Theory of Computation (CSC257)

Note By:

- **Roshan Bist**
- SNSC, Mahendranagar
- https://www.facebook.com/roshanbist.roshanbist.3
- https://www.facebook.com/notejunction/
- notejunction360@gmail.com

Unit-1
Basic Foundations

Purpose of TOC → To develop formal mathematical models of computation that reflect real-world computers.

⊛. Review of Set Theory, Logic, Functions, Proofs: [less imp]

1. Sets: A set is a collection of well defined objects. For example: The set of odd positive integer less than 15 can be expressed by:

$S = \{1, 3, 5, 7, 9, 11, 13\}$.

or, $S = \{x \mid x \text{ is odd and } 0 < x < 15\}$

A set is finite if it contains finite number of elements in a set, otherwise infinite. The empty set has no element and is denoted by $\emptyset$.

Cardinality of set → It represents number of elements within a set.

Subset → A set A is subset of a set B if each element of A is also element of B and is denoted by $A \subseteq B$.

Unit set → A set containing only one element.

Universal set → A set of all entities in the current context.

Power set → The set of possible subsets from any set is known as power set. A set with n elements has 2^n subsets.

⇒ There are some set operations like Union, Intersection, Complement, difference etc. we all know about them.

2. Logic: Logic is the study of the form of valid inference and laws of truth. A valid inference is one where there is a specific relation of logical support between the assumptions of the inference and its conclusion. In ordinary discourse, inferences maybe signified by words such as therefore, thus, hence and so on.

Propositional Logic → It is the way of joining or modifying propositions to form more complicated propositions as well as logical relationships. In propositional logic there are two types of sentences: simple sentences and compound sentences. Simple sentences express atomic propositions about the world. Compound sentences express logical relationships between the simpler sentences of which they are composed.

Propositions → Proposition is a declarative sentence that is either true or false but not both.

Example: $2+6=4$ (False), is a proposition
$2+3=5$ (True), is a proposition
$x>5$ (True and False based on value of x), is not a proposition.

Predicate → Any declarative statements involving variables often found in mathematical assertion and in computer programs, which are neither true nor false when the values of variables are not specified is called predicate.

For example: $5>3$, is not a predicate rather it is propositional statement because it is false.

But the statement "$x>4$" is predicate. The result is neither true nor false, it depends on the value of variable 'x'. The predicate "$x>4$" has two parts variable part (i.e, x) called subject and another relation part (i.e, >4). We can denote the statement "$x>4$" by $P(x)$. Once value is assigned to the propositional function then we can tell whether it is true or false. i.e, a proposition.

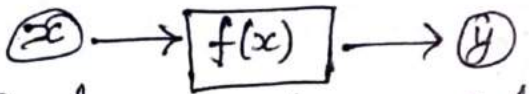
Quantifiers → Quantifiers are the tools that change the propositional function into a proposition. These are the word that refers to quantifiers such as "some" or "all" and indicates how frequently a certain statement is true. Construction of propositions from the predicates using quantifiers is called quantifications.

There are two types of quantifiers: Universal and Existential.

Universal quantifier → The phrase "for all" denoted by $\forall$, is called universal quantifier.

Existential quantifier → The phrase "there exist", denoted by $\exists$, is called existential quantifier.

3. Functions: If A and B are two sets, a function f from A to B is a rule that assigns to each element x of A an element f(x) of B. For a function f from A to B, we call A the domain of f and B the co-domain of f.



Input Function Output.

Relations → A binary relation on two sets A and B is a subset of $A \times B$. For example, if $A = \{1, 3, 9\}$, $B = \{x, y\}$, then $\{(1,x), (3,y), (9,x)\}$ is a binary relation on 2-sets.

A binary relation r is an equivalance relation if R satisfies:

- → R is reflexive i.e, for every x, $(x,x) \in R$.
- → R is symmetric i.e, for every x and y, $(x,y) \in R$ implies $(y,x) \in R$.
- → R is transitive i.e, for every x, y and z, $(x,y) \in R$ and $(y,z) \in R$ implies $(x,z) \in R$.

Closures → Closure is an important relationship among sets and is a general tool for dealing with sets and relationship of many kinds. Let R be a binary relation on a set A. Then the reflexive closure of R is a relation such that:

- → R' is reflexive (symmetric, transitive).
- → $R' \supseteq R$
- → If R'' is a reflexive relation containing R then $R \supseteq R$.

4. Methods of proofs / Proof Techniques: (Direct, Indirect, Contradiction)

These are the things exactly we read in discrete structure in 2nd sem. Not much more imp here so I'am leaving it. Now, important part of this chapter starts from below.

⊛. Theory of Computation (TOC):

It is a study of power and limits of computing having three interacting components: Automata Theory, Computability Theory and Complexity Theory.

Computability Theory

- What can be computed?
- Are there problems that no program can solve?

Complexity Theory

- What can be computed efficiently?
- Are these problems that no program can solve in a limited amount of time or space?

Automata Theory

- Study of abstract machine and their properties, providing a mathematical notation of "computer".
- Automata are abstract mathematical models of machines that perform computations on an input by moving through a series of states or configurations. If the computation of an automata reaches an accepting configuration it accepts that input.

Study of Automata

- For software designing and checking behaviour of digital circuits.
- For designing software for checking large body of text as a collection of web pages, patterns etc. like pattern recognization.
- Designing "lexical analyzer" of a compiler, that breaks input text into logical units called "tokens".

Abstract Model: An abstract model of computer system ~~(considered)~~ (considered either as hardware or software) constructed to allow a detailed and precise analysis of how the computer system works. Such a model usually consists of input, output and operations that can be performed and so can be thought of as a processor. E.g. an abstract machine that models a banking system can have operations like "deposit", "withdraw", "transfer" etc.

⊛. Basic Concepts of Automata Theory [Imp];

Alphabets → Alphabet is a finite non-empty set of symbols. The symbols can be the letters such as $\{a,b,c\}$, bits $\{0,1\}$, digits $\{0,1,2\ldots9\}$, Common characters like '$', #, * etc.
For Example:

$\Sigma = \{0,1\}$ → Binary alphabets

$\Sigma = \{+,-,*\}$ → Special Symbols.

Strings → String is a finite sequence of symbols taken from some alphabet. E.g. 0110 is a string from binary alphabet, "computation" is a string from alphabet $\{a,b,c,\ldots z\}$.

Length of string → The length of string w, denoted by $|w|$, is the number of symbols in w.
Example: string w = computation has length 11.

Empty string → It is a string with zero occurrences of symbols. It is denoted by 'ε' (epsilon). The length of empty string is zero. i.e, $|\varepsilon| = 0$.

Power of Alphabet → The set of all strings of certain length k from an alphabet is the k^{th} power of that alphabet i.e, $\Sigma^k = |w/w| = k$
If $\Sigma = \{0,1\}$ then,

$\Sigma^0 = \{\varepsilon\}$
$\Sigma^1 = \{0,1\}$
$\Sigma^2 = \{00,01,10,11\}$
$\Sigma^3 = \{000,001,010,011,100,101,110,111\}$

Kleen Closure:

The set of all the strings over an alphabet Σ is called kleen closure of Σ and is denoted by Σ^*. Thus, kleen closure is set of all the strings over alphabet Σ with length 0 or more.

$\therefore \Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$

E.g. $A = \{0\}$

$A^* = \left\{ \frac{0^n}{n=0,1,2\ldots} \right\}$

Positive Closure: The set of all the strings over an alphabet Σ except the empty string is called positive closure and is denoted by Σ^*.

$\therefore \Sigma^* = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$

Concatenation of Strings: If x and y are two strings over an alphabet, the concatenation of x and y is written xy and consists of the symbols of x followed by those of y.

For example:

$x = aaa$

$y = bbb$

$xy = aaabbb$

$yx = bbbaaa$

Note → Concatenating the empty string 'ε' with another string, the result is just the other string.

Suffix of a string: String is called suffix of a string w if it is obtained by removing zero or more leading symbols in w.

For example:

$w = abcd$

$S = bcd$ is suffix of w

S is proper suffix if $S \neq w$

Prefix of a string: A string is called prefix of a string w if it is obtained by removing zero or more trailing symbols of w.

For example:

w = abcd

s = abc is prefix of w.

s is proper ~~suffix of s~~ prefix if $s \neq w$.

Substring: A string s is called substring of a string w if it is obtained by removing zero or more leading or trailing symbols in w. It is proper substring if $s \neq w$.

Language: A language L over an alphabet Σ is subset of all the strings that can be formed out of Σ. i.e, a language is subset of kleen closure over an alphabet. Σ: $L \subseteq \Sigma^*$.
(Set of strings choosen from Σ^* defines language).

For example:

→ Set of all strings over $\Sigma = \{0,1\}$ that ends with 1.

$L = \{1, 01, 11, 011, 0111, \ldots\}$

→ English language is a set of strings taken from the alphabet

$\Sigma = \{a, b, c, \ldots z, A, B, C, \ldots, Z\}$.

$L = \{dbms, TOC, Graphics\}$.

Empty language: A language is empty if it does not have any strings within it. ϕ is an empty language and is a language over any alphabet. It does not contain any string.

Membership in a language: Membership in a language defines association of some string with the language. A membership problem is the question of deciding whether a given string is a member of some particular language or not. i.e, whether the string belongs to the given language or not.

Example: Given, $L = \{DBMS, TOC, GRAPHICS\}$, then for W="DBMS", the membership of w is true in L. ~~and hence w ∈ L~~ For W="HELLO", the membership of w is false in L. ~~and hence w~~

Unit-2

Introduction to Finite Automata:

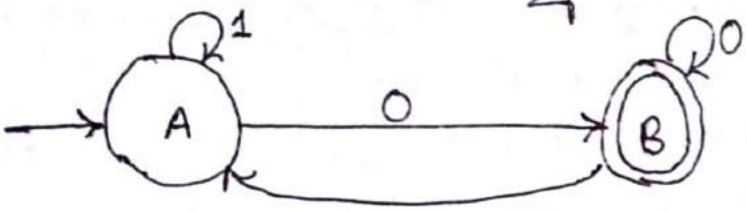
⊛. Finite Automata / Finite State Machine (FSM):

Finite Automata (FA) is the simplest machine to recognize patterns. The Finite state machine is an abstract machine which has five elements or tuple. It has a set of states and rules for moving from one state to another but it depends on applied input symbol. Basically it is an abstract model of digital computer. A finite automata consists of five tuples $(Q, \Sigma, q_0, F, \delta)$. where:-

$Q \rightarrow$ Finite set of states
$\Sigma \rightarrow$ Set of input symbols
$q_0 \rightarrow$ Initial state
$F \rightarrow$ Set of final states
$\delta \rightarrow$ Transition function.

automata is also called as automation

Example: Let we have following FSM:



Things I write inside circles are for understanding only

Now, five tuples for this FSM can be described as;

$Q = \{A, B\}$ ← set of all states

$\Sigma = \{0, 1\}$ ← set of all inputs

$q_0 = A$ ← Whenever we see an arrow from nowhere pointing to a state then it is start state or initial state

$F = \{B\}$ ← set of final states, final states are denoted by double rings. Initial state is only one but final states can be more than one.

$\delta = Q \times \Sigma$ which can be described by the table below:

Other tuples are same but formula for δ will be different for DFA, NFA, ε-NFA. $\delta = Q \times \Sigma$ is for DFA

	0	1
A	B	A
B	B	A

we write inputs at top (i.e, at column)

we write states at left (i.e, at row)

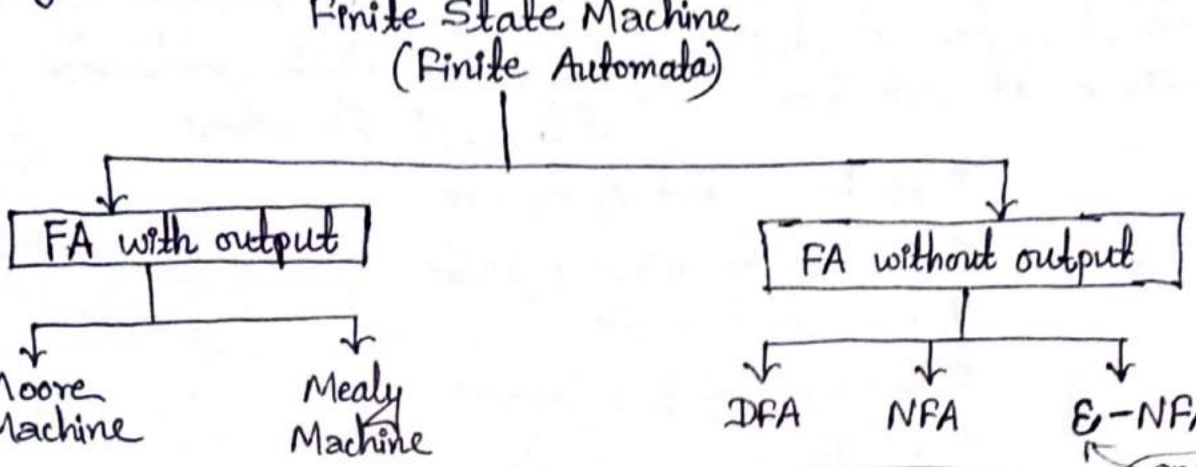
Initially we are at start state A. Now A on getting input 0 it goes to B. similarly for others

transition table (i.e, δ) for above diagram considering it as DFA.

Applications of FSM:

The finite state machines are used in applications in computer science and data networking. For example; finite-state machines are basis for programs for spell checking, indexing, grammar, searching large bodies of text, recognizing speech, transforming text using markup languages such as XML and HTML, and network protocols that specify how computers communicate.

Categories of FSM:

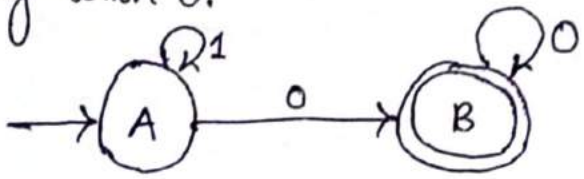


5 tuples already discuss गरेको भएर यसमा दोहोऱ्याएको छ exam मा आए 5 tuples पनि we we discuss गरेको did before

1) Deterministic Finite Automata (DFA):

It is the simplest model of computation which has a very limited memory. In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. In DFA null (or ε) move is not allowed i.e, DFA cannot change state without any input character. DFA consists of 5 tuples $\{Q, \Sigma, q_0, F, \delta\}$ which we already discussed. Write same for all except δ which is defined as $\delta: Q \times \Sigma \rightarrow Q$.

For example: The below DFA with $\Sigma = \{0, 1\}$ accepts all strings ending with 0.



⇒Note that, there can be many possible DFAs for a pattern. A DFA with minimum number of states is generally preferred.

⊛ Notations for DFA:

There are two preferred notations for describing DFA;

→ Transition table.

→ Transition Diagram.

Transition table:

Transition table is a conventional tabular representation of the transition function δ that makes the arguments from $Q \times \Sigma$ and returns a value which is one of the state of the automata. The row of the table corresponds to the states, while column of the table corresponds to the input symbol. The starting state of the table is represented by arrow ($\rightarrow$) followed by the state (i.e, represented as: $\rightarrow q$) where; q is considered any initial state. The final state is represented as $*q$ for q being final state. The entry for a row corresponding to state q and the column corresponding to input a, is the state $\delta(q, a)$.

For example: Consider a DFA as;

$Q = \{q_0, q_1, q_2, q_3\}$

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

$q_0 = q_0$

$F = \{q_0\}$

$\delta = Q \times \Sigma \rightarrow Q$.

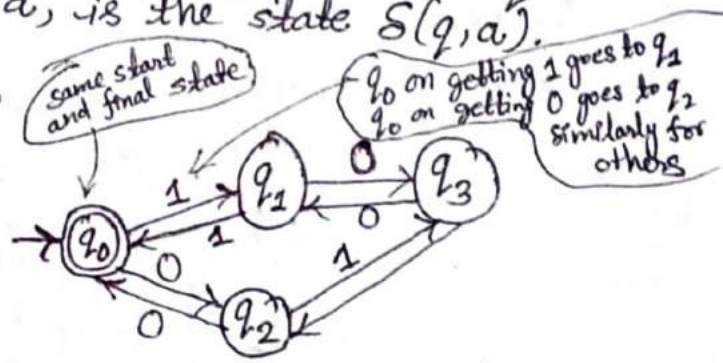


Fig: transition diagram.

Then, the transition table for above DFA is as follows:-

same start and final state

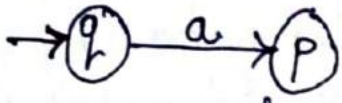
δ	0	1
$*\rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

look at figure q_0 on getting 0 goes to q_2 and q_0 on getting 1 goes to q_1 similarly for others

$\Rightarrow$ This DFA accepts strings having both an even number of 0's & even number of 1's.

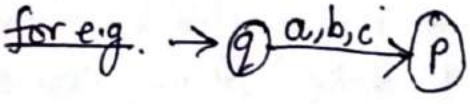
ii) Transition diagram:-

A transition diagram of a DFA is a graphical representation or a graph. For each q in Q and each input 'a' in Σ, if $\delta(q,a)=p$ then there is an arc from node q to p labelled as a in the transition diagram, as below;



(i.e, q on getting a goes to p).

If more than one input symbol cause the transition from state q to p then arc from q to p is labelled by a list of those symbols.

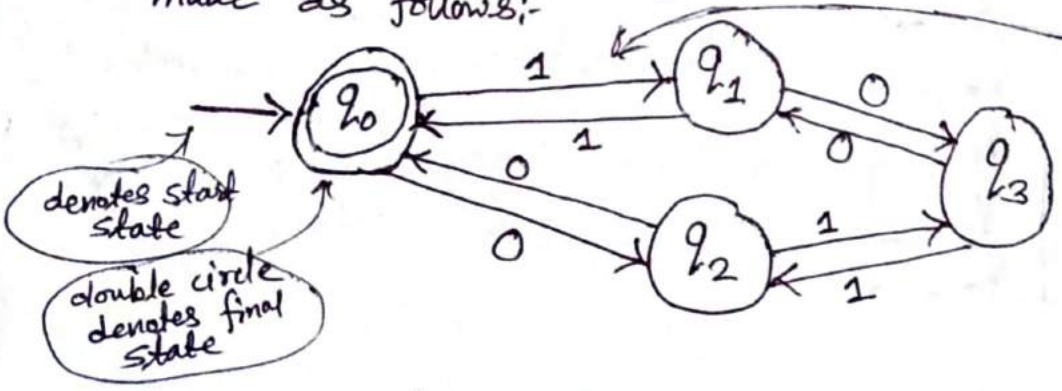
~~for e.g.~~ 

(i.e, q on getting a or b or c goes to p).

In transition diagram, start state is denoted by labelled by an arrow comming from nowhere and the final state or accepting state is marked by double circle. Let we have Following transition table;

* δ	0	1
→ q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Now the transition diagram for this transition table can be made as follows;-



from table
q_0 on getting 0 goes to q_2
& q_0 on getting 1 goes to q_1
similarly for others

order of tuples can change. Explain tuples also if asked in exam

(*). Language of DFA:-

The language of DFA, $M=(Q,\Sigma,\delta,q_0,F)$ denoted by $L(M)$ is a set of strings over Σ^* that are accepted by M. Σ^* is the set of all the input symbols. This means the language of DFA is the set of all strings (i.e, sequence of input symbols) that take DFA from start state to one of accepting states.

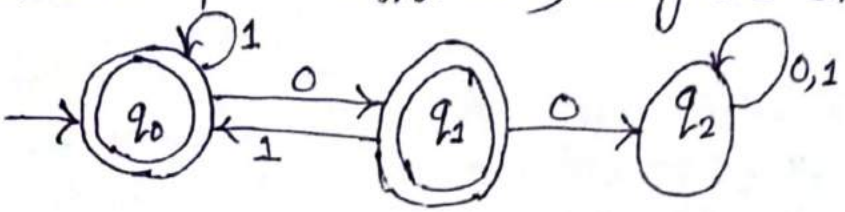
⊛ Extended Transition Function of DFA ($\hat{\delta}$):-

The extended transition function of DFA, denoted by $\hat{\delta}$ is a transition function that makes two arguments as input: one is the state q of Q and another is a string belonging to Σ^* and generates a state $p \in Q$. This state p is reached when the current state processes the sequence of inputs.

i.e, $\hat{\delta}(q, W) = p$

where, q = current state.
W = sequence of inputs.
P = generated or new reached state.

For Example: Compute $\hat{\delta}(q_0, 1001)$ using the DFA below:



Here,

$\hat{\delta}(q_0, 1001)$
$= \delta(\hat{\delta}(q_0, 100), 1)$
$= \delta(\delta(\hat{\delta}(q_0, 10), 0), 1)$
$= \delta(\delta(\delta(\hat{\delta}(q_0, 1), 0), 0), 1)$
$= \delta(\delta(\delta(\delta(\hat{\delta}(q_0, \varepsilon), 1), 0), 0), 1)$
$= \delta(\delta(\delta(\delta(q_0, 1), 0), 0), 1)$
$= \delta(\delta(\delta(q_0, 0), 0), 1)$
$= \delta(\delta(q_1, 0), 1)$
$= \delta(q_2, 1)$
$= q_2$, since q_2 is not final state, so the string 1001 is not accepted.

first expand until we get $\hat{\delta}(q_0, \varepsilon)$ ↑ empty

Now solve one by one. $\hat{\delta}(q_0, \varepsilon)$ is always q_0 since ε is empty

Now we solved $\delta(q_0, 1)$. q_0 on getting 1 (in the fig.) goes to q_0 itself so $\delta(q_0, 1) = q_0$. Similarly we proceed rest

Understanding what are string and other terms

Symbol → a, b, c, 0, 1, 2, 3 ...
Alphabet → Collection of symbols
e.g. {a, b}, {0, 1, 2} ...
String → Sequence of symbols
e.g. a, b, 0, 1, aa, bb, 01, aaa, 111 ...
Language → Set of strings e.g. {0, 1}, {aa, bb} ...

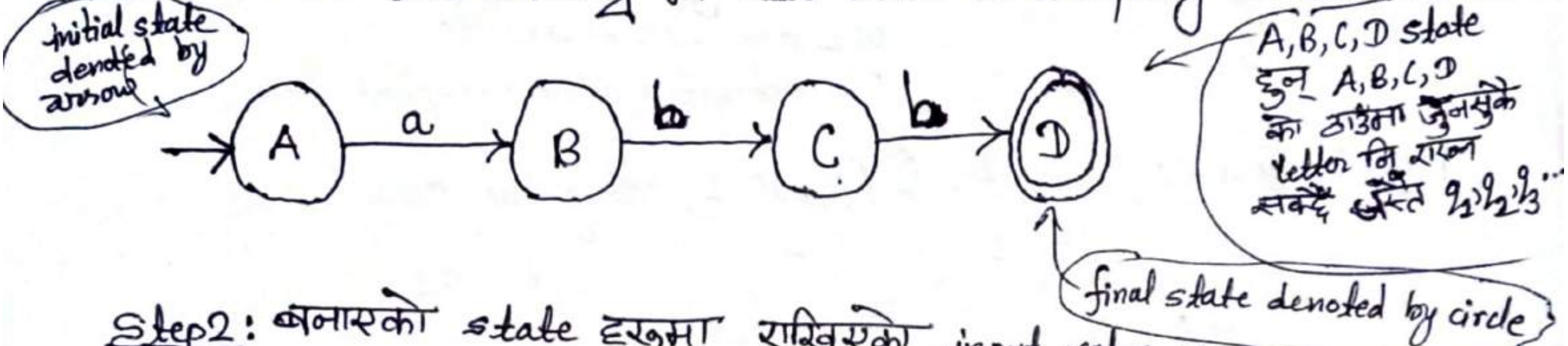
q_2 को सट्टा finally हामिले q_0 वा q_1 पाको भए string 1001 accept हुने थियो किनकि q_0 र q_1 final state हुन् जसलाई double circle ले denote गरिएको छ।

⊛. Constructing DFA with examples:-/Examples of DFA:

(for understanding only)

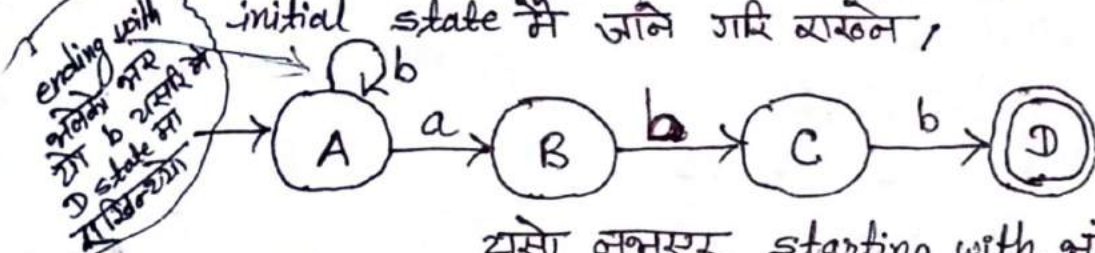
Method/Steps: Let we are going to construct a DFA, that accepts all strings over $\Sigma=\{a,b\}$ ~~that accepts~~ ending with abb.

Step1: Question मा दिइएको accept गर्ने condition अनुसार, सुरुमा त्यो condition accept गर्ने diagram मात्र बनाउने initial state देखि final state सम्म / According to above question taken ~~let~~ we are drawing for abb which is accepting condition.



Step2: बनाएको state हरूमा राखिएको input value Σ भन्दा अरु value ~~Σ~~ Σ मा भएको राख्दा दिएको condition लाई असर नगर्ने गरि मिलाउने जसमा तलका sub-step काम लाग्छन् ।

i) Question मा ending with भनेको छ भने start state मा already place गरिएको ~~input~~ input बाहेक Σ मा बचेका अरु input त्यहि initial state मै जाने गरि राख्ने ।

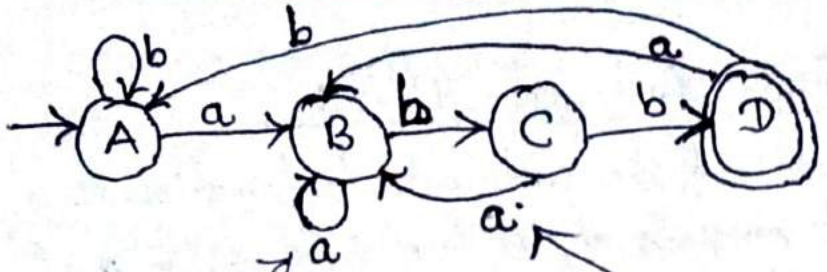


यसो नभएर starting with भनेको भए जसरी initial state मा राख्यौं त्यसै गरि initial state मा नभएर final state मा राखिन्छ। starting with, ending with केहि भनेको छैन र containing string, having string भनेको छ भने initial र final दुबैमा राखिन्छ ।

ii) बचेका state मा input गर्ने बाँकी string राख्ने र राख्दा transition (arrow) कुन state मा जान्छ analyse गर्ने, यदि दिइएको condition ~~बिगार्छ~~ बिगार्दैन भने त्यसै current state मा रहन्छ /

(बिगार्दैन या ~~यदि string~~ चाहिएको string फेरि पाउने सम्भावना छ भने)

→ ~~or~~ त्यसो नभएर यदि दिएको condition मा भएको string बिगार्ने word आउछ भने input मा (for eg. abb sequence बिगार्ने according to example) त्यतिबेला त्यो word first ~~a~~ चोटि पाएर कुन state मा पुगेका थिये त्यहि state मा arrow पुर्याउने /



③ Similarly D state मा कुनै पनि input दैन गरिसको so, a र b दुबैको लागि check गर्ने

① State B मा b already input दियो अब a को लागि check गर्ने। हामी condition abb हो यसमध्येको सुरुको word a पाए already B मा छ। अब यदि a अर्को b state मा string aabb type को हुन्छ जसमा देख्न सक्छौं aabb ending with abb condition अनुसार पाउने सम्भावना छ, त्यसैले यसलाई sequence बिगार्दैन त्यसैले Step 2 (ii) को अनुसार त्यहि state मा बस्यो।

② Similarly B state को जस्तै state C मा already b input भएर D मा गएको छ। So, अब बचेको अब a मात्र हो। abb condition मा state C मा a आउदा abab हुन्छ जसले दिएको condition abb को sequence बिगार्छ। abb पाउने सम्भावना दैन त्यसैले पहिलो चोटि a पाएर जुन state मा (state B) मा पुगेका थियौं त्यहि state मा जान्छ। (According to Step2 (ii) 2nd part.)

Now we get Complete DFA according to the question.

Step3: Note that कुनै word कुनै पनि state मा जान सकेन यो condition अनुसार भने त्यो नया state dead state मा जान्छ र dead state मा सबै input value त्यहि रहन्छन्।
Question हरु जस्मा restrictions हुन्छन् (जस्तै having length 2 only) यस्ता type मा dead state हुन्छ। Restriction नभएको case मा dead state हुँदैन जस्तो अहिले गरेको step सँगै को example मा।

⇒ Question मा does not भनेर आयो भने त्यसलाई पहिला Positive बनाएर solve गर्ने (for e.g. does not contain लाई contains बनाउने) र finally; final state जति सबैलाई non-final state बनाउने र non-final जति सबै state लाई final बनाउने।

There is another method to construct DFA. As we know constructing NFA is easy so first construct NFA then convert it to DFA. Conversion is dislussed later.

Practice DFA questions as much as you can. If you understand DFA then it is easy to understand whole chapter.

2) Non-Deterministic Finite Automata (NFA):

A non-deterministic finite automata/automation is a mathematical model that consists of; 5 tuples $(Q, \Sigma, \delta, q_0, F)$ where,

$Q \rightarrow$ set of states. (finite)

$\Sigma \rightarrow$ A set of input symbols (finite)

$\delta = Q \times \Sigma \rightarrow 2^Q$, which is a transition function that maps state symbol pair to sets of states.

$q_0 \rightarrow q_0 \in Q$, which is initial or starting state.

$F \rightarrow$ set of final states, where $F \subseteq Q$.

all are same as DFA except δ definition

There are two main things that we should keep in mind:

i) NFA does not contain any dead state.

ii) Unlike DFA, a transition function in NFA takes the NFA from one state to several states just with a single input.

For Example1:

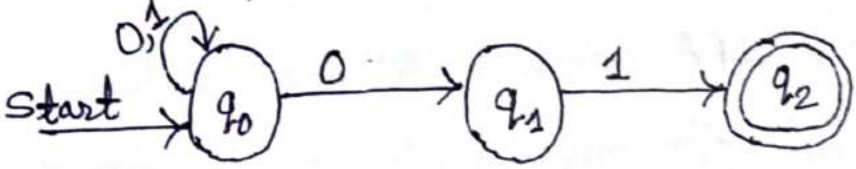
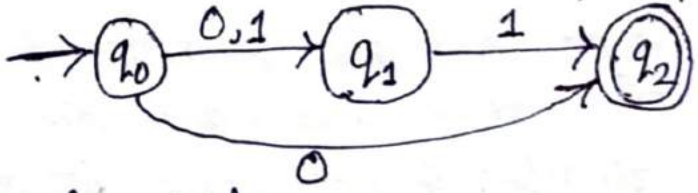


fig. NFA accepting all strings that end in 01.

Here, q_0 state on getting 0 goes to q_0 itself as well as q_1. But in DFA on getting particular input it goes to one particular state only.

Example2: NFA over {0,1} accepting strings {0, 01, 11}.



transition table:

δ	0	1
$\rightarrow q_0$	$\{q_0, q_2\}$	q_1
q_1	$\emptyset$	q_2
$*q_2$	$\emptyset$	$\emptyset$

* shows final state

$\emptyset \rightarrow$ denotes it goes no where i.e empty

Constructing NFA is easier than DFA. No need to draw dead state. Draw only what it accepts. With one input may go to more than one state.

DFA without Dead state is NFA which can go to many states on single input

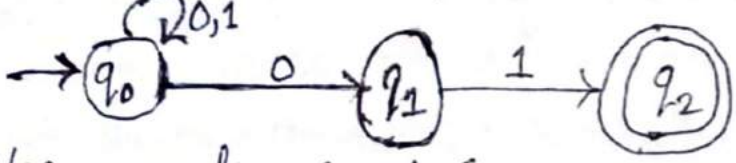
⊛. **Notations for NFA:** Notations for NFA are also same as DFA. [Theory write same only change example by writing example of NFA].

⊛. **Language of NFA:** Language for NFA is also same as DFA except definition of $\delta = Q \times \Sigma \to 2^Q$. In DFA $\delta = Q \times \Sigma \to Q$. [Write same theory as we wrote in DFA explaning 5 tuples].

⊛ **Extended Transition function of NFA:** ← Theory similar to DFA only change example

The extended transition function $\hat{\delta}$ is a function that takes a state q and a string of input symbol w and returns the set of states. In DFA it returns a single state but NFA can return more than one state.

Example: Consider a NFA:



Now computing for $\hat{\delta}(q_0, 01101)$

Solution:

$\hat{\delta}(q_0, 01101)$

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

$\hat{\delta}(q_0, 0) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 01) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

$\hat{\delta}(q_0, 011) = \delta(q_0, 1) \cup \delta(q_2, 1) = \{q_0\} \cup \{\phi\} = \{q_0\}$

$\hat{\delta}(q_0, 0110) = \delta(q_0, 0) = \{q_0, q_1\}$

$\hat{\delta}(q_0, 01101) = \delta(q_0, 1) \cup \delta(q_1, 1)$

$= \{q_0\} \cup \{q_2\}$

$= \{q_0, q_2\}$

$= \text{Accepted}.$

Since $(q_0, 0) = \{q_0, q_1\}$ in above step
so, $((q_0, 0), 1) = (q_0, 1) \cup (q_1, 1)$
q_0 q_2

$(q_0, 01) = \{q_0, q_2\}$
$((q_0, 01), 1) = (q_0, 1) \cup (q_2, 1)$

$(q_0, 011) = \{q_0\}$
$((q_0, 011), 0) = \{q_0, 0\}$
q_0 on getting 0 goes to q_0 & q_1

finally find गरेको set मा कुनै एउटा state पनि final state देखिए accept हुन्द अन्य reject

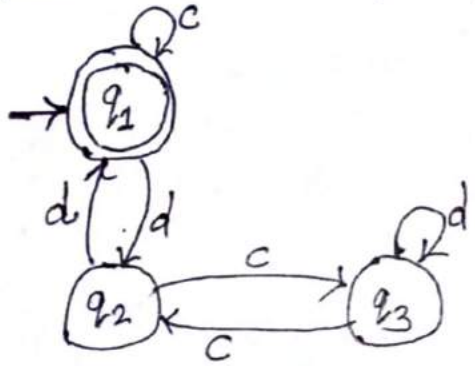
⊛ Equivalence of NFA & DFA:-

The equivalence of two finite automata determines whether the two automatas are equal or not. i.e, the two automatas perform the same function. The languages they accept will be same. So, in order to identify two automatas are equivalent or not there are certain steps we need to follow:-

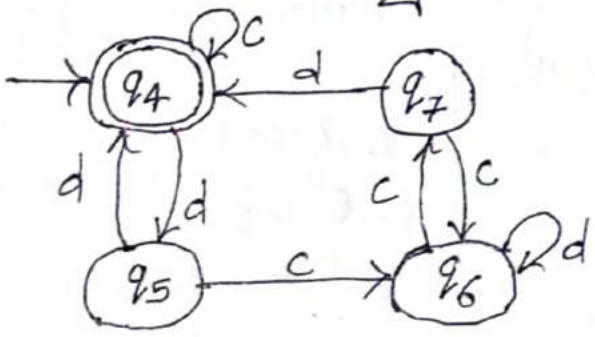
1) If initial state is final state of one automation, then in second automation also initial state must be final state for them to be equivalent.

2) For any pair of states $\{q_i, q_j\}$ the transition for input $a \in \Sigma$ is defined by $\{q_a, q_b\}$ where $\delta\{q_i, a\} = q_a$ and $\delta\{q_j, a\} = q_b$. The two automata are not equivalent if for a pair $\{q_a, q_b\}$ one is intermediate state and the other is final state.

Example: Let we have following two automaton namely A & B.



Automaton (A)



Automaton (B)

⇒ In automation A, q_1 is the initial state as well as final state and In automation B also q_4 is the initial state as well as the final state so it satisfies our first condition.

⇒ Now we check second condition, for that let we first construct pair of states from two automaton and check transition in the table as follows:-

States	Inputs	
	c	d
(q_1, q_4)	(q_1, q_4) FS FS	(q_2, q_5) IS IS
(q_2, q_5)	(q_3, q_6) IS IS	(q_1, q_4) FS FS
(q_3, q_6)	(q_2, q_7) IS IS	(q_3, q_6) IS IS
(q_2, q_7)	(q_3, q_6) IS IS	(q_1, q_4) FS FS

IS → Intermediate state
FS → Final state

Now we can see that all the pairs checked have transition in c and d both are either initial states or final states. This satisfies the second condition.

⇒ Hence both automation A and B are equivalent.

Note: While making or checking transition if we found two pairs are not same (i.e, both are not intermediate states or final states) then we stop there and conclude as not equivalent.

⊛ Conversion of NFA to DFA: (using subset construction method):

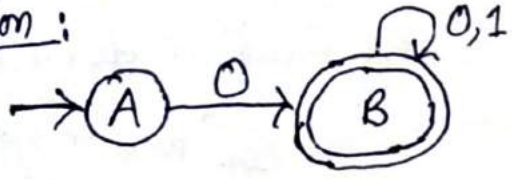
⇒ Remember that every DFA is an NFA but not vice versa. But there is an equivalent DFA for every NFA.

language

Example 1: Construct NFA for L = {Set of all strings over (0,1) that starts with '0'} then convert it to equivalent DFA.

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

Solution:



δ	0	1
→A	B	∅
*B	B	B

empty set, A on getting 1 goes nowhere

This is the required DFA with transition table for given question.

Now, to convert to equivalent DFA first we construct transition table for DFA looking at (or using) transition table of NFA.

δ	0	1
→A	B	C
*B	B	B
C	C	C

In NFA A on getting 1 goes to ∅ but in DFA transition can not go to ∅. So, we made new state C for transition A on getting 1. Now we proceed on the basis of new states.

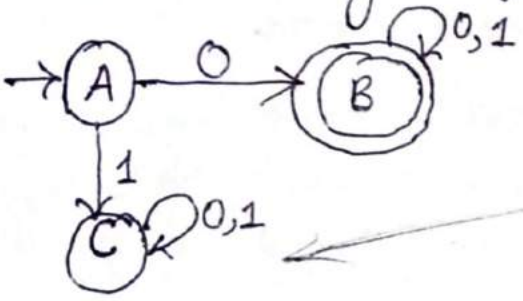
For this next step we look in first row and states that are not input till now are used as input. B is not taken input so we take B

Analysing table for NFA we know that B on getting 0 and 1 both goes to B

Now look before rows A, B. Here, C is not taken as input till now. So, we took C as input

∅ was changed to C. So, C is dead state, C on getting any input goes itself to C. Now A, B, C all are used as input no state remains so we stop there.

Now, based on this transition table for DFA we can construct DFA easily as follows:-



Here, C is dead state or trap state

Example 2: Find the equivalent DFA for the NFA given by $M=[\{A,B,C\},(a,b),\delta,A,\{C\}]$ where, δ is given by;

these are 5 tuples where, [states, input, δ, Initial state, final state]

δ	a	b
→A	A,B	C
B	A	B
*C	∅	A,B

Solution:-

For converting into equivalent DFA first we draw transition table as follows:-

δ	a	b
→A	AB	C
→AB	AB	BC
*C	D	AB
*BC	A	AB
D	D	D

In NFA on single input it can go to more than one state but in DFA on single input goes to single state only so we combine A and B as AB as one state

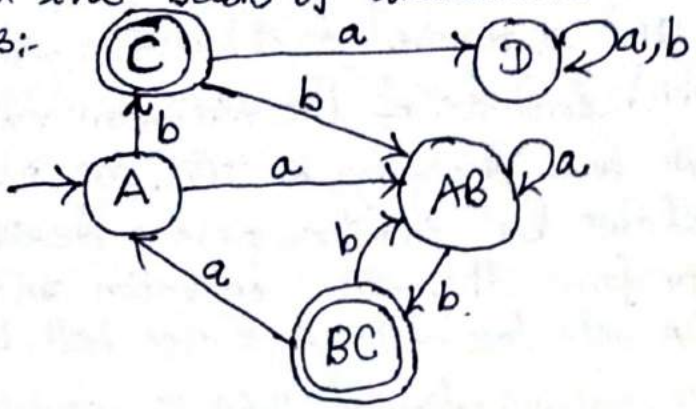
New state from first row, we combined AB two states as one so AB on getting a, the union operation of A and B on getting a in transition table of NFA is done. (i.e, $(A,B)UA \equiv A,B \equiv AB$)

∅ changed to new state

Since C state was final state in NFA so, C containing all states are final here

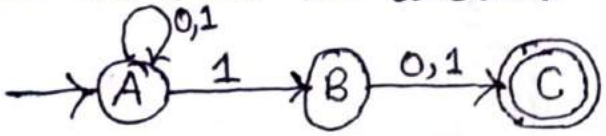
D is dead state so on getting any input goes to itself

Now on the basis of transition table equivalent DFA is as follows:-



Example 3: Design an NFA for a language that accepts all strings over {0,1} in which the second last symbol is always '1'. Then convert it to its equivalent DFA.

Solution:- Let first we construct NFA with its transition table:



δ	0	1
→A	A	A,B
B	C	C
*C	∅	∅

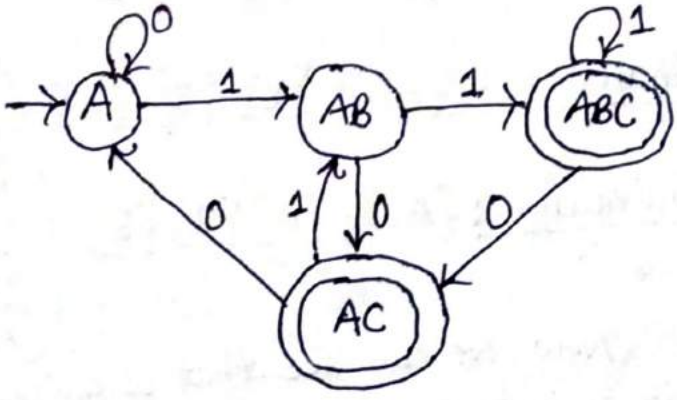
Now to convert into equivalent DFA, let we construct transition table for DFA using transition table of NFA.

δ	0	1
→A	A	AB
AB	AC	ABC
*AC	A	AB
*ABC	AC	ABC

Since DFA can not go more than one state so combined as one

Union operation of A and B on getting 0 from NFA table above. Similarly for ABC

AC and ABC are final states because they contain C which is final state in NFA



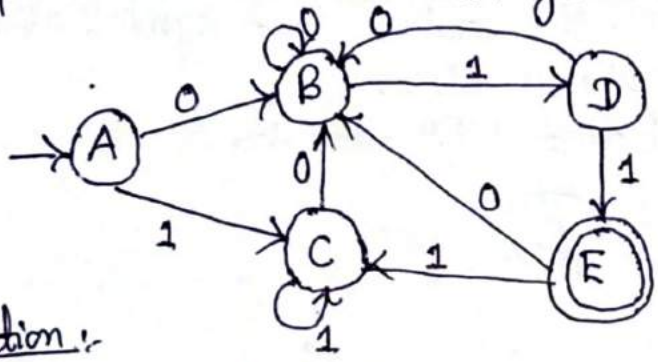
Note: Question मा NFA को figure दिएको भए DFA मा convert गर्न भने पहिलेगरि पहिला दिएको figure अनुसार NFA को transition table बनाउने अनिमात्र convert गर्ने भनेर गरि,

(*) Minimization of DFA:

(Not in micro-syllabus but maybe imp.)

Minimization of DFA is required to obtain the minimal version of any DFA which consists of the minimum number of states possible. Let we are designing a DFA, one person designed it with 5 states but another person designed with 4 states. But both perform the same operation and are equivalent. Obviously DFA with less states 4 states will be better with performance. Hence, minimization of DFA is required.

Example1: Minimize the DFA given below:



Solution:-

Let we construct transition table for given DFA.

S	0	1
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

Now based on transition table we construct equivalences as;

(for constructing 0 equivalence we construct 2 seperate sets. one set for all non-final states and another for all final states.)

0 Equivalence: {A,B,C,D} {E}

1 Equivalence: {A,B,C} {D} {E}

(for 1 equivalence take sets more than one elements from 0 equivalence then check if they are 1 equiva--- to one another or not. If equivalent keep in same set otherwise seperate set. for eg. checking for A and B. A and B on getting 0 goes to B and B which are on same set in 0 equivalence. & A and B on getting 1 goes to C and D are which are also inside same set in 0 equivalence. So they can stay in same set. Similarly proceed for all except single sets.)

(Once we know A&B are on same set now for checking C can be done with any of them either A or B)

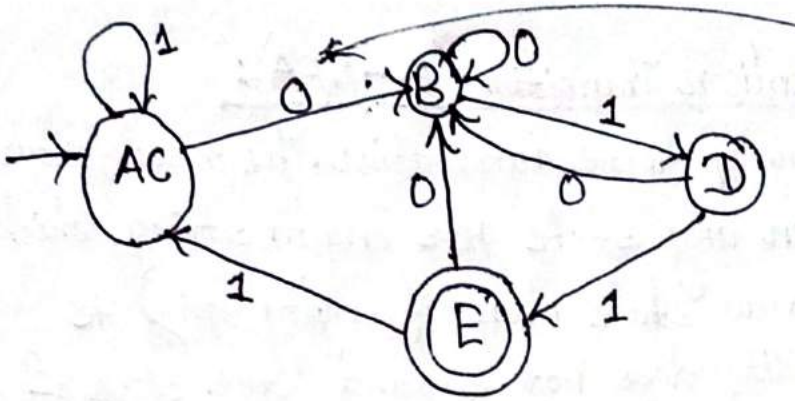
(using 1 equivalence row)

2 Equivalence: {A,C} {B} {D} {E}

3 Equivalence: {A,C} {B} {D} {E}

(using 2 equivalence row)

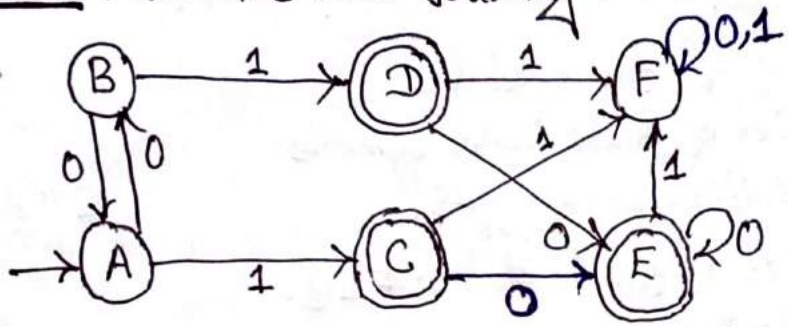
Now we can see that 2 equivalence and 3 equivalence are giving same results so we stop process and now we can construct minimized DFA combining states A & C as AC one.



Look at table and draw
A on getting 0 input goes to B & C also on getting 0 goes to B

Now we minimized 5 states DFA to 4 states DFA where both perform same operation.

Example 2: Minimize the following DFA:



Solution:-

S	0	1
→A	B	C
B	A	D
*C	E	F
*D	E	F
*E	E	F
F	F	F

Non-final states

final states

0 Equivalence: {A,B,F} {C,D,E}

1 Equivalence: {A,B} {F} {C,D,E}

2 Equivalence: {A,B} {F} {C,D,E}

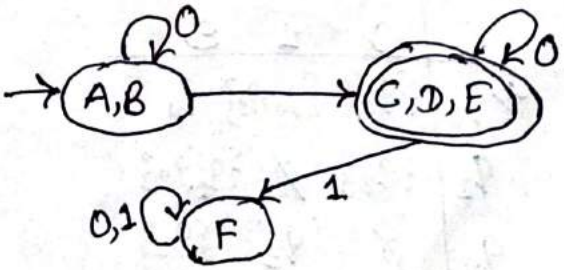
Now, we construct transition table for minimized DFA for easier.

S	0	1
→{A,B}	{A,B}	{C,D,E}
{F}	{F}	{F}
*{C,D,E}	{C,D,E}	{F}

A & B are combined now as one as AB one state. So, AB on getting 0 goes to B∪A = AB = {A,B}

A & B on getting 1 goes to CD, CD as new state is {C,D,E}

C, D, E is now one state



Note:- If there are any unreachable states (i.e, having no incoming arrow only outgoing arrow) then first we eliminate that state then we proceed as usual.

3) Finite Automaton with Epsilon Transition (ε-NFA):

This is extension of finite automation. The new feature ε allows a transition when any state gets empty string, which means it can switch to new state even if empty string is taken as input. This capability does not expand the class of languages that can be accepted by finite automata, but it does give some added "programming convenience".

Definition: A NFA with ε-transition is defined by five tuples $(Q, \Sigma, \delta, q_0, F)$ where;

Q = set of finite states

Σ = Set of finite input symbols

q_0 = Initial state, $q_0 \in Q$.

F = Set of final states; $F \subseteq Q$.

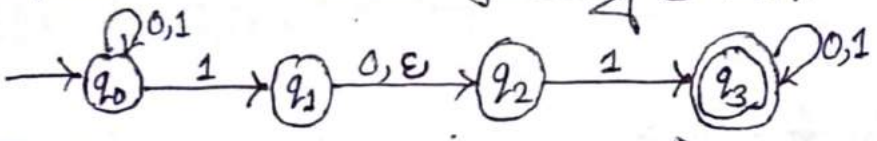
$\delta = Q \times \Sigma \cup \{\varepsilon\} \to 2^Q$, which is a transition function with two arguments.

→ a state Q

→ A member of $\Sigma \cup \{\varepsilon\}$ that is either an input symbol, or the symbol ε.

δ only different others are same as usual

Example: Consider the following ε-NFA.



If at least one ε symbol is seen in diagram then we have to know that it is ε-NFA

Now,

$Q = \{q_0, q_1, q_2, q_3\}$

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

$q_0 = q_0$

$F = \{q_3\}$

& δ is given as;

δ	0	1	ε
→q_0	$\{q_0\}$	$\{q_0, q_1\}$	q_0
q_1	$\{q_2\}$	$\emptyset$	$\{q_1, q_2\}$
q_2	$\emptyset$	$\{q_3\}$	$\{q_2\}$
*q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$

keep in mind that all states contain ε and every state on getting ε as input goes to itself.

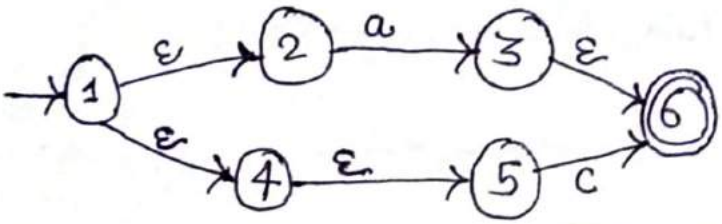
(ε नदिएको बेला त्यहि state मा जान्छ / ε दिएको ६ भने त्यो state र ε आउदा जानेको state दुबै मा जान्छ। for e.g. in fig. q_1 on getting ε goes to q_1 itself and q_2;

Notations for ε-NFA:- Notations for ε-NFA are also same as notations for DFA. [i.e, describing transition table and transition diagram taking examples of ε-NFA].

Epsilon Closure of a state:-(ε*):

Epsilon closure is the set of states that can be reached without reading any input symbol (i.e, only reading ε) from a particular state.

Example: Consider the ε-NFA below.



Then,

ε-closure(1) = {1, 2, 4, 5}

ε-closure(2) = {2}

ε-closure(3) = {3, 6}

ε-closure(4) = {4, 5}

ε-closure(5) = {5}

ε-closure(6) = {6}.

current state बाट ε मोर्दे हुँदा कुन कुन state मा पुग्न सकिन्छ (यसको list) 1 on getting ε goes to itself so, 1 also counted

goes to itself. No other outgoing arrow on getting ε. So, only 2 itself.

Extended transition function of ε-NFA:-

The extended transition function of ε-NFA denoted by $\hat{\delta}$ is defined by;

i) Basis Step:

$$\hat{\delta}(q, \varepsilon) = \varepsilon\text{-closure}(q)$$

ii) Induction step: Let w=xa be a string, where x is substring of w without last symbol a and $a \in \Sigma$ but $a \neq \varepsilon$.

Let $\hat{\delta}(q, x) = \{p_1, p_2, \ldots p_k\}$ i.e, p_i's are the states that can be reached from q following labeled x which can end with many ε & it can have many ε.

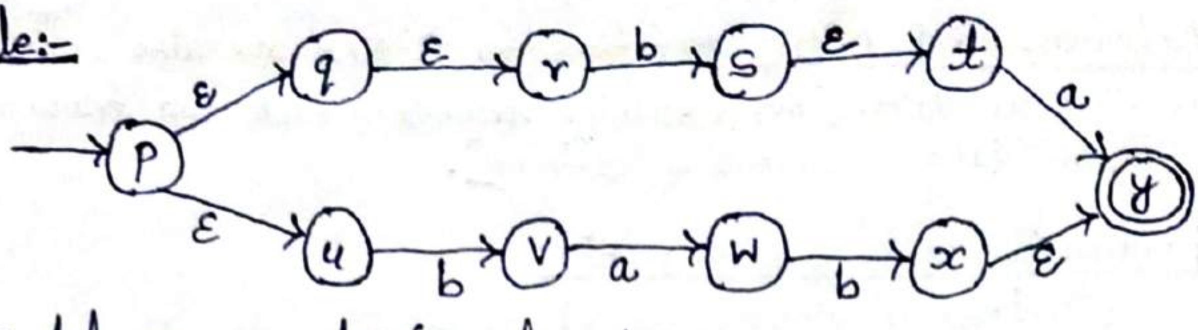
Also let,

$$\bigcup_{i=1}^{k} \delta(p_i, a) = (r_1, r_2, \ldots r_m)$$

i.e, union of $\delta(p_i, a)$ from $i=1$ to k

Then, $\delta(q, x) = \bigcup_{j=1}^{m} \varepsilon\text{-closure}(r_j)$.

Example:-



Now, let we compute for string ba.

(States that can be reached with input ε from state p)

i) $\delta(p, \varepsilon) = \varepsilon\text{-closure}(p) = \{p, q, r, u\}$

ii) Now we compute for b as;

$\delta(p,b) \cup \delta(q,b) \cup \delta(r,b) \cup \delta(u,b) = \{s, v\}$

(b is given input to all ε-closures(p) i.e p, q, r, u. Then we know δ(p, b) means state p on getting input b goes to nowhere. Similarly checking for others we get {s, v}.)

$\varepsilon\text{-closure}(s) \cup \varepsilon\text{-closure}(v) = \{s, t, v\}$

(i.e, states that can be reached from state s using input ε only. i.e, we get s, t) (union) (states that can be reached using ε from v. Only itself so v only.)

Similarly we compute for a as;

(latest states s, t, v are used now for input a)

$\delta(s,a) \cup \delta(t,a) \cup \delta(v,a) = \{y, w\}$

$\varepsilon\text{-closure}(y) \cup \varepsilon\text{-closure}(w) = \{y, w\}$

(final state is y now. एउटा पनि final state नभएमा सो set मा reject हुन्छ)

⇒ Now, the final result set i.e {y, w} contains the one of the final state so, the string is accepted.

(ε* → means epsilon closure)

⊛ Removing Epsilon Transition using concept of Epsilon Closure OR Converting ε-NFA to its equivalent NFA:

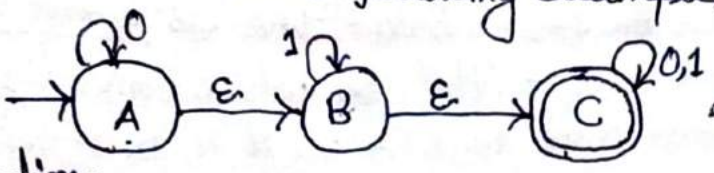
The procedure for converting any ε-NFA to its equivalent NFA is as follows:-

i) For each state we have to check that where does the state goes on ε*. [ε* is the set of all states that can be reached from a particular state only by seeing the ε symbol].

ii) Now set of states that we got in step (i), have to be checked on which state they go on getting a particular input.

iii) Now the set of states that we got in step (ii) are again checked on to which state do they go on ε* again.

Example: let we take following example, to understand better:



This is an example for converting ε-NFA to NFA

Solution:

We have, For NFA:

Start state = A

Input = {0,1}

Since there is no ε in NFA so ε as input is eliminated in NFA

Now we follow process for each state and input as follows:-

$\delta_N(A,0) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(A),0)$

$= \varepsilon\text{-closure}(\delta(\{A,B,C\},0))$

$= \varepsilon\text{-closure}(\{A,C\})$

$= \{A,B,C\}$

δ_N or simply δ.

for state A with input 0

Process step (1) completed in this line since ε-closure(A) is found i.e. states A is found, the states A on getting ε input are listed.

now A, B, C are checked where they go on getting input zero. A → A, B → ∅, C → C i.e. {A, C}

finally ε* for C done

ε*(A) = A, B, C

ε*(C) = C

therefore ε*(A, C) = {A, B, C}

$\delta_N(A,1) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(A),1)$

$= \varepsilon\text{-closure}(\delta(\{A,B,C\},1)$

$= \varepsilon\text{-closure}(\{B,C\})$

$= \{B,C\}$

for state A with input 1

$\delta_N(B,0) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(B),0)$

$= \varepsilon\text{-closure}(\delta(\{B,C\},0)$

$= \varepsilon\text{-closure}(\{C\})$

$= \{C\}$

$\delta_N(B,1) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(B),1)$

$= \varepsilon\text{-closure}(\delta(\{B,C\},1)$

$= \varepsilon\text{-closure}(\{B,C\})$

$= \{B,C\}$

$\delta_N(C,0) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(C),0)$

$= \varepsilon\text{-closure}(\delta(\{C\},0)$

$= \varepsilon\text{-closure}(\{C\})$

$= \{C\}$

$\delta_N(C,1) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(C),1)$

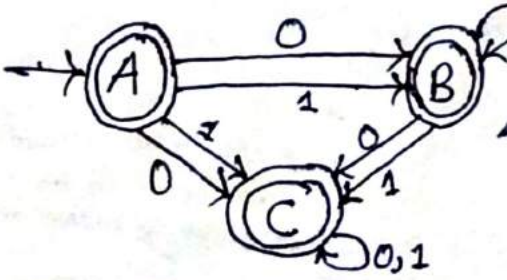
$= \varepsilon\text{-closure}(\delta(\{C\},1)$

$= \varepsilon\text{-closure}(\{C\})$

$= \{C\}$

⇒ Now, we can easily draw transition table and diagram for NFA as follows:-

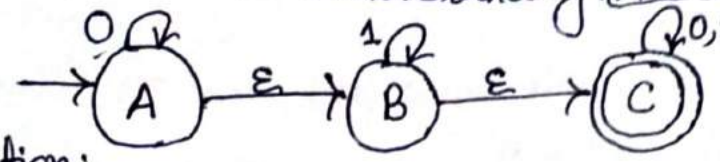
δ	0	1
*→A	{A,B,C}	{B,C}
*B	{C}	{B,C}
*C	{C}	{C}



Here A, B, C all are final states because. In NFA the final states are the states that can reach to final state of ε-NFA only by input ε

⊛ Converting ε-NFA to its equivalent DFA:

Converting ε-NFA process to its equivalent DFA is almost similar to converting equivalent DFA except that, while converting ε-NFA to NFA we used same start start in NFA as it is in ε-NFA but, now for converting ε-NFA to DFA, we will use ε-closure of start start in ε-NFA, as a start state in DFA. Let we take example below for clear understanding. Also step (i) is eliminated in this only step (ii) and (iii) applied.



Solution:

We have for DFA:

Start state = ε-closure(A)
= {A, B, C}

Input = {0, 1}

(different here is start state, also only process (ii) and (iii) done by checking new state each time.)

(δ' or δ_D can also be written as notation)

Now,

$\delta(\{A,B,C\}, 0) = \varepsilon\text{-closure}(\delta(\{A,B,C\}, 0))$
$= \varepsilon\text{-closure}(\{A, C\})$
$= \{A, B, C\}$

$\delta(\{A,B,C\}, 1) = \varepsilon\text{-closure}(\delta(\{A,B,C\}, 1))$
$= \varepsilon\text{-closure}(\{B, C\})$
$= \{B, C\}$.

(new states after checking state {A,B,C} are {A,B,C} & {B,C}. Since {A,B,C} already checked so {B,C} now.)

$\delta(\{B,C\}, 0) = \varepsilon\text{-closure}(\delta(\{B,C\}, 0))$
$= \varepsilon\text{-closure}(\{C\})$
$= \{C\}$

$\delta(\{B,C\}, 1) = \varepsilon\text{-closure}(\delta(\{B,C\}, 1))$
$= \varepsilon\text{-closure}(\{B, C\})$
$= \{B, C\}$

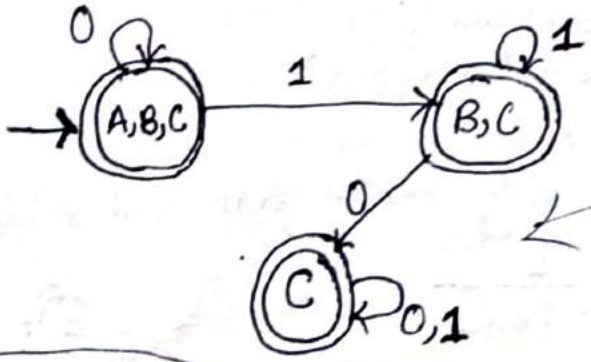
({C} & {B,C} are new states, but {B,C} already checked so remaining {C} now.)

$\delta(C, 0) = \varepsilon\text{-closure}(\delta(\{C\}, 0))$
$= \varepsilon\text{-closure}(\{C\})$
$= \{C\}$

$\delta(C, 1) = \varepsilon\text{-closure}(\delta(\{C\}, 1))$
$= \varepsilon\text{-closure}(\{C\})$
$= \{C\}$

No new states to be checked, all are checked so we stop here.
Now we can draw transition table and diagram as follows:-

δ	0	1
*→{A,B,C}	{A,B,C}	{B,C}
*{B,C}	{C}	{B,C}
*{C}	{C}	{C}



(If ∅ comes in transition table also draw it as a state. Since ∅ means dead state in DFA)

(C was final state in ε-NFA. So, all states containg C are final states in DFA)

Finite State Machines with output:

read as del

read as delta

1) Mealy Machine:

Mealy machine is defined by the six tuples $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where;

this is output which is new here

Q = finite set of states.

Σ = finite non-empty set of input alphabets.

Δ = The set of output alphabets.

δ = Transition function: $Q \times \Sigma \to Q$

λ = Output function: $\Sigma \times Q \to \Delta$

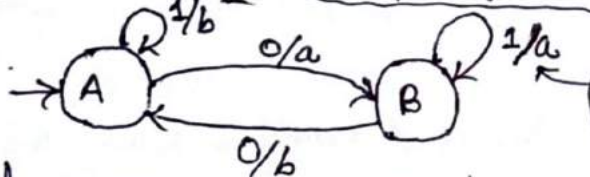
This λ is also different here, Inplace of final state F here we have out put function λ.

q_0 = Initial state/start state.

This output function Δ is associated to input Σ & current state Q.

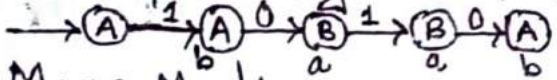
Example:

This means state A on getting input 1 goes to A itself producing output b.



Mealy machine मा output यसरि transition मा input सँग slash गरि राखिन्छ र output depend गर्छ input र current state दुवैमा

#Let we pass string 1010 and see output.



. Here we can see that on passing string 1010 we get output as baab [i.e, for n input we get n outputs]

2) Moore Machine:

Moore machine is also defined by six tuples $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where;

Q = finite set of states.

Σ = finite non-empty set of input alphabets.

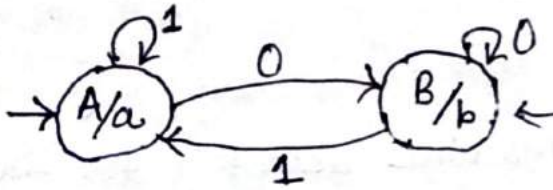
Δ = the set of output alphabets.

δ = Transition function: $Q \to \Delta$

q_0 = Initial state/Start state.

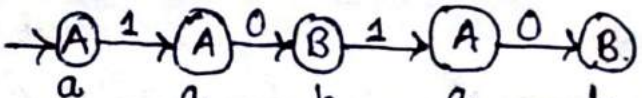
यसमा यो मात्र फरक हो अरु same. This means output function Δ is associated with current state Q only.

Example:



Moore machine मा output यसरि State सँग slash गरि राखिन्छ र output current state मा मात्र depend गर्छ, अरु फरक हो

Let we pass string 1010 and see output.



moore machine depends on current state only so without any input it can produce output on being in any state.

Here we can see that on passing string 1010 we get output as aabab. [i.e, for n inputs we get n+1 outputs in moore machine].

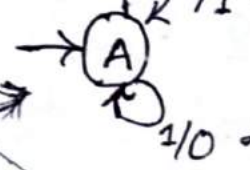
⊛ Construction of Mealy Machines [Examples]:-

Remember that there are no any final states in melay machine

Example 1: Construct a Mealy Machine that produces the 1's complement of any binary input string.

Solution:-

We know that 1's complement of any binary string is converting 1's to 0's and 0's to 1's. So, the required Mealy Machine is as follows:



On getting 0 as input provide output 1

And on getting 1 as input provide output as 0

Since length of string is not provided or any other requirement so, any length string can come. So it remains on same state No other state required.

Example 2: Construct a Melay Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string.

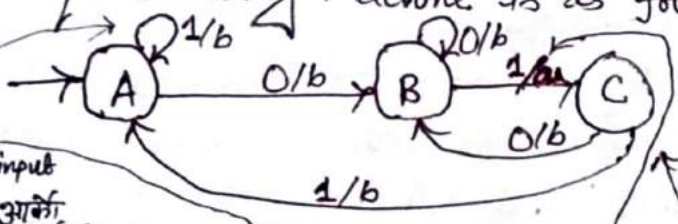
Solution:

$\Sigma = \{0, 1\}$ — inputs

$\Delta = \{a, b\}$ — outputs

b taken for all outputs other than a when 01 is not encountered

The required Melay Machine is as follows:-



① first draw 3 states for given input sequence 01. then complete it as we did for DFA.

② A मा already 0 input गियो / 1 आउदा आर्को state मा गए string लाई (ie, 01 sequence) अस्त गर्दै So त्यसै state मा रहन्छ।

③ Similarly C मा 0 र 1 आउदा already 01 पाइसकेपछि So, first चोटि 0 र 1 input आएर जुन state मा गियो त्यहि state मा जाने

④ B input मा B पुग्यो र 1 input मा C पुग्यो So, 01 sequence मिलेको ले question अनुसार C मा पुग्दा 'a' output गरिदियो र बाकि सबैमा b

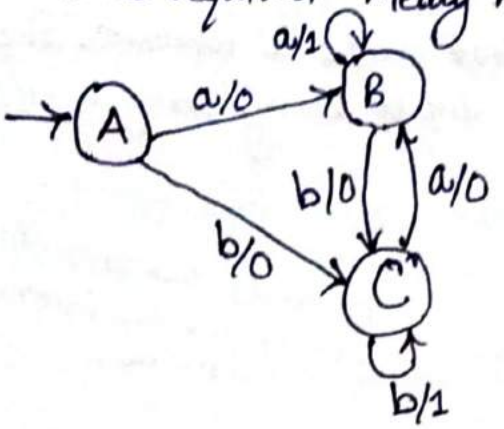
Example 3:- Design a Mealy Machine accepting the language consisting of strings from Σ^*, where $\Sigma = \{a, b\}$ and the strings should end with either aa or bb.

Solution

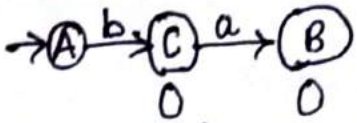
Σ^* means all any length of strings from $\Sigma = \{a, b\}$.

Our string should end with aa or bb. So, let if we get sequence aa or bb then we print output as 1. otherwise 0.

Now the required Mealy Machine is as follows:-

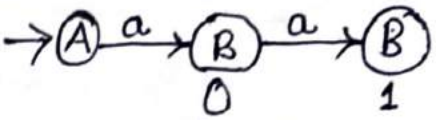


Let we check for string ba



Here outputs are 00 so no any 1's appeared. This means sequence aa or bb is not encountered.

Let similarly we check for string aa



Here outputs are 01 so, the 1 in it shows once the sequence aa or bb is encountered. [If we see more than 1 times let n then it means sequence encountered n times.]

Example 4: Construct a Mealy Machine that gives 2's complement of any binary input. (Assume that the last carry bit is neglected).

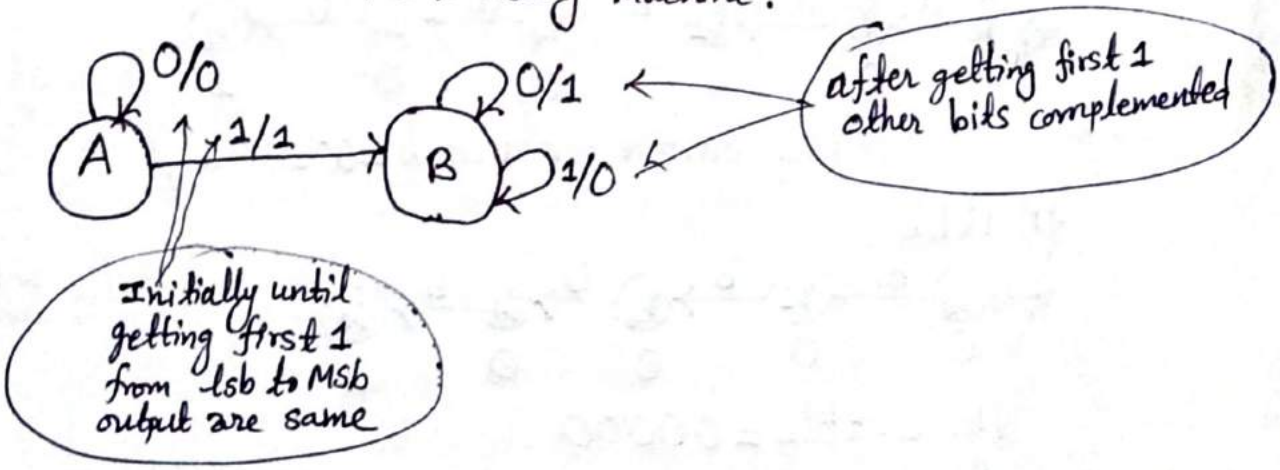
Solution: We know that 2's complement = 1's complement + 1.

Let we have 10100 then 2's complement = 01011 (MSB ← LSB)

$$\begin{array}{r} 01011 \\ +\quad 1 \\ \hline 01100 \end{array}$$

2's = 01100

Now, if we look in this example. we found that on going from LSB towards MSB of 2's complement (i.e, 01100) and given string (i.e, 10100) we can see that until we get first 1 all digits are same (i.e, 100) but after that 1 is changed to 0 and 0 is changed to 1 (i.e, rest bits are complemented). This gives us idea to build our mealy machine.



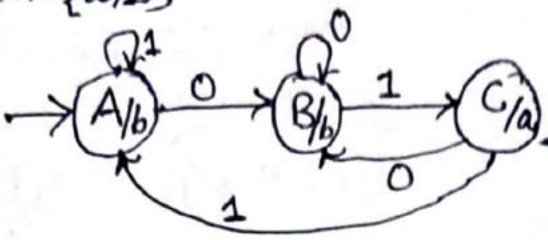
Construction of Moore Machine:

Example 1: Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string.

Solution:-

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

$\Delta = \{a,b\}$



Outputs are associated to states in moore machine.

prints a only on getting sequence 01 otherwise prints b.

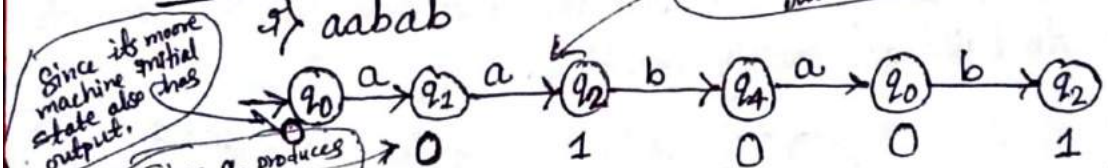
Example 2: For the following Moore Machine the input alphabet is $\Sigma = \{a,b\}$ and the output alphabet is $\Delta = \{0,1\}$. Run the following input sequences and find the respective outputs.

i) aabab ii) abbb iii) ababb.

States	a	b	Outputs
→ q_0	q_1	q_2	0
q_1	q_2	q_3	0
q_2	q_3	q_4	1
q_3	q_4	q_4	0
q_4	q_0	q_0	0

Solution:-

i) aabab



q_2 on getting a goes to q_2 but output of q_2 is 1

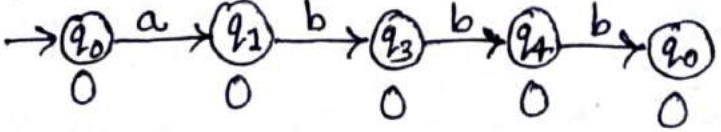
Since its moore machine initial state also has output.

Since q_1 produces output 0

If it was mealy machine then 0 output for initial state would not be here.

Hence, output for aabab = 001001

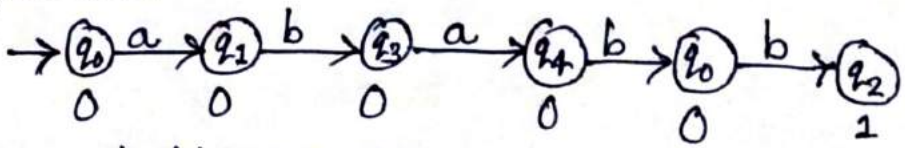
ii) abbb



for moore machine with n inputs there are n+1 outputs

Hence, abbb = 00000.

iii) ababb.



Hence, ababb = 000001.

Mealy Machine vs. Moore Machine

Mealy Machine	Moore Machine.
i) Output depends upon both the present state and present input.	i) Output depends upon only the present state.
ii) Generally it has fewer states than Moore Machine.	ii) Generally it has more states than Mealy Machine.
iii) For mealy machine with n inputs there are n outputs.	iii) For moore machine with n inputs there are n+1 outputs.
iv) The value of the output function is a function of transitions and the changes when the input logic on the present state is done.	iv) The value of output function is a function of the current state and the changes at the clock edges whenever state changes occur.
v) Mealy machines react faster to inputs since they react the same no. of clock cycles.	v) Moore machines react slower to inputs as they react generally one clock cycle later.

* **Language** → Set of strings. Eg. $\Sigma = \{0,1\}$

L_1 = set of all strings of length 2.
= $\{00, 01, 10, 11\}$ (i.e, finite).

L_2 = set of all strings of length 3.
= $\{000, 001, 010, 011, 100, 101, 110, 111\}$ (i.e, finite).

L_3 = set of all strings that begin with 0.
= $\{0, 00, 01, 000, 001, 0000, \ldots\}$ (i.e, infinite).

* **Powers of Σ:** Let $\Sigma = \{0,1\}$

Σ^0 = Set of all strings of length 0 : $\Sigma^0 = \{\epsilon\}$ (epsilon. that means empty)
Σ^1 = set of all strings of length 1 : $\Sigma^1 = \{0,1\}$
Σ^2 = set of all strings of length 2: $\Sigma^2 = \{00, 01, 10, 11\}$.
⋮
Σ^n = set of all strings of length n.

Σ^* = set of all strings of any length that can be formed using Σ.

Unit-3

Regular Expressions

Regular Languages → A language is said to be a regular language if and only if some finite state machine recognizes it.

The languages which are not recognized by any finite state machine and which require memory are not regular languages.

For Example:- let we have ababbababb. As we see this language should follow the rule that, the first five letters ababb is repeated again. So, in order to repeat next time we should have information about what to be repeated (i.e, ababb in this case) which requires memory to store ababb. As we know that memory of FSM is very limited and it cannot store or count string, so this language can not be designed using FSM making it a language that is not regular.

⊛ **Regular Expression** → Regular expressions are those algebraic expressions used for describing regular languages, the languages accepted by finite automation. Regular expressions offer a declarative way to express the strings we want to accept.

Any regular expression is composed of two components; symbols and operators. Symbol, Σ is a set of inputs & operators are union (U), concatenation (.), Kleen Closure (*), Positive closure (+).

Example: Consider a regular expression $(0 \cup 1)01^*$; where 0,1 are symbols and $\cup$, * are the operators. Then, the language described by this expression is the set of all binary strings:

→ that start with either 0 or 1. ← indicated by $(0 \cup 1)$ $\cup$ means (or operation)

→ for which the second symbol is 0

→ and that end with zero or more number of 1's. ← Indicated by 1^*.

Hence, the language described by this expression is {00, 001, 00111 ... , 10, 101, 10111, ...}.

Formal Definition of Regular Expression:

Let Σ be an alphabet, the regular expression over the alphabet Σ are defined inductively as follows;

⇒ $\emptyset$ is a regular expression representing empty language.

⇒ ϵ is a regular expression representing the language of empty strings.

⇒ If 'a' is a symbol in Σ, then 'a' is a regular expression representing the language {a}.

⇒ if 'r' and 's' are the regular expressions representing the language L(r) and L(s) then:

→ r U s is a regular expression denoting the language L(r) U L(s).

→ r.s is a regular expression denoting the language L(r).L(s).

→ r* is a regular expression denoting the language (L(r))*

→ (r) is a regular expression denoting the language (L(r)).

Operators of Regular Expressions:

i) Union (U/|): If L_1 and L_2 are any two regular languages then,

$L_1 \cup L_2 = \{s \mid s \in L_1, \text{ or } s \in L_2\}$

e.g. Let $L_1 = \{00, 11\}$, $L_2 = \{\epsilon, 10\}$

$L_1 \cup L_2 = \{\epsilon, 00, 11, 10\}$

Nothing, this simply means or operation. i.e s such that s belongs to L_1 or s belongs to L_2

but this symbol used here denotes empty string

ii) Concatenation (.): If L_1 and L_2 are any two regular languages then,

$L_1 . L_2 = \{l_1 . l_2 \mid l_1 \in L_1 \text{ and } l_2 \in L_2\}$

e.g. $L_1 = \{00, 11\}$, $L_2 = \{\epsilon, 11\}$

$L_1 . L_2 = \{11\}$.

Concatenation denotes and operation

iii) Kleen Closure (*): If L is any regular language then, Kleen closure of L is;

$L^* = L_0 \cup L_1 \cup L_2 \cup L_3 \cup \ldots$

i.e, $L^* = \bigcup_{i=0}^{\infty}$

i.e, set of all strings of length 0 or more

* has highest precedence
. has next higher precedence
U/|/+ has lowest precedence.

iv) Positive Closure (+): If L is any regular language then, Positive closure of L is: $L^+ = L_1 \cup L_2 \cup L_3 \cup \ldots$

i.e, $L^+ = L^* - L_0$

i.e, set of all strings expect of length 0 or empty

⊛ Applications of Regular Expression:

i) Validation → Determining that a string compiles with a set of formatting constraints, like email address validation, password validation etc.

ii) Search and Selection → Identifying a subset of items from a larger set on the basis of a pattern match.

iii) Tokenization → Converting a sequence of characters into words, tokens (like keywords, identifiers) for later interpretation.

iv) Lexer → Used as a lexer/tokenizer in lexical analysis step of compilers.

⊛. Algebraic laws for regular expression:

r, s, l, m, n used all letters are notations for any regular expression

1) Commutativity:

$r+1 = 1+r$ i.e, $r \cup 1 = 1 \cup r$

but $r.1 \neq 1.r$

2) Associativity

$1+(r+s) = (l+r)+s$

Also $1.(r.s) = (1.r).s$

3) Distributive law:

$l(m+n) = lm + ln$

Also, $(m+n)l = ml + nl$

4) Identity law:

$r+\phi = \phi+r = r$ i.e, $\phi \cup r = r$.

i.e, $\epsilon.r = r = r.\epsilon$ ← (empty symbol)

Also, $\epsilon + r^*r = r^*$

5) Annihilator:

$\phi.r = r.\phi = \phi$

6) Idempotent law of union:

$r+r = r$

7) Laws of closure:

$(r^*)^* = r^*$

Also, closure of $\phi = \phi^* = \epsilon$

Also, closure of $\epsilon = \epsilon^* = \epsilon$

Also, Positive closure of r, $r^* = rr^*$.

⊛ Regular Expression Examples:

1. Describe the following sets as Regular Expressions.

a) $\{0,1,2\}$ b) $\{\wedge, ab\}$ c) $\{abb, a, b, bba\}$ d) $\{\wedge, 0, 00, 000, \ldots\}$

e) $\{1, 11, 111, 1111, \ldots\}$

Solution,

a) $\{0,1,2\}$

$\Rightarrow R = 0+1+2$ ← Since this set can contain 0 or 1 or 2 not other than that. ↖ or means addition operation.

b) $\{\wedge, ab\}$ ← empty symbol $\wedge$ and ϵ are same any can be used

$\Rightarrow R = \wedge.ab$ ← during empty symbol we write using . operator

c) $\{abb, a, b, bba\}$ ← Since this set contains any of abb or a or b or bba not other than that so, + operator.

$\Rightarrow R = abb + a + b + bba$

d) $\{\wedge, 0, 00, 000, \ldots\}$ ← i.e, all the string that can be formed using 0 along with empty symbol i.e, all means closure * sign.

$\Rightarrow R = 0^*$

e) $\{1, 11, 111, 111, \ldots\}$ ← i.e, all string that can be formed by 1. But its not closure of 1. Since closure should contain empty symbol $\wedge$ or ϵ as well.

$\Rightarrow R = 1^+$ ← 1^+ denotes closure of 1 excluding empty symbol.

2. Consider $\Sigma = \{0,1\}$, some regular expressions over Σ;

Understand only

a) 0^*10^* ← 0^* means any number of 0's including empty. i.e, this means it contains single 1 because there's no possibility of other 1 comming.

b) $\Sigma^*1\Sigma^*$ ← This means it contains at least one 1. Since Σ^* means all string that can be formed using $\Sigma = \{0,1\}$ of any length.

c) $\Sigma^*001\Sigma^*$ ← this means contains string 001 as substring.

d) $(\Sigma\Sigma)^*$ or $((0+1)^*.(0+1)^*)$ ← this means string of even length.

e) $1^*(01^*01^*)^*$ ← this means string containing even number of zeros

f) $0^*10^*10^*10^*$ ← this means string with exactly three 1's

g) $(1+0)^*.001.(1+0)^* + (1+0)^*.(100).(1+0)^*$ ← this means string that have either 001 or 100

h) $1^*(0+\epsilon).1^*.(0+\epsilon).1^*$ ← this means strings that have at most two 0's within it

i) $(1+0)^*.(11)^*$ ← this means string ending with 11

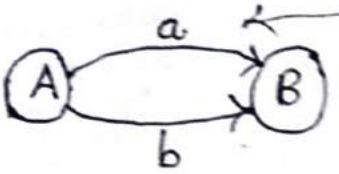
⊛ Equivalence of Regular Expression and Finite Automata:

For the conversion of regular expression to its equivalent finite automata we have some important basic rules which are as follows:-

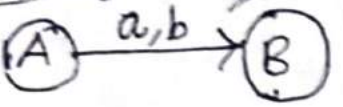
i) When we have expressions like (a+b)
OR (a U b)
OR (a | b)

since +, U & | operation denote same thing i.e, OR operation

then, we draw it as,



OR



since a+b, aUb, a|b means OR operation so on getting any of them can go to next state

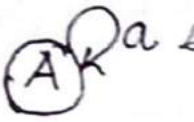
same thing drawn with single transition line

ii) If we have the expression of the form (a.b) then we simply draw new state for each input as follows:-

simply (ab)

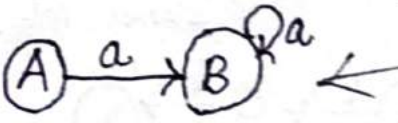
. means and operation so seperate states are made

iii) If we have the expression of the form a^* then we create a transition that goes to the state itself without creating next state as follows:-



Since a^* denotes zero to any numbes of a's.
i.e, $a^* = \{\epsilon, a, aa, aaa, \ldots\}$
ϵ denotes empty.

iv) If we have the expression of the form a^+ then we draw as follows:-



Since a^+ means any numbes of a's excluding zero.
i.e, $a^+ = \{a, aa, aaa, \ldots\}$.
This means at least 1 a should come. so we draw it as like this.

Example 1: Convert the following regular expressions to their equivalent Finite Automata.

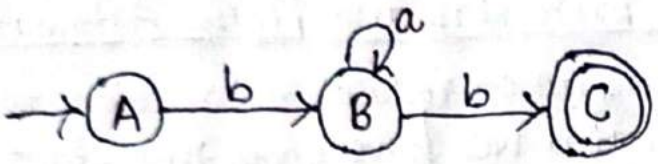
i) ba^*b

ii) $(a+b)c$

iii) $a(bc)^*$

i) b a*b

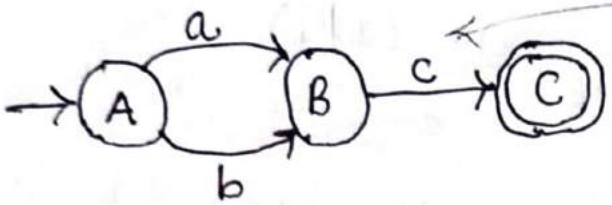
Solution:



Since for first a there is no any operation so we directly go to next state (according to rule ii). then we have a* in which we applied rule (iii) and finally for b again rule (ii) used. b input is final so C is final state.

ii) (a+b) c

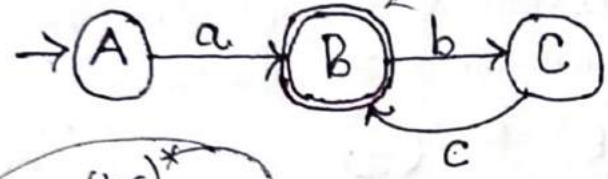
Solution:-



Since c is just and (i.e, dot operation) with (a+b). so directly sent to new state according to rule (ii)

iii) a(bc)*

Solution:-



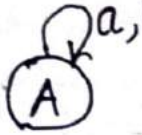
think (bc)* as form (a)* i.e, as one

Here a(bc)* means it contains strings like a, abc, abcbc, abcbcbc. B becomes final state because we may also get single a only as our string. Now if B is final state C is the final string on our question so, C on getting input c goes to final state B.

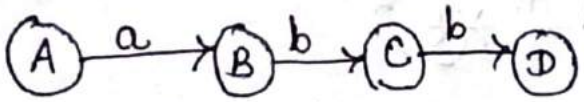
Example2: Convert the following Regular Expression to its equivalent Finite Automata:

(a|b)* (abb|a+b)

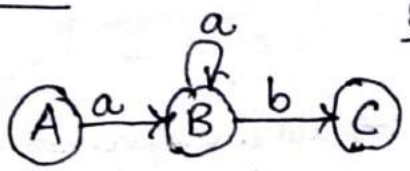
Solution:- For making easier to solve let we divide given expression in three parts as (a|b)* , abb & a+b and draw for each seperately.



for (a|b)*



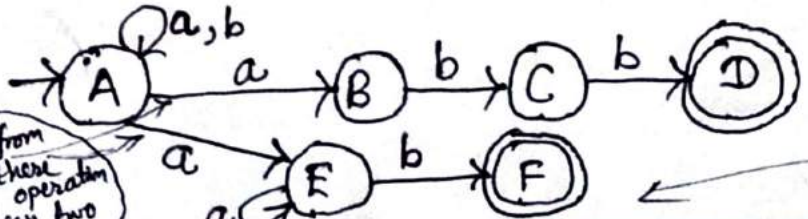
for abb



for a+b

for easier I did this we can directly draw if we know

Now we combine all of them in one as follows:-



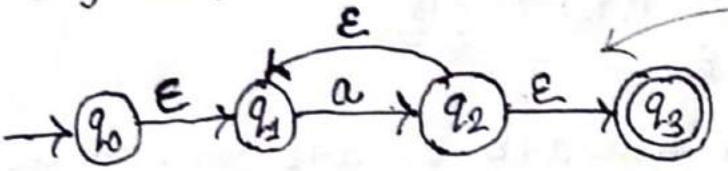
we draw directly from A since there is and operation between two

D and F are final states and came from A seperately because there is | i.e, OR operation between aab and a+b, so, on getting both we get final state

⊛ Reduction of Regular Expression to ε-NFA:- [Imp]

For the conversion of regular expression to its equivalent ε-NFA we have some important basic rules as we used before (including now ε) as follows:-

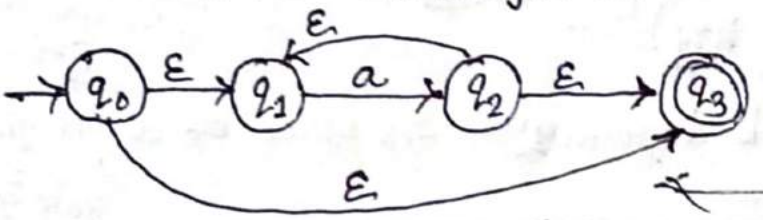
i) When we have expression of the form a^+ then its ε-NFA is as follows:-



यसमा अगाडि पछाडि ε input मा state परिवर्तन हुन् । q_2 बाट empty input मा फेरि q_1 मा जान मिल्ने भएले गर्दा any number of a we can get

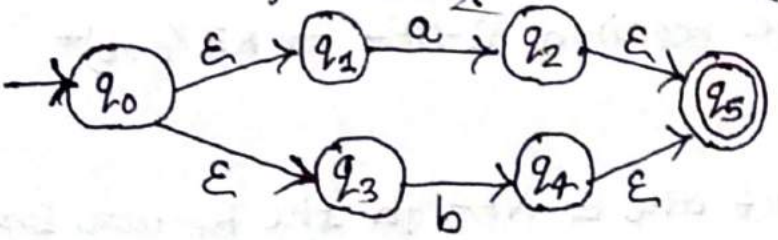
rule हरु हो यो यति याद गर्नै पर्ने ।

ii) When we have expression of the form a^* then its ε-NFA is as follows:-



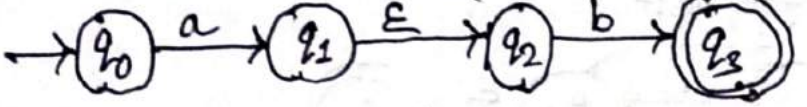
same diagram as for a^+. Only difference is this transition line from start state to final state on getting ε as input. This is because a^* contains any number of a's including zero (i.e, no input or ε)

iii) When we have expression of the form (a+b) OR (a∪b) OR (a|b) then its ε-NFA is as follows:-



a or b ε त्यसैले दुबै एक state बाट अर्को मा seperately र अगाडि र ~~साथी~~ ~~ε~~ states

iv) When we have expression of the form (a.b) or simply (ab), then its ε-NFA is as follows:-

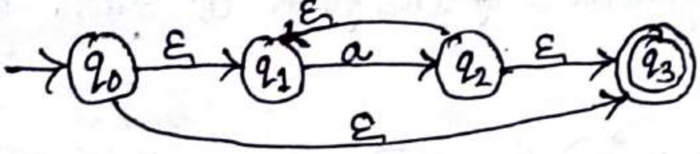


ab simply भाको बेला without any operation then यसरि बिचमा ε ले seperate हुन्छ

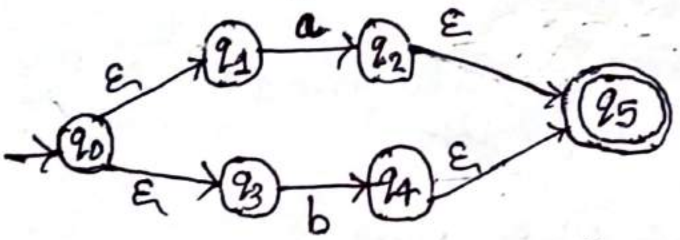
Example 1: Construct the ε-NFA for the Regular Expression $(a+b)^*$ where the alphabets are [a,b].

Solution:-

Let first we think $(a+b)^*$ as a^*, then according to the rule ε-NFA for a^* is as follows:-



But instead of a we have a+b so, a+b form can be represented according to the rule in ε-NFA as follows;

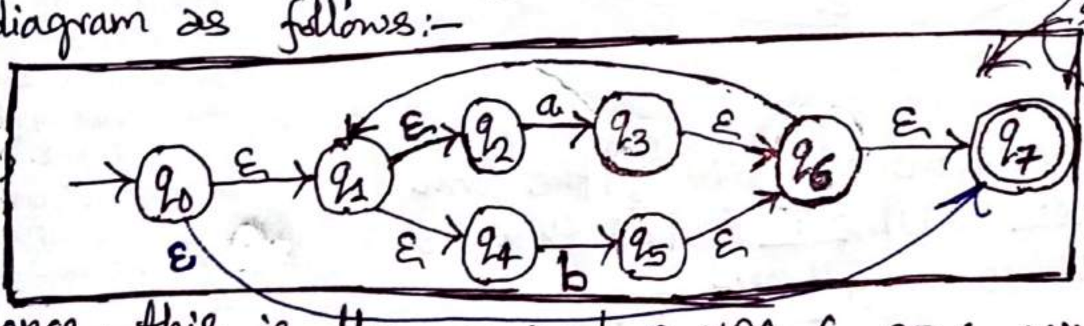


बुझ्नको लागि यसरि stepwise गरेको। यो सब rough मा गरि direct final diagram बनाउदा नि हुन्छ solution मा

Now, substituting a+b (diagram 2nd) in place of a in first diagram as follows:-

State को name हरु जे दिए नि हुन्छ unique to each but draw गरेपछि last मा नाम दिंदा सजिलो हुन्छ

final diagram यो हो अनि मात्र बनाउन box मा

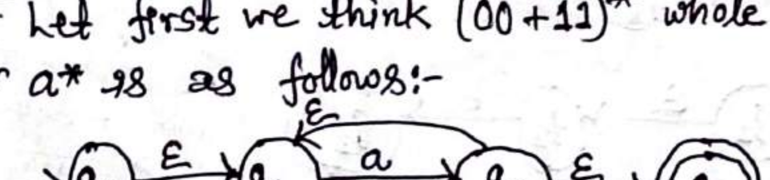


Hence, this is the required ε-NFA for RE $(a+b)^*$

Example 2:- Construct the ε-NFA for the Regular Expression $(00+11)^*$

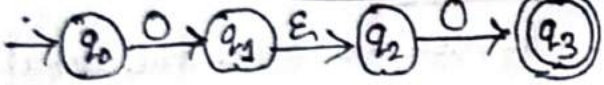
Solution:-

Let first we think $(00+11)^*$ whole as a^*, then ε-NFA for a^* is as follows:-

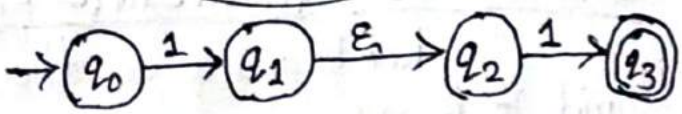


But instead of a we have 00+11. Again two inputs 00 or 11 can not remain at same place in diagram so we subdivide 00 and 11 and draw for each seperately as follows;

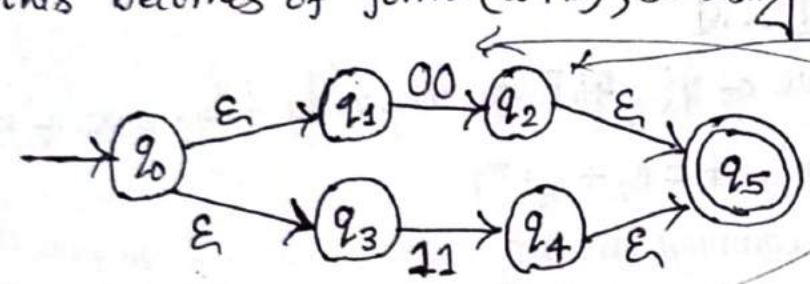
For 00 ← ab form so use rule (iv) to expand



For 11 ← ab form rule (iv) used



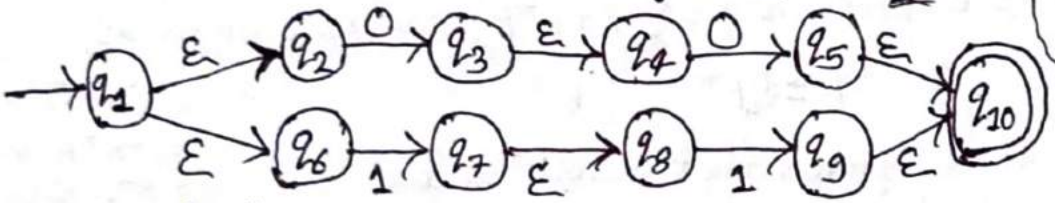
Now we got for 00 and 11, but we have 00+11 inplace of a so, we combine both. If we consider 00 as a and 11 as b then this becomes of form (a+b), so using rule (iii) we get,



Let
a = 00
b = 11

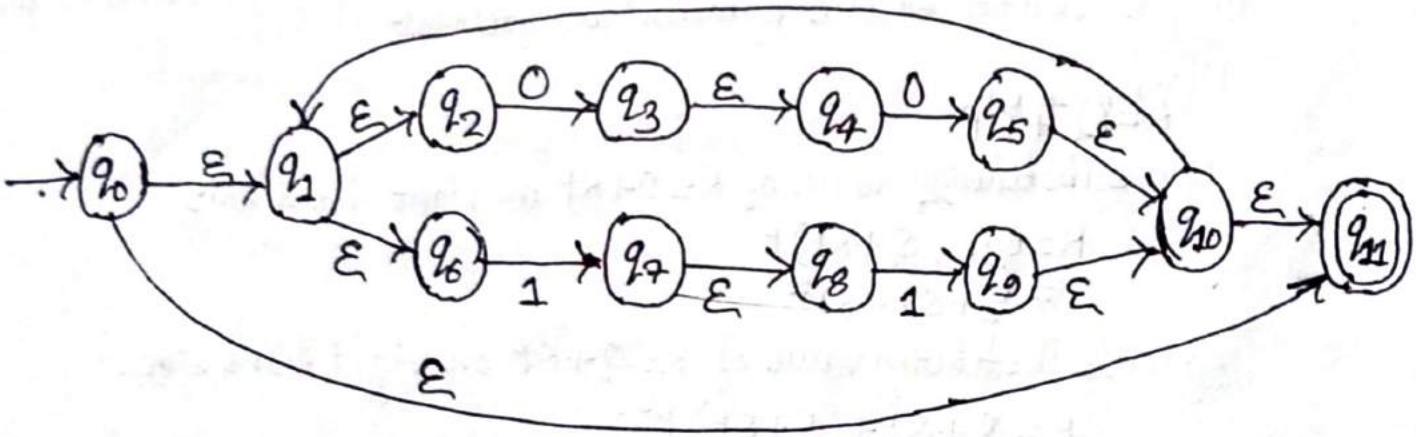
Do not ever focus on state names it will confuse you. Just know that goes to any next state on input and follow rules

But as we know that two strings 00 or 11 can not sit together as in above diagram so we replace them by the diagram we draw for 00 and for 11 as follows:-



माथिको fig मा q_1 to q_2 changed by q_0 to q_3 from fig for 00. Then name of states changed/renamed finally similarly for 11 done.

Now we got diagram for (00+11) now we substitute it to our first figure in place of a, and hence we get required ε-NFA as follows:-



Note:- Practice more questions [i.e, kec book examples] yourself.

⊛ Arden's Theorem:-

Let P and Q be the regular expressions over the alphabet Σ, and if P does not contain any empty string ε, then the following equation in R given by $R = Q + RP$ has a unique solution i.e. $R = QP^*$.

Proof:

(a) Here, $R = Q + RP$ ..... (i)

Let us put the value of $R = QP^*$ on the right hand side of relation (i)

$$R = Q + QP^*P$$

Now, taking Q as common we get;

$$R = Q(\varepsilon + P^*P)$$

In general math on taking common if nothing remains we use to put 1 but here if nothing remains we put ε i.e, empty symbol

Now, from the algebraic laws or regular identities that we discussed earlier we have $\varepsilon + R^*R = R^*$.

The above expression $R = Q(\varepsilon + P^*P)$ can be changed as;

$$R = QP^*$$

Since $\varepsilon + P^*P$ is of form $\varepsilon + R^*R$ which equals to R^*. So, $\varepsilon + P^*P$ is replaced by P^*.

Hence it is proved that $R = QP^*$ is the solution to the equation $R = Q + RP$. This proves the first part of Arden's Theorem.

यो solⁿ हो भनि माथि prove गरियो so, अब unique solⁿ हो भनि prove गरौं।

(b) Now we prove the second part of Arden's theorem (i.e, this is the unique solution to the equation) as follows:-

Here,

$R = Q + RP$

Now, substituting value of $R = Q + RP$ on right hand side:

$$R = Q + (Q + RP)P$$
$$= Q + QP + RP^2$$

Again, substituting value of $R = Q + RP$ on right hand side.

$$R = Q + QP + (Q + RP) \cdot P^2$$
$$= Q + QP + QP^2 + RP^3$$

Continuing in the same way, we will get as;

$$R = Q + QP + QP^2 + RP^3 + \cdots$$
$$= Q(\varepsilon + P + P^2 + P^3 + \cdots)$$

Since $\varepsilon + P + P^2 + P^3 + \cdots$ is the closure of P i.e, P^* So,

$$R = QP^*$$

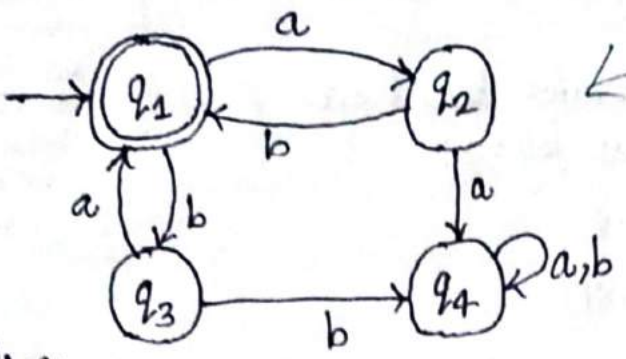
Hence, now we proved that this is the unique solution for equation $R = Q + RP$.

Page 20 continued

⊛. Conversion of DFA to Regular Expression:-

For solving this we should have knowledge of Arden's theorem and algebraic laws for RE.

Example 1: Find the Regular Expression for the following DFA.



i.e, Example of single final state, Later we will discuss for more than one.

Solution:-

Here, equations for each states based on incomming transitions are as follows:-

$$q_1 = \varepsilon + q_2 b + q_3 a \quad \text{——(i)}$$

$$q_2 = q_1 a \quad \text{——(ii)}$$

$$q_3 = q_1 b \quad \text{——(iii)}$$

$$q_4 = q_2 a + q_3 b + q_4 a + q_4 b \quad \text{——(iv)}$$

start : incomming transition comming from nowhere so ε.

$q_2 b$ means q_1 has incomming transition which is q_2 on getting input b similarly for $q_3 a$

q_4 on getting a,b goes to q_4 so, $q_4 a + q_4 b$ done. seperately for a & b both.

Now we take eqn (i). (i.e, final state q_1)

$$q_1 = \varepsilon + q_2 b + q_3 a$$

Putting values of q_2 and q_3 from eqn (ii) and (iii)

$$q_1 = \varepsilon + q_1 ab + q_1 ba$$

Taking q_1 common:

$$q_1 = \varepsilon + q_1 (ab + ba)$$

$q_1 = \varepsilon + q_1 (ab+ba)$, $R = Q + RP$

Now this is of the form $R = Q + RP$ so, we can write it in the form $R = QP^*$ according to Arden's theorem.

i.e, $q_1 = \varepsilon \cdot (ab+ba)^*$

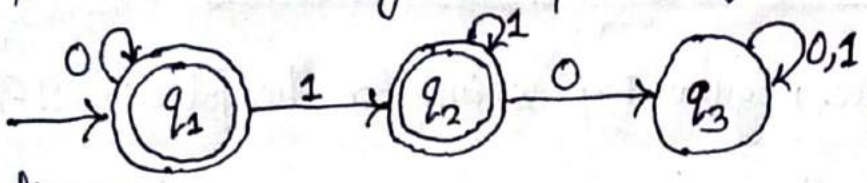
If we think ababa as one let R then it is of form $\varepsilon \cdot R$

Now, according to the algebraic laws for regular expressions we know that $\varepsilon \cdot R = R$. So, above expression can be written as;

$$q_1 = (ab+ba)^*$$

Since q_1 is final state and we get expression for q_1. Hence $q_1 = (ab+ba)^*$ is the required Regular Expression for given DFA.

Example2: Find the Regular Expression for the following DFA.



Example with more than one final state. For solving this we find R.E for all final states one by one as we did before then finally we do union (i.e, +) to get final RE

Solution:-

Here, equations for each states, based on incomming transitions are as follows;

$$q_1 = \varepsilon + q_1 0 \quad \text{—— (i)}$$

$$q_2 = q_1 1 + q_2 1 \quad \text{—— (ii)}$$

$$q_3 = q_2 0 + q_3 0 + q_3 1 \quad \text{—— (iii)}$$

Let we take final state q_1:

$$q_1 = \varepsilon + q_1 0$$

where, $q_1 = R$, $\varepsilon = Q$, $0 = P$

Now, this is of the form $R = Q + RP$, so using Arden's theorem we write it in the form $R = QP^*$ as follows:-

$$q_1 = \varepsilon \cdot 0^*$$

$$q_1 = 0^* \quad \text{(iv)}$$ [Since, from algebraic laws for R.E we know that $\varepsilon \cdot R = R$].

Again, let we take final state q_2:

$$q_2 = q_1 1 + q_2 1$$

$$q_2 = 0^* 1 + q_2 1$$ [Putting value of $q_1 = 0^*$ from eq" (iv)]

$q_2 = 0^*1 + q_2 1$, $R = Q + RP$

Now, if we think 0^*1 as one let Q, then this expression is of form $R = Q + RP$. So, using Arden's theorem we can write it in the form $R = QP^*$.

i.e, $q_2 = 0^* 1 (1)^*$

Now the Final Regular Expression can be obtained by the union of both final states:

i.e, $R.E = 0^* + 0^* 1 (1)^*$

$$R.E = 0^* (\varepsilon + 11^*)$$ [0^* Taken common]

According to the algebraic laws for R.E we have $\varepsilon + RR^* = R^*$.

So, $R.E = 0^* 1^*$ ← since $\varepsilon + 11^* = 1^*$

Hence, the required Regular Expression is $R = 0^* 1^*$ for given DFA.

⊛. Pumping Lemma for Regular Expression:-

Application of pumping lemma is to prove that a language is not regular.

Pumping lemma is used to prove that a language is not regular. It cannot be used to prove that a language is regular.

Statement: If A is a Regular language, then A has a Pumping length 'p' such that any string 'S' where $|S| \geq p$ may be divided into 3 parts $S = xyz$, such that the following conditions must be true:

i) $xy^iz \in A$ for every $i \geq 0$ — i.e, on increasing y any number of times $i \geq 0$, then the string obtained must also belong to A.

ii) $|y| > 0$ — i.e, length of y should be greater than 0;

iii) $|xy| \leq p$ — i.e, length of x & y together should be less than or equal to pumping length.

To prove that a language is not Regular using Pumping lemma follow the following steps: (We prove using Contradiction) :-

→ Assume that A is regular.
→ It has to have a Pumping Length (say p)
→ All strings longer than p can be pumped $|S| \geq p$.
→ Now find a string 'S' in A such that $|S| \geq p$.
→ Divide S into xyz.
→ Show that $xy^iz \notin A$ for some i.
→ Then consider all ways that S can be divided into xyz.
→ Show that none of these can satisfy all the 3 pumping conditions at the same time.
→ S cannot be Pumped == CONTRADICTION.

Example1:- Using Pumping lemma prove that the language $A = \{a^n b^n \mid n \geq 0\}$ is Not Regular.

Proof:

Assume that A is regular.
Pumping length = p.

Now, $S = a^p b^p$

Now, we need to divide S into three parts x, y, z, for that let we assume pumping length $p = 7$. Then we can write the string S as;

$S = aaaaaaabbbbbbb$.

Now, let us see all the possible ways in which we can divide this S into three parts x, y, z for that let we take cases as follows:-

तिनवटै condition मिले मात्रै satisfy हुन्छ यसरी condition पनि नमिले satisfy हुन्छ।

take x & y length in such a way that $|xy| \leq p$ Also, $|y| > 0$.

Case-I: The y is in the 'a' part.

i.e, $\underbrace{a}_{x}\underbrace{aaaaaa}_{y}\underbrace{abbbbbb}_{z}$

Case-II: The y is in the 'b' part.

i.e, $\underbrace{aaaaaaa}_{x}\underbrace{bb}_{y}\underbrace{bbbbb}_{z}$

Case-III: The y is in the 'a' and 'b' part.

i.e, $\underbrace{aaaaaa}_{x}\underbrace{aab}_{y}\underbrace{bbbbb}_{z}$

This means 2 times i.e. y string is repeated one time more

Now for each cases we check condition xy^iz. Here we take $i = 2$. i.e, xy^2z.

For Case-I:

$\underbrace{aa}_{x}\underbrace{aaaa}_{y}\underbrace{aaaa}_{y}\underbrace{abbbbbbb}_{z}$

y^2

No. of a's = 11
No. of b's = 7

No. of a's ≠ No. of b's
So, $xy^2z \notin A$.

does not belong to A

For Case-II

aaaaaaabb bbbb bbbbb

No. of a's = 7
No. of b's = 11

No. of a's ≠ No. of b's.
So, $xy^2z \notin A$.

Case II with y part repeated similarly case-III

यदि case मा $\notin A$ आ does not follow question pattern भएर सबै regular हुन्छ

For Case-II

i.e, पहिलो a हरू मात्रै हुनु पर्छ र त्यसपछि b हरू, बीचमा mix हुन हुन्न

aaaaa aabb aabb bbbbb

Since this string does not follow the pattern $a^n b^n$ as given by question. So, this does not lie in our language.

So, we proved that xy^iz on taking $i=2$ all cases does not lie in our language. ~~Hence the given language is not regular.~~ This means S cannot be pumped which leads to contradiction. Hence, the language is not regular.

Example2:- Using Pumping Lemma prove that the language $A=\{yy \mid y \in \{0,1\}^*\}$ is not regular.

i.e, yy means first part of language must be same as 2nd part. where y has 0,1.

Proof: Assume that A is regular, then it must have a pumping length. Let Pumping length be 'P'.

We formed String S in A such that $|S| \geq P$. & first and 2nd part made same ie, 0^P1.

Now, $S = 0^P 1 0^P 1$

O मा मात्रै power P राख्दा $|S| \geq P$ condition satisfy हुन्छ So, सजिलो को लागि O मा मात्रै राखेको x string बनाको $0^P1\,0^P1$ sodre गर्दा सजिलो हुन्छ भनेर

Now, we need to divide S into three parts x, y, z, for that let we assume pumping length P=7. Then we can write the string S as;

1st part (y) 2nd part (y).

$S = 00000001\,00000001$

Now let we see all the possible ways in which we can divide this S into three parts x,y,z for that we take cases as;

Case-I; The y is in the first part.

i.e, 0 | 0000000 | 1 | 00000001
x y z

Here, $|y| > 0$ & $|xy| \leq P$ i.e, $|xy| \leq 7$

Case-II: The y is in the second part.

i.e, 00000001 | 0000000 | 1
x y z

Case-III: The y is in the first and second part.

i.e, 00000000 | 1 | 00000001
x y z.

Now for each cases we check condition xy^iz. Let we take $i=2$. i.e, xy^2z.

For Case-I 00000000 | 1 | 00000001
first part 2nd part.

Here, we see that first part is not equal to second part. i.e, it does not follow pattern YY. So, this does not lie in the language A. This means S cannot be pumped which leads to contradiction. Hence the language is not regular.

⊛. Closure Properties of Regular languages:-

i) Closure under Union → If L and M are regular languages, so is L∪M. Let L and M be the languages of regular expressions R and S respectively. Then R+S is a regular expression whose language is L∪M.

ii) Closure under Intersection → If L and M are regular languages, so is L∩M. Let L and M be the languages of regular expressions R and S respectively then R∩S is a regular expression whose language is L∩M.

iii) Closure under Concatenation → If L and M are regular languages, so is L.M. If L and M be the languages of regular expressions R and S respectively then R.S is a regular expression whose language is L.M.

iv) Closure under Kleene closure → If L is the regular language of regular expression R then R* is a regular expression whose language is L*.

v) Closure under complement → The complement of a language L (with respect to an alphabet Σ such that Σ* contains L) is Σ*−L. Since Σ* is surely regular by the property of closure under Kleen closure, the complement of a regular language is always regular.

⊛ Minimization of DFA: (Table Filling Algorithm) :-

also known as Myhill-Nerode Theorem

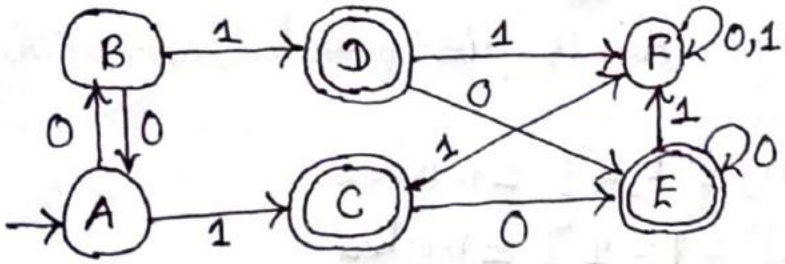
Steps:-

1) Draw a table for all pairs of states (P, Q).
2) Mark all pairs where $P \in F$ and $Q \notin F$.
3) If there are any Unmarked pairs (P, Q) such that $[\delta(P,x), \delta(Q,x)]$ is marked, then mark $[P, Q]$ and repeat this process until no markings can be done.
4) Combine all the Unmarked Pairs and make them a single state in the minimized DFA.

Pair मा कुनै एक final state को set मा पर्ने र अर्को पर्दैन भने Mark गर्ने । दुबै state पर्ने या पर्दैननन् भने नगर्ने

i.e, Check a pair where it goes which is obtained by state P & Q on getting particular input x.

Example :- Minimize the following DFA using Table fill Algorithm.



Solution:

	A	B	C	D	E	F
A						
B						
C	✓	✓				
D	✓	✓				
E	✓	✓				
F	✓	✓	✓	✓	✓	

Full table महत्वको diagonally divide गरि upper part राख्ने repeat गरेको same pair जस्ते AA, FF, यसैगरि AB र BA मात्रै हो so, एउटा pair BA check गर्दा पुग्या so, repeating rooms eliminated & pair checked and ticked.

F,A and F,B are empty not marked initially. they are later marked after checking pairs which unmarked.

(F,E) pair मा F → final state हुने E → final state हो यसरि फरक भए mark हुन्छ यदुवै भए हुन्न

Now, we check the Unmarked pairs as follows:-

For pair (B, A)

input 0 taken for pair

$[\delta(B,0), \delta(A,0)] = [A, B] \Rightarrow$ unmarked

input 1 taken for pair

$[\delta(B,1), \delta(A,1)] = [D, C] \Rightarrow$ unmarked

[A, B] table मा mark हुन यसैले unmarked राखेको । marked भको भए marked राखिन्थ्यो

DC unmarked in table so, unmarked

For pair DC

$[\delta(D,0), \delta(C,0)] = [E,E]$ = unmarked ← table मा लेख्दा नि unmarked

$[\delta(D,1), \delta(C,1)] = [F,F]$ = unmarked

For pair EC

$[\delta(E,0), \delta(C,0)] = [E,E]$ = unmarked

$[\delta(E,1), \delta(C,1)] = [F,F]$ = unmarked

For pair ED

$[\delta(E,0), \delta(D,0)] = [E,E]$ = unmarked

$[\delta(E,1), \delta(D,1)] = [F,F]$ = unmarked

For pair FA

$[\delta(F,0), \delta(A,0)] = [F,B]$ = unmarked.

$[\delta(F,1), \delta(A,1)] = [F,C]$ = marked

So, the pair (F,A) is distinguishable. (i.e, (F,A) is marked now)

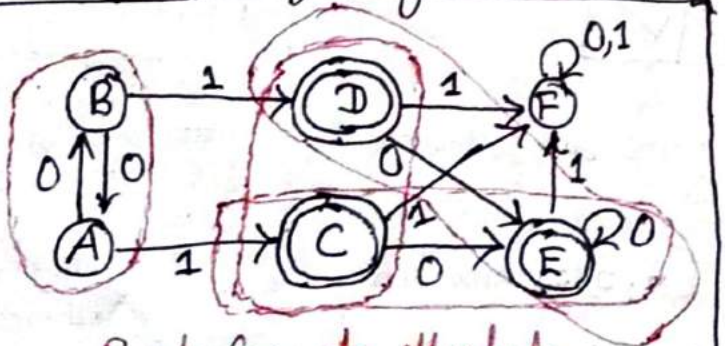
For pair FB

$[\delta(F,0), \delta(B,0)] = [F,A]$ = marked

$[\delta(F,1), \delta(B,1)] = [F,D]$ = marked

So, the pair (F,B) is distinguishable. (i.e, (F,B) is marked now)

Now, finally we combine unmarked pairs (B,A), (D,C), (E,C), (E,D) as one single state. i.e, (B,A) is now single state. But in (D,C), (E,C), (E,D) there is common C in (D,C), (E,C) and also common E in (E,C), (E,D) this shows all these 3 pairs can also be combined as one single state, which is illustrated by a rough figure below. Hence, the minimized DFA is as follows:-



Rough fig:- to illustrate (D,C), (E,C), (E,D) as one state since on combining them each of them are overlapping (i.e, common).

यो rough fig. rough मा बनाउने solⁿ मा जरुरी छैन / कुनै सम्झिने लाई मात्र

Unit - 4

Context Free Grammar (CFG):

Grammars are used to generate the words of a language and to determine whether a word is in a language. Context free grammars are used to define syntax of almost all programming languages, in context of computer science. Thus an important application of context free grammars occurs in the specification and compilation of programming languages.

Formal Definition of Context Free Grammar:

Context Free Grammar is defined by 4 tuples as $G = \{V, T, S, P\}$ where,

V = Set of variables or Non-terminal Symbols.
T = Set of Terminal Symbols
S = Start Symbol
P = Production Rule.

> Context Free Grammar has production rule of the form $A \rightarrow \alpha$
> where, $\alpha = \{V \cup T\}^*$ and $A \in V$.
>
> ($\{V \cup T\}^*$: closure of V union T; $\in V$: belongs to)

Components of CFG:

There are four components in CFG.

1). There is a finite set of symbol that form the strings of language being defined. We call this alphabet the terminal symbol. In above tuples it is represented by T.

2). There is a finite set of variables also called non-terminal symbols. Each variable represents the language, i.e, a set of strings. It is represented by V in above tuples.

3). One of the variables represent the language being defined. It is called the start symbol. It is represented by S in above tuples.

4). There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of:

a) A variable that is being defined by the production. This is called head of production.

b) The production symbol →

c) A string of zero or more terminals and variables. This string is called the body of the production, represents one way to form the string in the language of the head.

Note: The variable symbols are often represented by capital letters. The terminals are analogous to the input alphabet and are often represented by lower case letters. One of the variables is designed as a start variable. It usually occurs on the left hand side of the topmost rule.

Example:

$S \rightarrow \epsilon$

$S \rightarrow 0S1$

This is a CFG defining the grammar of all the strings with equal no. of 0's followed by equal no of 1's.

Here, the two rules define the production P,
$\epsilon, 0, 1$ are the terminals defining T,
S is a variable symbol defining V,
And S is start symbol from where production starts.

CGF vs RE:

The CGF are more powerful than the regular expressions as they have more expressive power than the regular expression. Generally regular expressions are useful for describing the structure of lexical constructs as identical keywords, constants etc. But they do not have the capability to specify the recursive structure of the programming constructs. However, the CFG are capable to define any of the recursive structure also. Thus, CFG can define the languages that are regular as well as those languages that are not regular.

Use of CFG:

Context-free grammars are used in compilers and in particular for parsing, taking a string-based program and figuring out what it means. Typically, CFGs are used to define the high-level structure of a programming language. Figuring out how a particular string was derived tells us about its structure and meaning.

Meaning of context free: Consider an example:

$S \rightarrow aMb$
$M \rightarrow A \mid B$
$A \rightarrow \varepsilon \mid aA$
$B \rightarrow \varepsilon \mid bB.$

Now, Consider a string aaAb, which is an intermediate stage in the generating of aaab. It is natural to call the strings "aa" and "b" that surround the symbol A, the "context" of A in this particular string. Now, the rule $A \rightarrow aA$ says that we can replace A by the string aA no matter what the surrounding strings are; in other words, independently of the context of A.

When there is a production of form $lw_1r \rightarrow lw_2r$ (but not of the form $w_1 \rightarrow w_2$), the grammar is context sensitive since w_1 can be replaced by w_2 only when it is surrounded by the strings "l" and "r".

Context Free Language (CFL):

Context free language (CFL) is a language which is generated by a context-free grammar or type-2 grammar and gets accepted by a Pushdown Automata. The set of all context-free language is identical to the set of languages accepted by pushdown automata, and the set of regular languages is a subset of context-free languages. An inputed language is accepted by a computational model if it runs through the model and ends in an accepting final state. Context-free languages and context-free grammars have applications in computer science and linguistics such as natural language processing and computer language design.

⊛. Derivation using Grammar Rule:

The process of producing strings using the production rules of the grammar is called derivation. There are two possible approaches of derivation:

1) Body to head (Bottom Up) approach:-

Here, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is the language of the variables in the head.

Consider grammar,

$S \to S+S$
$S \to S/S$
$S \to (S)$
$S \to S-S$
$S \to S*S$ ..... Grammar(2)
$S \to a$

or this can also be written as:
$S \to S+S \mid S/S \mid (S) \mid S-S \mid S*S \mid a$
in single line

nothing just named the grammer

first body solve गर्ने पछाडि head

Here given $a+(a*a)/a-a$

Now, by this approach we start with any terminal appearing in the body and use the available rules from body to head.

S.N	String Inferred	Variable	Production	String Used.
1.	a	S	$S \to a$	a; String 1
2.	a*a	S	$S \to S*S$	a*a; String 2
3.	(a*a)	S	$S \to (S)$	String 1 & 2; String 3
4.	(a*a)/a	S	$S \to S/S$	String 1 & 3; String 4
5.	a+(a*a)/a	S	$S \to S+S$	String 1 & 4; String 5
6.	a+(a*a)/a-a	S	$S \to S-S$	String 1 & 5; String 6

(3) We write String here based on Production rule taken.

this may not apply for all so follow sequence, Body- Left- Right

(2) Looking at question. we start with terminal (i.e, here for this small letter 'a'). Then we proceed as BODMAS rule as in math. First we complete bracket, Then divide, then multiply, then add, finally sub. Here, multiply is done earlier because bracket must be completed first

OR Body- Left (tail) Right (Head)

(1)

(4) i.e, String used ; New String named

for e.g. String 4 is formed by string 1 & 3 as follows;

(a*a)/a
String 3 String 1

2) Head to body (Top Down) approach:

Here, we use production from head to body. We expand the start symbol using a production, whose head is the start symbol. Here, we expand the resulting string until all strings of terminal are obtained. Here, we have two approaches:

a) **Leftmost Derivation:** Here leftmost symbol (variable) is replaced first. (LMD)

b) **Rightmost Derivation:** Here rightmost symbol is replaced first. (RMD)

For Example:- Consider the previous example of deriving string $a+(a*a)/a-a$ with the grammar (2). [i.e, $S \to S+S \mid S/S \mid (S) \mid S-S \mid S*S \mid a$]

#Now the leftmost derivation for the given string is;

$S \to S+S$ (Rule $S \to S+S$)
$S \to a+S$ (Rule $S \to a$)
$S \to a+S-S$ (Rule $S \to S-S$)
$S \to a+S/S-S$ (Rule $S \to S/S$)
$S \to a+(S)/S-S$ (Rule $S \to (S)$)
$S \to a+(S*S)/S-S$ (Rule $S \to S*S$)
$S \to a+(a*S)/S-S$ (Rule $S \to a$)
$S \to a+(a*a)/S-S$ (Rule $S \to a$)
$S \to a+(a*a)/a-S$ (Rule $S \to a$)
$S \to a+(a*a)/a-a$ (Rule $S \to a$)

Start from left and Replace from left so we start with left part towards body

#Now the rightmost derivation for the given string is;

$S \to S-S$ (Rule $S \to S-S$)
$S \to S-a$ (Rule $S \to a$)
$S \to S+S-a$ (Rule $S \to S+S$)
$S \to S+S/S-a$ (Rule $S \to S/S$)
$S \to S+S/a-a$ (Rule $S \to a$)
$S \to S+(S)/a-a$ (Rule $S \to (S)$)
$S \to S+(S*S)/a-a$ (Rule $S \to S*S$)
$S \to S+(S*a)/a-a$ (Rule $S \to a$)
$S \to S+(a*a)/a-a$ (Rule $S \to a$)
$S \to a+(a*a)/a-a$ (Rule $S \to a$)

Start from right and move towards body and replace from right

Q. Consider a Grammar;

$S \to aAS \mid a$

$A \to SbA \mid SS \mid ba$

This | symbol is or.
i.e, $S \to aAS$ } S can be replaced by any of these two
or, $S \to a$ }
If ε symbol comes then it is empty symbol.

Given a string aaabaaa, give leftmost and rightmost derivation.

Solution:

Leftmost Derivation: (LMD or lm)

$S \to aAS$
$S \to aSSS$ (Rule $A \to SS$)
$S \to aaSS$ (Rule $S \to a$)
$S \to aaaASS$ (Rule $S \to aAS$)
$S \to aaabaSS$ (Rule $S \to ba$)
$S \to aaabaaS$ (Rule $S \to a$)
$S \to aaabaaa$ (Rule $S \to a$)

Rightmost Derivation: (RMD or rm)

$S \to aAS$
$S \to aAa$ (Rule $S \to a$)
$S \to aSSa$ (Rule $A \to SS$)
$S \to aSaASa$ (Rule $S \to aAS$)
$S \to aSaAaa$ (Rule $S \to a$)
$S \to aSabaaa$ (Rule $A \to ba$)
$S \to aaabaaa$ (Rule $S \to a$)

⊛ Sentential Form :-

Derivations from the start symbol produce strings that have a special role. We call these "sentential forms" i.e, if $G_1 = (V, T, P, S)$ is a CFG, then any string α is in $(V \cup T)^*$ such that $S \to^* \alpha$ is a sentential form. If $S \to^*_{lm} \alpha$, then α is a left sentential forms, and if $S \to^*_{rm} \alpha$, then α is a right sentential form.

⊛. Language of Grammar (Context Free Grammar):-

Let $G_1 = (V, T, P, S)$ is a context free grammar. Then the language of G_1 denoted by $L(G_1)$ is the set of terminal strings that have derivation from the start symbol in G_1.

i.e, $L(G_1) = \{x \in T^* \mid S \to^* x\}$

The language generated by a CFG is called the Context Free Language (CFL).

⊛ Parse Tree / Derivation Tree:

Parse tree is a representation of strings of terminals using the productions defined by the grammar. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. Parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order.

Formally, given a Context Free Grammar $G=(V,T,P,S)$, a parse tree is a n-ary tree having the following properties;

→ The root is labeled by the start symbol.
→ Each interior node of parse tree are variables.
→ Each leaf node of parse is labeled by a terminal symbol or ε. (epsilon)
→ If an interior node is labeled with a non terminal A and its childrens are $x1, x2, \ldots xn$ from left to right then there is a production P as;

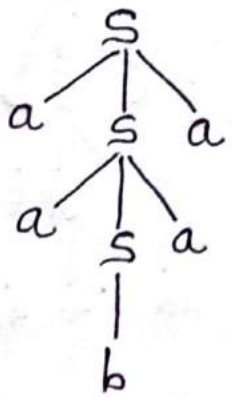
$$A \rightarrow x1, x2, \ldots, xn \text{ for each } xi \in T.$$

(each xi belonging to T)

Example: Consider the grammar $S \rightarrow aSa|a|b|\varepsilon$.
Now, for string $S \rightarrow^* aabaa$.
We have,

$S \rightarrow aSa$
$S \rightarrow aaSaa$ (Rule $S \rightarrow aSa$)
$S \rightarrow aabaa$ (Rule $S \rightarrow b$).

So, the parse tree is →:



Q. Construct the parse tree for a*(a+b00), Considering S→I | S+S | S*S | (S). And I→a | b | Ia | Ib | I0 | I1.

Solution: The parse tree is;

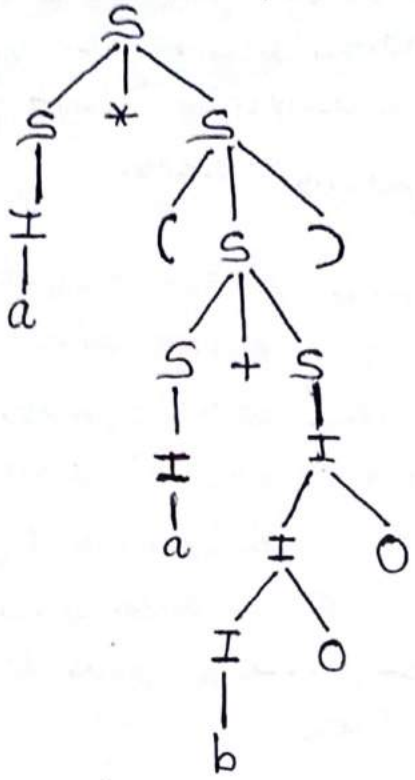


Fig. Parse tree for a*(a+b00).

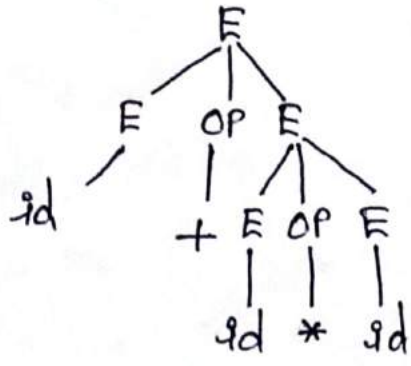
Q. Construct a grammar defining arithmetic expression and generate parse tree for id+id*id and (id+id)*(id+id).

E→EOPE | (E) | id

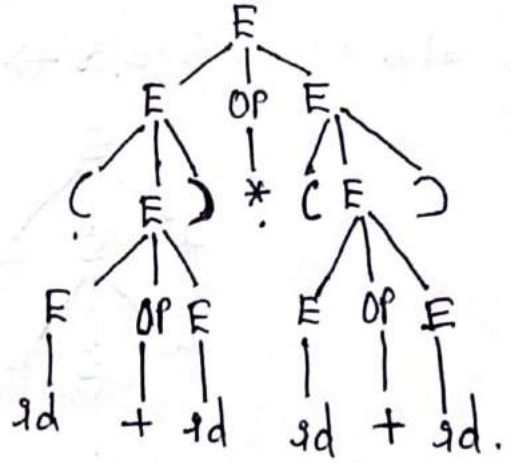
OP→+ | − | * | / | ↑.

Solution:-

Parse tree for id+id*id



Parse tree for (id+id)*(id+id)



⊛. Ambiguity in Grammar:-

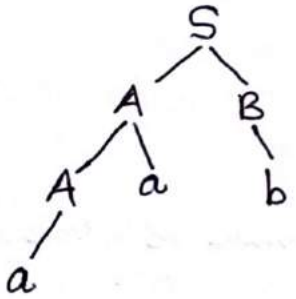
A Grammar $G = (V, T, P, S)$ is said to be ambigious if there is a string $w \in L(G)$ for which we can derive two or more distinct derivation tree rooted at S and yielding w. In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for the same string in the language of the grammar.

Example: $S \to AB \mid aaB$
$A \to a \mid Aa$
$B \to b$

For any string aab; we have two leftmost derivations as;

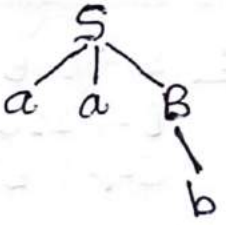
$S \to AB$
$S \to AaB$ (Rule $A \to Aa$)
$S \to aaB$ (Rule $A \to a$)
$S \to aab$ (Rule $B \to b$).

the parse tree for this derivation is;



Also,
$S \to aaB$
$S \to aab$ (Rule $B \to b$)

the parse tree for this derivation is;



⊛. Inherent Ambiguity:-

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous.

E.g. $L = \{ 0^i 1^j 2^k \mid i=j \text{ or } j=k \}$.

⊛. Regular Grammar:

A regular grammar represents a language which is represented by regular expressions. The regular grammar is accepted by NFA and DFA and the language of grammar is called regular language. A regular grammar is a subset of CFG. The regular grammar may be either left or right linear.

i) Right Linear Regular Grammar:

A regular grammar in which all of the productions are of the form $A \to \omega B$ or $A \to \omega$ where $A, B \in V$ and $\omega \in T$ is called right linear.

omega, i.e, any string ω

For Example: $S \to 00B \mid 11C$

$B \to 11C \mid S \mid 1$

$C \to 00B \mid 00$

ii) Left Linear Regular Grammar:

A grammar in which all of the productions are of the form $A \to B\omega$ or $A \to \omega$, where $A, B \in V$ and $\omega \in T$ is called left linear.

For Example: $S \to B00 \mid C11$

$B \to C11 \mid S \mid 1$

$C \to B00 \mid 00$

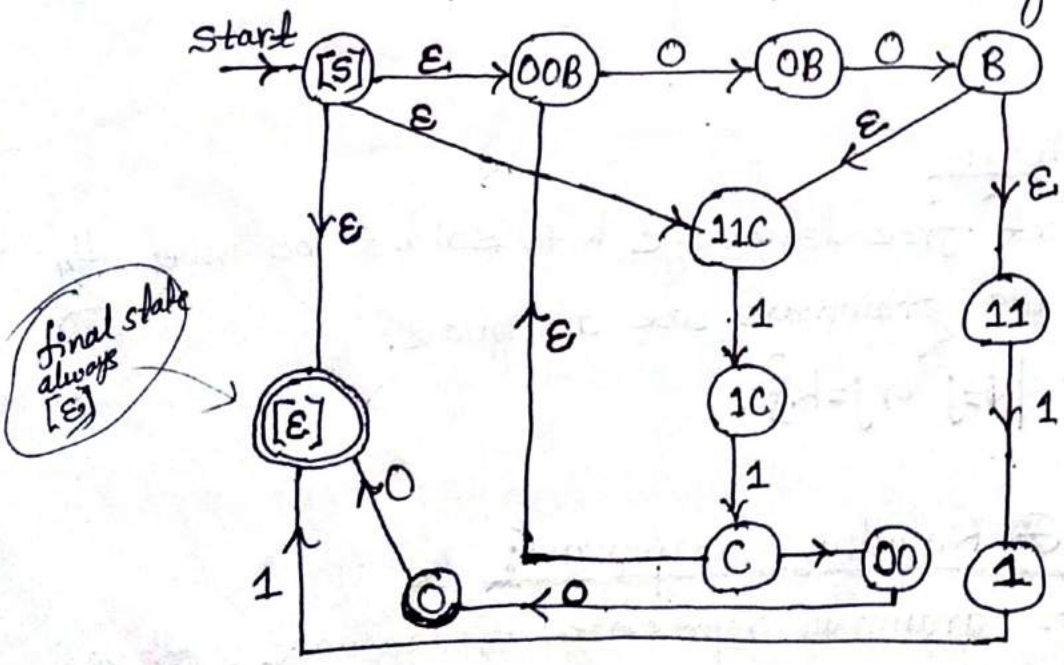
⊛. Equivalence of Regular Grammar and Finite Automata:

Example 1: $S \to 00B \mid 11C \mid \varepsilon$

$B \to 11C \mid 11$

$C \to 00B \mid 00$

Solution: The finite automation for the above grammar is given as;

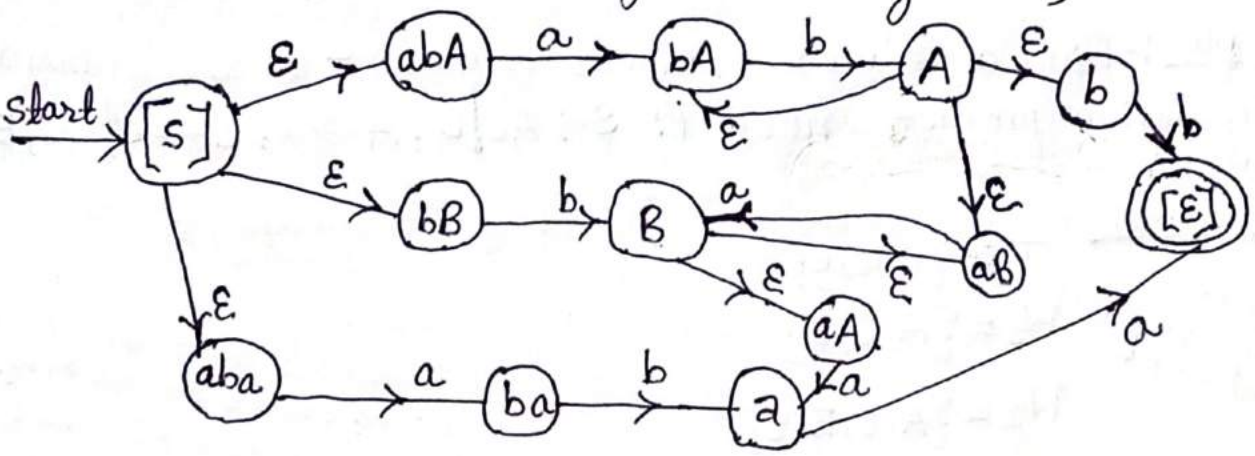


Process

First we started with our start variable [S]. which goes to 00B, 11C and [ε] on getting empty value ε. [ε] is a final state. Now 00B and 11C are extended until when single B or C remains. Now, B goes to 11C and 11 on ε & C goes to 00B and 00 on ε. Now, 00B & 11C are already extended So remaining 11 & 00 are extended and finally sent to final state.

Example 2:- $S \rightarrow abA \mid bB \mid aba$
$A \rightarrow b \mid aB \mid bA$
$B \rightarrow aB \mid aA$

Solution: Here, ε does not appear on the body part of any productions, but we introduce state [ε] and select it as accepting state. Thus the finite automation for the above grammar is given as;



⊛. Simplification of CFGs:

In CFGs sometimes all the production rules and symbols are not needed for the derivation of strings. Besides this, there may also be some NULL Productions and UNIT Productions. Elimination of these productions and symbols is called Simplification of CFG. Simplification consists of: i) Reduction of CFG ii) Removal of Unit Productions iii) Removal of Null Productions.

i) Reduction of CFG: CFGs are reduced in two phases: (This is also called elimination of useless symbols.)

Phase 1: Derivation of an equivalent grammar G', from the CFG, G, such that each variable derives some terminal string.

Derivation Procedure:

Step 1: Include all Symbols W_1, that derives some terminal and initialize $i=1$.

Step 2: Include symbols W_{i+1}, that derives W_i.

Step 3: Increment i and repeat step 2 until $W_{i+1} = W_i$.

Step 4: Include all production rules that have W_i in it.

(No need to remember it will be clear on solving example and comparing e.g. with procedure — Ivdo Neso Academy)

Phase 2: Derivation of an equivalent grammar G'', from the CFG, G', such that each symbol appears in a sentential form.

Derivation Procedure:

Step1: Include the start Symbol in Y_1 and initialize $i=1$.

Step2: Include all symbols Y_{i+1}, that can be derived from Y_i.

Similar to Phase1 Only Step1 differ in Y_1 used instead of W_1

Step3: Increment i and repeat Step2 until $Y_{i+1} = Y_i$.

Step4: Include all production rules that have Y_i in it.

Example:- Find a reduced grammar equivalent to the grammar G_1, having production rules P: $S \to AC|B, A \to a, C \to c|BC, E \to aA|e$.

Solution:

Phase1:

$T = \{a, c, e\}$ ← Set of terminal symbols

all symbols that can derive terminal symbol

$W_1 = \{A, C, E\}$

$W_2 = \{A, C, E, S\}$ ← all symbols that can derive symbols that are in W_1. Here S can derive AC which is also present in W_1. So, S is included.

Now, $W_3 = W_2$ i.e, $W_{i+1} = W_i$ So, we stop here

$W_3 = \{A, C, E, S\}$ ← All symbols that can be derive symbols that are in W_2.

Now,

$G' = \{(A, C, E, S), \{a, c, e\}, P, (S)\}$

recent non-terminal symbols that we get from W_3.

Terminal symbols that can be derived from (A, C, E, S)

Since $G = \{V, T, S, P\}$

Now, Production Rule can be defined as;

Any production rule.

Start symbol

P: $S \to AC, A \to a, C \to c, E \to aA|e$ ←

First look at P: given in question now copy it eliminating symbols that are not in G'

Phase2:

In phase 2 we will look at grammer G' and Production P obtained in phase1

$Y_1 = \{S\}$ ← only start symbol included

$Y_2 = \{S, A, C\}$ ← All symbols that Y_1 i.e, S can derive.

$Y_3 = \{S, A, C, a, c\}$ ← All symbols that Y_2 can derive

$Y_4 = \{S, A, C, a, c\}$ ← Since $Y_{i+1} = Y_i$ we stop here

Now,

$G' = \{(A, C, S), \{a, c\}, P, \{S\}\}$

P: $S \to AC, A \to a, C \to c$ ←

First look at P in Phase1 then copy by reducing. No E symbol in above G' so reduced in P also

Since, this P in phase 2 has lesser symbols than that of P in question. So we have reduced the given grammar.

ii) Removal of Unit Productions:

Any production rule of the form $A \to B$ where $A, B \in$ Non Terminals is called Unit Production.

Procedure:

Step1: To remove $A \to B$, add production $A \to x$ to the grammar rule whenever $B \to x$ occurs in the grammar. [$x \in$ Terminal, x can be Null].

Step2: Delete $A \to B$ from the grammar.

Step3: Repeat from step1 until all Unit Productions are removed.

Example: Remove Unit Productions from the Grammar whose production rule is given by P: $S \to XY, X \to a, Y \to Z|b, Z \to M, M \to N, N \to a$.

Solution:

Here, $Y \to Z, Z \to M, M \to N$ are the Unit Productions given by P in the given question. Now we have to remove them as follows:

For $M \to N$

Since, $N \to a$, we add $M \to a$ (according to step 1)

P: $S \to XY, X \to a, Y \to Z|b, Z \to M, M \to a, N \to a$ (P now changed as this)

For $Z \to M$

Since $M \to a$, we add $Z \to a$

P: $S \to XY, X \to a, Y \to Z|b, Z \to a, M \to a, N \to a$.

For $Y \to Z$

Since $Z \to a$, we add $Y \to a$.

P: $S \to XY, X \to a, Y \to a|b, Z \to a, M \to a, N \to a$.

(Here, $Y \to Z|b$ since $Z \to a$ so, $Y \to a|b$ this b remains as it is)

Hence, we removed all Unit Productions. But here if we look at P above the symbols Z, M and N are unreachable symbols since, all these are giving a. Here, S is giving XY, X is giving a, where a is terminal symbol, also Y gives a & b which are also terminal symbols. There is no way from start symbol S through which we can reach Z, M and N. Hence, we have to remove the unreachable symbols and our production rule finally becomes as;

P: $S \to XY, X \to a, Y \to a|b$.

iii) Removal of Null Productions:

In a CFG, a Non-Terminal Symbol 'A' is a nullable variable if there is a production $A \to \varepsilon$ or there is a derivation that starts at 'A' and leads to ε. (like $A \to \ldots \to \varepsilon$).

Procedure:

Step1: To remove $A \to \varepsilon$, look for all productions whose right side contains A.

Step2: Replace each occurences of 'A' in each of these productions with ε.

Step3: Add the resultant productions to the Grammar.

Example: Remove Null Productions from the following Grammar.

$S \to ABAC,\ A \to aA|\varepsilon,\ B \to bB|\varepsilon,\ C \to c$

Solution:

Here, in the given question $A \to \varepsilon$ and $B \to \varepsilon$ are two null productions. So we remove them as follows;

To eliminate $A \to \varepsilon$

$S \to ABAC$ ← According to step1.

$S \to ABC|BAC|BC$ ← Different cases on replacing different A by ε (i.e, empty).
Replacing 1st A with ε = BAC
" 2nd A with ε = ABC
replacing both = BC

Also,

$A \to aA$ ← Since according to step1 $A \to aA|\varepsilon$ also contains A on right side so this is also replaced by ε

$A \to a$ ← on replacing A by ε (i.e, empty)

New Production: $S \to ABAC|ABC|BAC|BC$

$A \to aA|a$ ← we have now a so ~~a changed by~~ ε no longer remains as eliminated by a

$B \to bB|\varepsilon$

$C \to c$

To eliminate $B \to \varepsilon$ ← for this we look at new production above

$S \to ABAC|ABC|BAC|BC$ ← Since it contains B on right side as step1

$S \to AAC|AC|C$ ← ABAC if B is ε = AAC, For ABC if B is ε then AC, for BAC if B is ε then AC but AC already there so no need to write. Similarly BC, B on getting ε give C.

Also,

$B \to bB$

$B \to b$ ← bB, B on getting ε

Others remains same

New Production: $S \to ABAC|ABC|BAC|BC|AAC|AC|C$

$A \to aA|a$

$B \to bB|b$

$C \to c.$

Hence, all Null productions are removed. //

⊛ Chomsky Normal Form (CNF):-

In Chomsky Normal Form (CNF) we have a restriction on the length of RHS, which is: elements in RHS should either be two variables or a Terminal. A CFG is in Chomsky Normal Form if the productions are in the following forms:

$A \to a$
$A \to BC$ where A, B and C are non-terminals & a is a terminal.

Steps to convert a given CFG to Chomsky Normal Form:

Step1: If the start symbol S occurs on some right side, create a new start symbol S' and a new production $S' \to S$.

Step2: Remove Null Productions. (By Method that we discussed earlier).

Step3: Remove Unit Productions. (By Method that we discussed earlier).

Step4: Replace each Production $A \to B_1 \ldots . B_n$ where $n > 2$, with $A \to B_1 C$ where $C \to B_2 \ldots . B_n$. Repeat this step for all productions having two or more symbols on the right side.

Step5: If the right side of any Production is in the form $A \to aB$ where 'a' is a terminal and A and B are non-terminals, then the production is replaced by $A \to XB$ and $X \to a$. Repeat this step for every Production which is of the form $A \to aB$.

Example: Convert the following CFG to CNF:

P: $S \to ASA \mid aB$, $A \to B \mid S$, $B \to b \mid \varepsilon$.

Solution:

Step1: Since S appears in RHS, we add a new state S' and $S' \to S$ is added to the production.

P: $S' \to S$, $S \to ASA \mid aB$, $A \to B \mid S$, $B \to b \mid \varepsilon$.

Step2: Remove the Null Productions: $B \to \varepsilon$ and $A \to \varepsilon$.

After Removing $B \to \varepsilon$: P: $S' \to S$, $S \to ASA \mid aB \mid a$, $A \to B \mid S \mid \varepsilon$, $B \to b$

After Removing $A \to \varepsilon$: P: $S' \to S$, $S \to ASA \mid aB \mid a \mid AS \mid SA \mid S$, $A \to B \mid S$, $B \to b$

always look at latest production P:

Step3: Remove the Unit Productions: $S \to S$, $S' \to S$, $A \to B$ and $A \to S$:

After Removing $S \to S$: P: $S' \to S$, $S \to ASA \mid aB \mid a \mid AS \mid SA$, $A \to B \mid S$, $B \to b$

After Removing $S' \to S$: P: $S' \to ASA \mid aB \mid a \mid AS \mid SA$,
$S \to ASA \mid aB \mid a \mid AS \mid SA$,
$A \to b \mid S$, $B \to b$

i.e. $S \neq S$

Here S is not removed directly from S', instead replaced by value of S becoz LHS ≠ RHS

Here S is directly removed because it is $S \to S$, LHS and RHS same so.

(यसमा B लाई replace गरिन्छ)

After Removing A→B:

P: S'→ASA|aB|a|AS|SA, S→ASA|aB|a|AS|SA,
A→b|S, B→b.

After Removing A→S:

P: S'→ASA|aB|a|AS|SA, S→ASA|aB|a|AS|SA,
A→b|ASA|aB|a|AS|SA, B→b.

Step4: Now find out the productions that has more than two variables in RHS.

S'→ASA, S→ASA and A→ASA.

After removing these, we get:

P: S'→AX|aB|a|AS|SA, S→AX|aB|a|AS|SA,
A→b|AX|aB|a|AS|SA, B→b, X→SA.

Step5: Now change the productions: S'→aB, S→aB and A→aB

Finally we get: P: S'→AX|YB|a|AS|SA,
S→AX|YB|a|AS|SA,
A→b|AX|YB|a|AS|SA,
B→b,
X→SA,
Y→a.

(Since X was already used so we took next variable Y)

⊛. Greibach Normal Form (GNF):-

A CFG is in Greibach Normal Form if the productions are in following forms:

A→b
A→bC1C2....Cn.

where A, C1,....,Cn are Non-Terminals and b is a Terminal.

Steps to Convert a given CFG to GNF:-

Step1: Check if the given CFG has any Unit Productions or Null Productions and Remove if there are any (using Unit & Null Productions removal methods discussed before).

Step2: Check whether the CFG is already in Chomsky Normal Form (CNF) and convert it to CNF if it is not.

Step3: Change the names of the Non-Terminal Symbols into some A_i in ascending order of i.

Step4: Alter the rules so that the Non-Terminals are in ascending order, such that, If the Production is of the form $A_i \to A_j x$, then, $i < j$ and should never be $i \geq j$.

Example: Convert the given CFG to GNF:

$S \to CA \mid BB,\ B \to b \mid SB,\ C \to b,\ A \to a$

Last step4 is hard to understand better to study with video by neso academy

Solution:

Step1: Since there are no any Unit Productions or Null Productions in the given question, so we are done with step 1.

Step2: The given CGF is already in Chomsky Normal Form so, we are also done with step2.

Step3: Here we have S, C, A and B as our Non-Terminal Symbols. Now, we have to change these as follows:

Replace: S with A_1
C with A_2
A with A_3
B with A_4

But remember we should rename these in ascending order as the production given in question. Order should not be changed.

Now, we get equivalent production as;

$A_1 \to A_2A_3 \mid A_4A_4$
$A_4 \to b \mid A_1A_4$
$A_2 \to b$
$A_3 \to a$

Step4:

$A_1 \to A_2A_3 \mid A_4A_4$
$A_4 \to b \mid A_2A_3A_4 \mid A_4A_4A_4$
$A_4 \to b \mid bA_3A_4 \mid A_4A_4A_4$

Let we take ~~For~~, $A_1 \to A_2A_3$ we have form $A_i \to A_j x$. Now check if $i < j$. Here $1 < 2$, so no need to alter. but if $i \geq j$ then we replace A_j with value of A_j that is in any other part of production

If it is of form $bG_1, \ldots G_n$ then it is in GNF so, now this part is okay.

A_1 repraced by $A_2A_3 \mid A_4A_4$ and others copied as they are but this again not following the rule.

If $i = j$ then it is left recursion, so it must be removed by rules as...

Since we encounter $i = j$, So, now we remove it by left recursion by introducing a new variable. Let new variable be Z. Now we make new production for Z by taking variables following the problematic one and once write it with new variable and once without it.

$Z \rightarrow A_4A_4Z \mid A_4A_4$

Now production for A_4 becomes;

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$A_4A_4A_4$ was our problem last A_4 is problematic one followed by A_4A_4. So, once write A_4A_4Z and once A_4A_4

Since we had $A_4 \rightarrow b \mid bA_3A_4 \mid A_4A_4A_4$.
Now, $b \mid bA_3A_4 \mid$ is written as it is. Then the term following b is changed by Z once and once written by adding Z to bA_3A_4
Now this is in GNF

Now the Grammar is:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Here A_2, A_4 at start violates the form of GNF so they are replaced by corresponding values.

A_4 in A_1 replaced by b in A_4 and others copied as it is

Here,

$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z.$

$Z \rightarrow bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4 \mid bA_4Z \mid bA_3A_4Z \mid bZA_4Z \mid bA_3A_4ZA_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Now we have everywhere terminal symbols at beginning so we are done and this is required GNF

⊛. Bakus-Naur Form (BNF):

This is another notation used to specify the CFG. It is named so after John Bakus, who invented it, and Peter Naur, who refined it. Here, concept is similar to CFG, only the difference is instead of using symbol "→" in production, we use symbol ::= . We enclose all non-terminals in angle brackets, <>.

For Example: The BNF for identifiers is as;

<identifier> ::= <letter or underscore> | <identifier> | <symbol>

<letter or underscore> ::= <letter> | <_>

<symbol> ::= <letter or underscore> | <digit>

<letter> ::= a | b | .... | z.

<digit> ::= 0 | 1 | 2 | .... | 3.

⊛. Context Sensitive Grammar:

A context sensitive grammar (CSG) is a formal grammar in which left hand sides and right hand sides of any production may be surrounded by a context of terminal and non-terminal symbols.

Formally CSG can be defined as;

A formal grammar, $G=(V,T,p,S)$ is the context sensitive if all the production 'P' are of the form;

$\alpha A\beta \rightarrow \alpha\gamma\beta$, where $A \in V, \alpha\beta \in (V \cup T)^*$ and $\gamma \in (V \cup T)^+$.

The name context sensitive is explained by α and β the form the context of A and determine whether A can be replaced with γ or not. Thus the production $A \rightarrow \gamma$ can only be applied in contexts where α occurs to left of A and β occurs to right of A.

Example: $\{a^n b^n c^n \mid n \geq 1\}$ is a context sensitive defined as;

$S \rightarrow aSBC \mid aBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bc \rightarrow bc$

$cC \rightarrow cc$

⊛. Chomsky Hierarchy:

Grammar Type	Grammar Accepted	Language Accepted	Automation
Type-0	Unrestriced Grammar	Recursively Enumerable Language	Turning Machine
Type-1	Context Sensitive Grammar	Context Sensitive Language	Linear Bounded Automation
Type-2	Regular Grammar	Context Free Language	Pushdown Automata
Type-3	Regular Grammar	Regular Language	Finite State Automation.

⊛ Pumping lemma for CFL:

(application of pumping lemma)

The pumping lemma for context free language is used to prove that a language is Not Context Free.

If A is a Context Free Language, then, A has a Pumping Length 'P' such that any string 'S', where $|S| \geq P$ may be divided into 5 pieces $S = uvxyz$ such that the following conditions must be true:

i) uv^ixy^iz is in A for every $i \geq 0$
ii) $|vy| > 0$
iii) $|vxy| \leq P$

(everything similar to pumping lemma for Regular Expression discussed before only difference is condition ① i.e, instead of xy^iz here we have uv^ixy^iz)

To prove that a language is Not Context Free, using Pumping Lemma (for CFL) we follow the steps given below: (We prove using CONTRADICTION).

→ Assume that A is Context Free.
→ It has to have a Pumping Length (say P).
→ All strings longer than P can be pumped $|S| \geq P$.
→ Now find a string 'S' in A such that $|S| \geq P$.
→ Divide S into uvxyz.
→ Show that $uv^ixy^iz \in A$ for some i.
→ Then consider the ways that S can be divided into uvxyz.
→ Show that none of these can satisfy all the 3 pumping conditions at the same time.
→ S cannot be pumped == CONTRADICTION.

Example:- Show that $L = \{a^N b^N c^N \mid N \geq 0\}$ is Not Context Free (CFL).

Solution:- We prove this by contradiction. Let we assume that given language L is context free. If it is context free then we will have pumping length (say P). Now we take a string S such that $S = a^P b^P c^P$. Let we take $P = 4$ then, $S = a^4 b^4 c^4$. Now we divide S into parts uvxyz. So, there are two cases as follows:

Case-I: v and y each contain only one type of symbol.

i.e, aaaabbbbcccc (i.e, $a^4b^4c^4$)

u v x y z

($|vy| > 0$ & $|xvy| \geq P$ | since, $P=4$, $x=5$, $v=2$, $y=1$, $xvy = 8$, $\therefore |xvy| \geq P$ i.e, $|8| \geq 4$)

Now for this case let we check condition uv^ixy^iz taking $i=2$ i.e, uv^2xy^2z.

$\Rightarrow$ aaaaaabbbbccccc

Here, we get $a^6b^4c^5$

(यो कुराहरूलाई मात्र हो directly 1 line मा लेख्दा $a^6b^4c^5 \notin L$ as me ege लेख्ने)

Since $a^6b^4c^5$ does not follow the pattern $a^Nb^Nc^N$ given in the question which means it does not belongs to given language L. We assumed earlier that L is context free which means it must satisfy all three conditions, but it does not satisfy which lead to contradiction.

(first condition नै match नगरे so no need to check second & third conditions. If it matches then we check) (does not satisfy first condition)

Case-II: Either v or y has more than one kind of symbols.

i.e, aaaabbbbcccc

u v x y z

Now, for this case let we check condition first i.e, uv^ixy^iz taking $i=2$ i.e, uv^2xy^2z.

$\Rightarrow$ aaaabbaabbbbbcccc

Here the pattern of language $a^Nb^Nc^N$ is not followed. So this string also does not belong to L.

So, we proved that xv^ixy^iz taking $i=2$ all cases does not lie in our language. This means S cannot be pumped which leads to contradiction. Hence, the language is not CFL.

Example 2: Show that $L = \{WW \mid W \in \{0,1\}^*\}$ is Not Context Free.

Solution:-

Assume that L is Context Free then, L must have a Pumping length (say P). Now we take a string S such that $S = 0^P1^P0^P1^P$. If we take $P=5$ then, $S = 0^51^50^51^5$.

Now we divide S into parts uvxyz. So, we have cases as follows:

Case 1: vxy does not straddle a boundary.

(Straddle means it does not lie on either sides of boundary but it lies in one part only without crossing boundary.)

00000111110000011111

u vxy z

(v is 1^{st} 1, x is 2^{nd} 1, y is 3^{rd} 1 respectively)

Now we check our first condition uv^ixy^iz taking $i=2$.
i.e, uv^2xy^2z.

$\Rightarrow 000001111111000001111$

Here we get, $0^5 1^7 0^5 1^5$

(W any string
W=W in question,
but Here
i.e, W≠W according
to as our
question.)

Since First half $0^5 1^7 \neq$ second half $0^5 1^5$ so, this does not belong to L. i.e, $0^5 1^7 0^5 1^5 \notin L$.

Case II: vxy straddles the first boundry.

i.e, 00000111111000001111
u v x y z

(we can take in any way
Here, v as 00
x as 1
& y as 11)

Now we check for our first condition uv^ixy^iz taking $i=2$.
i.e, uv^2xy^2z.

$\Rightarrow 000000011111110000011111$

Here we get, $0^7 1^7 0^5 1^5$.

Since $0^7 1^7 \neq 0^5 1^5$, So, $0^7 1^7 0^5 1^5 \notin L$.

Case III: vxy straddles the third boundry.

i.e, 00000111110000011111
u v x y z

(taken
v as 00
x as 1
y as 11)

Now again we check for our first condition uv^ixy^iz taking $i=2$.
i.e, uv^2xy^2z.

$\Rightarrow 000001111100000011111$

Here we get, $0^5 1^5 0^7 1^7$

Since $0^5 1^5 \neq 0^7 1^7$. So, $0^5 1^5 0^7 1^7 \notin L$.

Case IV: vxy straddles the midpoint.

i.e, 00000111110000011111
u v x y z

(taken
v as 11
x as 1
& y as 00)

Now again we check for our first condition taking $i=2$. i.e, uv^2xy^2z.

$\Rightarrow 000001111111000000011111$

Here we get, $0^5 1^7 0^7 1^5$.

Since, $0^5 1^7 \neq 0^7 1^5$. So, $0^5 1^7 0^7 1^5 \notin L$.

Since any of the cases does not satisfy our conditions, it leads to contradiction for our assumption. Hence, the given language is not CFL.

⊛ Closure Properties of Context Free Languages (CFL):

Following are some of the principal closure properties of context free languages:

i) The context free language are closed under union:

Given any two context free languages L1 and L2, their union $L1 \cup L2$ is also context free language.

ii) The context free language are closed under concatenation:

Given any two context free languages L1 and L2, their concatenation $L1.L2$ is also context free language.

iii) The context free language are not closed under intersection:

Given any two context free languages L1 and L2, their intersection $L1 \cap L2$ is not context free language.

iv) The context free language are closed under Kleen closure:

v) The context free language are not closed under complement.

Unit-5

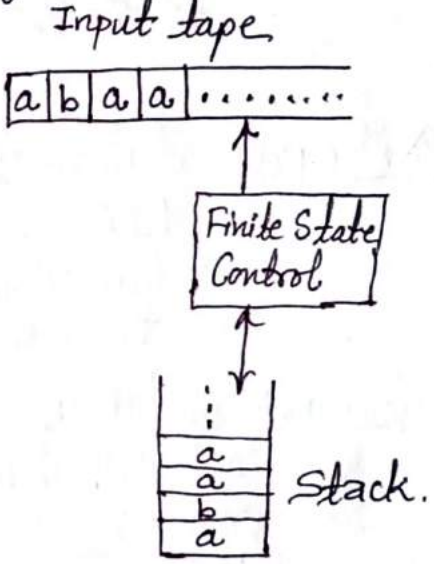
Push Down Automata (PDA)

Introduction:- The context free languages have a type of automaton that defines them. This automaton is called "pushdown automaton" which can be thought as a ε-NFA with the addition of stack. The presence of stack means that, the pushdown automata can ~~rember~~ remember infinity amount of information. However, the pushdown automaton can only access the information on its stack in a last-in-first-out way.

We can define PDA informally as shown in figure:

PDA is an abstract machine determined by following three things:

- → Input table
- → Finite state control
- → A stack.



Each moves of the machine is determined by three things;

- → The current state.
- → Next input symbol
- → Symbol on the top of stack.

The moves consist of;

- → Changing state | Staying on same state.
- → Replacing the stack top by string of zero or more symbols.

Poping the top symbol off the stack means replacing it by ε.

Pushing on the stack means replacing stack's top, say x, by yx.

Formal Definition:

A PDA is defined by seven tuples $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ (Gamma, del)

where,

Q = A finite set of states.

Σ = A finite set of input symbols.

Γ = A finite stack Alphabets. (Gamma or uppercase gamma)

δ = transition function

q_0 = The start state.

z_0 = The start stack Symbol

F = The set of Final/Accepting States.

Here, δ takes as argument a triple $\delta(q, a, X)$ where:

i) q is a state in Q.
ii) a is either an input symbol in Σ or $a = \varepsilon$.
iii) X is a stack symbol, that is a member of Γ.

The output of δ is finite set of pairs (p, γ) where;

p is a new state.
γ is a string of stack symbols that replaces X at the top of the stack.

Example:
If $\gamma = \varepsilon$, then the stack is popped.
If $\gamma = X$ then the stack is unchanged.
If $\gamma = YZ$ then X is replaced by Z and Y is pushed onto the stack.

⊛ Representation of PDA:

PDA can be represented by using two techniques:
→ transition table
→ transition diagram.

i) **Transition table:-** Consider a context free language defined by the grammar $a^n b^n$. Then the transition table used to represent this language is given as;

q_0 on getting ε, pops out top of stack

initial stack symbol

Move Number	State	Input	Stack Symbol	Move (δ)
1	q_0	A	Z_0	(q_0, aZ_0).
2	q_0	A	A	(q_0, aa).
3	q_0	ε	A	(q_0, a)
4	q_1	B	A	(q_1, ε)
5	q_1	ε	Z_0	(q_2, Z_0).

State q_0 on getting input A goes to top of stack Z_0

Our last input was A so, top of stack is now A. Now state q_0 on getting another A's stays on q_0 and push input symbol to stack

q_1 on getting input B to top of stack A switches the state q_1 and pop the stack

If state q_1 is reading nothing (ε) and top of stack is Z_0 then it switches to final state q_2

Here, q_0 is the starting state of PDA, q_2 is the final state of PDA and Z_0 is the initial stack symbol.

ii) Transition Diagram:

We can use transition diagram to represent a PDA, where

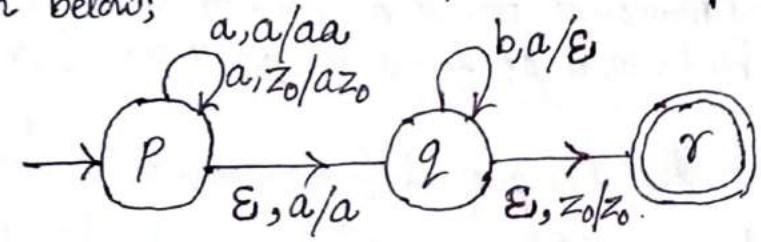
→ Any state is represented by a node in a diagram.

→ Any arrow comming from nowhere to a state indicates the start state and doubly circled states are accepting/final states.

→ The arc corresponds to transition of PDA as;

- Arc labelled as $a, X|\alpha$ means: $\delta(q, a, x) = (p, \alpha)$ for arc from state p to q.

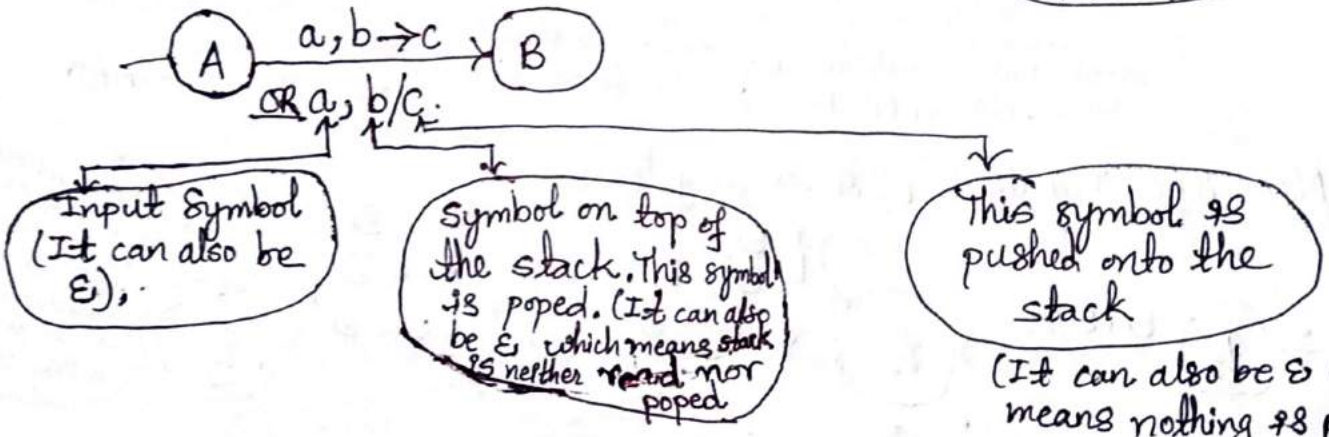
The PDA of $a^n b^n$ can be represented using transition diagram as shown below;



where, p, q, r are state names we can name any

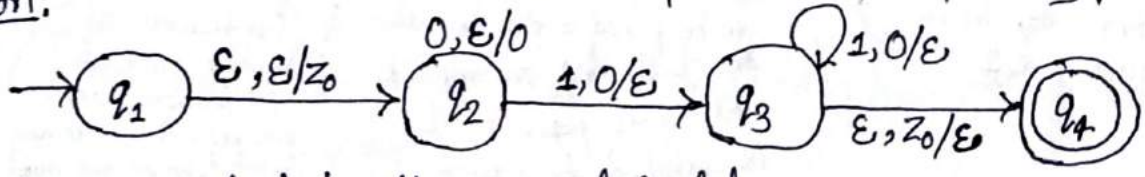
This explains Operation and move of PDA including examples

Graphical Notation of Push down automata:



Example 1: Construct a PDA that accepts $L = \{0^n 1^n | n \geq 0\}$.

Solution:



Here, we started with any initial state q_1. In q_1 we have transitions $\varepsilon, \varepsilon/Z_0$ where ε is input. ε/Z_0 denotes initially stack contains ε(empty) symbols and Z_0 is pushed symbol on stack which denotes it is the first (or bottom most) element of stack. Now we are on state q_2 where we can get input 0, or 1 according to the question. Now on getting 0 as input we do not pop anything from stack (i.e, ε), but we push the comming 0's to stack. Similarly on getting input 1 we go to next state q_3. Here $0/\varepsilon$ denotes 0 is the topmost element of stack which should be poped from the stack and ε denotes

nothing to be pushed onto stack.

Now in state q_3 on getting input 1, check if 0 is on the topmost position of stack, if yes then, pop it and nothing has to be pushed onto the stack. Now on getting input ε, we check if z_0 is on the topmost position of stack, if yes then nothing has to be pushed onto the stack and we reached to final state q_4 accepting the particular string.

Example 2: Construct a PDA that accepts Even Palindromes of the form $L = \{WW^R \mid W = (a+b)^+\}$.

Solution:

We know that palindromes is a word or sequence that reads the same backwards as forwards. e.g. NOON, 123321, abba etc.

Let us understand the language before constructing PDA.

$L = \{WW^R \mid W = (a+b)^+\}$

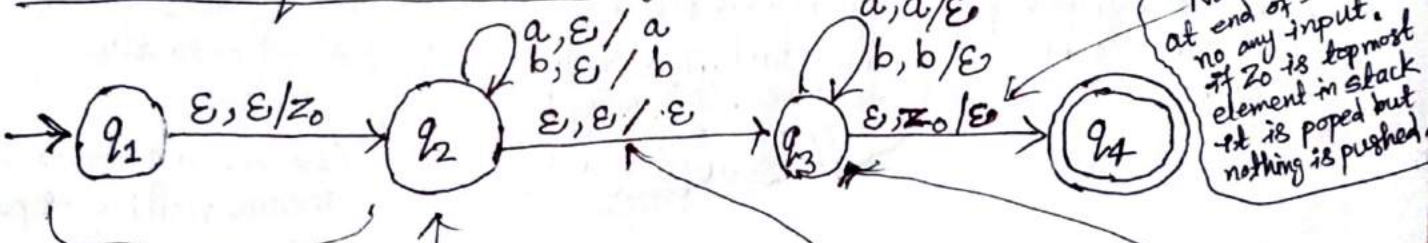
4 letters so even palindrome

this +ve closure shows that there must be atleast one symbol be there, it cannot be empty & a, b are our inputs.

first W represents first half of palindrome for e.g. NO for NOON

second W with R represents reverse of first half. i.e, ON. (if we take NOON)

Now, the required PDA is as follows:-



Now we reached at end of string so no any input. If z_0 is topmost element in stack it is poped but nothing is pushed.

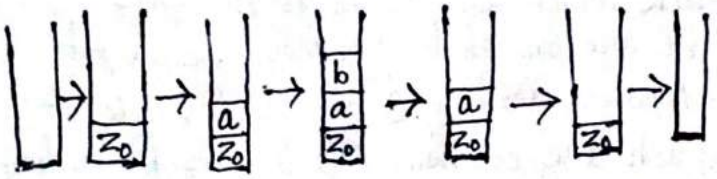
This part we discussed in example 1

q_2 on input a, nothing to be poped and a is pushed to stack. & q_2 on input b nothing is poped and b is pushed to stack

On reaching at middle part of palindrome word we do not input any symbol, we do not pop and we do not need to push anything

On q_3 getting a we check if topmost symbol is a on stack if it is then pop it & nothing to be pushed. Similarly for input b.

Now let we check if abba is accepted:



Hence accepted.

#If stack does not remain empty finally on tracing then that string is not accepted by PDA. (like string abab).

⊛ Instantaneous Description for PDA:

In short we say it as ID

Any configuration of a PDA can be described by a triplet (q, w, γ) where,

q is the state
w is the remaining input.
γ is the stack contents.

$\in$ → belongs to
ε → epsilon.

Such a description by triple is called an instantaneous description of PDA (ID). Instantaneous description is helpful for describing changes in the state, input and stack of the PDA.

Let $P=\{Q, \Sigma, \Gamma, \delta, q_0, z_0, F\}$ be a PDA. Then, we define a relation $\vdash$, "yields" as; $(q, aw, z\alpha) \vdash (p, w, \beta\alpha)$ if $\delta(q, a, z)$ contains (p, β) and may be a ε. This move reflects the idea that, by consuming "a" from the input, and replacing z on the top of stack by β, we can go from state q to state p.

Example: For the PDA described earlier accepting language WW^R, [see example 2] the accepting sequence of ID's for string 1001 can be shown as; $(q_0, 1001, \varepsilon)$

$\vdash (q_1, 1001, z_0)$ — Here 1001 is input string

$\vdash (q_1, 001, 1z_0)$

$\vdash (q_1, 01, z_0)$ — Here POP if there is mismatch between Input & z_0 content i.e, 0 and 1 mismatch here

$\vdash (q_1, 1, 0z_0)$

$\vdash (q_1, \varepsilon, z_0)$

$\vdash (q_2, \varepsilon, \varepsilon)$ Accepted.

Therefore $(q_0, 1001, z_0) \vdash^* (q_2, \varepsilon, \varepsilon)$.

$\vdash$ we read it as turnstile. It connects the two or more ID's. * after $\vdash$ symbol denotes moves can be more than one.

⊛ Language of a PDA:

We can define acceptance of any string by a PDA in two ways;

i) Acceptance by final state:

Given a PDA, P the language accepted by final state, $L(P)$ is;

$\{w \mid (q, w, z_0) \vdash^* (p, \varepsilon, \gamma)\}$ where $p \in F$ and $\gamma \in \Gamma^*$.

i.e, finally anything can be in stack no matter

ii) Acceptance by empty stack:

Given a PDA, P the language accepted by empty stack, $L(P)$ is

$\{w \mid (q, w, z_0) \vdash^* (p, \varepsilon, \varepsilon)\}$ where $p \in Q$.

i.e, finally stack should be empty

⊛. Deterministic Pushdown Automata (DPDA):-

A PDA is said to be deterministic if for every input alphabet 'a' (may be ε) and top of stack symbol, PDA has either unique transition (i.e, moves to only one state) or its transition is not defined.

Formally A pushdown automata $P=(Q,\Sigma,\Gamma,\delta,q_0,z_0,F)$ is deterministic pushdown automata if there is no configuration for which P has a choice of more than one moves. i.e, P is deterministic if following two conditions are satisfied;

i) For any $q\in Q, x\in\Gamma$, if $\delta(q,a,x)\neq\Phi$ for some $a\in\Sigma$ then $\delta(q,\varepsilon,x)=\Phi$
i.e, if $\delta(q,a,x)$ is non-empty for some a, then $\delta(q,\varepsilon,x)$ must be empty.

ii) For any $q\in Q$, $a\in\Sigma\cup\{\varepsilon\}$ and $x\in\Gamma$, $\delta(q,a,x)$ has at most one element.

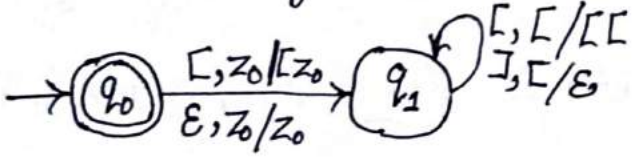
Example 1: Construct DPDA that accepts balanced parenthesis i.e, equal no. of opening and closing braces. eg: [[]], []] etc.

Solution:

The transition table for the given DPDA is as follows;

Move number	state	Input	stack Symbol	Move(s)
1.	q_0	[	z_0	$(q_1, [z_0)$
2.	q_1	[	[	$(q_1, [[)$
3.	q_1	]	[	(q_1, ε)
4.	q_1	ε	z_0	(q_0, z_0)

The transition diagram for this is as shown below;



Example 2: Construct a DPDA accepting language $L=\{wCw^R / w\in(0+1)^*\}$.

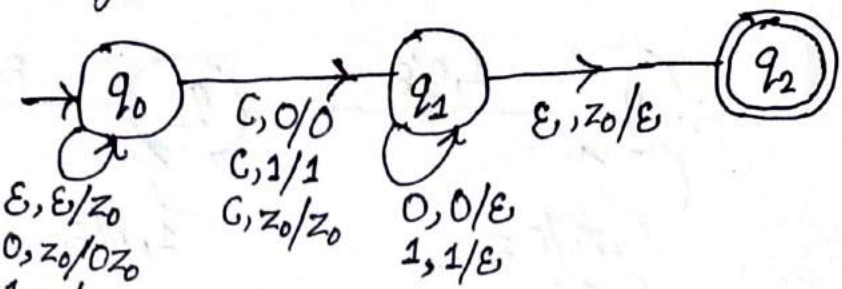
Solution:

$P=(\{q_0,q_1,q_2\},\{0,1\},\{0,1,z_0\},\delta,q_0,z_0,\{q_2\})$ where δ is defined as;

1. $\delta(q_0,\varepsilon,\varepsilon)=(q_0,z_0)$
2. $\delta(q_0,0,z_0)=(q_0,0z_0)$
3. $\delta(q_0,1,z_0)=(q_0,1z_0)$
4. $\delta(q_0,0,0)=(q_0,00)$

5. $\delta(q_0,1,1) = (q_0,11)$
6. $\delta(q_0,0,1) = (q_0,01)$
7. $\delta(q_0,1,0) = (q_0,10)$
8. $\delta(q_0,C,0) = (q_1,0)$
9. $\delta(q_0,C,1) = (q_1,1)$
10. $\delta(q_0,C,z_0) = (q_1,z_0)$
11. $\delta(q_1,0,0) = (q_1,\varepsilon)$
12. $\delta(q_1,1,1) = (q_1,\varepsilon)$
13. $\delta(q_1,\varepsilon,z_0) = (q_2,\varepsilon)$.

Now, the general notation or transition diagram is as follows:



⊛. Non Deterministic PDA (NPDA):

A nondeterministic pushdown automaton (npda) is basically an nfa with a stack added to it. A nondeterministic pushdown automaton is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$. It is an NFA which is a 5-tuple, and added two things to it:

Γ is a finite set of symbols called the stack alphabet and $z_0 \in \Gamma$ is the stack start symbol. Other 5 tuples are as usual we can describe them.

We also need to modify δ, the transition function, so that it manipulates the stack. The transition function for an npda has the form:

$$D: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to \text{finite subsets of } Q \times \Gamma^*.$$

⊛ Construction of PDA by Final State:

A PDA accepts a string when after reading the entire string in the final state. From the starting state we can make moves that end up in a final state with any stack values. The stack values are irrelevent as long as we end up in a final state.

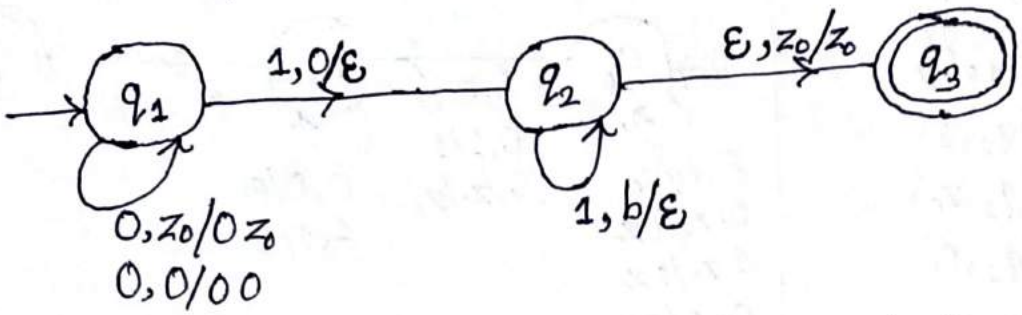
For a PDA $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ the language accepted by the set of final states F as;

$$L(PDA) = \{w \mid (q_0, w, z_0) \vdash^* (q, \varepsilon, \alpha)\} \text{ where } q \in F.$$

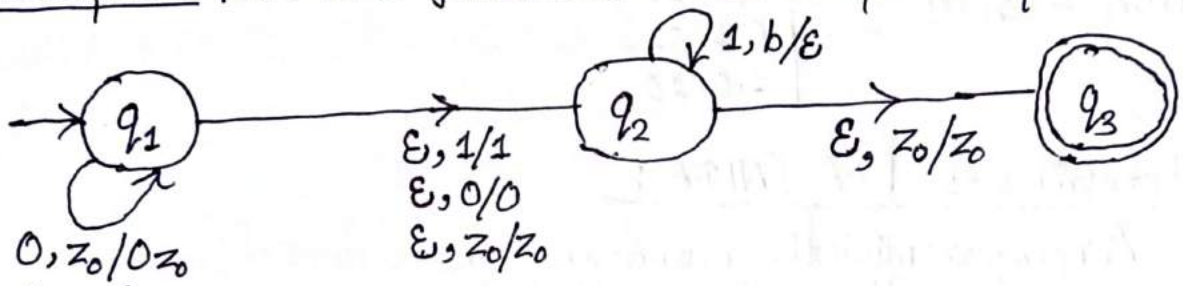
(* denotes moves may be more than one)

(such that) (ID consisting of triplets) (turnstein notation connecting IDs) (another ID consisting triplets)

Example1:- PDA with final state with equal number of zeros (0's) followed by equal numbers ones (1's).



Example2: PDA with final state that accept the palindrome.

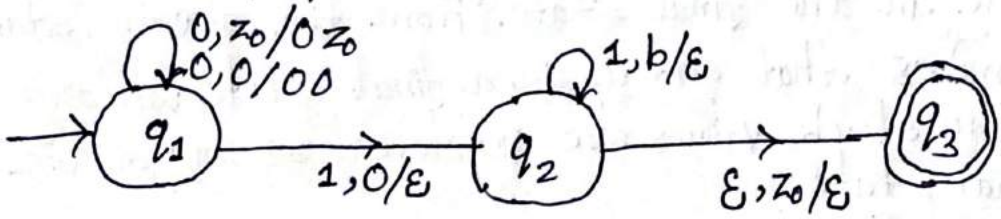


⊛. Construction of PDA by Empty Stack:-

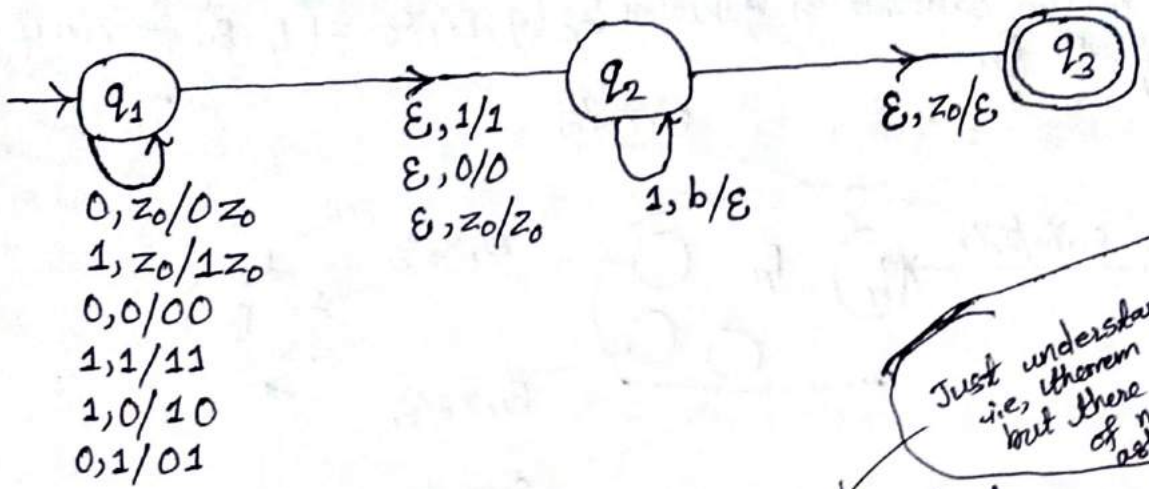
A PDA accepts a string when, after reading the entire string, when PDA has emptied its stack. For a PDA $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ the language accepted by empty stack is;

$$L(PDA) = \{w \mid (q_0, w, z_0) \vdash^* (q, \varepsilon, \varepsilon)\} \text{ where, } q \in Q.$$

Example1: PDA with empty stack with equal number of zeros (0's) followed by equal numbers ones (1's).



Example2: PDA with empty stack that accept the palindrome.



Just understand theories i.e, theorem may be asked but there is less chance of numerical being asked for this and vice-versa

⊛. Conversion of PDA accepting by Empty stack to accepting by final state:

Let us consider the PDA P_N that accepts a language L by empty stack and P_F is it's equivalent PDA that accepts L by final state.

Since no final state F written as we don't care about F during acceptance by empty stack

Theorem: If $L = L(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, then there is a PDA P_F such that $L = L(P_F)$.

Proof: The construction of P_F from P_N is done as follows:

1. Introduce the new symbol X_0 ($X_0 \notin \Gamma$) and place this symbol into the bottom of the stack P_F.
2. Introduce a new start state p_0, whose sole function is to push Z_0, the start symbol of P_N, onto the top of stack and enter state q_0 (the start state of P_N).
3. Copy the other states and transition functions of P_N in P_F.
4. Finally add another new state p_1, which is the accepting state of P_F; this PDA transfers to state p_F whenever it discoveres that P_N would have emptied its stack.

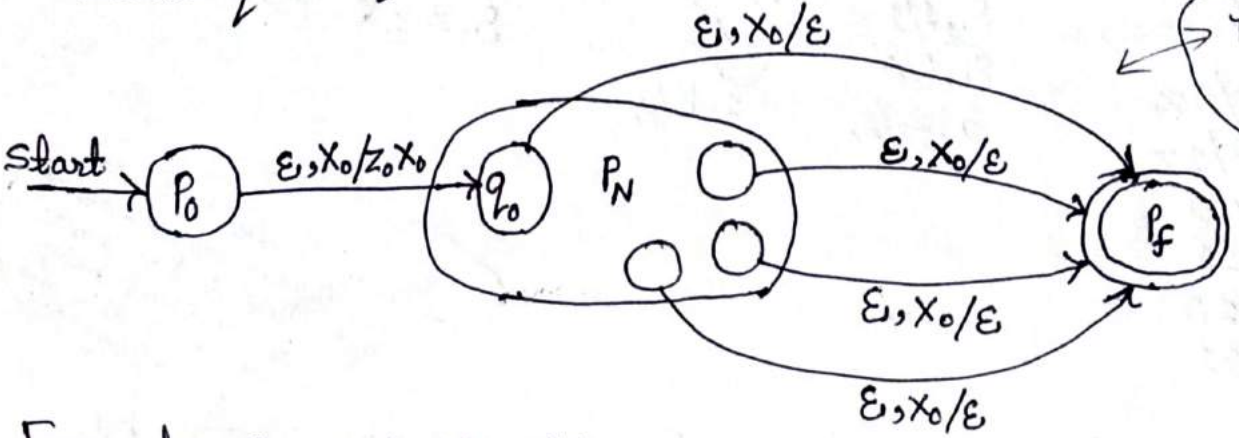
Thus the new constructed P_F has the components:

$$P_N = (Q \cup \{p_0, p_1\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_F\}).$$

where δ_F is defined by:

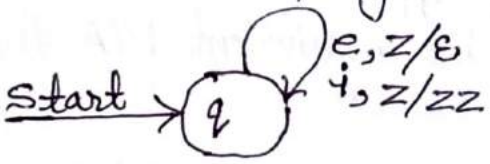
1. $\delta_F(p_0, \varepsilon, X_0) = (q_0, Z_0 X_0)$
2. For all states q in Q, inputs a in Σ or $a = \varepsilon$, and stack symbols Y in Γ, $\delta_F(q, a, Y)$ contains all the pairs in $\delta_N(q, a, Y)$. i.e, Copy all the transition functions of P_N.

3. Add another transition function $\delta_F(q, \varepsilon, X_0) = (p_f, \varepsilon)$ for every state q in Q.

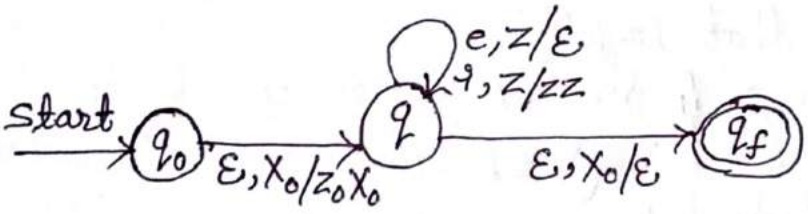


This figure illustrates whole process that we discussed

Example: Convert the following PDA that accepts the string using empty stack into its equivalent PDA that accepts the string by entering into the accepting state.



Solution:



⊛. Conversion of PDA accepting by final state to accepting by empty stack:

Theorem: Let L be $L(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Then there is a PDA P_N such that $L = N(P_N)$.

Proof: The construction of P_N from P_F is done as follows:

Steps 1 & 2 same as before only P_N and F_F names are interchanged.

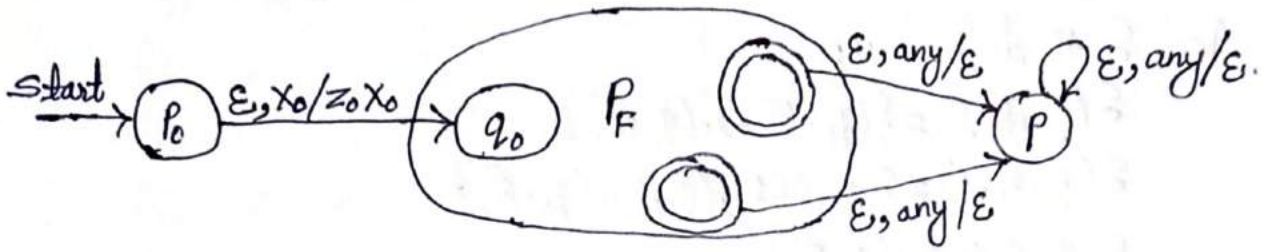
1. Introduce a new symbol X_0 ($X_0 \notin \Gamma$) and place this symbol into the bottom of the stack of P_N.
2. Introduce a new start state p_0, whose sole function is to push Z_0 (the start symbol of P_F), onto the top of the stack and enter state q_0 (the start state of P_F).
3. For each accepting state of P_F, add a transition on ε to a new state p.

Thus the new constructed P_N has the components:

$$P_N = \left(Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0 \right)$$

where, δ_N is defined by; (1 & 2 are same as before)

1. $\delta_N(p_0, \varepsilon, X_0) = (q_0, z_0X_0)$.
2. For all states q in Q, inputs a in Σ or $a=\varepsilon$, and stack symbols Y in Γ, $\delta_N(q,a,Y)$ contains all the pairs in $\delta_F(q,a,Y)$. i.e, Copy all the transition functions of P_N.
3. For all accepting states q in F and stack symbols Y in Γ or $Y=X_0$, $\delta_N(q,\varepsilon,Y)=(p,\varepsilon)$. By this rule, whenever P_F accepts, P_N can start emptying its stack without consuming any more input.
4. For all stack symbols Y in Γ or $Y=X_0$, $\delta_N(q,\varepsilon,Y)=(p,\varepsilon)$.



⊛ Conversion of CFG to PDA:

Given a CFG, $G_1 = (V, T, P \text{ and } S)$, we can construct a push down automata, M which accepts the language generated by the grammar G_1. i.e, $L(M) = L(G_1)$.

The machine can be defined as;

$$M = (\{q\}, T, V \cup T, \delta, q, S, \Phi)$$

where,

$Q = \{q\}$ is only the state in the PDA.

$\Sigma = T$

$\Gamma = V \cup T$ (i.e, PDA uses terminals and variables of G_1 as stack symbols).

$z_0 = S$ (i.e, initial stack symbol is start symbol in grammar).

$F = \Phi$

And δ is defined as;

i) $\delta(q, \varepsilon, A) = \{(q, \alpha) / A \to \alpha$ is a production P of $G_1\}$.

ii) $\delta(q, a, a) = \{(q, \varepsilon)$, for all $a \in T\}$.

Example: Convert the grammar defined by following production into PDA;

$S \to 0S1 | A$

$A \to 1S0 | S | \varepsilon$.

Solution: Let $G_1 = (V, T, P \text{ and } S)$ defined by following productions;

$S \rightarrow 0S1 | A$

$A \rightarrow 1S0 | S | \varepsilon$

PDA equivalent to this grammar is as;

$$M = (\{q_0\}, \{0,1\}, \{0,1,S,A\}, \delta, q_0, S, \phi)$$

where,

$Q = \{q_0\}$

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

$\Gamma = \{0,1,S,A\}$

$z_0 = S$

$F = \phi$

And δ is defined as;

$\delta(q_0, \varepsilon, S) = \{(q_0, 0S1), (q_0, A)\}$

$\delta(q_0, \varepsilon, A) = \{(q_0, 0S1), (q_0, S), (q_0, \varepsilon)\}$

$\delta(q_0, 0, 0) = \{(q_0, \varepsilon)\}$

$\delta(q_0, 1, 1) = \{(q_0, \varepsilon)\}$

Example 2: Construct a PDA equivalent to following grammar defined by;

$S \rightarrow aAA$

$A \rightarrow aS | bS | a$. Also trace acceptance of aaabaaaaa.

Solution:

Let $G = (V, T, P \text{ and } S)$ be the grammar defined by the production;

$S \rightarrow aAA$

$A \rightarrow aS | bS | a$

Now, PDA equivalent to this grammar is as;

$$M = (\{q_0\}, \{a,b\}, \{a,b,S,A\}, \delta, q_0, S, \phi)$$

where δ is defined as;

$\delta(q_0, \varepsilon, S) = \{q_0, aAA\}$

$\delta(q_0, \varepsilon, A) = \{(q_0, aS), (q_0, bS), (q_0, a)\}$

$\delta(q_0, a, a) = \{q_0, \varepsilon\}$

$\delta(q_0, b, b) = \{q_0, \varepsilon\}$

Now, we trace the acceptance of aaabaaaaa.

$(q_0, aaabaaaaa, S)$

$\vdash (q_0, aaabaaaaa, aA)$

$\vdash (q_0, aabaaaaa, AA)$

$\vdash(q_0, aabaaaaa, aSA)$
$\vdash(q_0, abaaaaa, SA)$
$\vdash(q_0, abaaaaa, aAAA)$
$\vdash(q_0, baaaaa, AAA)$
$\vdash(q_0, baaaaa, bSAA)$
$\vdash(q_0, aaaaa, SAA)$
$\vdash(q_0, aaaaa, aAAAA)$
$\vdash(q_0, aaaa, AAAA)$
$\vdash(q_0, aaaa, aAAA)$
$\vdash(q_0, aaa, AAA)$
$\vdash(q_0, aaa, aAA)$
$\vdash(q_0, aa, AA)$
$\vdash(q_0, aa, aA)$
$\vdash(q_0, a, A)$
$\vdash(q_0, a)$

In CFG1

$S \to aAA$
$\to aaSA$
$\to aaaAAA$
$\to aaabSAA$
$\to aaababAAAA$
$\to aaababaAAA$
$\to aaababaaAA$
$\to aaababaaaA$
$\to aaababaaaaa$

Left most derivation that we read in regular expression chapter. In this we expand left most non-terminal in our production and we keep expanding until we reach required string.

for eg. A is replaced by aS in 2nd line.

⊛. Conversion of PDA into its equivalent CFG:

Given a PDA $M=(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$; $F=\phi$, we can obtain equivalent CFG1, $G_1=(V, T, P \text{ and } S)$ which generates the same language as accepted by the PDA M.

The set of variables in the grammar consist of;
→ The special symbol S, which is start symbol.
→ All the symbols of the form $[pXq]$; $p, q \in Q$ and $X \in \Gamma$.
ie, $V=\{S\} \cup \{[pXq]\}$
The terminal in the grammar $T=\Sigma$.

The production of G_1 is as follows;
→ For all states $q \in Q$, $S \to [q_0, z_0, q]$ is a production of G_1.
→ For any states $q, r \in Q$, $X \in \Gamma$ and $a \in \Sigma \cup \{\varepsilon\}$,
If $\delta(q, a, X)=(p, \varepsilon)$ then $[pXq] \to a$
→ For any states $q, r \in Q$, $X \in \Gamma$ and $a \in \Sigma \cup \{\varepsilon\}$,
If $\delta(q, a, X)=(r, Y_1 Y_2 \ldots Y_k)$; where $Y_1, Y_2, \ldots Y_k \in \Gamma$ and $k \geq 0$.
Then for all lists of states $r_1, r_2, \ldots r_k$, G_1 has the production
$[pXq] \to a\,[rY_1r_1]\,[r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$.

This production says that one way to pop X and go from stack q to state r_k is to read "a" (which may be ε), then use some input to pop Y_1 off the stack while going from state r to r_1, then read some more input that pops Y_2 off the stack and goes from r_1 to r_2 and so on.....

Example1: Convert the PDA given below that recognizes a language $L = \{a^n b^n \mid n > 0\}$ defined as;

1. $\delta(q_0, a, z_0) = (q_1, az_0)$
2. $\delta(q_1, a, a) = (q_1, aa)$
3. $\delta(q_1, b, a) = (q_1, \varepsilon)$
4. $\delta(q_1, \varepsilon, z_0) = (q_1, \varepsilon)$. Also show the acceptance of aaabbb.

Solution: Let $G_1 = (V, T, P \text{ and } S)$ be the equivalent CFG for the given PDA where,

$V = \{S\} \cup \{[pXq] / p, q \in Q, X \in \Gamma\}$
S = is the start state
$T = \Sigma$

And P is defined by following production;

1. $S \rightarrow [q_0 z_0 q_0] \mid [q_0 z_0 q_1]$
   i.e, $S \rightarrow [q_0 z_0 r_2]$; for r_2 in $\{q_0, q_1\}$.
2. From the fact that $\delta(q_0, a, z_0)$ contains $(q_1, a z_0)$, we get production
   $[q_0 z_0 r_2] \rightarrow a [q_1 a r_1] [r_1 z_0 r_2]$; for r_1 in $\{q_0, q_1\}$.
3. From the fact that $\delta(q_1, a, a)$ contains $(q_1 a a)$, we get production
   $[q_1 a r_2] \rightarrow a [q_1 a r_1] [r_1 a r_2]$ for r_1 in $\{q_0, q_1\}$.
4. From the fact that $\delta(q_1, b, a)$ contains (q_1, ε), we get
   $[q_1 a q_1] \rightarrow b$
5. From the fact that $\delta(q_1, \varepsilon, z_0)$ contains (q_1, ε), we get
   $[q_1 z_0 q_1] \rightarrow \varepsilon$

Now the acceptance of aaabbb can be shown as;

$S \rightarrow [q_0 z_0 r_2]$
$\rightarrow a [q_1 a r_1] [r_1 z_0 r_2]$
$\rightarrow aa [q_1 a r_1] [r_1 a r_2] [r_1 z_0 r_2]$

$\rightarrow aaa[q_1 a r_1][r_1 a r_2][r_1 a r_2][r_1 z_0 r_2]$

$\rightarrow aaab[r_1 a r_2][r_1 a r_2][r_1 z_0 r_2]$

$\rightarrow aaabb[r_1 a r_2][r_1 z_0 r_2]$

$\rightarrow aaabbb[r_1 z_0 r_2]$

$\rightarrow aaabbb\ \varepsilon = aaabbb.$

Example2: Convert the PDA $P=(\{p,q\},[0,1],\{X,Z_0\},\delta,q,Z_0)$ to a CFG if δ is given by;

$\delta(q,1,Z_0)=(q,XZ_0)$

$\delta(q,1,X)=(q,XX)$

$\delta(q,0,X)=(q,X)$

$\delta(q,\varepsilon,Z_0)=(q,\varepsilon)$

$\delta(q,0,Z_0)=(q,Z_0).$

Solution:

The equivalent grammar can be written as;

$G_1=(V,\Sigma,P,S)$ where P consists of productions as;

$S\rightarrow[qZ_1r_2]$; r_2 in $\{p,q\}$.

$[qZ_0r_2]\rightarrow 1[qXr_1][r_1Z_0r_2]$; for r_1 in $\{p,q\}$.

$[qXr_2]\rightarrow 1[qXr_1][r_1Xr_2]$; for r_1 in $\{p,q\}$.

$[qXr_2]\rightarrow 0[pXr_2]$; for r_2 in $\{p,q\}$.

$[qXq]\rightarrow\varepsilon$

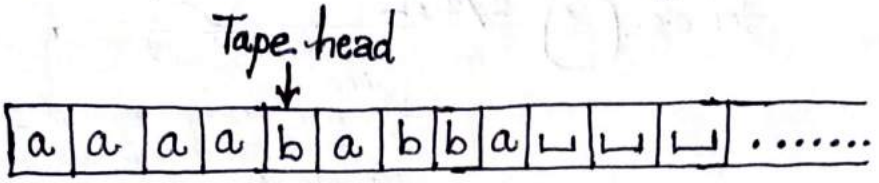
$[pXp]\rightarrow 1$

$[pZ_0r_2]\rightarrow 0[qZ_0r_2]$; for r_2 in $\{p,q\}$.

Unit-6

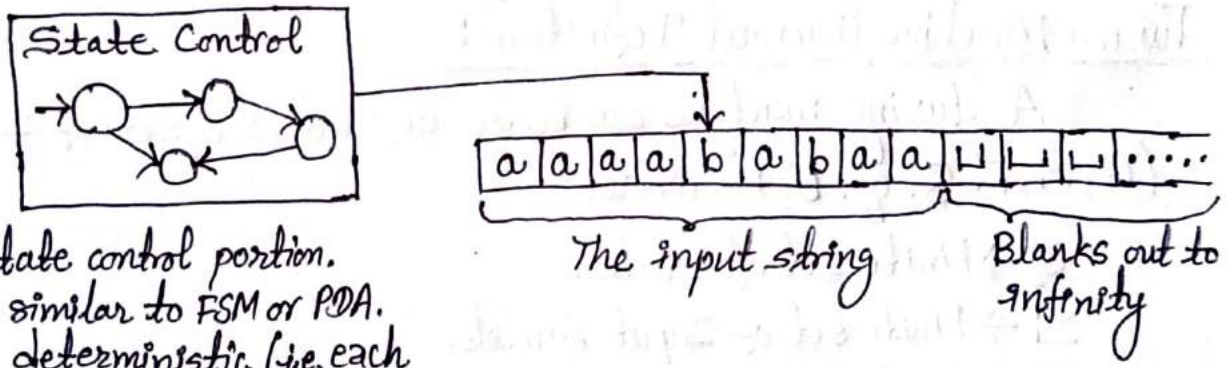
Turing Machines

Basics: In turing machine we have some sort of data structure known as tape. Tape looks like as the diagram shown below:-



As we see tape is a sequence of infinite symbols over which there is an arrow known as tape head, which is positioned over the symbol on which current control is present. Tape head can move either one step left or one step right at a single time of computation. Tape can contain any kind of alphabets like 0,1,a,b,x etc, and it also contains a special symbol i.e, the blank ␣ symbol. It is a special symbol used to fill the infinite tape. Read, Write, Move the tape head one step LEFT, Move the tape head one step RIGHT are the operations that can be performed on tape.

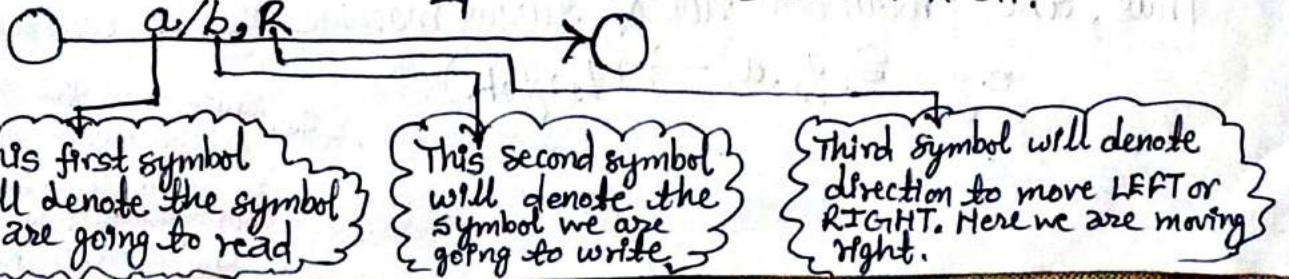
Turing Machine:



The state control portion.
It is similar to FSM or PDA.
It is deterministic (i.e, each state must be defined for all input symbols).

Rules of Operation:

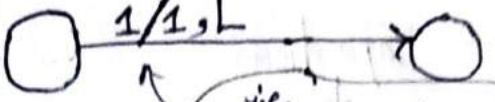
Rule1: At each step of the computation:
- → Read the current symbol
- → Write the same cell.
- → Move exactly one cell either LEFT or RIGT.



Note:

i) If we are at the left end of the tape and trying to move LEFT, then do not move, just stay at left end.

ii) If we don't want to write/update the cell, then just write the same symbol.

for e.g. 

i.e, We are reading symbol 1 on which we go to next state. but we do not want to update the cell, so we write same symbol 1 to cell again. L denotes move tape head to left.

Rule 2:

→ Initial State
→ Final states: (there are two final states)
  i) The ACCEPT state.
  ii) The REJECT state.
→ Computation can either;
  i) HALT and ACCEPT
  ii) HALT and REJECT
  iii) LOOP (the machine fails to HALT).

Turing Machine (Formal Definition):

A turing machine can be de defined as a set of 7 tuples $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where;

$Q \rightarrow$ Finite set of states.
$\Sigma \rightarrow$ Finite set of Input Symbols.
$\Gamma \rightarrow$ Finite set of Tape Symbols.
$\delta \rightarrow$ Transition function defined as;

$$Q \times \Sigma \rightarrow \Gamma \times (R/L) \times Q$$

i.e, when we are on a particular state, on getting a particular input symbol, we write something on the tape sequence and we move right or left on the tape and then we go to the next state.

$q_0 \rightarrow$ Initial state
$B \rightarrow$ Blank Symbol
$F \rightarrow$ Set of Final States (Accept State & Reject state).

Thus, the production rule of turing machine will be written as;

e.g. $\delta(q_0, a) \rightarrow (q_1, y, R)$.

i.e, Initially we are on q_0 state, on getting particular input a, we go to another state q_1 writing the symbol y on tape sequence and move right on tape head

⊛. Instantaneous Description for TM:

A configuration of TM is described by Instantaneous description (ID) of TM as like PDA. A string $x_1 x_2 \ldots\ldots x_{i-1} q x_i x_{i+1} \ldots\ldots x_n$ represents the I.D. of TM in which;

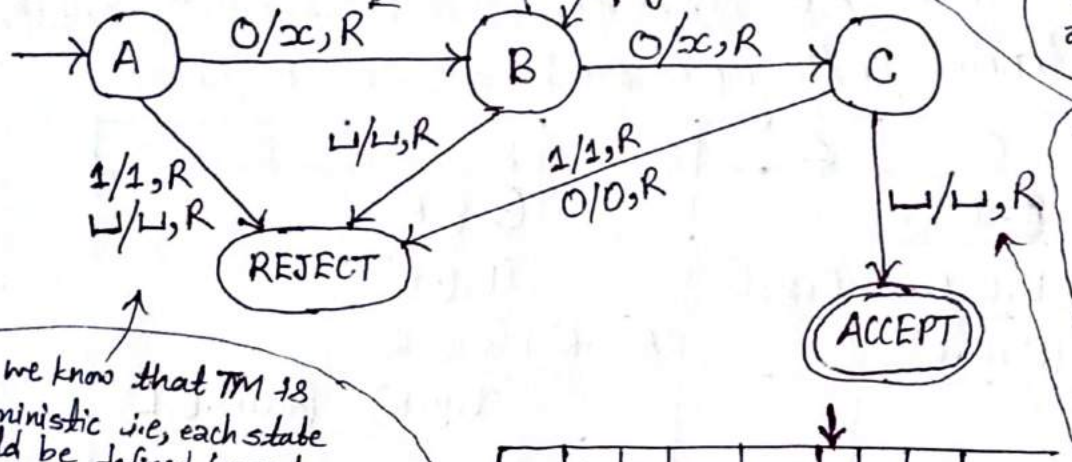
→ q is the state of TM.

→ the tape head scanning the i^{th} symbol from the left.

→ $x_1 x_2 \ldots\ldots x_n$ is the portion of tape between the leftmost and rightmost non-blank.

Example 1: Let we design a Turing Machine (TM) which recognizes the language $L = 01^*0$.

Here;



i.e, State A on getting input 0 goes to state B writing x on tape and moving arrow right similarly for others

i.e, B state on getting any number of 1's should remain in state B writing y in ~~each~~ tape each time and moving tape arrow right each time.

It is similar to NFA with additional tape symbol and move and seperate accept state

i.e, C state on getting blank symbol ␣ goes to accept state writing same blank symbol (becoz we do not need it to update) and moving tape arrow right.

as we know that TM is deterministic i.e, each state should be defined for each symbol i.e, for input symbol (0,1 here) and blank symbol (␣) as well. Sometimes we will see Reject state and transitions to it may be missing, then do not think it as wrong. It is right because, In any TM if any transition is missing then it goes to Reject state by default.

Example 2: Design a TM which recognizes the language $L = 0^N 1^N$. (i.e, the no. of 0's must be equal to no. of 1's & first 0's should occur then 1's).

Solution:

It can be designed with the concept of algorithm as below:

→ Change "0" to "x"

→ Move RIGHT until we get first "1"

If None: REJECT

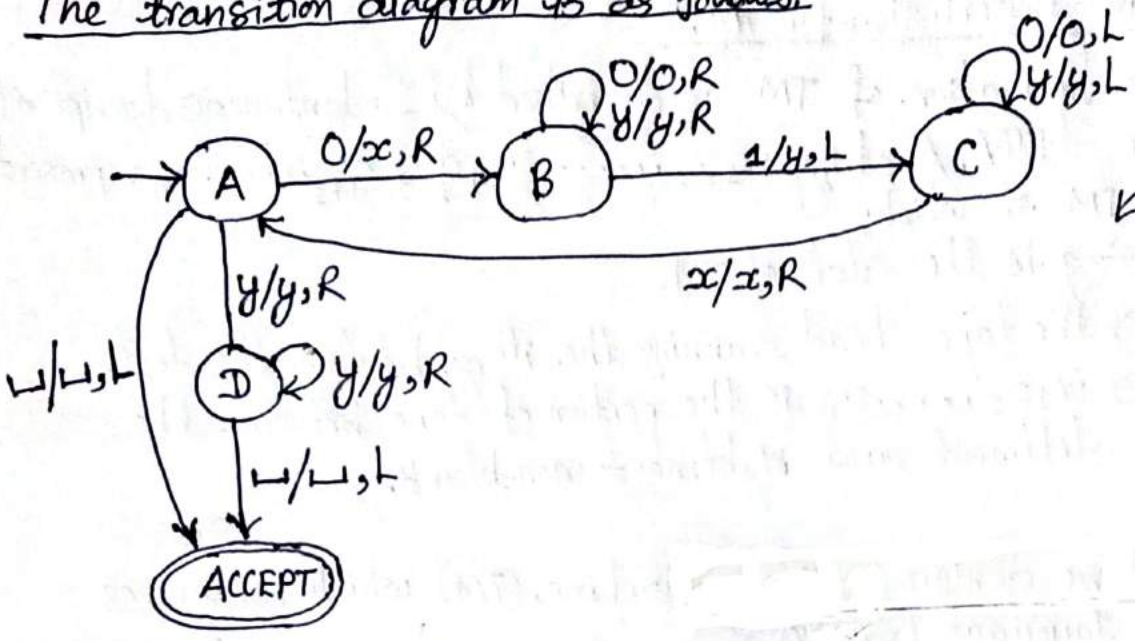
→ Change "1" to "y"

→ Repeat the above all steps until no more "0"s.

→ Make sure no more "1"s remain.

this algorithm will help to understand how the TM can be constructed. If difficult to understand refer video by neso academy TM example-2

The transition diagram is as follows:-



Here we see no reject state and some transitions missing. This means missing some transitions by default goes to Reject state so no need to draw

Now the TM for this can be represented as;

$TM = (\{A, B, C, D, ACCEPT\}, \{0,1\}, \{0,1,x,y,B\}, \delta, q_0, B, \{ACCEPT\})$.

Transition table for the moves can be described as follows:-

	0	1	x	y	B
A	(B,x,R)			(D,y,R)	
B	(B,0,R)	(C,y,L)		(B,y,R)	
C	(C,0,L)		(A,x,R)	(C,y,L)	
D				(D,y,R)	(ACCEPT,B,L)
ACCEPT					

Now, let we check the acceptance of string 0011 by the TM:

start state → A 0011
given string →

$\vdash$ xB011
$\vdash$ x0B11
$\vdash$ xC0y1
$\vdash$ xA0y1
$\vdash$ xxBy1
$\vdash$ xxyB1
$\vdash$ xxCyy
$\vdash$ xCxyy
$\vdash$ xxAyy
$\vdash$ xxyDy
$\vdash$ xxyyDB
$\vdash$ xxyyB ACCEPT B

Hence, Halt and accept.

e.g. मा accepting state को नाम ACCEPT राखेका छन् त्यसैले यो ACCEPT भनेको A,B,C जस्ते रहरा state को नाम हो। If accept state हामिले E राखेको भए माथि यो लाइन xxyyBEB हुन्थ्यो।

Let similarly we check the acceptance for string 0110:

A 0110 |- xB110
|- Cxy10
|- xAy10
|- xyD10

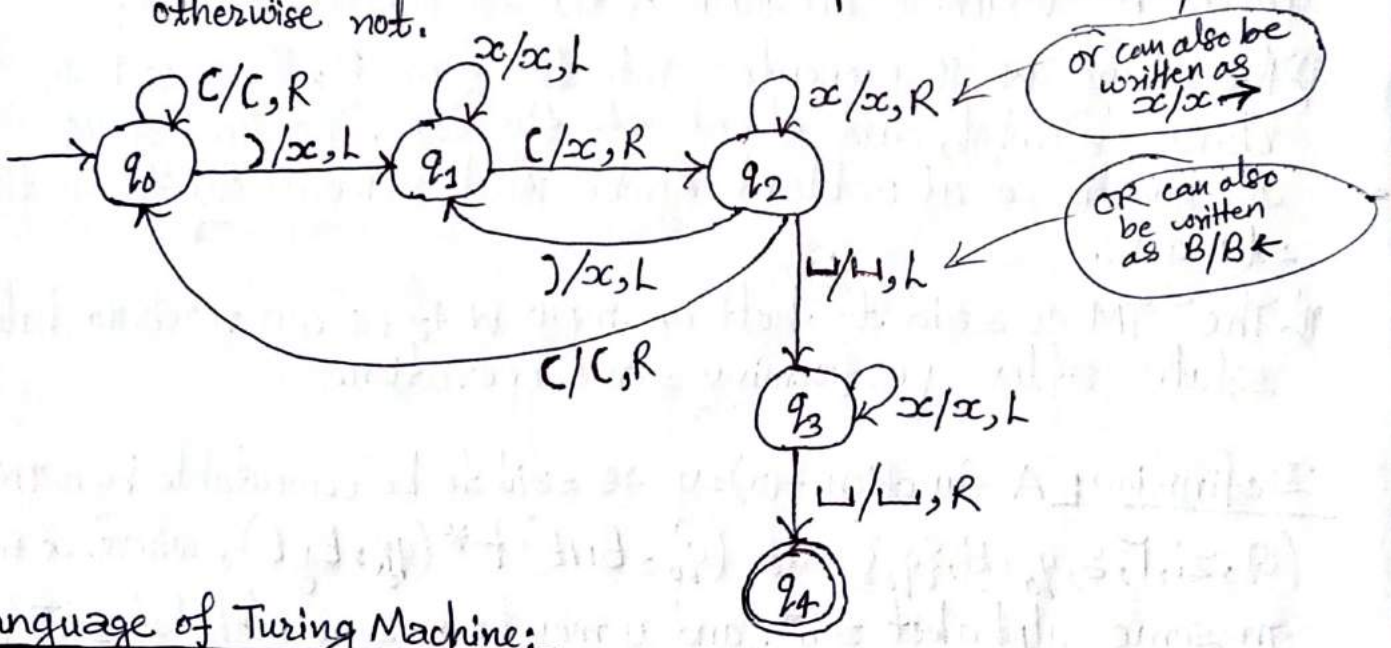
Halt and reject, since D has no move on symbol 1.

Example3: Design a TM that accepts well formed string of parenthesis.
i.e, L = {(0), 0(0), (0)0, ...}.

Solution:

Idea:

→ Find the first (, then replace it with x and find it's corresponding) and replace it with x.
→ Perform the above step until all the symbols are scanned.
→ Finally if the tape consists of only x or ␣ then accept otherwise not.



⊛Language of Turing Machine:

If $T = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is a turing machine and $w \in \Sigma^*$, then language accepted by T, $L(T) = \{w \mid w \in \Sigma^* \text{ and } q_0 w \vdash^* \alpha p \beta\}$ for some $p \in F$ and any type string α and β.

The set of languages that can be accepted using TM are called recursively enumerable languages.

Roles of TM:

→ As a language recognizor
→ As a computer of function.
→ As a enumerator of string of a language.

⊛. Turing Machine as a Language Recognizer:

A Turing Machine can be used as a language recognizer to accept strings of certain language.

For Example:- A TM accepting $\{0^n1^n \mid n \geq 1\}$ as we discussed earlier.

A Turing Machine T recognizes a string x (over Σ) if and only if when T starts in the initial position and x is written on the tape, T halts in a final state. A Turing Machine T does not recognize a string x, if T does not halt in a state that is not final.

⊛Turing Machine as a Computing a Function:-

A Turing Machine can be used to compute functions. For such TM, we adopt the following policy to input any string to the TM which is an input of the computation function.

i) The string w is presented into the form BwB, where B is blank symbol, and placed onto the tape. The head of TM is positioned at a blank symbol which immediately follows the string w.

ii) The TM is said to halt on input w if we can reach to halting state after performing some operation.

Definition: A function $f(x)=y$ is said to be computable by a TM $(Q, \Sigma, \Gamma, \delta, q_0, B, \{q_f\})$ if $(q_0, BwB) \vdash^* (q_f, ByB)$, where x may be in some alphabet Σ_1^*. and y may be in some alphabet Σ_2^* and $\Sigma_1 . \Sigma_2 \subseteq \Sigma$.

Example: Design a TM which compute the function $f(x)=x+1$ for each x that belongs to the set of natural numbers.

Solution:

Given function $f(x) = x+1$.

Here, we represent input x on the tape by a number of 1's on the tape. For example, for $x=1$, input will be B1B.

for $x=2$, input will be B11B.

for $x=3$, input will be B111B and so on.

Similarly output can be seen by the number of 1s on the tape when machine halts.

Let TM = $(Q, \Sigma, \Gamma, \delta, q_0, B, \{q_f\})$; where $Q = \{q_0, q_f\}$, $\Gamma = \{1, B\}$ and halt state = $\{q_0\}$. Then, δ can be simulated as;

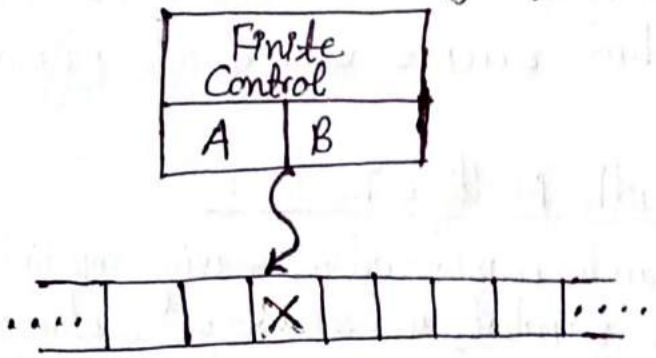
Q	B	1
q_0	$(q_0, 1, S)$	$(q_f, 1, R)$
q_f	–	–

So, let the input be x=4. So, input tape at initial step consists of B1111B.

Then $(q_0, B1111B) \vdash (q_0, B11111) \vdash (q_f, B11111B)$. Which means output 5 is accepted.

⊛ Turing Machine with Storage in the state:

In turing machine, any state represents the position in the computation. But a state can also be used to hold a finite amount of data. The finite control of machine consists of a state q and some data portion. In this case a state is considered as a tuple – (state, data). Following figure illustrates the model.



δ is defined by:

$\delta([q, A], X) = ([q_1, X], Y, R)$ means q is the state and data portion of q is A. The symbol scanned on the tape is copied into second component of the state and moves right entering state q_1 and replacing tape symbol by Y.

Example:- This model of TM can be used to recognize languages like $01^* + 10^*$, where first symbol (0 or 1) that it sees, and checks that it does not appear else where in the input. For this, it remembers the first symbol in finite control.

⊛ Turing Machine as Enumerator of Strings of Language:

Consider a multi-tape turing machine TM that uses a tape as an output tape, in which a symbol once written can never be changed, and those whose tape head never moves left. Suppose also that on the output tape, TM writes strings over some alphabet Σ separated by a marker symbol #.

We can define L(TM) to be set of w in Σ^* such that w is eventually printed between a pair of #'s on the output tape. Thus the language of this type of Turing machine is called Turing Enumerable languages.

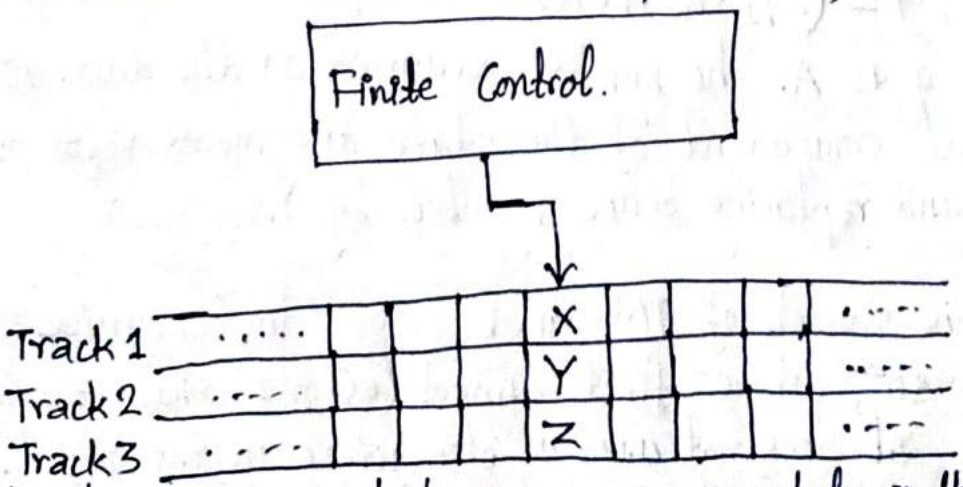
⊛. Turing Machine as Subroutine:

Subroutines are collection of interacting components. A TM subroutine is set of states that perform some useful process. This set of states includes a start state and other states that serves as the return to pass control to other set of states called subroutine. The "call" of a subroutine occurs whenever there is a transition to its initial state.

Example: A TM that accepts even & odd palindrome.

⊛ Turing Machine with Multiple Tracks:

The tape of TM can be considered as having multiple tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each track. Following figure illustrates the TM with multiple tracks;



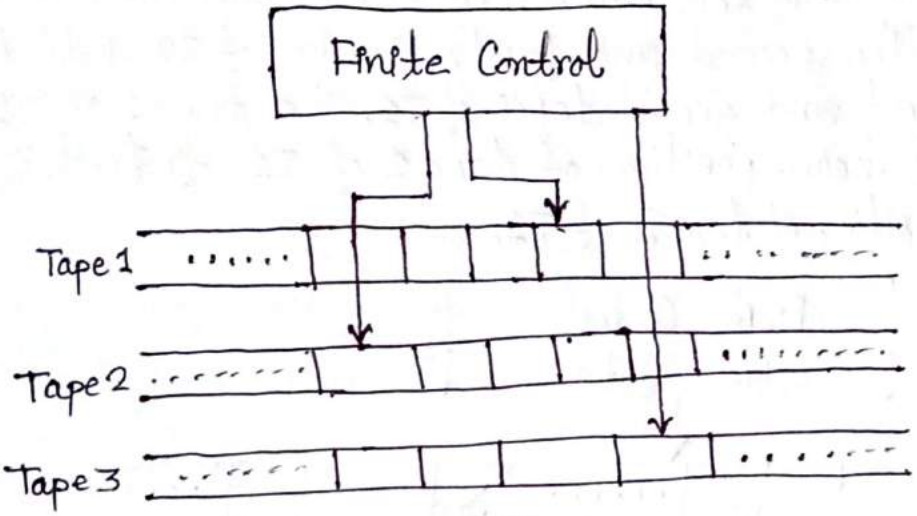
The tape head moves up and down scanning symbols in the tape at one position.

The tape alphabet Γ is a set consisting tuples like,

$$\Gamma = \{(X, Y, Z), \ldots\}.$$

⊛. Turing Machine with Multiple Tapes:

A TM can be multi-tape in which there is more ~~one~~ than one tape. Adding extra tape adds no power to the computational model, but only the ability to accept the language is increased. A multi-tape TM consists of finite control and finite number of tapes. Each tape is divided into cells and each cell can hold any symbol of finite tape alphabets. The set of tape symbols include a blank and the input symbols.



In the multi-tape TM, initially;

→ The Input (finite sequence of input symbols) w is placed on the first tape.
→ All other cells of the tapes hold blanks.
→ TM is in initial state q_0.
→ The head of the first tape is at the left end of the input.
→ All other tape heads are at some arbitrary cell.

A move of multi-tape TM depends on the state and the symbol scanned by each of the tape head. In one move, the multi-tape TM does following;

→ The control enters in a new state, which may also be same previous state.
→ On each step, a new symbol scanned is written on the cell, these symbols may be same as the symbols previously there.
→ Each of the tape head make a move either left or right or remains stationary. Different head may move different direction independently.

⊛ Equivalence of One-tape and Multi-tape TM's:

OR simulating one tape TM with multi-tape TM

Theorem: Every language accepted by a multitape TM is recursively enumerable.

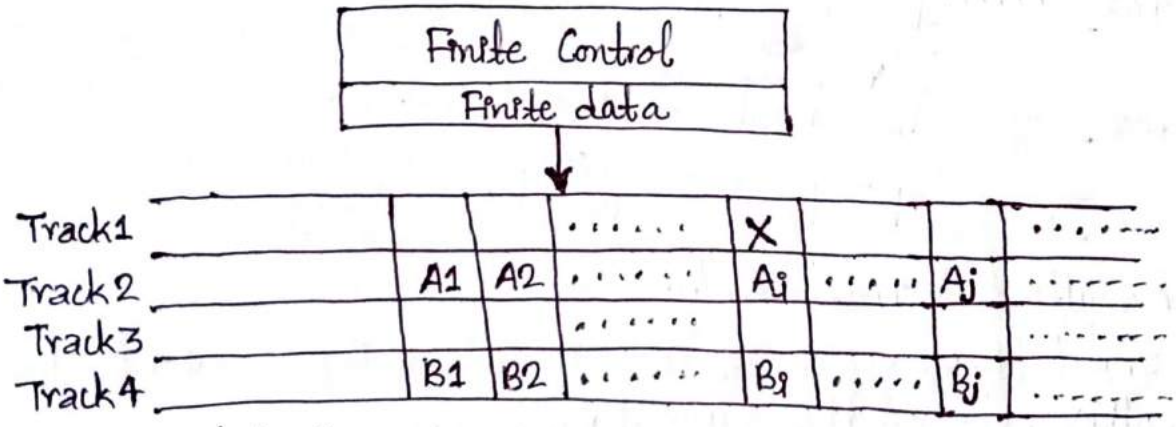
OR Any languages that are accepted by a multi-tape TM are also accepted by one tape Turing machine.

Proof:

T1 denotes a name of TM with n-tape

Let L is a language accepted by a n-tape TM, T1. Now, we have to simulate T1 with a one-tape TM, T2 considering there are 2n tracks in the tape of T2.

For simplicity, let us assume n=2, then for n>2 is the generalization of this case. Then total number of tracks in T2 will be 4. The second and fourth tracks of T2 hold the contents of first and second tapes of T1. The track1 in T2 holds ~~the contents of~~ head position of tape 1 of T1 & track 3 in T2 holds head position of tape2 of T1.



Now, to simulate the move of T1,

→ T2's head must visit the n-head markers so that it must remember how many head markers are to its left at all times. That count is stored as a component of T2's finite control.

→ After visiting and storing scanned symbol, T2 knows what tape symbols are being scanned by each of T1's head.

→ T2 also knows the state of T1, which it stores in T2's own finite control. Thus T2 knows what move T1 will make.

→ T2 now revisits each of the head markers on its tape, changes the symbol in track representing corresponding tapes T1 and moves the head marker left or right if necessary.

Finally, T2 changes the state of T1 as recorded in its own finite control. Hence T2 has simulated one move of T1. We select T2's accepting states, all those states that record T1's state as one of the accepting state of T1. Hence, whatever T1 accepts T2 also accepts.

⊛. Non-Deterministic Turing Machines:-

A non-deterministic Turing Machine (NTM) is exactly the same as ordinarly TM, except the value of transition function δ. In TM any state on getting any particular input can go to only one particular next state writing a particular symbol onto tape. But in NTM any state on getting any particular input can go to any state among multiple next states writing corresponding symbol onto tape, based on path selected.

In NTM, the values of the δ are subsets, rather than a single element of set, $Q \times \Gamma \times \{R, L, S\}$. Here the transition function δ for each state q and tape symbol x, is $\delta(q,x)$ which is a set of triples as;

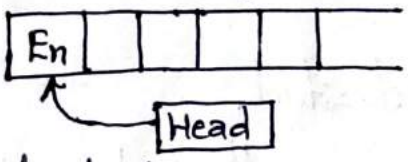
$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots\ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer.

The NTM can choose any of the triples to be the next move at each step.

⊛ Restricted Turing Machines:-

1. Semi-Infinite Tape: A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.



It is a two track tape;

i) Upper track → It represents the cells to the right of the initial head position.

ii) lower track → It represents the cells to the left of the initial head position in ~~reser~~ reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.

The machine starts from the initial state q_0 and head scans from the left end. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head one tape cell left or right. A transition function determines the actions to be taken.

It has two special states called accept state and reject state. If at any point of time if it enters into the accept state then the input is accepted. and if it enters into the reject state then the input is rejected. In some cases it continues to run infinitely without being accepted or rejected for some certain input symbols.

2. Muti Stack Machines:

Multi Stack Machines are the generalizations of the PDA's. TM's can accept languages that are not accepted by any PDA with one stack, but PDA with two stacks can accept any language that a TM can accept.

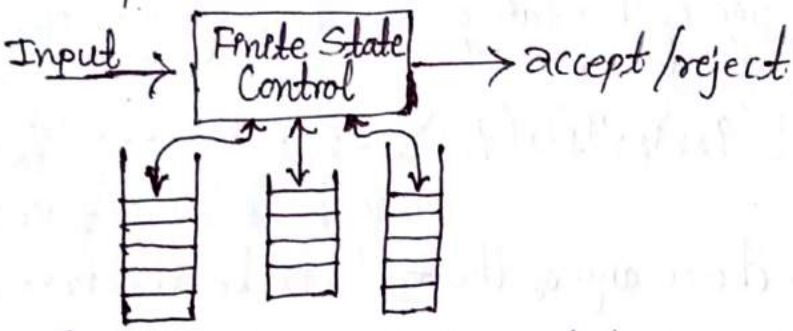


fig. A machine with three stacks.

It obtains its input source rather than having the input placed in a tape. A move of a multistack machine is based on;

→ The state of finite control.
→ The input symbol read.
→ The top stack symbol on each stack.

In one move a multistack machine can change to a new state and can replace the top symbol of each stack with a string of zero or more stack symbols.

3. Counter Machines:

Counter Machine are offline TM's whose storage tapes are semi-infinite and whose tape alphabets contain only two symbols Z and B. The symbol Z serves as a bottom of stack marker appearing initially on the cell scanned by the tape head and may never appear on any other cell. B is the Blank Symbol.

A stored number can be incremented or decremented by moving the tape head left or right. We can test whether a number is zero, but cannot directly test whether two numbers are equal. Instantaneous description of a counter machine can be described by the state, the input tape contents, the position of input head, and the distance of the storage heads from the symbol Z.

⊛. Church. Turing Thesis:

It is a mathematically un-provable hypothesis about the computability. This hypothesis simply states that - "Any algorithmic procedure that can be carried out at all can be carried out by a TM." This statement was first formulated by Alonzo Church a mathematician and a logician in 1930s. According to Church, "No computational procedure will be considered an algorithm unless it can be represented by a Turing Machine".

After adopting Church - Turing Thesis, we are giving precise meaning of the term: An algorithm is a procedure that can be executed on a Turing Machine. Another use of Church-Thesis is that, when we want to describe a solution to a problem, we will often satisfy with a verbal description of the algorithm, translating it into detailed Turing Machine Implementations.

⊛. Universal Turing Machine:

A machine that can simulate the behaviour of an arbitrary TM is called a Universal Turing Machine. Thus, we can describe a Universal Turing Machine T_u as a TM, that on input $\langle M, w \rangle$; where M is a TM and w is a string of input alphabets, simulates the computation of M on input w. and satisfies two properties;

i) Tu accepts $\langle M, w \rangle$ iff M accepts w.
ii) Tu rejects $\langle M, w \rangle$ iff M rejects w.

⊛. Encoding of Turing Machine:-

For the representation of any arbitrary TM T_1, and an input string w over as arbitrary alphabet, as binary strings $e(T_1)$, $e(w)$ over some fixed alphabet, a notational system should be formulated. Encoding the TM T_1, and input w into $e(T_1)$ and $e(w)$, it must not destroy any information. For encoding of TM, we use alphabet $\{0,1\}$, although the TM may have a much larger alphabet.

To represent a TM $T_1 = (Q_1, \{0,1\}, \Gamma, \delta, q_1, B, F)$ as binary string, we first assign integers to the states, tape symbols and directions. We assume two fixed infinite sets $Q = \{q_1, q_2, q_3, \ldots\}$ and $S = \{a_1, a_2, a_3, \ldots\}$ so that Q_1 is subset of Q and Γ is subset of S. Now we have a subscript attached to every possible state and tape symbols, we can represent a state or a symbol by a string of 0's of the appropriate length. Here, 1's are used as seperators.

Once we have established an integer to represent each state, symbol and direction, we can encode the transition function δ. Let one transition rule is:

$$\delta(q_i, a_j) = (q_k, a_l, D_m) \text{ for some integer } i, j, k, l, m.$$

Then we shall code this rule by the string $s(q_i)1s(a_j)1s(q_k)1s(a_j)1s(D_m)$, say this as m_1, where s is the encoding function defined below. A code for entire TM T1 consist of all the codes for the transitions, in some order, seperated by pairs of 1's:

$$m_1 11 m_2 11 \ldots m_n.$$

Now the code for TM and input string w will be formed by separating them by three consecutive ones i.e, 111.

The Encoding function s:-

First, associate a string of 0's, to each state, to each of the three directions, and to each tape symbol. Let the function s is defined as;

$S(B) = 0$

$S(a_i) = 0^{i+1}$ for each $a_i \in S$.

$S(q_i) = 0^{i+2}$ for each $q_i \in Q$.

$S(S) = 0$

$S(L) = 00$

$S(R) = 000$.

Example: Consider an example where, TM T is defined as;

$T = (\{q_1, q_2, q_3\}, \{a, b\}, \{a, b, B\}, \delta, q_1, B, F)$, where δ is defined as;

$\delta(q_1, b) = (q_3, a, R) \rightarrow m_1$

$\delta(q_3, a) = (q_1, b, R) \rightarrow m_2$

$\delta(q_3, b) = (q_2, a, R) \rightarrow m_3$

$\delta(q_3, B) = (q_3, b, L) \rightarrow m_4$.

Now, using the encoding function s defined above, as the rule, we have

$S(q_1) = 000$

$S(q_2) = 0000$

$S(q_3) = 00000$

$S(a_1) = 00$

$S(a_2) = 000$

$S(B) = 0$

$S(R) = 000$

$S(L) = 00$

$S(S) = 0$

Considering $a_1 = a$ & $a_2 = b$.

Now, encoding for rules;

$e(m_1) = S(q_1)1S(b)1S(q_3)1S(a)1S(R)$

$= 0001000100000100 1000$

$e(m_2) = S(q_3)1S(a)1S(q_1)1S(b)1S(R)$

$= 000001001000100 01000$

$e(m_3) = S(q_3)1S(b)1S(q_2)1S(a)1S(R)$

$= 000001000100001001000$

$e(m_4) = S(q_3)1S(B)1S(q_3)1S(b)1S(L)$

$= 000001010000010001 00$

Now, the code for TM T is $e(m_1)11\,e(m_2)11\,e(m_3)11\,e(m_4)$.

i.e, 000100010000010010001100000100100010000100100011000001010000010001 00.

For this machine T, for any input w, where w=ab, the code will be e(T)111e(w), where e(w)= s(a)1s(b)
= 001000.

⊛. Turing Machines and Computers:

The Turing Machines and Computers both can accept the same, recursively enumerable languages. Since the notation of "a common computer" is not well defined mathematically, the arguments in this are necessarily informal. Turing Machines and Computers can be divided into two parts as follows:

i) A computer can simulate a Turing Machine.

ii) A Turing machine can simulate a computer, and can do so in an amount of time that is at most some polynomial in the number of steps taken by the computer.

⊛. Difference between TM and Other Automata (FSA and PDA):

The most significant difference between the TM and the simpler machine (FSA and PDA) is that; in a TM, processing a string is no longer restricted to a single left to right pass through input. The tape head can move in both directions and erase or modify any symbol it encounters. The machine can examine part of the input, modify it, take time execute some computation in a different area of the tape, return to re-examine the input, repeat any of these actions and perhaps stop the processing before it has looked at all input.

⊛ Enumerating the Binary Strings:-

We shall need to assign integers to all the binary strings so that each string corresponds to one integer, and each integer corresponds to one string. If w is a binary string, then treat 1w as a binary integer i. Then we shall call w the ith string. That is ε is the first string, 0 is the second, 1 is the third, 00 the fourth, 01 the fifth and so on. Equivalently, strings are ordered by length, and strings of equal length are ordered lexicographically. Hereafter, we shall refer to the ith string as w_i.

Unit-7
Undecidability and Intractability:-

⊛ Computational Complexity:-

The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much resource machine of that type require for the solution of that problem. Complexity Measure is a means of measuring the resource used during a computation. In case of Turing Machines, during any computation, various resources will be used, such as space and time. When estimating these resources, we are always interested in growth rates than absolute values.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The time and storage are the main complexity measures to be used. Other complexity measures are also used, such as the amount of communication, the number of gates in a circuit and the number of processors.

⊛ Time and Space Complexity of a Turing Machine:

When a turing machine answers a specific instance of a decision problem we can measure time as number of moves and the space as number of tape squares, required by the computation. The most obvious measure of the size of any instance is the length of input string. The worst case is considered as the maximum time or space that might be required by any string of that length.

The time and space complexity of a TM can be defined as;

Let T be a TM. The time complexity of T is the function T_t defined on the natural numbers as; for $n \in N$, $T_t(n)$ is the maximum number of moves T can make on any input string of length n. If there is an input string x such that for $|x|=n$, T loops forever on input T, $T_t(n)$ is undefined.

The space complexity of T is the function S_t defined as $S_t(n)$ is the maximum number of the tape squares used by T for any input string of length n. If T is multi-tape TM, number of tape squares means maximum of the number of individual tapes. If for some input of length n, it causes T to loop forever, $S_t(n)$ is undefined.

⊛. Intractability:-

An algorithm for which the complexity measures $S_t(n)$ increases with n, no more rapidly than a polynomial in n is said to be polynomially bounded; if one in which it grows exponentially is said to be exponentially bounded. Intractability is a technique for solving problems not to be solvable in polynomial time. The problems that can be solved within reasonable time and space constraints are called tractable. The problems that cannot be solved in polynomial time but requires exponential time algorithm are called intractable or hard problems.

To introduce intractability theory, the class P and class NP of problems solvable in polynomial time by deterministic and non-deterministic TM's are essential. When the space and time required for implementing the steps of particular algorithm are reasonable, we can say that the problem is tractable. Problems are intractable if the time required for any of the algorithm is at least $f(n)$, where f is an exponential function of n.

⊛. Complexity Classes:-

In computational complexity theory, a complexity class is a set of problems of related resource-based complexity. A typical complexity class has a definition of the form: "The set of problems that can be solved by an abstract machine M using $O(f(n))$ of resource R, where n is the size of the input." The simpler complexity classes are defined by the following factors;

The type of computational problem: The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, optimization problems etc.

The model of computation: The most common model of computation is the deterministic Turing machine, but many complexity classes are based on non-deterministic Turing machines, Boolean circuits etc.

The resources that are being bounded and the bounds: These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth" etc.

Class P: The class P is the set of problems that can be solved by deterministic TM in polynomial time. A language L is in class P if there is some polynomial time complexity T(n) such that L=L(M), for some deterministic Turing Machine M of time complexity T(n).

Class NP: The class NP is the set of problems that can be solved by non-deterministic TM in polynomial time. A language L is in the class NP if there is a non-deterministic TM, M, and a polynomial time complexity T(n), such that L=L(M) and when M is given an input of length n, there are no sequences of more than T(n) moves of M.

NP-Complete: The complexity class NP-complete (NP-C or NPC) is a class of problems having two properties,

→ It is the set of NP (nondeterministic polynomial time) problems: Any given solution to the problem can be verified quickly (in polynomial time).

→ It is also in the set of NP-hard problems: Any NP problem can be converted into this one by a transformation of the inputs in polynomial time.

Properties of NP-Complete problems:-

→ No polynomial time algorithms has been found for any of them.

→ It is not established that polynomial time algorithm for these problems do not exist.

→ If polynomial time algorithm is found for any of them, there will be polynomial time algorithm for all of them.

→ If it can be proved that no polynomial time algorithm exists for any of them, then it will not exist for every one of them.

⊛. Problems and its types:

i) **Abstract Problems:** Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions. For e.g. Minimum spanning tree of a graph G1 can be viewed as a pair of the given graph G1 and MST graph T.

ii) **Decision Problems:** Decision problem D is a problem that has an answer as either "true", "yes", "1" or "false", "no", "0". For e.g. if we have the abstract shortest path with instances of the problem and the solution set as {0,1}, then we can transform that abstract problem by reformulating the problem as "Is there a path from u to v with at most k edges." In this situation the answer is either yes or no.

iii) **Optimization Problems:** We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with the best value. This kind of problem is called optmization problem. For e.g. given the graph G1, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not directly deal with optimizations problems; however we can translate the optimization problem to the decision problem.

iv) **Function Problems:** A function problem is a computational problem where a single output is expected for every input, but the output is more complex than that of a decision problem, which isn't just Yes or No. For e.g, The travelling salesman problem, which asks for the route taken by the salesman, and the Inter Factorization problem, which asks for the list of factors.

⊛. Reducibility:-

Reducibility is a way of converging one problem into another in such a way that, a solution to the second problem can be used to solve the first one.

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. There are many different type of reductions based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. For complexity classes larger than P, polynomial-time reductions are commonly used.

⊛ Circuit Satisfiability:-

The circuit satisfiability problem (also known as CircuitSAT, CSAT etc) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true. In other words, it asks whether the inputs to a given Boolean circuit can be consistently set to 1 or 0 such that the circuit outputs 1. If that is the case, the circuit is called satisfiable. Otherwise, the circuit is called unsatisfiable.

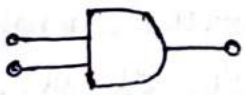


fig: satisfiable circuit.

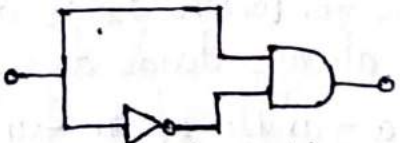


Fig: unsatisfiable circuit.

Cook's Theorem:

Lemma: SAT is NP-hard

Proof: Take a problem $V \in NP$, let A be the algorithm that verifies V in polynomial time (this must be true since $V \in NP$). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to the bits of the inputs x and y of A and which outputs 1 precisely when A(x,y) returns yes:

For any instance x of V let A_x be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of A_x from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

Theorem: SAT is NP-complete.

Proof: To show that SAT is NP-complete we have to show two properties as given by the definition of NP-complete problems. The first property i.e, SAT is NP & the second property i.e, SAT is NP-hard.

Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

This claims that SAT is NP. Now it is sufficient to show the second property holds for SAT. The proof for the second property i.e, SAT is NP-hard is from above lemma. This completes the proof.

⑧. Undecidability:-

In computational theory, an undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct "yes" or "no" answer. It consists of a family of instances for which a particular yes/no answer is required, such that there is no computer program that, for any given problem instance as input, terminates and outputs the required answer after a finite number of steps. More formally, an undecidable problem is a problem whose language is not a recursive set or computable or decidable.

Undecidable Problems:

1. Posts Correspondance Problem (PCP):

The input of the problem consists of two finite lists $U = \{u_1, u_2, \ldots, u_n\}$ and $V = \{v_1, v_2, \ldots, v_n\}$ of words over the alphabet Σ having at least two symbols. A solution to this problem is a sequence of indices i_k;

$$1 <= k <= n, \text{ for all } k, \text{ such that}$$

$$u_{i1} u_{i2} \ldots\ldots u_{ik} = v_{i1} v_{i2} \ldots\ldots v_{ik}$$

We say $i_1, i_2, \ldots\ldots, i_k$ is a solution to this instance of PCP.
Here, the decision problem is to decide whether such a solution exists or not.

Example: Consider the following two lists:

U		
u_1	u_2	u_3
a	ab	bba

V		
v_1	v_2	v_3
baa	aa	bb

A solution to this problem would be the sequence (3,2,3,1), because

$u_3 u_2 u_3 u_1 = bba + ab + bba + a = bbaabbbaa$

$v_3 v_2 v_3 v_1 = bb + aa + bb + baa = bbaabbbaa.$

Furthermore, since (3,2,3,1) is a solution, so all of it's "repetitions", such as (3,2,3,1,3,2,3,1) etc. are infinitely many solutions of this repetitive kind.
However, if the two lists had consisted of only u_2, u_3 and v_2, v_3 then, there would have been no solution.

2. Halting problem and its proof:

"Given a Turing Machine M and an input w, do M halts on w?"

Algorithms may contain loops which may be infinite or finite in length. The amount of work done in an algorithm usually depends on data input. Algorithms may consist of various number of loops nested or in sequence. Thus, the halting problem asks the question; "Given a program and an input to the program, determine if the program will eventually stop when it is given that input." The question is simply whether the given program will ever halt on a particular input.

Trial Solution: Just run the program with the given input. If the program stops we know that the program halts. But if the program does not stop in reasonable amount of time, we cannot conclude that it won't stop. May be we did not wait long enough!

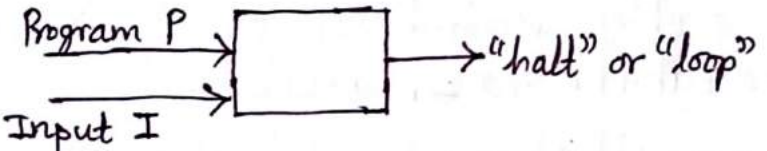
The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

Sketch of a proof that the Halting Problem is undecidable:

Suppose we have a solution to the halting problem called H. Now H takes two inputs:

i) A program P.
ii) An input I for the program P.

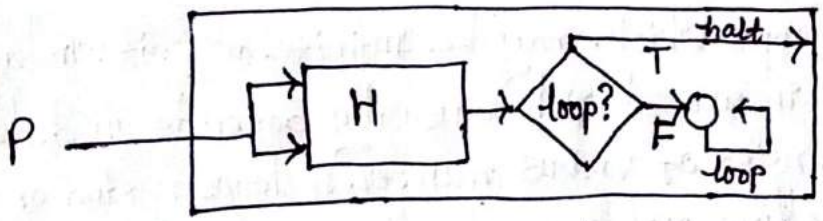
H generates an output "halt" if H determines that P stops on input I or it outputs "loop" otherwise.



We can treat the program as data and therefore a program can be thought of as input. So, now H can be revised to take P as both inputs (the program and its input) and H should be able to determine if P will halt on P as its input.

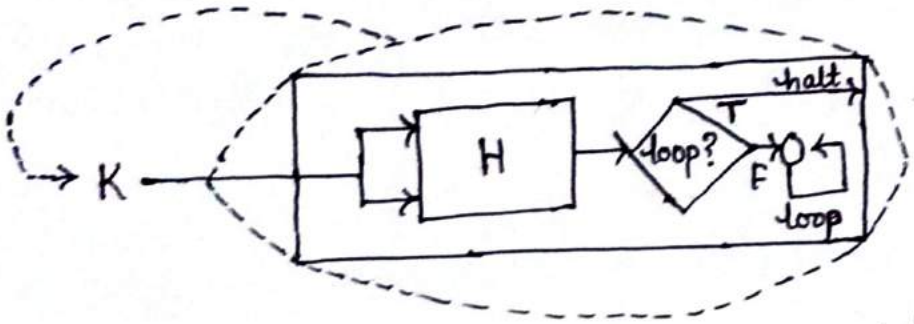
Let us construct a new, simple algorithm K that takes H's output as its input and does the following:

i) If H outputs "loop" then K halts.
ii) Otherwise H's output of "halt" causes K to loop forever.




```
function K(){ if(H()=="loop"){
                    return;
              }
              else{
                    while(true); // loop forever
              }
}
```

Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it). If H says that K loops then K will halt.
In either case H gives the wrong answer for K. Thus, H cannot work in all cases. We've shown that it is possible to construct an input that causes any solution H to fail. Hence Proved!

⑧. Undecidable Problems about Turing Machines:

A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.

Examples:-

i) Ambiguity of context-free languages: Given a context-free language, there is no Turing Machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.

ii) Equivalence of two context-free languages: Given two context-free languages, there is no Turing Machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.

iii) Completeness of CFG: Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet is undecidable.

iv) Regularity of CFL:- Given a CFL, determining whether this language is regular is undecidable.