classmate
Date
Page

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) ∀ inputs belonging to the YES instance of P, then the device outputs YES / ACCEPT
ii) ∀ 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

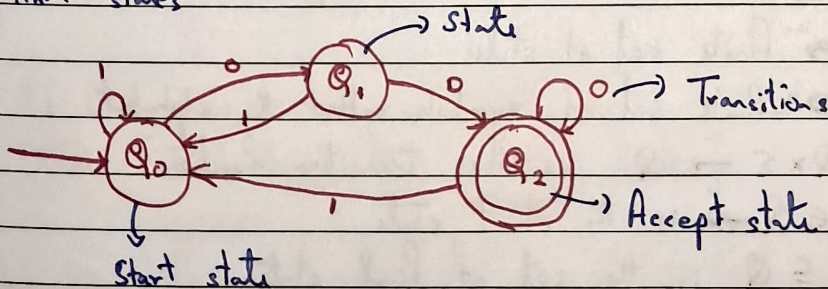
Strings / Words:- Finite sequence of symbols from an alphabet

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

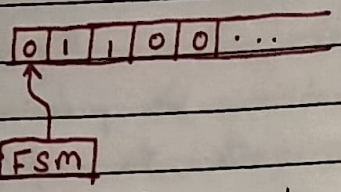
Generally, ε denotes the empty string

Deterministic Finite State Automata Model [DFA]

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



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



One way infinite Tape.

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 execution 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 accept is :-

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

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

This is also known as the language of the automaton.

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

i) $\forall\, w \in L$, $M(w)$ accepts

ii) $\forall\, w \notin L$, $M(w)$ rejects

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

Q is a finite set of state

Σ is a finite set of symbols called the alphabet

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

$q_0 \in Q$ is the start state

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

11
110
1001
1111

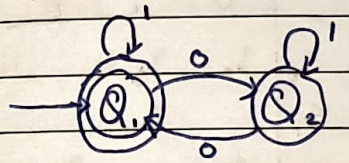
classmate
Date
Page

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 ~~ending in 00~~ that result in an Accept state at the end of the computation.

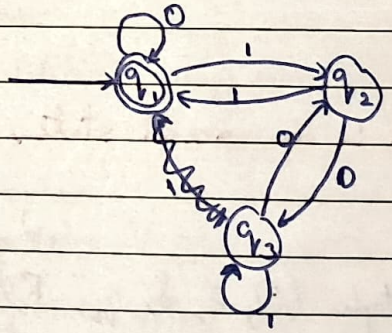
Examples:- (i) $\Sigma = \{0, 1\}$, $L(m) = \{\omega \mid \omega$ has an even no of 0s$\}$



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

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

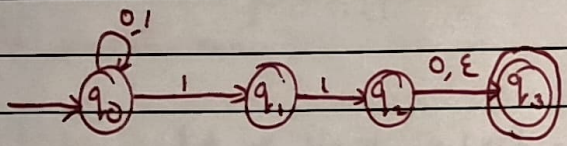
If ω is divisible by 3, the after reading a 0 it should stay in the final state, and so on.



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.



	0	1	ε
q_0	q_0	q_0, q_1	
q_1		q_2	
q_2	q_3		q_3
q_3			

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.
$q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{\varepsilon} q_3 \xrightarrow{0}$ 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)
$\delta \Rightarrow$: $Q \times \Sigma \longrightarrow P(Q)$, is the transition fn
$q_0 \in Q \Rightarrow$ start state
$F \subseteq Q \Rightarrow$ set of accepting states

$P(Q)$ → powerset of Q

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

The languages accepted by DFAs are a subset of the languages accepted by NFAs → L_2

$L_2 \subseteq L_1$

$L_1 \subseteq L_2$ (Proof upcoming) → L_1

Hence $L_1 = L_2$ → Both are equally powerful.

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

Steps :-

(i) Let R be a remembering DFA

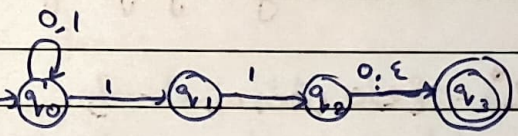
(ii) R on an input enters a state labelled by all possible states that the NFA can enter on that input

(iii) This trims away the non-determinism of the NFA without losing its language

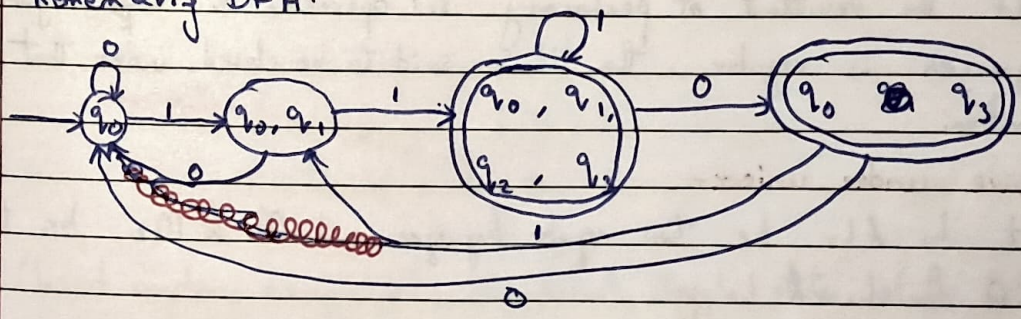
(iv) Also, if the NFA has k states, R has at most 2^k states [Power sets]

Example :-

NFA =>



Remembering DFA:



Unit-2 :-

Regular Languages:- A language is called regular if ∃ 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 → 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 → iii) Star :- $L_1^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$

Countably ∞ ↙ ↳ concat any no. of strings from L_1

Ex :- $\Sigma = \{a\}$; $\Sigma^* = \{\varepsilon, a, aa, \ldots\}$

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

ε belongs to the * of any language by convention

$L = \{0,1\} \rightarrow L^* = \{\varepsilon, 0, 1, 00, 01, \ldots\}$

Closure of Regular Languages

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

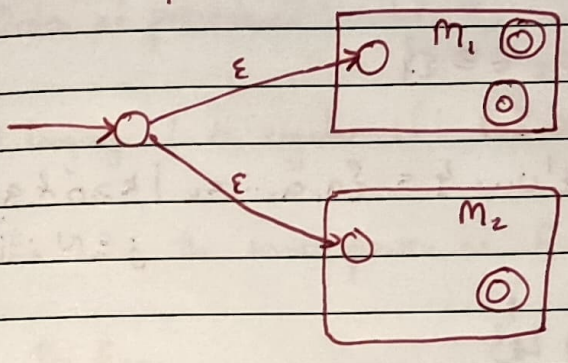
(i) **Closure under union:-**

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 ε-

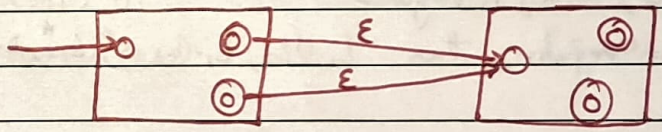
transitions to the start states of M_1 & M_2. This decides $L_1 \cup L_2$. Hence Union is closed

Math proof in slides [Lec-3]



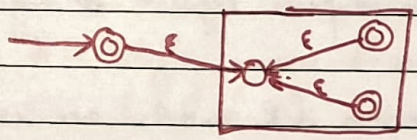
(ii) Closure under concatenation :-

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) $\Sigma^0 = \{\varepsilon\}$

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

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

iv) $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i = \{\Sigma^0 \cup \Sigma^1 \cup \cdots\} = \{a_1 a_2 \cdots a_k \mid k \geq 0 \text{ \& } a_j \in \Sigma \; \forall j \in N, j \leq k\}$

$L \subseteq \Sigma^*$ 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) ϕ 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) ϕ is a regular expression. $L(\phi) = \phi$

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

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, $L(R^*) = (L(R))^*$

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

Regular Expressions are like the alphabet of the Regular Language

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

Order of precedence :- (), *, ., +

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)*	{ε, 0, 1, 00, 01, ...}
(01+ε)1	{011, 1}
(0+1)*01	{01, 001, 101, 0001, ...}
(0+10)*(ε+1)	{ε, 0, 10, 00, 001, 010, 0101, ...}

Check slides for examples - Lec3. I'm a copy ;) 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

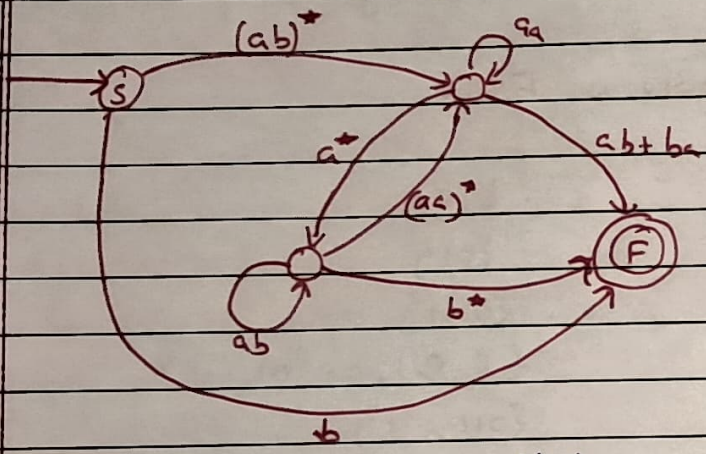
iii) The label of the arrow is the RE corresponding to the language accepted by M.

→ (S) —(R1+R2)R3*+R4→ (E) } → Generalized NFA.

hello cold world hello h & 11 &

Generalized NFAs

→ Transitions may have REs
→ Unique start state with only outgoing arrows
→ Unique final state with only incoming arrows
→ During a run on 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, aaabbbaa, etc.

↳ Uses ε transitions [think!]

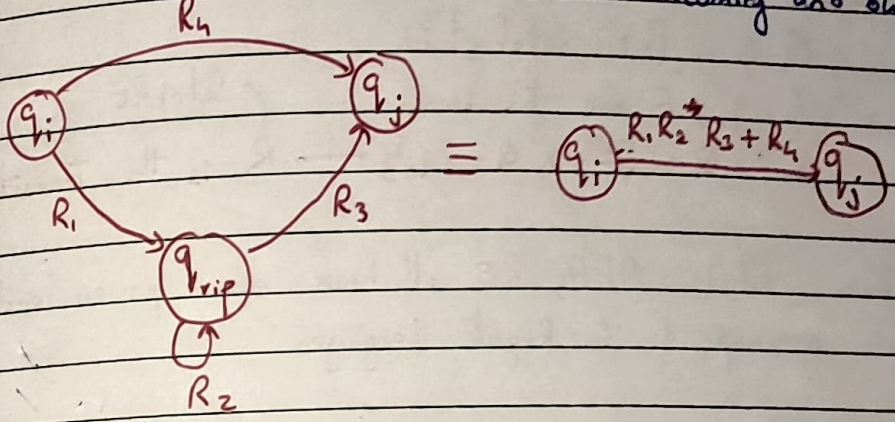
"All transitions can be ε ↵
due to it being RE!

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 2 states.

(i) Add a new start state with an ε transition to the old start states.
Add a new final state with ε transitions from all the old final states

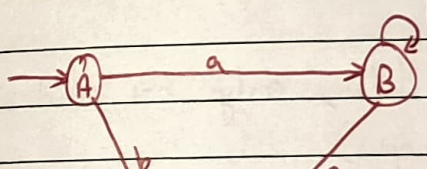
(ii) & Pick any state [! = start / finish] of the GNFA
Call it q_{rip}

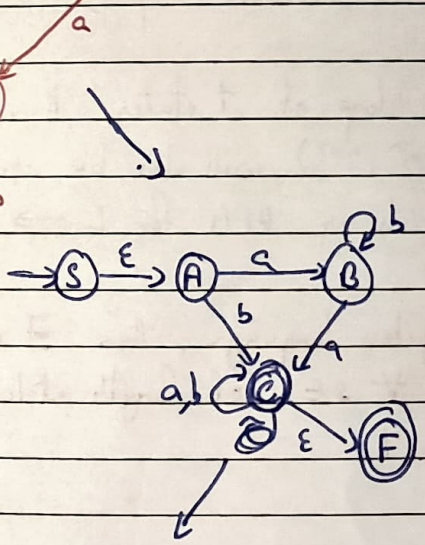
(iii) Rip out q_{rip} and adjust the states of the GNFA.
Account all incoming and outgoing edges

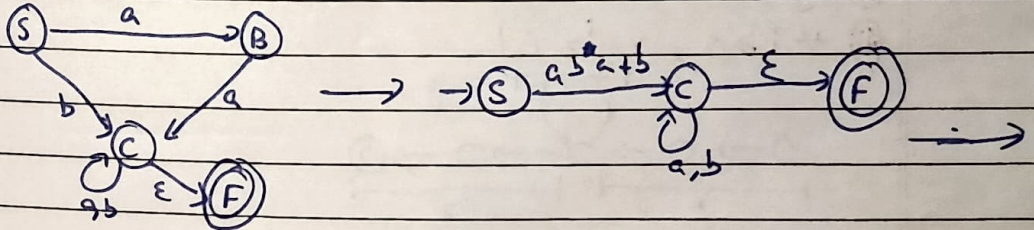


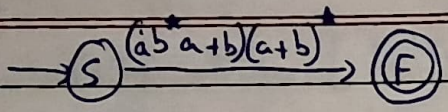
Ex:-

DFA →









Formally, a GNFA 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

$\delta : Q - \{F\} \times Q - \{q_0\} \longrightarrow R$ is the transition fn. (R = set of RE)

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

Pumping Lemma:-

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

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

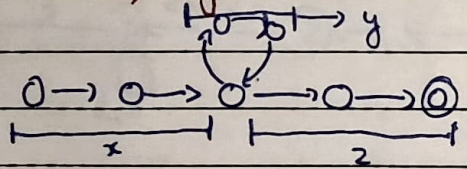
If there is a loop of t states, then even $0^{n+t}1^n$ (or perhaps $0^n 1^{n+t}$) will also be read, which means that there exists no DFA for L $\longrightarrow$ L is not regular.

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) $|xy| \leq p$

(ii) $|y| \geq 1$

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



We usually prove a language is not regular by contraposition

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

Proof:- Say the DFA M has p states.

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

Let $s = s_1 s_2 \ldots s_n$ with $n \geq p$ and suppose $r_1 r_2 \ldots r_{n+1}$ be the sequence of states encountered while ~~implied~~ implementing a run of s in M.

There would be two states $r_i = r_j$ where $i \neq j$. We can hence divide s into three parts → $\underbrace{s_1 s_2 \ldots s_{i-1}}_{x}$ $\underbrace{s_i \ldots s_j}_{y}$ and $\underbrace{s_{j+1} \ldots s_n}_{z}$

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

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

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

Context Free Grammars

Grammars:- provide a way to generate strings belonging to a language
↳ set of rules → Systematic procedures

Contain variables and terminals
↳ 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

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

↳ Has at least one variable on the left hand side

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

Ex:- G: $X \longrightarrow 1X$
$X \longrightarrow 0Y$ $\quad S = X$
$Y \longrightarrow 0X$ $\quad V = [X, Y]$
$Y \longrightarrow 1Y$ $\quad \Sigma = [0, 1]$
$Y \longrightarrow \varepsilon$

The seq 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 $\exists$ a derivation for it.

$L(G)$ is $\{ \omega \in \Sigma^+ \mid S \Rightarrow \omega \}$
↳ derivation

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

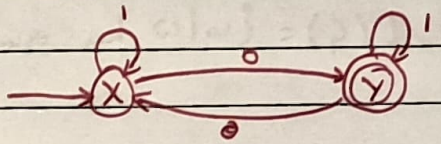
Var → Ter Var

Var → Ter

Var → ε .

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 ~~CFG~~s

Left linear grammar: Var → Var Ter

Var → Ter

Var → ε

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 → (~~V∪Σ~~) (V ∪ T)*, then such a Grammar is Context Free.

Regular Grammars ⊂ CFG

Ex:- S → 0S1 | ε → L(G) = {ω | ω = 0^n 1^n, n ≥ 0}

CFGs are more powerful than Regular Grammars / DFA / NFA

S → 0S1 | SS | ε → If 0 & 1 are {}, then it generates balanced paranthesis

Tips:-

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

$S \rightarrow S_1 | S_2$

$S_1 \rightarrow \ldots$

$S_2 \rightarrow \ldots$

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

Q) Construct a CFG s.t $L(G) = \{w | w$ has equal 0s & 1s$\}$

$S \rightarrow SS$

$S \rightarrow 0S1$

$S \rightarrow 1S0$

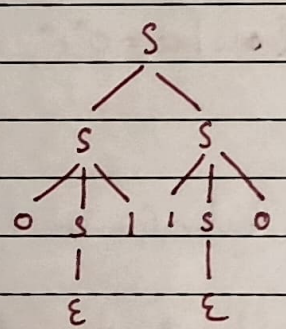
$S \rightarrow \varepsilon$

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.

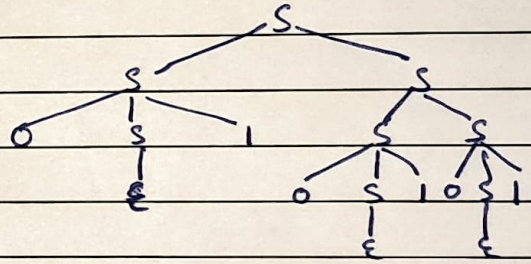
Definition:- If a string is derived by always replacing the leftmost var, it is called the leftmost derivation.

Definition:- Similar for rightmost derivation.

Derivations need not be leftmost or rightmost.

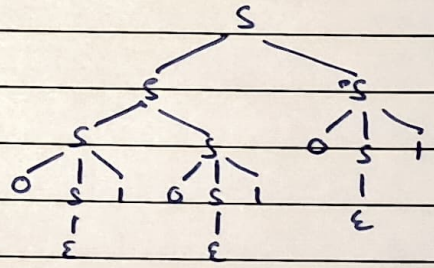
Ambiguous Grammars:- A CFG is called ambiguous if $\exists\ w \in L(G)$ such that there are two or more parse trees / left derivations / right derivations for w.

Consider $w = 01\,01\,01$ with $S \to 0S1 \mid SS \mid \varepsilon$



Leftmost :- $S \to SS \to 0S1S \to 01S \to 01SS \to 010S1S \to 01010S1$

$0101010 \leftarrow 0101051$ ↵



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

We can remove paranthesis or assign precedence while breaking 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

Var → Var Var
Var → ter
Start var → ε

(not this)

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 $2n-1$ for input strings w of length n} {Any CFL can be generated by a CFG in CNF} ②

Algo:-
i) Convert G to CNF
ii) List all derivations of $2n-1$ steps where $|w| = n$
iii) 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 $2|w|-1 = 1$ step

Induction 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 sub divided 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

$2^{|A|-1} + 2^{|B|-1} + 1 \rightarrow 2^{(|A|+|B|)-1}$

$2^{k+1-1} \leftarrow 2^{(k+1)-1} \leftarrow 2^{|w|-1}$

Hence proved.

(2) To convert a CFG G into a CNF

i) Add a new Start variable $S' \rightarrow S$

ii) Remove ε rules (nullable symbols and rules)

iii) Remove unit (short) rules of the form $A \rightarrow B$

iv) Remove long rules of the form $A \rightarrow u_1 u_2 \cdots u_k$

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

$B \rightarrow uAvAw \Rightarrow B \rightarrow uAvAw \mid uAvw \mid uvAw \mid uvw$

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

~~$S \rightarrow 0A1 \mid 2$~~

(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.

$S \rightarrow A \mid 11$ $\quad$ $S \rightarrow B \mid 1 \mid 11$ $\quad$ $S \rightarrow B \mid 1 \mid 11$ $\quad$ $S \rightarrow B \mid 1 \mid 11$

$A \rightarrow B \mid 1$ $\rightarrow$ $A \rightarrow B \mid 1$ $\rightarrow$ $A \rightarrow S \mid 0 \mid 1$ $\rightarrow$ $A \rightarrow S \mid 0 \mid 1$

$B \rightarrow S \mid 0$ $\quad$ $B \rightarrow S \mid 0$ $\quad$ $B \rightarrow S \mid 0$ $\quad$ $B \rightarrow 1 \mid 11 \mid 0$

$S \rightarrow 0 \mid 1 \mid 11$ $\quad$ $S \rightarrow 0 \mid 1 \mid 11$

$A \rightarrow 0 \mid 1 \mid 11$ $\leftarrow$ $A \rightarrow S \mid 0 \mid 1$

$B \rightarrow 0 \mid 1 \mid 11$ $\quad$ $B \rightarrow 0 \mid 1 \mid 11$

(iv) Add new variables !

$A \to u_1 u_2 \cdots u_k \Rightarrow A \to u_1 A_1$

$A_1 \to u_2 \cdots u_k \Rightarrow A_1 \to u_2 A_2$

Loop till done

Now add newer variables like

$A \to u_1 A_1 \Rightarrow$ ~~$A_{-1} A_1$~~ $A \to A_{-1} A_1$

where $A_{-1} \to u_1$ & $A_1 \to A_{-2} A_2 \cdots$

Ex:- $S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \varepsilon$

(i) $S' \to S$

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \varepsilon$

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

$S \to ASA \mid aB \mid a$

$A \to B \mid S \mid \varepsilon$

$B \to b$

(b) $S' \to S$

$S \to ASA \mid aB \mid a \mid AS \mid SA \mid S$

$A \to B \mid S$

$B \to b$

(iii) (a) Remove $S \to S$ (b) Remove $S' \to S$ (c) $S' \to ASA \mid aB \mid a \mid AS \mid SA \mid S$

$S \to$ "

$A \to S \mid b$

$B \to b$

(d) $S' \to ASA \mid aB \mid a \mid As \mid SA \mid S$

$S \to$ "

$A \to b \mid$ "

$B \to b$

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

$S' \to AU \mid VB \mid a \mid AS \mid SA$

$S \to$ "

$A \to b \mid$ "

$B \to b$

$U \to SA$

$V \to a$

Push down Automata :-

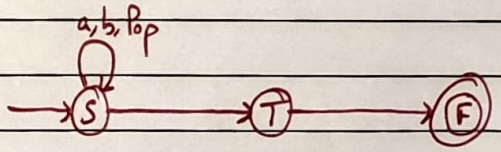
Since there is a need for unbounded memory, the automata for a CFL = FSM + Memory device. By ignoring the memory device, it can emulate a DFA / NFA.

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 pops from the stack.



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

Syntax :- a, b, cmd c → read a
pop b
exec cmd arg → c

Another Syntax :- a, b → c → read a
pop b
push c

If b ≠ ε, 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
$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition fns
q_0 is the start state.
$F \subseteq Q$ is a set of accepting states

A PDA accepts a string $\omega \in L$ if $\exists$ a run such that
→ it reaches the final state when the entire string is read
OR
→ 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, push c and transition from q_i to q_j
a, b, c can be ε

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

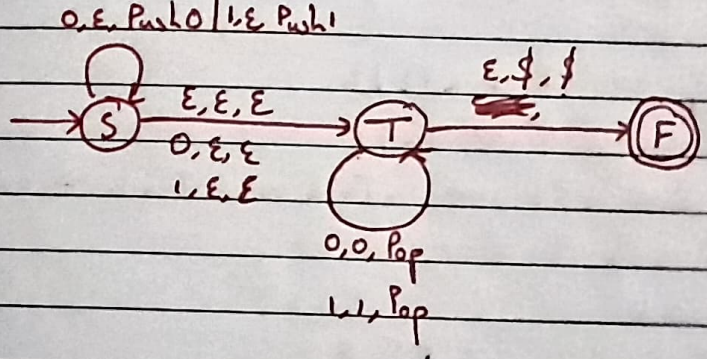
The stack alphabet need not be the same as the input alphabet

Q) Construct the PDA deciding L = {w | w is palindromic}

Intuition :- Push first half onto stack. Verify if 2nd 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



Equivalence blw CFLs & PDA

(i) If L is CFL, then ∃ a PDA that recognizes it
(ii) If a PDA recognizes L, then L is a CFL

DPDAs are weaker than PDAs

↓

DCFL ⊆ CFL

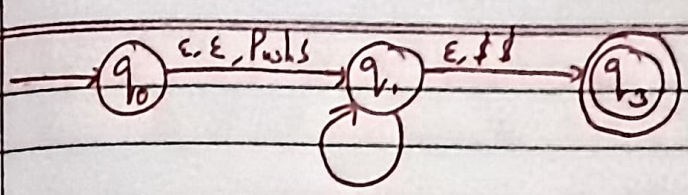
For any L, we can write a CFG that can generate all strings that are in L.
Any string w is generated by the CFG if $\exists$ a derivation $S \overset{*}{\Rightarrow} w$.

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

Intuition:-
→ The PDA begins by pushing S into the stack
→ The PDA pops the top variable, pick a random rule $A \to x$ (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 :- $S \to aTb \mid b$
$T \to Ta \mid \varepsilon$

$w = aaab \Rightarrow S \to aTb \to aTab \to aTaab \to aaab$
→ Push S onto the stack
→ Pop S and
  (i) Push b [ε, S, Push b]
  (ii) Push T [ε, ε, Push T]
  (iii) Push a [ε, ε, 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 × 3
→ Now stack is empty and input is read, accept w.



ε, S, Push aTb
ε, T, Push Ta
ε, S, Push b
ε, T, Pop
a, a, Pop
b, b, Pop

→ Multiple states, but loop back to q_1, as read by 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 $A \to x$ and return back to q_1
$\delta(q_1, \varepsilon, A) = (q_1, x)$

Pop a, i.e. $\delta(q_1, a, a) = (q_1, \varepsilon)$, matching the input string 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 is a CFL, $\exists p \in \mathbb{Z}, p \geq 1$ (pumping len) s.t.
$\forall s \in L$ s.t $|s| \geq p$, $\exists u, v, x, y, z \in \Sigma^*$ s.t $s = uvxyz$
(i) $|vy| \geq 1$
(ii) $|vxy| \leq p$
(iii) $uv^n xy^n z \in L \ \forall n \geq 0$

To show L is not a CFL

① Assume L is a CFL

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

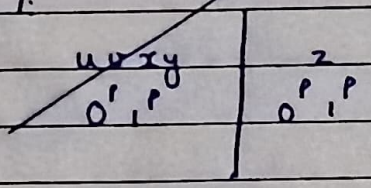
Assume that L is a CFL.

Suppose $\exists p$ for L.

Choose $w = 0^p 1^p 0^p 1^p$

Valid decomposition of w.

Case 1:



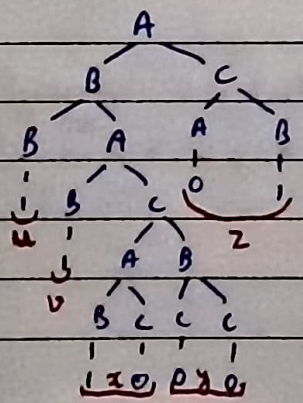
All CFLs satisfy the conditions of the pumping lemma.
In order to recognize very long strings, the model of computation must repeat certain steps.
This can be repeated to create longer and longer strings.

Ex:- $A \rightarrow Bc \mid 0$

$B \rightarrow BA \mid 1 \mid CC$

$C \rightarrow AB \mid 0$

Consider the derivation of $w = 11100001$



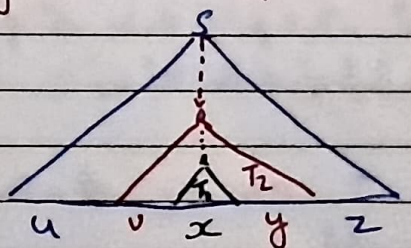
If we look at the longest path to a terminal, len $> |V|$ (: 3)
Then $\exists$ at least one variable that is repeated.

For example, A.
Replace the lowermost occurence of A with the middle occurence 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:-
Let L be a CFL and G be s.t $L = Lang(G)$ 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 atmost 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 pigeonhole 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 $w = uvxyz$



$h \leq |V|+1$

max

The ↑ no of terminals of the upper R at $h = |V|+1$
$= d^{|V|+1} \geq |uxy|$.
But we also established $p = d^{|V|+1}$
Hence $|uxy| \leq p$ ①

By replacing T_2 with T_1 i times $\forall i > 0$
$uv^i xy^i z$.

We replace T_2 with T_1 for $i=0$, and it still holds
Hence $uv^i xy^i z \in L$ $\forall i \geq 0$ ②

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

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

Thus, we've proved the pumping lemma.

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

Let p be the pumping length and $\omega = 0^p 1^p 2^p \in L$
$\omega = uvxyz$.
Since $|vxy| \leq p$, and no of 0's blw 0 & L is p
, vxy can have at most 2 of 3 symbols

$vxy = 0^k$ or 1^k or 2^k, $k \leq p$, then $\omega' \notin L$
Since no. of 0s ≠ no of 1s ≠ no of 2s.

$vxy = 0^m 1^n$ or $1^m 2^n$, $m+n \leq p$, then $\omega' \notin L$
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 $S' \to S_1 | S_2$

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

Closed under star $(S \to S_1 S | \varepsilon)$

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 $\{0^n 1^n\}$ 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 Morgans)

If L is a CFL & R is RL
$L \cup R$ is CFL (Union of CFLs cause RL $\subseteq$ CFL)
$L \cap R$ is also a CFL
↓ By constructing a Product PDA.
If the states of PDA P: $Q = (q_1, q_2, \ldots q_n)$ & DFA D: $Q' = \{d_1, d_2, \ldots d_n\}$
then the states of the Product PDA X: $Q = \{(q, d) \forall q \in Q, d \in Q'\}$

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 $\delta(q_i, \varepsilon, b) = (q_j, c)$ & $\delta(d_k, \varepsilon) = \phi$, then for

$$X: \delta((q_i, d_k), \varepsilon, b) = ((q_j, d_k), c)$$

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

$L(x) = L(P) \cup L(D)$, let final states (q_r, d_s)
such that q_r, 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 ~~consists~~ 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 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 L/R → Read a, write b, move L/R

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

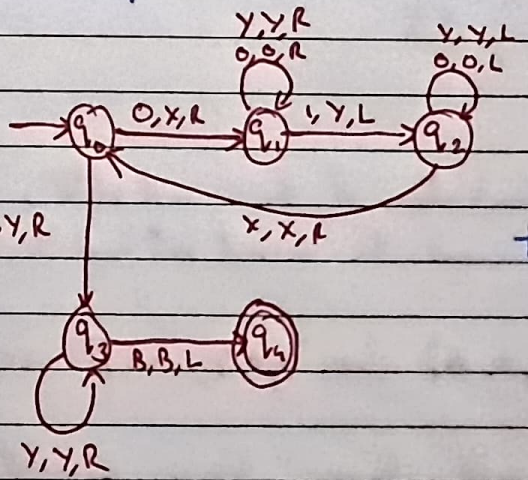
The TM might never reach a halting state and might loop forever.

⇓

the input string still does not belong to the language

E_1 :- $L = \{0^n 1^n \mid n \geq 1\}$ (is a CFL)

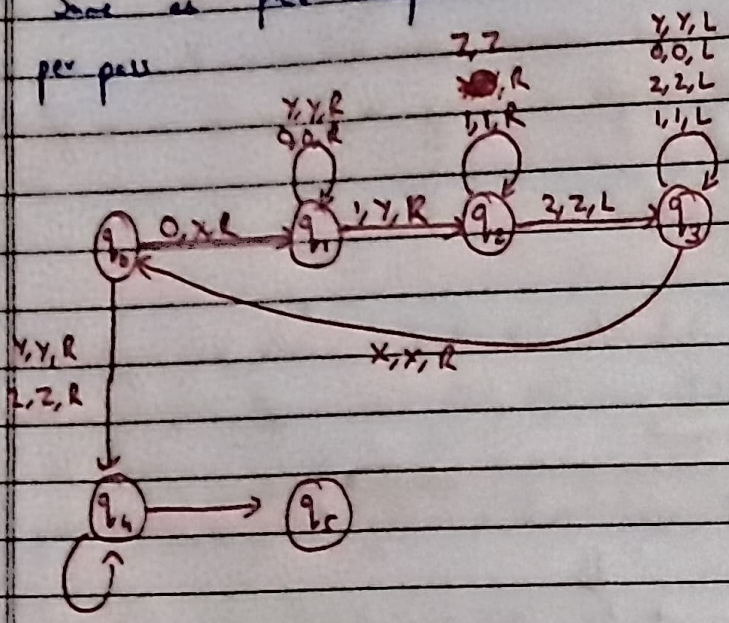
(i) Mark the first 0 with a symbol (say X)
(ii) Continue to move right until 1st 1 is encountered.
(iii) Mark the first 1 with a symbol (say Y)
(iv) Repeat until all 0s & 1s have been marked
(v) Accept if no 0s & 1s are left.



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

Ex:- $L = \{0^n 1^n 2^n \mid n \geq 1\}$

Same as prev example but mark with ● x, y & z per pass



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

Q is the finite set of states
Σ is the set of input alphabet
Γ is the tape alphabet where $\{B\} \subseteq \Gamma$ & $\Sigma \subseteq \Gamma$
$\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ 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 tape head location

Formally it is a triple (q, a, x) where $q \in Q$, $a \in \Gamma$, $x \in Z_+$

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

→ C_1 C_2 ...

step: [C_1, C_2 are configurations]

Defn:- A TM M accepts w if ∃ a seq of configurations C_1 to C_k where

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

A language L is recognized by a TM M st
$L(M) = \{w \mid M \text{ accepts } w\}$

Variations of Turing Machine Models.

A TM M_1 is equivalent to M_2 if M_1 can be simulated by M_2 and vice versa.

(i) Lazy Turing Machine:-
The head can move L/R or stay put S.

LTMs can simulate TMs trivially.

For the other direction, we add multiple transitions, where we read/write whatever we encounter and move back to the original position

(ii) k-tape TM:-

Again k-tape TMs can simulate TMs trivially.

For the other direction:
We use a spl symbol $ to demarcate the tapes. We add new symbols, each with an underscore beneath them to mark the

positions of the heads.

The TM makes two passes, first reading all the symbols under the virtual heads, and then a second pass to update the tapes according to this.

To make extra space, run an insertion-sort-esque algorithm to make space

(iii) Two way infinite TM:-

Can simulate TMs trivially

For the reverse, break the two way inf tape to a two tape TM which has been proved above.

(iv) Enumerators:-

TMs with a printer, uses the print output print tape to output strings. The language of E is the set of strings it prints out. If E does not halt, it may print out inf many strings in any order

Proof:- TM simulate Enumerators

(i) Lexicographically generate all binary strings sequentially. $\exists$ a OTO correspondence with Λ.

(ii) Number them as $s_1, s_2, \ldots$

(iii) for i in range $(1, \ldots)$:

for j in range $(1, i)$:

Run M with string s_j for i steps

If any string is accepted, print it

(ii) Enumerators simulate TM:-

(i) Compare all the output strings with the input string $w \rightarrow$ accept if same.

(v) Non deterministic TM :-

At any point in the computation, several possible configurations are available.

$$\delta_{NTM} : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

There are multiple branches at each configuration, where if one branch accepts, NTM accepts.

Trivially $TM \subseteq NTM$

For the converse :-

The TM can search through all possible configs of the NTM, but DFS will not work.

But an NTM can be simulated using three tapes

Tape 1 stores the input string

Tape 2 generates runs lexicographically (using an Enumerator)

Tape 3 simulates one branch of the config tree corresponding to the run taken.

And a 3-tape TM is equivalent to a TM.

While all variations are equivalent in power, they vary greatly in terms of efficiency.

P = Set of all problems where a DTM takes polynomial time in n

NP = Set of all problems where an NTM takes polynomial time in n

Total Turing Machines:- A Total TM is a TM that $\forall w \in \Sigma^*$, M(w) accepts or rejects, but never loops / runs infinitely.

On every input, M halts. $\begin{cases} \forall w \in L ; M(w) \text{ accepts} \\ \forall w \notin L ; M(w) \text{ rejects} \end{cases}$

Algorithms are TTMs

Class R

Decidable / Recursive Languages:- A language is said to be decidable if $\exists$ a TTM for it.

Recursively

Recognizable / Enumerable Languages:- A language is said to be recognizable if $\exists$ a TM M s.t.

$\forall w \in L$; M(w) accepts

$\forall w \notin L$; M(w) rejects or loops.

This is the RE class of languages

The class R is a subset of RE

$R \subset RE$

Co-recursively Enumerable Language:- A language is co-RE ($\overline{RE}$) if

$\forall w \in L$; M(w) accepts or loops

$\forall w \notin L$; M(w) rejects

$R = RE \cap \overline{RE}$

If language is in R, its complement is also in R

Formally any problem that is not decidable / recursive are not decidable.

There are problems in RE not in R

(11)

Partially Decidable Languages:- Belong to RE-R

Completely Undecidable Languages:- Belong to co-RE and the beyond.

Encoding:- Turing Machines can themselves be encoded as strings

TMs can accept multiple strings as input

Ex:- $M(\langle M_1, \omega \rangle) \rightarrow$ Accept if $M_1(\omega)$ accepts
$\rightarrow$ Reject if $M_1(\omega)$ rejects

M simulates M_1 on input ω.

Universal Turing Machines:- TMs that accept the description of a TM, and an input string, and it then simulates a run of the TM on the input string.

Recall that a TM is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

Any state could be encoded in binary where
$\langle q_0 \rangle = 0$, $\langle q_1 \rangle = 01$, etc.

Symbols in Σ & Γ can be encoded as
$\langle 0 \rangle = 0$, $\langle 1 \rangle = 1$, etc.

The directions $\langle L \rangle = 0$ & $\langle R \rangle = 1$.

The transition fn is just a k tuple of k finite binary strings

Our TM is now a 7-tuple of binary strings.

To combine all of these into one string

We claim that we can use a delimiter # and the following map

$0 \rightarrow 00$

$1 \rightarrow 01$

$\# \rightarrow 11$ (!)

By 0 in odd position, the symbol that follows is part of the substring, while a 1 in an odd position is a delim.

We have created an injective map from $TM \rightarrow \{0,1\}^*$

We use the Schröder-Bernstein Theorem to create an injection from $\{0,1\}^* \rightarrow TM$.

For every $k \in \{0,1\}^*$ (or Σ^*), create a TM M_k which just overwrites it's tape with k

This creates unique TMs for unique $k_s \in \{0,1\}^*$

Hence $\{0,1\}^* \rightarrow TM$.

Hence, we have a bijective map $\{0,1\}^* \rightleftharpoons TM$

Universal Turing Machine:-

A UTM, denoted as U_{TM} accepts as input

(i) the encoding of a TM M

(ii) an input string w

and simulates M running over w.

$U_{TM}(\langle M, w\rangle) = \begin{cases} \text{Accept} & \text{; if } M(w) \text{ accepts} \\ \text{Reject} & \text{; if } M(w) \text{ rejects} \\ \text{Loop} & \text{; if } M(w) \text{ loops} \end{cases}$

(Un) Decidability :-

Examples of Decidable languages

(i) $A_{DFA} = \{\langle DFA, w\rangle \mid w \in L(DFA)\}$
- ↳ Simulate DFA on w
- ↳ Accept if accept, reject if reject

(ii) $E_{DFA} = \{\langle DFA\rangle \mid L(DFA) = \phi\}$
- ↳ Mark start state
- ↳ Perform BFS
- ↳ If final state is marked, reject, else accept

(iii) $A_{CFG} = \{\langle CFG, w\rangle \mid w \in L(CFG)\}$
- ↳ Convert to CNF
- ↳ List all derivations of $2|w|-1$ steps
- ↳ Compare and search for match.
- ↳ Accept on match, reject else.

Undecidability :-

$A_{TM} = \{\langle M, w\rangle \mid M \text{ accepts } w\}$

Does ∃ a TTM A_{TM} that accepts if M accepts w, and rejects if M rejects or loops.

$A(w, w) = \begin{cases} \text{Accepts} & \text{; if } M_w(\langle M_w\rangle) \text{ accepts} \\ \text{Rejects} & \text{else.} \end{cases}$

We prove that A_{TM} does not exist by contradiction.

We construct a TTM D that accepts an input w and uses A as a subroutine to simulate $A(\langle w, w \rangle)$

D(w) = Run $A(\langle w, w \rangle)$ and negate the output

(or)

$D(\langle M_w \rangle)$ = Run $M_w(\langle M_w \rangle)$ and negate the output

Now if we run $D(\langle D \rangle)$.
We reach a contradiction (accept if reject & vice versa)

Hence D is not a TTM $\longrightarrow$ A is not a TTM

A_{TM} is in class RE
- ↳ Accept if Accept
- ↳ Reject if Reject
- ↳ Loop otherwise.

U_{TM} recognizes A_{TM}