

Automata Theory :-

Problems are a set of problem instances. A problem instance is a specific instance of a problem. We predominantly deal with yes/no problems which have 2 sets, the accept/yes and the reject/no sets.

A problem is computable only if all the instances are computable/solvable by the model.

A computational model solves a problem P if (and only if?)

i) If inputs belonging to the Yes instance of P , then the device outputs YES / ACCEPT

ii) If inputs belonging to the No instance of P , then the device outputs NO / REJECT

If i) & ii) hold, P is computable.

Alphabet:- Any finite, non-empty set of symbols

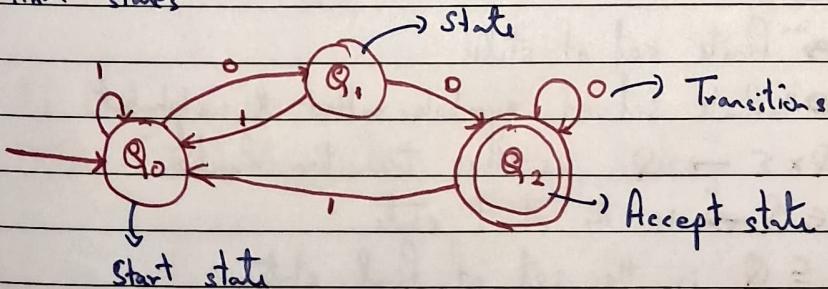
Strings / Words:- Finite sequence of symbols from an alphabet

Language:- Set of words/strings from the current alphabet

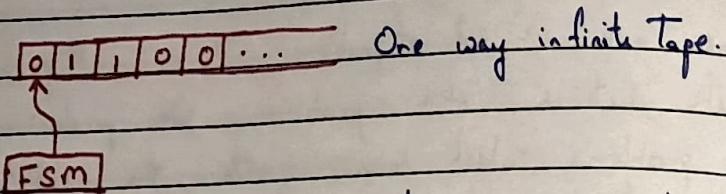
Generally, ϵ denotes the empty string

Deterministic Finite State Automata Model [DFA]

Has a single start state, unique transitions and zero or more final states



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



After the input is read entirely, and the machine is in the final/accept state, then it returns YES (ACCEPT); else it outputs NO (REJECT).

Run:- The total computation done during the reading and executing of the entire input string. The string is accepted or rejected after the complete run.

Definition:- Let the DFA be M . Then the language that M accepts is :-

$$L(M) = \{\omega \mid \omega \text{ results in an accepting run}\}$$

$\{\omega \mid \text{set of all strings } \omega \text{ such that } M(\omega) \text{ accepts}\}$

This is also known as the language of the automaton.

Definition:- For any language L , we say M solves/decides L if

- i) $\forall \omega \in L, M(\omega) \text{ accepts}$
- ii) $\forall \omega \notin L, M(\omega) \text{ rejects}$

Definition:- A DFA M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of states

Σ is a finite set of symbols called the alphabet

$\delta: Q \times \Sigma \rightarrow Q$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of final states

11
110
001
111

classmate

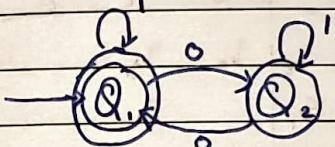
Date _____
Page _____

Characteristics of a DFA:-

- (i) Single Start State
- (ii) Unique Transitions
- (iii) Zero or more final states

Language of a DFA is the set of all binary strings ^{that result in} ending in 00 an accept state at the end of the computation.

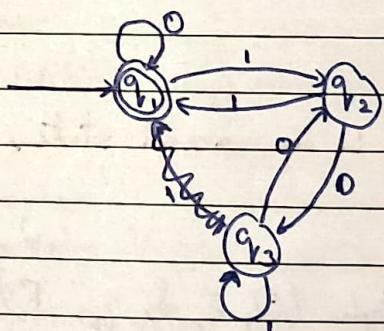
Example:- i) $\Sigma = \{0, 1\}$, $L(m) = \{\omega \mid \omega \text{ has an even no of } 0s\}$



ii) $\Sigma = \{0, 1\}$, $L(m) = \{\omega \mid \omega \text{ is divisible by } 3\}$

Any input string would leave remainders 0, 1, 2
DFA will have 3 states

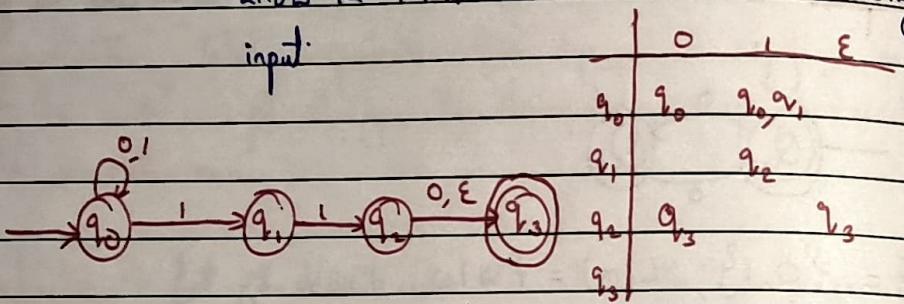
If ω is divisible by 3, then after reading a 0 it should stay in the final state, and so on.



If L is solvable by a DFA, then $\neg L$ is also solvable by a DFA. [Just toggle the accepting states]

Non-deterministic finite state Automata (NFA)

- Characteristics:-**
- i Single start state
 - ii Zero or more final states
 - iii Multiple transitions are possible on the same input state
 - iv Some transitions might be missing.
 - v ϵ -transitions or empty transitions exist which allow the NFA to transition without reading any input.



A given input can have multiple runs.

Take 10110 as the input string. It could accept, reject or crash in this case.

Crashes occur when the machine has no transition for an input

For the 10110 example, a possible crash run is -

$q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{\epsilon} q_3 \xrightarrow{0} \text{crash}$

Crashes are rejecting runs.

If \exists at least one run results in an accept state, the input is accepted, else rejects

Definition :- An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

$Q \Rightarrow$ finite set of states

$\Sigma \Rightarrow$ finite set of alphabets (contains ϵ)

$\delta \Rightarrow Q \times \Sigma \rightarrow P(Q)$, is the transition fn

$q_0 \in Q \Rightarrow$ start state

$F \subseteq Q \Rightarrow$ set of accepting states

Intuitively NFAs are more powerful than DFAs because of non-determinism.

$\rightarrow L_2$

The languages accepted by DFAs are a subset of the languages accepted by NFAs.

\downarrow

L_1

$L_2 \subseteq L_1$

$L_1 \subseteq L_2$ (Proof upcoming)

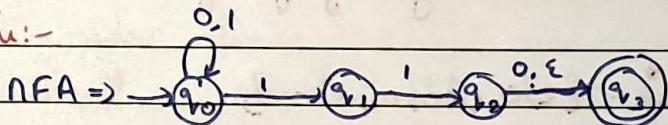
Hence $L_1 = L_2 \rightarrow$ Both are equally powerful.

We prove this by showing that we can construct a DFA to accept the same languages as ~~NFA~~ ^{an}. These DFAs are called Remembering DFAs,

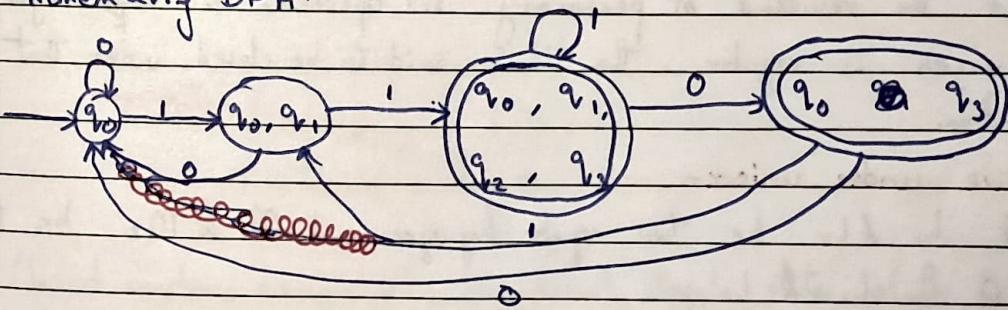
Steps :-

- i) Let R be a remembering DFA
- ii) R on an input enters a state labelled by all possible states that NFA can enter on that input.
- iii) This trims away the non-determinism of the NFA without losing its language
- iv) Also, if NFA has k states, R has at most 2^k states [Power state]

Example:-



Remembering DFA:



Fact-1 :-

Regular Languages:- A language is called regular if \exists a finite state automaton that decides it.

Let M be a finite state automaton and,

$$L(M) = \{w \mid w \text{ is accepted by } M\}$$

then $L(M)$ is regular.

Any language has a set of operations that can be performed on it associated with it.

Regular Operations:- Let L_1, L_2 be languages

Binary \rightarrow i) Union :- $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$

ii) Concatenation :- $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$

Unary \rightarrow iii) Star :- $L_1^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$

Countably $\infty \leftarrow$ \hookrightarrow L, concat any no. of strings from L.

④ Σ :- $\Sigma = \{a\} ; \Sigma^* = \{\epsilon, a, aa, \dots\}$

$\Sigma = \{\emptyset\} ; \Sigma^* = \{\emptyset\}$

ϵ belongs to the * of any language by convention

$$L = \{0, 1\} \rightarrow L^* = \{\epsilon, 0, 1, 00, 01, \dots\}$$

Closure of Regular Languages

If the resultant of performing set operation on ~~two~~ regular languages is regular, then it is said to be closed under that operation.

i) Closure under union:-

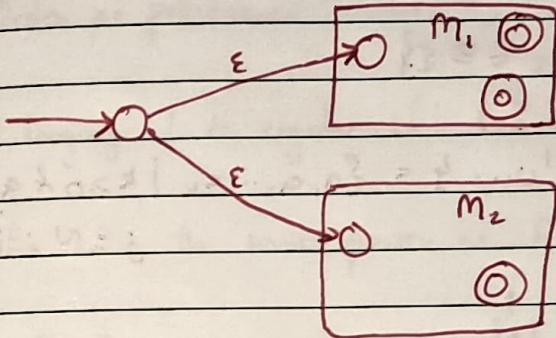
Let L_1 & L_2 be two regular languages and M_1 & M_2 be the FSA for L_1 & L_2 .

Construct an NFA by having a common start state with 2^Σ

transition to the start states of M_1 & M_2 . This decides ~~not~~ L, UL.

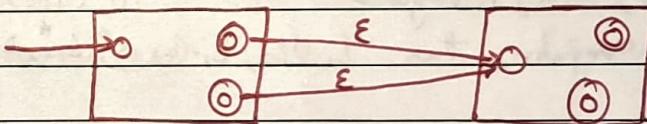
Hence Union is closed

Math proof in slides [Lec-3]



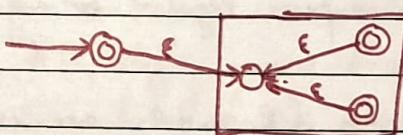
(ii) Closure under concatenation :-

Construct an NFA with start state as that of M_1 and ϵ transitions b/w final states of M_1 to the start state of M_2 .



(iii) Closure under star :-

Construct a new start state which is an accepting state, have an ϵ transition to the old start state, and have ϵ -loop backs from the final states to the old start state.



(iv) Closed under complement:- [Just toggle states of the DFA] Not for NFA ↴

v) Closure under intersection:- Yes, proof using De Morgan's Law with Union and Complement.

Notation:- Σ is an alphabet

i) $\Sigma^0 = \{\epsilon\}$

ii) $\Sigma^2 = \{a_1 a_2 \mid a_1, a_2 \in \Sigma\}$

iii) $\Sigma^k = \{a_1 \dots a_k \mid a_i \in \Sigma\}$

iv) $\Sigma^+ = \bigcup_{i \geq 0} \Sigma^i = \{\Sigma^0 \cup \Sigma^1 \cup \dots\} = \{a_1 a_2 \dots a_n \mid k \geq 0 \text{ & } a_j \in \Sigma \forall j \in \mathbb{N}; i \leq k\}$

$L \subseteq \Sigma^*$ and $L^* = \bigcup_{i \geq 0} L^i$

- First Principles:- [Ab initio]

Re Lang alt def:- Let Σ be an alphabet. Then the following are Re langs over Σ

i) \emptyset is regular

ii) for each $a \in \Sigma$, $\{a\}$ is regular

iii) If L_1 & L_2 are regular, then $L_1 \cup L_2$, $L_1 \cdot L_2$, L_1^* , L_2^* are regular.

Syntax for Regular Expressions :- R is said to be a regular expression if it has one of the following forms:-

i) \emptyset is a regular expression. $L(\emptyset) = \emptyset$

ii) ϵ is a regular expression. $L(\epsilon) = \{\epsilon\}$

iii) Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$

Regular Expressions are ways to express regular languages

iv) $R_1 + R_2$ is a regular expression if R_1 and R_2 are regular expressions, $L(R_1 + R_2) = L(R_1) \cup L(R_2)$

v) R^* is a regular expression if R is a regular expression, $L(R^*)$

vi) $R_1 R_2$ is a regular expression if R_1, R_2 are regular expressions, $L(R_1 R_2) = L(R_1) \cdot L(R_2)$

regular expression:-

Regular Expressions are like the alphabet of the Regular Language

vii) (R) is a regular expression if R is a regular expression. $L((R)) = L$

Order of precedence :- $(), +, \cdot, ^$

A language L is regular iff for some regular expression R , $L(R) = L$

REs have the same power as FSA.

Reg Expr	$L(R)$
01	$\{01\}$
$01 + 1$	$\{01, 1\}$
$(0+1)^*$	$\{\epsilon, 0, 1, 00, 01, \dots\}$
$(01+E)^\epsilon$	$\{01, \epsilon\}$
$(0+1)^*01$	$\{01, 001, 101, 0001, \dots\}$
$(0+10)^*(E+1)$	$\{\epsilon, 0, 10, 00, 001, 010, 0101, \dots\}$

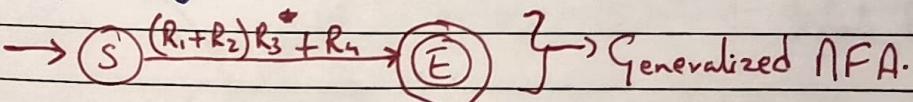
Check slides for example - Lec 3. I'm ~~deccpy~~ ^{high} ;)

DFA to RE:-

If a lang is regular, it accepts an RE.

Given a DFA, we recursively construct a 2 state generalized NFA with

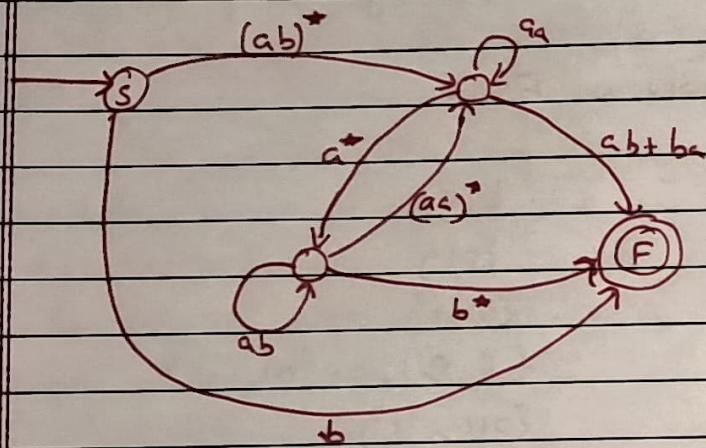
- i) a start and end state [Single; no incoming arrows to start, outgoing from end]
- ii) A single arrow from start to end
- iii) The label of the arrow is the RE corresponding to the language accepted by M.



hello cold world hello h ε / /

Generalized NFAs

- Transitions may have REs
- Unique start state with only outgoing arrows
- Unique final state with only incoming arrows
- During a run in a GNFA, blocks of symbols might be read instead of single symbols. [These correspond to REs]



This GNFA has multiple accepting strings like
b, abababab, aabbbab, etc.

↳ Uses ϵ transitions [think!]

"All transitions can be ϵ ←
due to it being RE!"

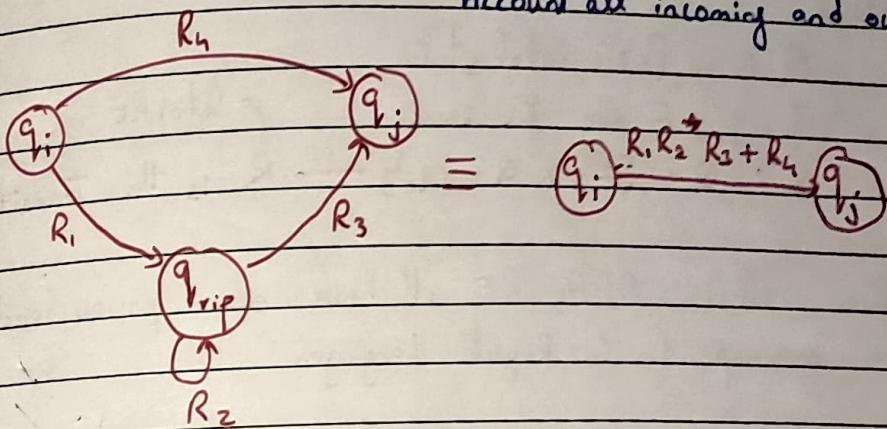
Starting from a DFA, we will begin by constructing a GNFA on k states. We then follow a recursive procedure to keep reducing the no. of states by 1 until we reach a GNFA with 1 state.

i) Add a new start state with same transitions to the old start state.

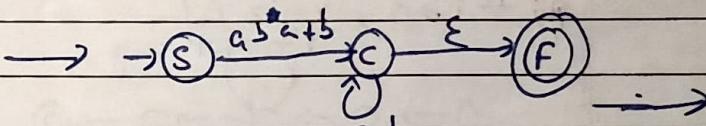
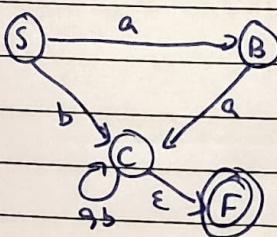
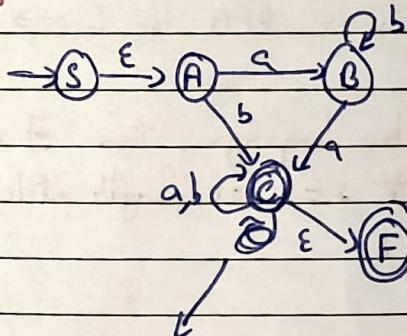
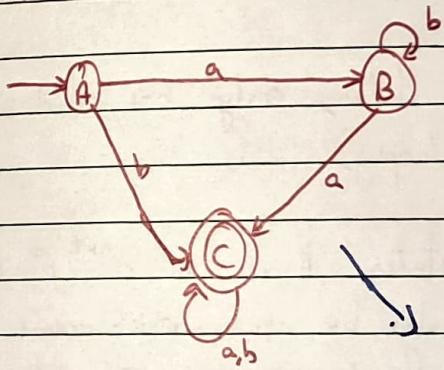
Add a new final state with ϵ transitions from all the old final states.

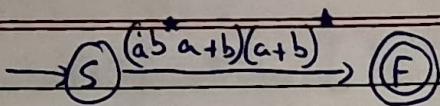
8 Pick any state [$i = \text{start/finish}$] of the GNFA
 Call it q_{rip} .

iii Rip out q_{rip} and adjust the states of the GNFA.
 Account all incoming and outgoing edges



Ex:-

DFA \rightarrow 



Formally, a GFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$
where

Q is a finite set of states

Σ, q_0, F are the same \uparrow Set of RE

$\delta : Q - \{F\} \times Q - \{q_0\} \rightarrow R$ is the transition fn.

Thus DFAs, NFAs, RE all have equal power and all of them correspond to Regular Languages

Pumping Lemma:-

Let $\Sigma = \{0, 1\}$. Consider $L = \{0^n, n \geq 0\}$.

For a DFA of n -distinct states, only $n-1$ sized strings can be accepted (without looping)

If there is a loop of t states, then even $0^{n+t} 1^n$ or (or perhaps $0^n 1^{n+t}$) will also be read, which means that \exists there exists no DFA for $L \rightarrow L$ is not regular.

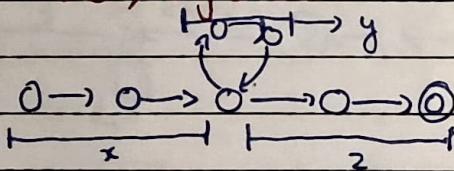
If L is a regular language, then \exists a number p (pumping length) where $\forall s \in L$ of length at least p , $\exists x, y, z$ s.t

$s = xyz$, s.t

i) $|y| \leq p$

ii) $|y| \geq 1$

iii) $\forall i \geq 0, xy^i z \in L$



We usually prove a language is not regular by contraposition.

Contraposition = If p s.t. $s \in L$ with $|s| \geq p$, $\nexists (x, y, z)$, with $s = xyz$ s.t. $\neg (|xyz| \leq p) \vee \neg (|y|^i \geq 1) \vee \neg (\forall i \geq 0, xy^i z \in L)$

Proof:- Say the DFA M has p states.

By the pigeonhole principle, any string $\geq p$ would encounter a loop.

Let $s = s_1 s_2 \dots s_n$ with $n \geq p$ and suppose $v_1 v_2 \dots v_m$ be the sequence of states encountered while implemting a run of s in M .

There would be two states $v_i = v_j$ where $i \neq j$. We can hence divide s into three parts $\rightarrow \underbrace{s_1 s_2 \dots s_i}_{x}, \underbrace{s_{i+1} \dots s_j}_{y}$ and $\underbrace{s_{j+1} \dots s_n}_{z}$

i) We can traverse the loop bit any number of times and so $\forall i \geq 0$, $xv^i z \in L$

ii) Also, as $j \neq i$, $|y| \geq 1$.

iii) The DFA reads $|xyz|$ before the end of the first loop, hence $|xyz| \leq p$

Context Free Grammars

Grammars - provide a way to generate strings belonging to a language
 \hookrightarrow set of rules \rightarrow systematic procedures

Contain variables and terminals

\hookrightarrow consist of strings over the alphabet corresponding to the language that the grammar generates

Formally, Grammars are 4-tuples (V, Σ, P, S) such that

- i) V is the set of variables
- ii) Σ is the set of terminals (disjoint from V)
- iii) P is the set of production rules
- iv) S is the start variable

$$P: (V \cup \Sigma)^* V (V \cup \Sigma)^* \rightarrow (V \cup \Sigma)^*$$

\hookrightarrow Has at least one variable on the left hand side

S is generally the variable on the LHS of the first rule.

Ex:- $G: X \rightarrow 1X$

$$X \rightarrow 0Y \quad S = X$$

$$Y \rightarrow 0X \quad V = [X, Y]$$

$$Y \rightarrow 1Y \quad \Sigma = [0, 1]$$

$$Y \rightarrow \epsilon$$

The process of substitutions using the rules of G required to obtain a certain string is called a derivation.

The generated string is said to be derived by the grammar

$$X \rightarrow 1X \rightarrow 11X \rightarrow 110Y \rightarrow 1101Y \rightarrow 1101$$

is a valid derivation.

A string belongs to the language of a Grammar if \exists derivation for it.

$$L(G) \text{ is } \{ w \in \Sigma^* \mid s \Rightarrow^* w \}$$

\hookrightarrow derivation

Regular Grammar :- If the rules of the Grammar G are of the form

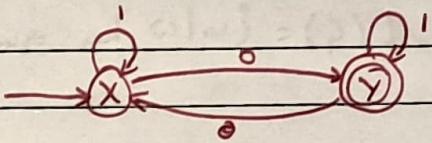
$\text{Var} \rightarrow \text{Ter Var}$

$\text{Var} \rightarrow \text{Ter}$

$\text{Var} \rightarrow \epsilon$

then the language of the grammar is regular. These are also called Right-linear grammar (a single var to the right of the terminals in the RHS)

DFA:-



Regular Grammar

A run in the DFA are analogous to the derivations in CFGs

Left linear grammar:- $\text{Var} \rightarrow \text{Var Ter}$

$\text{Var} \rightarrow \text{Ter}$

$\text{Var} \rightarrow \epsilon$

Mixing Left and Right Linear Grammars does not result in Regular Langs

Context Free Grammars:- If the rules of the Grammar is of the form $V \rightarrow (\cancel{V} \cancel{V} \cancel{S}) (V \cup T)^*$, then such a Grammar is Context Free.

Regular Grammars \subset CFG

Ex:- $S \rightarrow 0S1 \quad | \quad S \rightarrow \epsilon$ $\Rightarrow L(G) = \{ \omega \mid \omega = 0^n 1^n, n \geq 0 \}$

CFGs are more powerful than Regular Grammars / DFA / NFA

$S \rightarrow 0S1 \quad | \quad S \epsilon \quad | \quad \epsilon$ \Rightarrow If $0S1$ are $\{\}\},$ then it generates balanced parenthesis

Tips:-

1) Check if the CFL is a union of simpler languages

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow \dots$$

$$S_2 \rightarrow \dots$$

2) Grammars with rules like $S \rightarrow aSb$ help construct machines that might need unbounded memory.

Q) Construct a CFG to s.t $L(G) = \{w \mid w \text{ has equal } a \& b\}$

$$S \rightarrow SS$$

$$S \rightarrow OS1$$

$$S \rightarrow 1SO$$

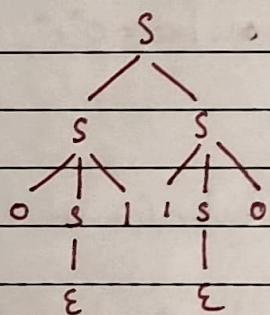
$$S \rightarrow \epsilon$$

Parse Trees for CFGs

Ordered trees that provide alt representations of the derivations of a grammar.

Parsing is used in compilers.

Root of the tree is the start node, leaves are the terminals and the leaves are read from left to right. We move from a node to its children by applying the rules of the CFG.



Any string can have multiple derivations, but the parse trees are the same.

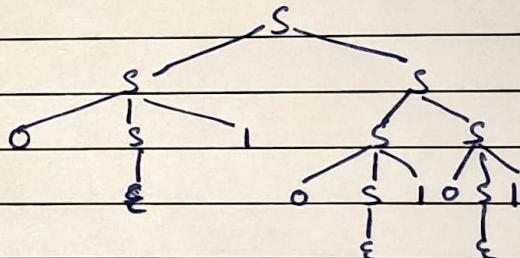
Definition:- If a string is derived by always replacing the leftmost var, it is called the leftmost derivation.

Definition:- Similar for rightmost derivation.

Derivations need not be leftmost or rightmost.

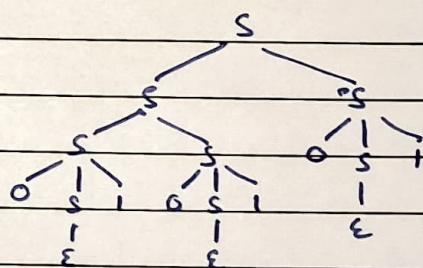
Ambiguous Grammars:- A CFG is called ambiguous if $\exists w \in L(G)$ such that there are two or more parse trees/ left derivations /right derivations for w .

Consider $w = 010101$ with $S \rightarrow 0S1 \mid SS \mid \epsilon$



Leftmost :- $S \rightarrow SS \rightarrow OS1S \rightarrow 01S \rightarrow OISS \rightarrow 010S1S \rightarrow 0101S \rightarrow 010101$

$010101 \leftarrow 01010S1 \leftarrow$



Ambiguity is not desirable, esp in compilers, since there would be different interpretations and hence different outcomes.

We can remove parenthesis or assign precedence while branching and change of rules. add vars

Chomsky Normal Form:-

A CFG G is said to be in CNF if all of its rules are of this form

$$\begin{cases} \text{Var} \rightarrow \text{Var Var} \\ \text{Var} \rightarrow \text{ter} \\ \text{Start var} \rightarrow \epsilon \end{cases}$$

where Var is any variable incl. the Start Var.

Suppose we have an algorithm, which given a CFG G and an input str w s.t it outputs Yes if it accepts or No if it rejects.

One idea is to try all derivations. But this means that the algorithm would never halt if G does not generate w .

But CNF can help as it guarantees that {if $w \in L(G)$, then a CFG in CNF has derivations of len $\leq n-1$ for input strings w of length n } {Any CFL can be generated by a CFG in CNF} ②

Algo:-

- Convert G to CNF
- List all derivations of $\leq n-1$ steps where $|w|=n$
- Pattern match to find out.

Proof:- ① Base Case :- Let $|w|=1$. Then one application of the 2nd rule is sufficient. So the derivation of w would need $\leq |w|-1 = 0$ step

Inductive Hypothesis:- Assume that this is true for any string of length at most k where $k \geq 1$.

Induction Step:- Take $|w|=k+1$. By the first rule, w can be subdivided into two parts A, B where $|A|, |B| \leq k$

So both these parts are derived in $2|A|-1$ & $2|B|-1$ respectively, by induction.

So, to derive w , we would take

$$2|A|-1 + 2|B|-1 + 1 \rightarrow 2(|A|+|B|)-1$$
 ~~$\cancel{+1}$~~ ~~$\cancel{2(k+1)-1}$~~ $\leftarrow 2|w|-1 \leftarrow$

Hence proved.

29

② To convert into a ~~BCFG~~ into a LTI CNF

- Add a new start variable $S' \rightarrow S$
- Remove ϵ rules (nullable symbols and rules)
- Remove unit (short) rules of the form $A \rightarrow B$
- Remove long rules of the form $A \rightarrow u, u, \dots, u$

ii) For each occurrence of 'A' on the right hand side, add a new rule with one occurrence of A deleted.

$$B \rightarrow uA \vee A w \implies B \rightarrow uA \vee Aw \mid uA \vee w \mid uw$$

If we had a rule like $B \rightarrow A$, we would have needed to add $B \rightarrow \epsilon$ (unless the rule has been removed), since B is nullable. Loop until all ϵ -rules are removed

$$S \rightarrow 0 \mid x \mid 0 \mid \cancel{x} \mid \cancel{x}$$

iii) We remove rules like $A \rightarrow B$ by using transitivity (i.e., if $B \rightarrow u$, add a new rule $A \rightarrow u$) unless the rule was removed. Loop until finished.

$$S \rightarrow A \mid u \quad S \rightarrow B \mid I \mid II \quad S \rightarrow B \mid I \mid II \quad S \rightarrow B \mid I \mid II$$

$$A \rightarrow B \mid I \rightarrow A \rightarrow BI \mid I \rightarrow A \rightarrow SI \mid I \rightarrow A \rightarrow SI \mid I$$

$$B \rightarrow SI \quad B \rightarrow SI \quad B \rightarrow SI \quad B \rightarrow I \mid II \mid O$$

$$S \rightarrow 0 \mid I \mid II \quad S \rightarrow 0 \mid I \mid II$$

$$A \rightarrow 0 \mid I \mid II \leftarrow A \rightarrow SI \mid I \leftarrow$$

$$B \rightarrow 0 \mid I \mid II \quad B \rightarrow 0 \mid I \mid II$$

(iv) Add new variables !

$A \rightarrow u_1, u_2, \dots, u_k \Rightarrow A \rightarrow u_1 A_1$

$A_1 \rightarrow u_2, \dots, u_k \Rightarrow A_1 \rightarrow u_2 A_2$

Loop till done

Now add newer variables like

$A \rightarrow u_1 A_1 \Rightarrow A_1 A_1 \rightarrow A \rightarrow A_1 A_1$

where $A_1 \rightarrow u_1 \wedge A_1 \rightarrow A_2 A_2 \dots$

Ex:- $S \rightarrow ASA | aB$

$A \rightarrow B | S$

$B \rightarrow b | \epsilon$

(i) $S' \rightarrow S$

$S \rightarrow ASA | aB$

$A \rightarrow B | S$

$B \rightarrow b | \epsilon$

(ii) a) $S' \rightarrow S$

$S \rightarrow ASA | aB | a$

$A \rightarrow B | S | \epsilon$

$B \rightarrow b$

b) $S' \rightarrow S$

$S \rightarrow ASA | aB | a | AS | SA | S$

$A \rightarrow B | S$

$B \rightarrow b$

(iii) a) Remove $S \rightarrow S$ b) Remove $S' \rightarrow S$

c) $S' \rightarrow ASA | aB | a | AS | SA | S$

$S \rightarrow \epsilon$

$A \rightarrow S | b$

$B \rightarrow b$

(d) $S' \rightarrow ASA|abA|AS|SA|S$

$S \rightarrow "$

$A \rightarrow b | "$

$B \rightarrow b$

(iv) Replace non unitary SA with V, a with V

$S' \rightarrow AV|VB|a|AS|SA$

$S \rightarrow "$

$A \rightarrow b | "$

$B \rightarrow b$

$V \rightarrow SA$

$V \rightarrow a$

Push down Automata :-

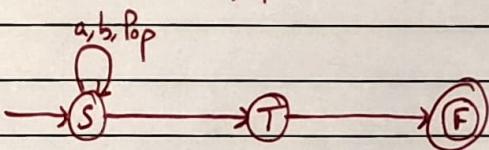
Since there is a need for unbounded memory, the automata for a CFL = FSM + memory device. By ignoring the memory device, it can emulate a DFA / NFA.

The memory device used is a stack

Transition can happen either based on what it reads, or the element popped from the top.

It can push new elements into the stack

It can also pop from the stack.



Use slides for notes [Lec - 5 or 6 idk]
example runs

Syntax :- $a, b, \text{cmd } c \rightarrow \text{read } c$
 $\text{pop } b$
 $\text{exec cmd arg} \rightarrow c$

Another Syntax :- $a, b \rightarrow c \rightarrow \text{read } a$
 $\text{pop } b$
 $\text{push } c$

If $b \neq \epsilon$, cmd must be Push

Formally, a PDA M is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$
 where

Q is a finite set called the states

Σ is a set of input alphabet

Γ is a set of stack alphabet

$\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$ is the transition func

q_0 is the start state.

$F \subseteq Q$ is a set of accepting states

A PDA accepts a string $w \in L$ if \exists a run such that
 \rightarrow it reaches the final state when the entire string is read

or

\rightarrow the stack is empty when the entire string is read

They are equivalent cause one can be converted into the other.

Transition fn:-

$\delta(q_i, a, b) = (q_j, c)$: If c is read and b is popped
 , pushed and transition from q_i to q_j .
 $a, b, c \in \Sigma \cup \Gamma$

The language L of the PDA P is
 $L = \{w \mid P \text{ accepts } w\}$

The stack alphabet need not be the same as the input alphabet

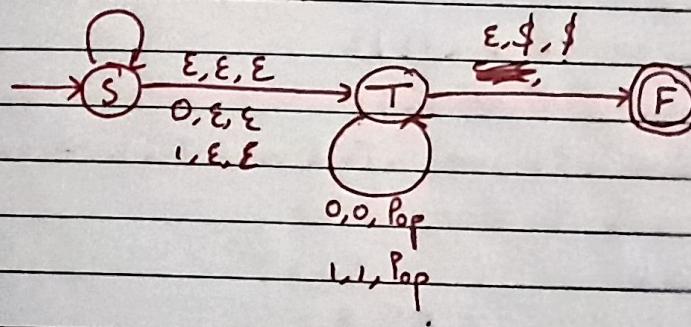
(Q) Construct the PDA deciding Paliind $L = \{w \mid w \text{ is palindromic}\}$

Intuition :- Push first half onto stack . Verify if $w^{2^{\text{nd}}}$ half mirrors the 1st half by popping and comparing.

We The PDA finds the middle of the input non-deterministically

For odd length palindromes , ND helps again

O.E. Push 0 / 1, E Push 1



Equivalence b/w CFLs & PDA

- i) If L is CFL, then \exists a PDA that recognizes it
- ii) If a PDA recognizes L , then L is CFL

DPDAs are weaker than PDAs,



DCFL \subseteq CFL

For any L , we can write a CFG that can generate all strings that are in L .

Any string w is generated by the CFG if \exists derivation $S \xrightarrow{*} w$.

We must build a PDA that can simulate any derivation $S \xrightarrow{*} w$

Intuition:-

- The PDA begins by pushing S onto the stack.
- The PDA pops the top variable, pick a random rule $A \rightarrow x$ (x can be a sequence of variables and terminals) and push x onto the stack.
- Read input symbol if the top of the stack is a terminal. (Tries to match part of the input non-deterministically)

Ex:- Consider $G := S \rightarrow aTb \mid b$

$$T \rightarrow Ta \mid \epsilon$$

$$w = aab \Rightarrow S \rightarrow aTb \rightarrow aTa b \rightarrow aTaa b \rightarrow aab$$

→ Push S onto the stack

→ Pop S and

(i) Push b [ϵ, S, b] Push b

(ii) Push T [$\epsilon, \epsilon, \text{Push } T$]

(iii) Push a [$\epsilon, \epsilon, \text{Push } a$]

→ Pop Top and match

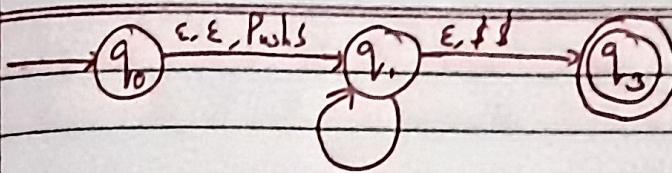
→ Pop Top and push rule

→ Pop T and push Ta

→ Pop T and push ϵ

→ Pop top and match $\times 3$

→ Since stack is empty and input is read, accept w .



$\epsilon, S, \text{Push } aTb$

$\epsilon, T, \text{Push } Ta$

$\epsilon, S, \text{Push } b$

ϵ, T, Pop

a, a, Pop

b, b, Pop

→ Multiple states, but loop back to
 q_1 as reqd for the rules
by

Proof:- The PDA will have three kinds of states $\{q_0, q_1, q_2\}$

The PDA always pushes S and transitions to q_1 first

At q_1 , PDA implements the rules of CFG.

Pop A and push x if the rule is $A \rightarrow x$ and return back to q_1 ,
 $\delta(q_1, \epsilon, A) = (q_1, x)$

Pop a , i.e. let $\delta(q_1, a, \epsilon) = (q_1, \epsilon)$, matching the input
giving with the terminals on the stack.

Hence

$$RL \equiv RG \equiv RE \equiv NFA \equiv DFA \subseteq CFL \equiv CFG \equiv PDA$$

Pumping Lemma for CFL

If $L \in \text{CFL}$, $\exists p \in \mathbb{Z}, p \geq 1$ (pumping len) s.t.

$\#s \in L$ s.t $|s| \geq p$, $\exists u, v, x, y, z \in \Sigma^*$ s.t $s = uxvxyz$

i) $|vxy| \geq 1$

ii) $|vxy| \leq p$

iii) $uv^nxy^nz \in L \quad \forall n \geq 0$

To show L is not a CFL

(1) Assume L is a CFL

(2) Show that $L = \{ww^{\dagger}w \in \{0,1\}^*\}$ is not a CFL

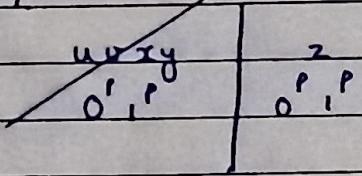
Assume that L is a CFL.

Suppose $\exists p$ for L .

Choose $w = 0^r 1^r 10^r 1^r$

Valid decomposition of w :

Case 1:



All CFLs satisfy the conditions of the pumping lemma.

In order to recognize very long strings, the model of computation must repeat certain steps.

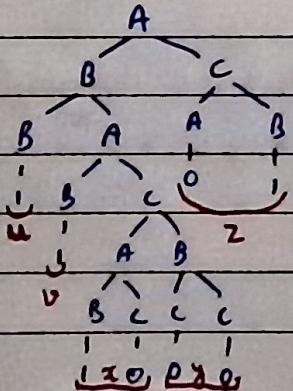
This can be repeated to create longer and longer strings.

Ex:- $A \rightarrow BC10$

$B \rightarrow BA111CC$

$C \rightarrow AB10$

Consider the derivation of $w = 11100001$



If we look at the longest path to a terminal, $\text{len} > |V| + 3$
 Then \exists at least one variable that is repeated.

For example, A -

Replace the lowermost occurrence of A with the middle occurrence of A to obtain a newer, longer tree

This belongs to L since the rules of the grammar are followed.

Properties of a parse tree:-

① Let L be a CFL and G be s.t $L = \text{Lang}(G)$ and $w \in L$.

Consider a parse tree T_w^G of G that yields w. Then:-

i) A path from root to a leaf is a seq of terminals variables ending in a Terminal

ii) Let d be the max no. of variables / terminals on the RHS of any rule of G.

iii) Any T_w^G that has at level l at most d^l nodes. Thus any T_w^G of height h has atleast d^h terminals.

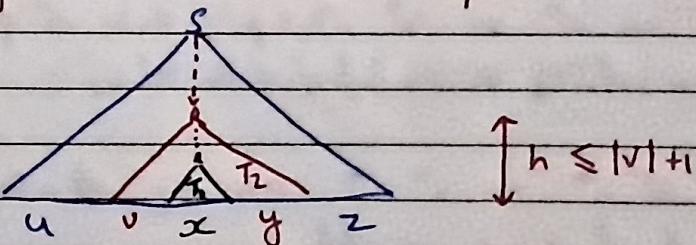
Let $|V|$ be total no of vars in G.

If $w \in L$ s.t $|w| = p = d^{|V|+1}$, the underlying parse tree would have a height $\geq |V| + 1$

The longest path to a terminal from S $\geq |V| + 1$.

By pigeon hole principle some variable has to be repeated within the lowest $|V| + 1$ variables in that path.

Then any w s.t $|w| \geq p$ can be partitioned as $w = u v x y z$



max
 The \uparrow no. of terminals of the upper R at $h = |V| + 1$,
 $|w| = d^{|V|+1} \geq |wxy|$.
 But we also established $p = d^{|V|+1}$
 hence $|wxy| \leq p$ ①

By replacing T_0 with T_i $\forall i \geq 0$
 $wxyz$.

We replace T_2 with T_i for $i=0$, and it still holds
 Hence $wxyz \in L \quad \forall i \geq 0$ ②

In case G is ambiguous, we pick the smallest parse tree generating W .

Assume $|wyl| = 0$. Then, \exists a shorter parse tree that
 doesn't go from $R \rightarrow R$, which is a contradiction.
 Hence $|wyl| \geq 1$.

∴ Thus, we've proved the pumping lemma.

Ex:- $\{0^n 1^n 2^n\}_{n \geq 0}^3$ is not a CFL

Let p be the pumping length and $w = 0^p 1^p 2^p \in L$
 $w = uxyz$.

Since $|wxy| \leq p$, and no of 1s b/w 0 & 2 ~~is~~ ^{is} $\leq p$
 , $wxyz$ can have ^{at most} ~~only one of~~ 2 of 3 symbols

$wxy = 0^k$ or 1^k or 2^k , $k \leq p$, then $w' \notin L$
 since no. of 0s \neq no. of 1s \neq no. of 2s.

$wxy = 0^m 1^n$ or $1^m 2^n$, $m+n \leq p$ \Leftrightarrow , then $w' \notin L$
 since count of 0, 1, 2 mismatch

Hence the lang is not CF

Closure Properties of CFL

Set of all CFLs is closed under union.

Create a new production rule with a new $S' \rightarrow S_1 | S_2$

Closed under concatenation ($S' \rightarrow S_1 S_2$)

Closed under star ($S \rightarrow S_1 S_2 | \epsilon$)

But CFLs are not closed under intersection!

$$L_1 = \{0^n 1^n 2^m \mid m, n \geq 0\}$$

$$L_2 = \{0^m 1^n 2^n \mid m, n \geq 0\}$$

L_1 is a concatenation of $\{0^n\}$ and $\{2^m\}$

L_2 is a concatenation of $\{0^m\}$ and $\{1^n 2^n\}$

Intersection is $L_I = \{0^n 1^n 2^n\}$ which is not a CFL

Same with complement (De Morgan's)

If L is a CFL & R is RL

LUR is CFL (Union of CFLs cause $RL \subseteq \text{CFL}$)

LNR is also a CFL

↓ By constructing a Product PDA.

If the states of PDA P: $Q = \{q_1, q_2, \dots, q_n\}$ & DFA D: $Q' = \{d_1, d_2, \dots, d_m\}$
 then the states of the Product PDA X: $Q = \{(q, d) \mid q \in Q, d \in Q'\}$

If $\delta(q_i, a, b) = (q_j, c)$ and $\delta(d_k, a) = d_l$.

then for

$$x : \delta((q_i, d_k), a, b) = ((q_j, d_l), c)$$

So x is a PDA

If $\delta(q_i, \epsilon, b) = (q_j, c)$ & $\delta(d_k, \epsilon) = \emptyset$, then for

$$x : \delta((q_i, d_k), \epsilon, b) = ((q_j, d_k), c)$$

Final states

$L(x) = L(P) \cap L(D)$, let final state = (q_s, d_s)
such that q_s, d_s is the final state of P and D.

$L(x) = L(P) \cup L(D)$, let final states (q_w, d_s)
such that q_w, d_s is the final state of P or D.

What lies beyond CFLs?

Computable Languages.

Turing Machines:-

Solves all computable problems

A Turing Machine is an FSM that has access to an infinite tape as memory.

The input tape ~~considers~~ contains the input string followed by blanks.

The TM can both r/w on the tape using an r/w head.

The r/w head can move left or right, one cell at a time.

At each step, the TM :-

- i) Replaces the symbol of the current cell with a new one
- ii) Transitions b/w states or stays where it was
- iii) The r/w head moves either left or right.
- iv) The TM accepts a string as it reaches an accept state or vice versa.
- v) Rejects on halts.

Transitions :- $a, b \xrightarrow{\quad} L/R \rightarrow$ Read a, write b, move L/R

Accept / Rejected States :- TM halts and accepts / rejects on these states

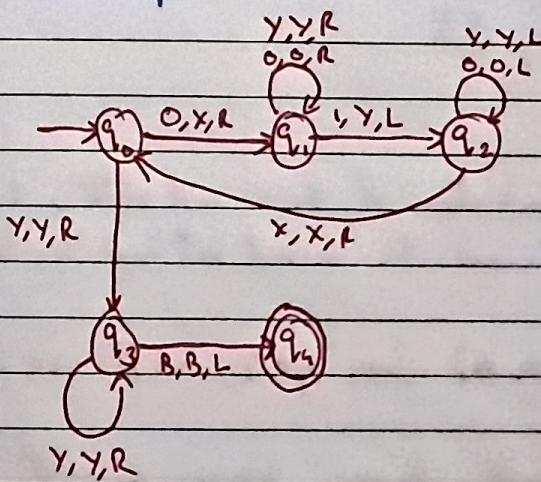
The TM might never reach a halting state and might loop forever.



the input string still does not belong to the language

Ex :- $L = \{0^n 1^n \mid n \geq 1\}$ (i.e. CFL)

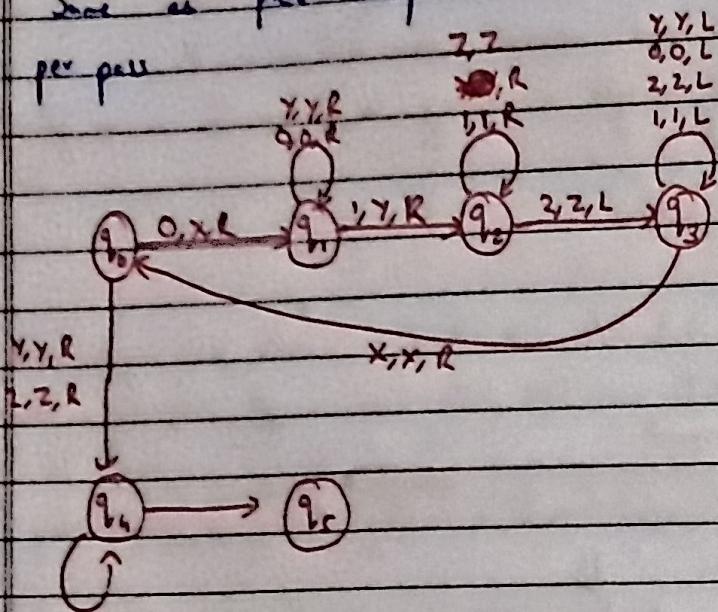
- i) Mark the first 0 with a symbol (say x)
- ii) Continue to move right until 1st 1 is encountered.
- iii) Mark the first 1 with a symbol (say y)
- iv) Repeat until all 0s & 1s have been marked
- v) Accept if no 0s & 1s are left.



All missing transitions lead to
the reject state & the input is rejected

$$\text{Ex:- } L = \{0^n, 1^n \mid n \geq 1\}$$

Same as prev example but mark with $\otimes x, y, z$
per pair



Formally, a TM is a 7 tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$

$Q \neq \emptyset$ is the finite set of states

Σ is the set of input alphabet

Γ is the tape alphabet where $\{\#\} \subseteq \Gamma \& \Sigma \subseteq \Gamma$

$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition fn.

$q_0 \in Q$ is the start state

$q_{acc} \in Q$ is the accepting state

$q_{rej} \in Q$ is the rejecting state

Configuration of a TM - Combination of the current state, the current state tape contents and the current head head location

Formally it is a triple (q, a, x) where $q \in Q, a \in \Gamma, x \in \Sigma^*$

At each step, the TM config changes. We say C_1 yields C_2 if the TM changes from C_1 to C_2 in one

step. [C_1, C_2 are configurations]

Defn:- A TM M accepts w if \exists a seq of configurations $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_k$ where

- i) C_1 is the start config of M on w .
- ii) Each $C_i \rightarrow C_{i+1}$ yields C_i .
- iii) C_k is an accepting config.

A language L is recognized by TM M st

$$L(M) = \{w \mid M \text{ accepts } w\}$$