HASIFA A S
1RV18IS016
IV Sem

THEORY OF COMPUTATION

PAPER I

Nov/Dec - 19

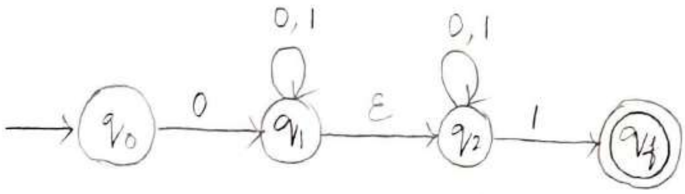
PART - A

1) (1.1) ~~*~~bab

(1.2) $(1^*01^*01^*01^*)^*$

(1.3) L_1 = Irregular
L_2 = Irregular
L_3 = Regular
L_4 = Irregular

(1.4) $0(0+1)^*1$



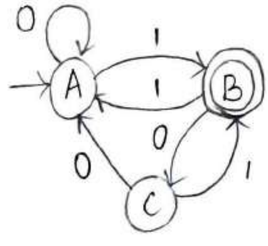
(1.5) Left derivation,

$S \to aaB$
$aab \quad \because B \to b$

$S \to AB$
$AaB \quad \because A \to Aa$
$aaB \quad \because A \to a$
$aab \quad \because B \to b$

∴ ~~Non~~ Ambiguous grammar
since two unique left derivations are possible from the same grammar.

(1.6) DFA $D = (\{A, B, C\}, \{0, 1\}, \delta, A, B)$



Grammar, $A \to 0A \mid 1B$
$B \to 0C \mid 1A \mid \varepsilon$
$C \to 0A \mid 1B$

PDA $P = (\{q\}, \{0,1\}, \Sigma, \{q\}, z_0, \delta)$, using empty stack

$\delta(q, \varepsilon, z_0) = (q, Az_0) \cdot \{(q, 0Az_0), (q, 1Bz_0)\}$

$\delta(q, \varepsilon, A) = \{(q, 0Az_0), (q, 1Bz_0)\}$

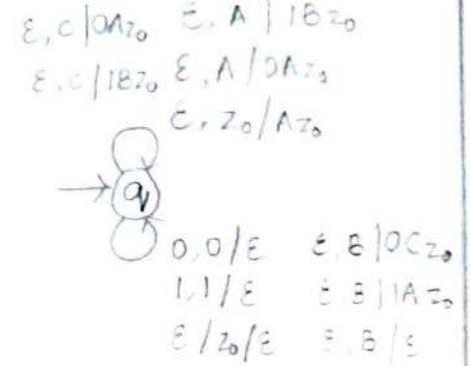
$\delta(q, \varepsilon, B) = \{(q, 0Cz_0), (q, 1Az_0), (q, \varepsilon)\}$

$\delta(q, \varepsilon, C) = \{(q, 0Az_0), (q, 1Bz_0)\}$

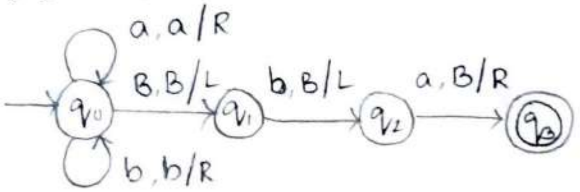
$\delta(q, 0, 0) = (q, \varepsilon)$

$\delta(q, 1, 1) = (q, \varepsilon)$

$\delta(q, \varepsilon, z_0) = (q, \varepsilon) \Rightarrow$ End state.



(1.7) $L = \{(a+b)^n ab \; ; \; n \geq 0\}$



(1.8) Deterministic

$\cancel{\delta(q, X)}$ $\delta(q \times X) \rightarrow q \times X \times \{\text{Left, Right, Stay}\}$

(1.9) Closure and decision.

(1.10)

Old V	New V	Productions
ϕ	S, A, D	$S \rightarrow a$ $A \rightarrow \varepsilon$ $D \rightarrow dd$
S, A, D	S, A, B, D	$S \rightarrow aA$ $B \rightarrow Aa$
S, A, B, D	S, A, B, D	$S \rightarrow B$ $A \rightarrow aB$
S, A, B, D	S, A, B, D	—

P'	T'	V'
—	—	S
$S \rightarrow a \mid aA \mid B$	a	S, A, B
$A \rightarrow aB \mid \varepsilon$	a, ε	S, A, B
$B \rightarrow Aa$	a	S, A, B

$S \rightarrow a \mid aA \mid B$

$A \rightarrow aB \mid \varepsilon$

$B \rightarrow Aa$

(1.11) i) A recursive language L is a formal language for which there exists a Turing machine that will halt and accept an input string in L, and halt and reject otherwise.

ii) A language is recursively enumerable if some Turing machine accepts it

Let L be a recursively enumerable language and M the Turing machine that accepts it

For string w,

If $w \in L$ then M halts in a final ~~state~~ state.

If $w \notin L$ then M halts in a non-final state and loops forever

(1.12) When a problem A is polynomial time reducible to a problem B, it means that given an instance of A, there is an algorithm for transforming instances of A into instances of B. This is often done to derive hardness results: if there was a fast algorithm for some problem, there would also be ~~so~~ a fast algorithm for some other problem.

(1.13) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Equal number of a's and b's

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

(1.14) A PDA is deterministic if there is never a choice of move in any situation. These choices are of two types.

For PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ to be deterministic the following conditions should hold,

(i) $\delta(q, a, X)$ has atmost one member for any q in Q, $a \in \Sigma$ or $a = \varepsilon$ and X in Γ.

(ii) If $\delta(q, a, X)$ is non-empty for some a in Σ, then $\delta(q, \varepsilon, X)$ is empty.

PART - B

2) (a) Pumping lemma for Regular languages:

Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

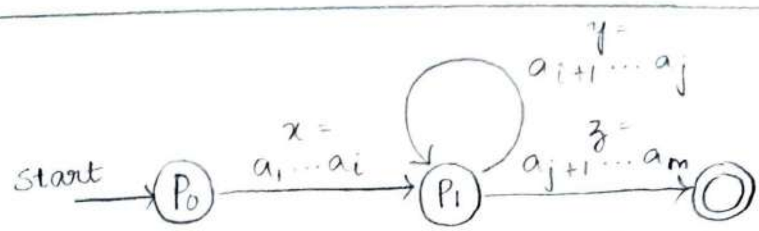
That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k=0$), keeps the resulting string in the language L.

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1 a_2 \dots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \dots, n$ define state p_i to be $\delta(q_0, a_1 a_2 \dots a_i)$ where δ is the transition function of A, and q_0 is the start state of A. This is, p_i is the state A is in after reading the first i symbols of w. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n+1$ different p_i's for $i = 0, 1, \dots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1 a_2 \dots a_n$
2. $y = a_{i+1} a_{i+2} \dots a_j$
3. $z = a_{j+1} a_{j+2} \dots a_m$

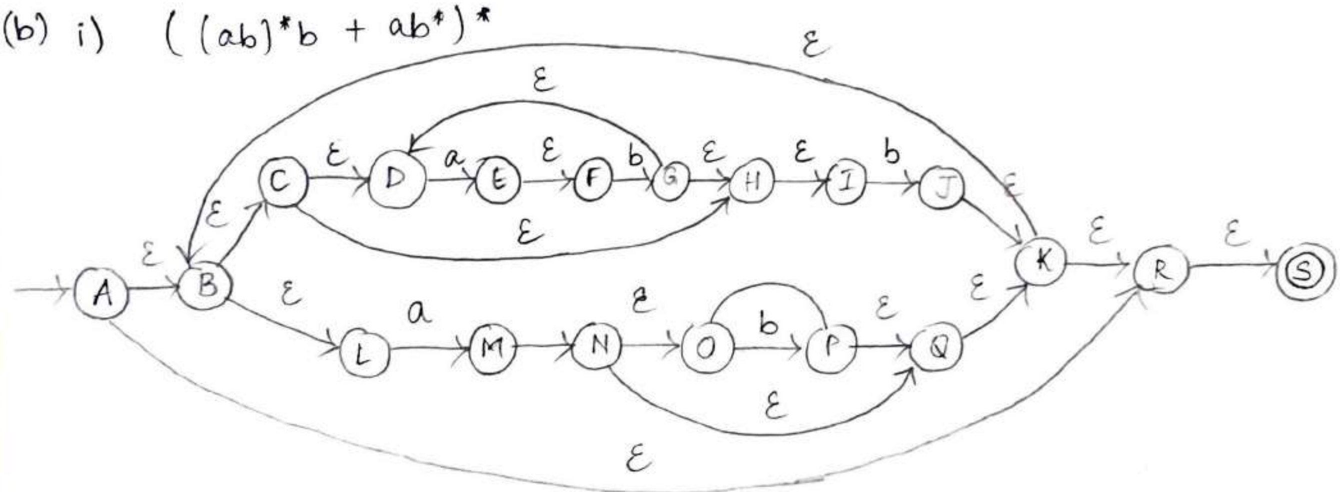
That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w. The relationships among the strings and states are suggested by the state diagram. Note that x may be empty, in the case that $i=0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i ~~is~~ $< j$.



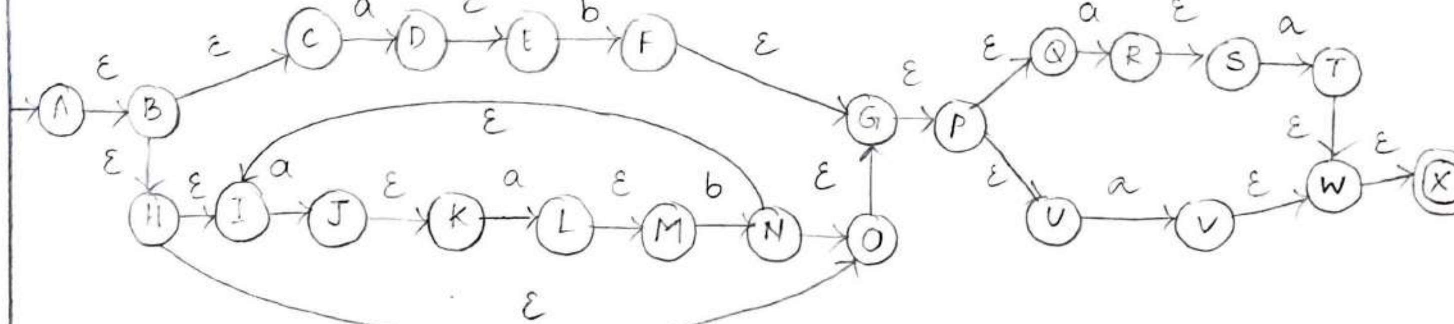
Now, consider what happens if the automaton A receives the input xy^kz for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x. Since p_i is also p_j, it must be that A goes from p_i to the accepting state shown in the state diagram on input z. Thus, A accepts ~~a~~ xz.

If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z. Thus, for any $k \geq 0$, xy^kz is also accepted by A; that is, xy^kz is in L.

(b) i) $((ab)^*b + ab^*)^*$

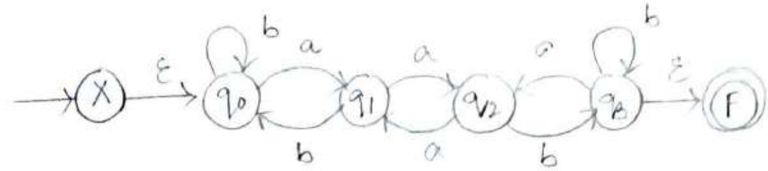


ii) $(ab + (aab)^*)(aa + a)$



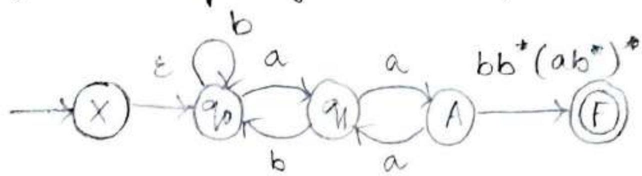
(c)

Step 1:



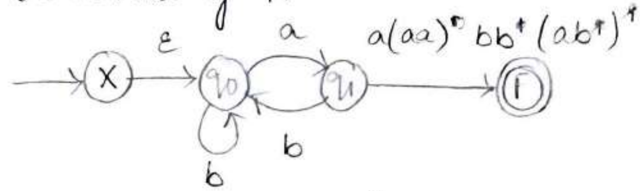
Removal of incoming edge on initial state.
Removal of incoming edge on the final state.

Step 2:

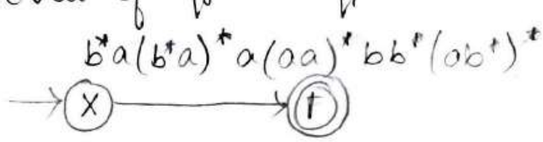


Eliminating q_2

Step 3: Eliminating A



Step 4: Removal of q_0 and q_1



Regular Expression : $b^*a(b^*a)^*a(aa)^*bb^*(ab^*)^*$

3) (a) Given CNF, $S \rightarrow AaA \mid CA \mid BaB$

$A \rightarrow aaBa \mid CDA \mid aa \mid DC$

$B \rightarrow bB \mid bAB \mid bb \mid aS$

$C \rightarrow Ca \mid bC \mid D$

$D \rightarrow bD \mid \varepsilon$

Step 1: Eliminating start symbol from RHS

$S_0 \rightarrow S$

Step 2: ~~A~~ Eliminating nullable variables.

Nullable variables = { D, C, A, S }

$S \rightarrow aA \mid Aa \mid BaB \mid C \mid A$

$A \rightarrow aaBa \mid CD \mid CA \mid DA \mid aa \mid C \mid D$

$B \rightarrow bB \mid bAB \mid bb \mid a \mid aS$

$C \rightarrow Ca \mid a \mid b \mid bC \mid D$

$D \rightarrow bD \mid b \mid D$

Step 3 : Eliminating unit production

unit productions :- $S \to C$, $S \to A$

$A \to C$, $A \to D$

$C \to D$

$D \to D$

Hence, $D \to bD \mid b$

$C \to Ca \mid a \mid b \mid bC \mid bD \mid b$

$B \to bB \mid bAB \mid bb \mid a \mid aS$

$A \to aaBa \mid CD \mid CA \mid DA \mid aa \mid Ca \mid a \mid b \mid bC \mid bD \mid b$

$S \to aA \mid Aa \mid BaB \mid aaBa \mid CD \mid CA \mid DA \mid aa \mid Ca \mid a \mid bc \mid bD \mid b$

Step 4 :

CNF :- $S \to ZA \mid XB \mid WX \mid CD \mid CA \mid DA \mid W \mid CZ \mid a \mid VC \mid VD \mid b$

$Z \to a$

$X \to Ba \mid BX'$

$X' \to Z$

$W \to aa \mid ZZ$

$V \to b$

$A \to WX \mid CD \mid CA \mid DA \mid W \mid CZ \mid a \mid b \mid VC \mid VD \mid b$

$B \to VB \mid VV \mid a \mid ZS$

$C \to CZ \mid a \mid b \mid VC \mid VD \mid b$

$D \to VD \mid b$

(b) $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

Left factoring,

$SaaS \to S'$

$aS \to X$

CNF after left factoring,

$S \to bSS' \mid a$

$S' \to SaX$

$X \to aS \mid Sb$

(c) i) $L = \{uvwv^R \ ; \ u, v, w \in \{a,b\}^+, \ |v| = |w| = 2\}$

$S \rightarrow AB$

$B \rightarrow aBa \mid bBb \mid aAa \mid bAb$

$A \rightarrow aa \mid ab \mid ba \mid bb$

ii) $L = \{a^n b^m \ ; \ n \leq m+3\}$

$S \rightarrow AAAB$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow aBb \mid Bb \mid \varepsilon$

4) (a) Left Recursion: A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS. A grammar containing a production having left recursion is called as Left Recursive grammar.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$A \rightarrow A\alpha \mid \beta \quad \left\{ \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array} \right.$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

Hence, $T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

(b) $S \rightarrow aAS \mid a \mid SS$

$A \rightarrow SbA \mid ba$

String: aabbaa

i) Left most derivation.

$S \rightarrow aAS$

$aSbAS \quad \because A \rightarrow SbA$

$aabAS \quad \because S \rightarrow a$

$aabbaS \quad \because A \rightarrow ba$

$aabbaa \quad \because S \rightarrow a$

ii) Right most derivation,

$S \rightarrow aAS$

$aAa \quad \because S \rightarrow a$

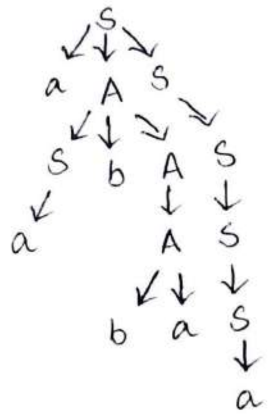
$aSbAa \quad \because A \rightarrow Sba$

$aSbbaa \quad \because A \rightarrow ba$

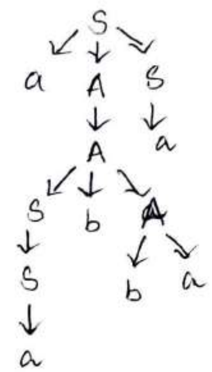
$aabbaa \quad \because S \rightarrow a$

iii) Derivation trees

Left derivation tree,



Right derivation tree,



(c) Context Free Grammar (CFG) is of great practical importance. It is used for following purposes –

- For defining programming languages.
- For parsing the program by constructing syntax tree
- For translation of programming languages.
- For describing arithmetic expressions.
- For construction of compilers.
- Simplicity of proofs:

There are plenty of proofs around context-free grammars, including reducability and equivalence of automata. Those are the simpler the most restricted set of grammars you have to deal with. Therefore, normal forms can be helpful here.

As a concrete example, Greibach normal form is used to show (constructively) that there is an ε-transition-free PDA for every CFL (that does not contain ε).

- They are used in an essential part of the Extensible Markup Language (XML) called the Document Type Definition.

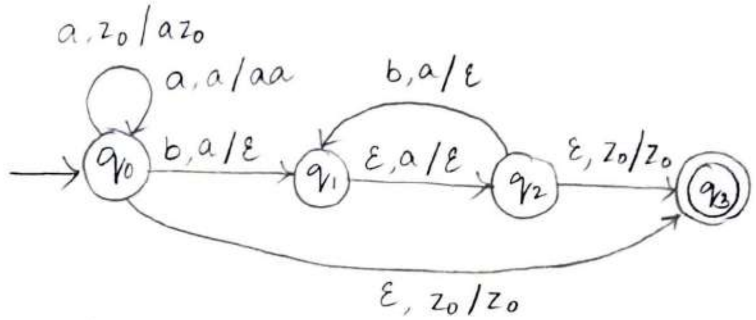
• Enables parsing

While PDAs can be used to parse words with any grammar, this is often inconvenient. Normal forms can give us more structure to work with, resulting in easier parsing algorithms.

As a concrete example, the CYK algorithm uses Chomsky Normal form. Greibach normal form, ~~so~~ on the other hand, enables recursive-descent parsing; even though backtracking may be necessary, space complexity is linear.

5) (a)

$L = \{ a^{2n} b^n \; ; \; n \geq 0 \}$



String: aaaabb

$(q_0, aaaabb, z_0) \vdash (q_0, aaabb, az_0)$
$\vdash (q_0, aabb, aaz_0)$
$\vdash (q_0, abb, aaaz_0)$
$\vdash (q_0, bb, aaaaz_0)$
$\vdash (q_1, b, aaaz_0)$
$\vdash (q_2, b, aaz_0)$
$\vdash (q_1, \varepsilon, az_0)$
$\vdash (q_2, \varepsilon, z_0)$
$\vdash (q_3, \varepsilon, z_0)$ final state

(b) Closure properties of CFL

Under Intersection:

If L1 and L2 are two context free languages, their intersection $L1 \cap L2$ need not be context free. For example,

$L1 = \{ a^n b^n c^m \mid n \geq 0 \text{ and } m \geq 0 \}$ and

$L2 = \{ a^m b^n c^n \mid n \geq 0 \text{ and } m \geq 0 \}$

$L3 = L1 \cap L2 = \{ a^n b^n c^n \mid n \geq 0 \}$ need not be context free

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by push down automata, hence not context free.

Therefore, CFLs are not closed under Intersection.

(c) $P = (\{q_0, q_1, q_f\}, \{a, b\}, \{A, z_0\}, \delta, q_0, z_0, \phi)$

CFL $G = (V, T, P, S)$

PDA by empty stack. Starting move,

$S \rightarrow [q_0 z_0 q_0] \,/\, [q_0 z_0 q_1] \,/\, [q_0 z_0 q_f]$

Erasing moves,

$\delta(q_0, b, A) = (q_1, \varepsilon)$

$[q_0 A q_1] \rightarrow b$

$\delta(q_1, b, A) = (q_1, \varepsilon)$

$[q_1 A q_1] \rightarrow b$

Non-Erasing moves,

$\delta(q_0, a, z_0) = (q_0, Az_0)$

$[q_0 z_0 q_0] \rightarrow a\, [q_0 A q_0][q_0 z_0 q_0]$

$[q_0 z_0 q_0] \rightarrow a\, [q_0 A q_1][q_0 z_0 q_0]$

$[q_0 z_0 q_0] \rightarrow a\, [q_0 A q_f][q_f z_0 q_0]$

$[q_0 z_0 q_1] \rightarrow a\, [q_0 A q_0][q_0 z_0 q_1]$

$[q_0 z_0 q_1] \rightarrow a\, [q_0 A q_1][q_1 z_0 q_1]$

$[q_0 z_0 q_1] \rightarrow a\, [q_0 A q_f][q_f z_0 q_1]$

$[q_0 z_0 q_f] \rightarrow a\, [q_0 A q_0][q_f z_0 q_f]$

$[q_0 z_0 q_f] \rightarrow a\, [q_0 A q_1][q_f z_0 q_f]$

$[q_0 z_0 q_f] \rightarrow a\, [q_0 A q_f][q_f z_0 q_f]$

$\delta(q_0, a, A) = (q_0, AA)$

$[q_0 A q_0] \rightarrow a\, [q_0 A q_0][q_0 A q_0]$

$[q_0 A q_0] \rightarrow a\, [q_0 A q_1][q_1 A q_0]$

$[q_0 A q_0] \rightarrow a\, [q_0 A q_f][q_f A q_0]$

$[q_0 A q_1] \rightarrow a\, [q_0 A q_0][q_0 A q_1]$

$[q_0 A q_1] \rightarrow a\, [q_0 A q_1][q_1 A q_1]$

$[q_0 A q_1] \rightarrow a\, [q_0 A q_f][q_f A q_1]$

$$[q_0 A q_f] \rightarrow a\,[q_0 A q_0][q_0 A q_f]$$
$$[q_0 A q_f] \rightarrow a\,[q_0 A q_1][q_1 A q_f]$$
$$[q_0 A q_f] \rightarrow a\,[q_0 A q_f][q_f A q_f]$$

Acceptance by empty stack,

$\delta(q_0, \varepsilon, z_0) = (q_f, z_0)$

$$[q_1 z_0 q_f] \rightarrow \varepsilon\,[q_1 z_0 q_f]$$

6) (a) Languages accepted by PDA :

i) Acceptance by final state

Let $P = (Q, \Sigma, \tau, \delta, q_0, z_0, F)$ be a PDA ~~For some state q in F~~ ~~for any~~ Then $L(P) = \{w \mid (q_0, w, z_0) \vdash_P^w (q, \varepsilon, \alpha)\}$ For some state q in F for any stack symbol α that is starting with the initial ID w waiting on the input. PDA P consumes w from the input and enters an accepting state. Contents of the stack at that time is irrelavant.

ii) Acceptance by empty stack.

Let $P = (Q, \Sigma, \tau, \delta, q_0, z_0, F)$ be a PDA. Then $N(P) = \{w \mid (q_0, w, z_0) \vdash^w (q, \varepsilon, \varepsilon)\}$ Here $N(P)$ is a set of inputs that a machine P can consume and at the same time empty its stack. Here the set of accepting states is irrelavant.

The stack shouldn't even contain z_0.

(b) Conversion of CFG to PDA :

Let $G = (V, T, q, S)$ be a CFG. PDA P accepts the CFG $L(G)$ by empty stack is as follows.

1. When you do not have any input symbol

For start variable, $\delta(q, \varepsilon, z_0) = (q, S z_0)$

2. For each variable,

$\delta(q, \varepsilon, A) = \{(q, \beta), (q, c)\}$

if $A \rightarrow \beta$, $A \rightarrow c$ productions exist

3. For each terminal a,

$\delta(q, a, a) = \{(q, \varepsilon)\}$

4. & Final transition,

$\delta(q, \varepsilon, z_0) = (q, \varepsilon)$

$S \to aABB \mid aAA$

$A \to aBB \mid a$

$B \to bBB \mid A$

$C \to a$

PDA $P = (Q\{q\}, \{a,b\}, \{a,b,A,B,Z_0\}, q, Z_0, \delta, \phi)$

$\delta(q, \varepsilon, Z_0) = (q, SZ_0)$

$= (q, aABBZ_0), (q, aAA)$

$\delta(q, \varepsilon, A) = (q, aBBZ_0), (q, aZ_0)$

$\delta(q, \varepsilon, B) = (q, bBBZ_0), (q, AZ_0)$

$= (q, bBBZ_0), (q, aBBZ_0), (q, aZ_0)$

$\delta(q, a, a) = (q, \varepsilon)$

$\delta(q, b, b) = (q, \varepsilon)$

(c) $L = \{a^n b^n c^n \mid n \geq 0\}$

Let us assume that L is context free, then by Pumping lemma the following rules hold good for an integer n such that for all $x \in L$ ~~(1)~~ with $|x| \geq n$, there exists $u, v, w, x, y \in \Sigma^*$, such that $x = uvwxy$, and

(1) $|vwx| \leq n$ (2) $|vx| \geq 1$ (3) for all $i \geq 0$, $uv^iwx^iy \in L$

For L, if (1) and (2) hold then $x = a^n b^n c^n = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$.

(1) tells that vwx does not contain both a and c. Thus either vwx has no a's, or vwx has no c's. Remaining two more cases, Suppose vwx has no a's. By (2), vx contains a 'b' or a 'c'. Thus uwy has 'n' a's and uwy either has less than 'n' b's or has less than 'n' c's.

But (3) tells that $uwy = uv^0wx^0y \in L$.

So, uwy has an equal number of a's, b's and c's gives a contradiction. The case where vwx has no c's is similar and also gives a contradiction.

Thus, L is not context-free.

7) (a)

$$f(x) = \begin{cases} x/2 , & x \text{ is even} \\ x+1/2 , & x \text{ is odd} \end{cases}$$

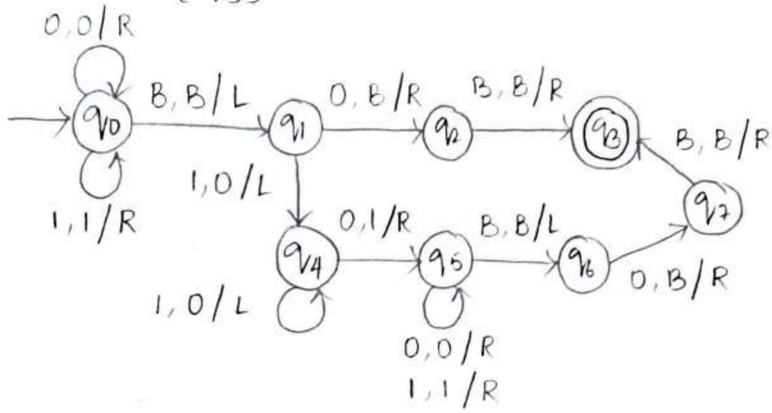
Turing machine $T = (Q, \Sigma, \tau, \delta, q_0, B, F)$ is given by,

$Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$

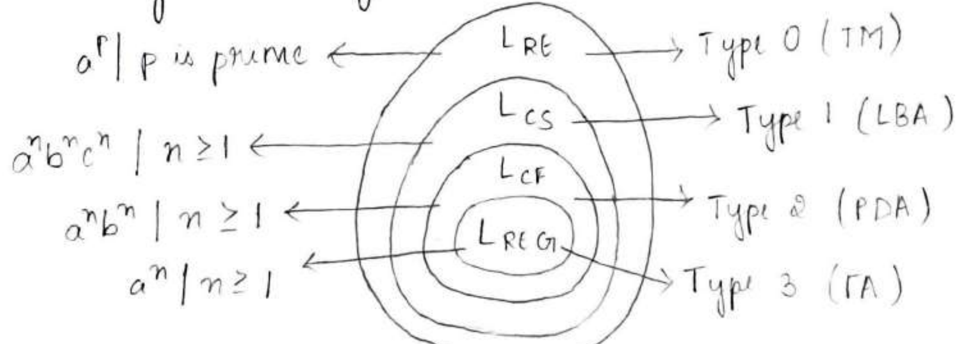
$\Sigma = \{0, 1\}$

$\tau = \{B, 0, 1\}$

$F = \{q_3\}$



(b) Chomsky Hierarchy



1. Type 0 Grammar

A grammar $G = (V, T, P, S)$ is said to be Type 0 (or) Unrestricted (or) Phrase structured grammar. If all productions are of form $\alpha \to \beta$ where $\alpha \in (V \cup T)^+$ and $\beta \in (V \cup T)^*$

Ex: $S \to aAb \,/\, \varepsilon$

$aA \to bAA$

$bA \to a$

2. Type 1 Grammar

A grammar $G = (V, T, P, S)$ is said to be Type 1 (or) Context - free sensitive if all productions are of type $\alpha \to \beta$ here there is a restriction in β, where the length of β should be atleast the length of α i.e. $|\beta| \geq |\alpha|$ and $\alpha, \beta \in (V \cup T)^+$

Ex : $S \rightarrow aAb$

$aA \rightarrow bAA$

$bA \rightarrow aa$

3. Type 2 Grammar

A grammar $G = (V, T, P, S)$ is said to be Type 2 (or) Context - free grammar if all the productions are of the form $A \rightarrow \alpha$ where $\alpha \in (V \cup T)^*$ i.e. ε can be in RHS

Ex : $S \rightarrow aB \mid bA \mid \varepsilon$

$A \rightarrow aA \mid b$

$B \rightarrow bB \mid a \mid \varepsilon$

4. Type 3 Grammar

A grammar $G = (V, T, P, S)$ is said to be Type 3 or regular if the grammar is either rightly linear or left linear.

A grammar is said to be right linear if

$A \rightarrow wB$ or $A \rightarrow w$

Ex : $S \rightarrow aaB \mid bbA \mid \varepsilon$

$A \rightarrow aA \mid b$

$B \rightarrow bB \mid a \mid \varepsilon$

A grammar is said to be left linear if

$A \rightarrow Bw$ or $A \rightarrow w$

Ex : $S \rightarrow Baa \mid Abb \mid \varepsilon$

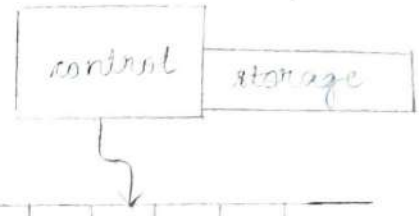
$A \rightarrow Aa \mid b$

$B \rightarrow Bb \mid a \mid \varepsilon$

(c) Every language accepted by a multi-tape Turing Machine is recursively enumerable.

PROOF : Suppose language L is accepted by a k-tape TM M. We simulate M with a one-tape TM. N whose tape we think of as having 2k tracks. Half these tracks hold the tapes of M; and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure assumes k = 2. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1 and track 3 holds the position of the second tape head

Simulation of a two-tape Turing machine by a one tape turing machine.



Track 1	...				x			...
Track 2	...	A_1	A_2	...	A_i	...		...
Track 3	...						x	...
Track 4	...	B_1	B_2	...	B_i	...	B_j	...

To simulate a move of M, N's head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control.
Thus, N knows what move M will make.
N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M.
We select as N's accepting states all those states that record M's ~~start~~ state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts and M does not accept otherwise.

8)(a) 3-SAT is polynomial time reducible to CLIQUE.

PROOF:

Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$

We reduce this boolean formula into an undirected graph. G
This is done by grouping the nodes in G into k groups of three nodes, each called a triple, $t_1, \ldots, t_k$.
Each of these triples corresponds to one of the clauses in the formula.
Each individual node within a triple corresponds to a literal in the corresponding clause.

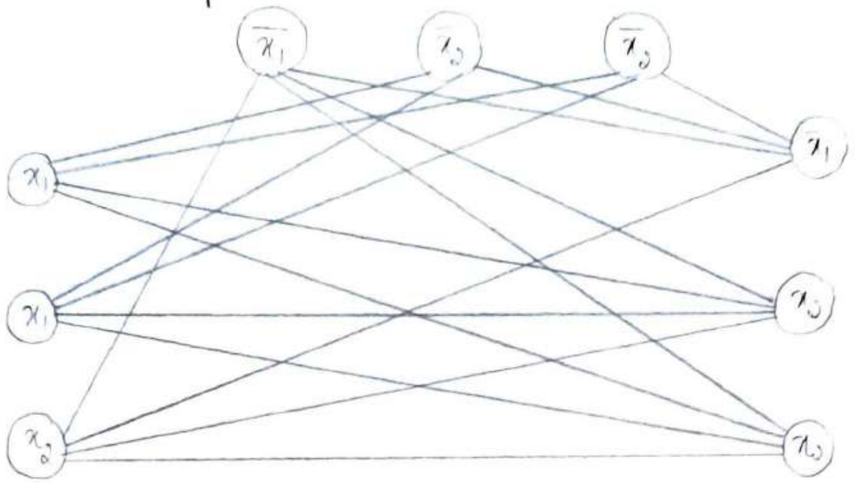
In the resulting graph G, all nodes are connected by an edge except:

- between nodes in the same triple
- between complementary nodes.

Boolean formula,

$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

converted to Graph,



We must show that a Boolean formula is satisfiable iff G has a k-clique. Suppose the Boolean formula is satisfiable. In the satisfying assignment at least one literal in each clause is true, so we select that node corresponding mode in each triple of the graph G.

- If more than one literal is true, pick one arbitrarily.

All nodes selected form a k-clique since we choose one from each of the k triples. Each pair of selected nodes is:

- Joined by an edge, because no pair fits one of the exceptions previously mentioned.
- Not from the same triple, because only one node was selected from each triple.
- Not conflicting labels, because the associated literals were both true in the assignment.

Therefore G contains a k-clique.

(b) Primitive Recursive functions.

They define a set of functions that contain only computable functions, using only basic operations (ex: the operation "add 1"), and basic ways of putting functions together (ex: composition). The model is as simple as possible and guarantees that all functions generated are computable.

A function $f(x_1, \dots, x_n)$ is primitive recursive if either:

1. f is the function that is always 0, i.e. $f(x_1, \dots, x_n) = 0$; This is denoted by Z when the number of arguments is understood. This rule for deriving a primitive recursive function is called the Zero rule.
2. f is the successor function, i.e. $f(x_1, \dots, x_n) = x_i + 1$; This rule for deriving a primitive recursive function is called the Successor rule.
3. f is the projection function, i.e. $f(x_1, \dots, x_n) = x_i$; This is denoted by Π_i when the number of arguments is understood. This rule for deriving a primitive recursive function is called the Projection Rule.
4. f is defined by the composition of primitive functions, i.e. if $g_1(x_1, \dots, x_n), g_2(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n)$ are primitive recursive and $h(x_1, \dots, x_k)$ is primitive recursive, then
$$f(x_1, \dots, x_n) = h(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$$
is primitive recursive. This rule for deriving a primitive recursive function is called the Composition rule.
5. f is defined by recursion of two primitive recursive functions, i.e., if $g(x_1, \dots, x_{n-1})$ and $h(x_1, \dots x_{n+1})$ are primitive recursive then the following function is also primitive recursive.
$$f(x_1, \dots, x_{n-1}, 0) = g(x_1, \dots, x_{n-1})$$
$$f(x_1, \dots, x_{n-1}, m+1) = h(x_1, \dots, x_{n-1}, m, f(x_1, \dots, x_{n-1}, m))$$

This rule for deriving a primitive recursive function is called the Recursion rule. It is very powerful rule and is why these functions are called 'primitive ~~functions~~ recursive' functions.