ALC Assignment II.

I. What are the normal forms of CFG's? What are the eliminations or simplifications to be made to the grammar before converting it into a CNF?

Normal forms of CFG's: Once a grammar is simplified i.e all unnessary symbols are eliminated then the grammar can be converted into Normal forms. There are 2 types of normal forms for CFG's.

1. **CNF:** (Chomsky Normal form.)

Any CFG without ϵ can be generated by a grammar in which all the productions are of the form:

$$NT \longrightarrow NT\ NT \quad (or)$$

$$NT \longrightarrow T$$

where NT is a NonTerminal and T is a terminal.

Eg:
$$A \longrightarrow BC$$
$$A \longrightarrow a$$
$$B \longrightarrow b$$
$$C \longrightarrow c$$

The above grammar is in CNF.

2. **GNF:** (Greibach Normal form). To convert a given CFG to GNF it is first converted into CNF. The CNF is then converted to GNF. All productions of GNF are of the form $A \longrightarrow a\alpha$ where A is an NT, 'a' is a T and α is a string of variables which may be empty. Eg: $A \longrightarrow a$ or $A \longrightarrow aB \mid b$, $B \longrightarrow bA$

Simplifications: To optimize CFG's, i.e to remove unnecessary symbols from a CFG, simplifications are made to the grammar. These simplifications reduce the length of the grammar and increase the efficiency. There are 3 simplifications or elimations that can be done to a grammar before converting it to Normal forms CNF/GNF

1. Elimination of ϵ-productions or Null productions:
   i.e productions of the form $A \rightarrow \epsilon$.
2. Elimination of Unit productions:
   i.e productions of the form $A \rightarrow B$
3. Elimination of Useless symbols: If a symbol X is not derivable from the starting symbol S or if X in turn does not derive a terminal then X is useless.

a) 1. Remove unit prod's:

$S \rightarrow AA$, $A \rightarrow B \mid BB$, $B \rightarrow abB \mid b \mid bb$

Unit prod's

$A \rightarrow B$

Non Unit prod's

$S \rightarrow AA$

$A \rightarrow BB$

$B \rightarrow abB \mid b \mid bb$

$A \rightarrow B \rightarrow abB \mid b \mid bb$ $\therefore$ $A \rightarrow abB \mid b \mid bb$

$\therefore$ The final grammar is:

$S \rightarrow AA$

$A \rightarrow BB \mid abB \mid b \mid bb$

$B \rightarrow abB \mid b \mid bb$

2. Eliminate ϵ-prods from:

$$S \rightarrow ABa \mid bC$$
$$A \rightarrow BC \mid b$$
$$B \rightarrow b \mid \epsilon$$
$$C \rightarrow c \mid \epsilon$$

B and C are directly nullable. If we substitute B, C in A, A also becomes null. ∴ Substituting ϵ for A, B, C in the above grammar we get,

$$S \rightarrow a \mid Ba \mid Aa \mid b$$
$$A \rightarrow B \mid C \mid b$$

We make a union of these additional productions with the given grammar and eliminate directly nullable productions and get,

$$S \rightarrow ABa \mid bC \mid a \mid Ba \mid Aa \mid b$$
$$A \rightarrow BC \mid b \mid B \mid C$$
$$B \rightarrow b$$
$$C \rightarrow c$$

This is the final grammar with no ϵ productions.

3. Remove useless symbols from:

$S \longrightarrow aAa$, $A \longrightarrow Sb \mid bCC \mid DaA$

$C \longrightarrow abb \mid DD$, $E \longrightarrow aC$, $D \longrightarrow aDA$

Since D does not derive a terminal D is useless.

Eliminating D we get,

$S \longrightarrow aAa$

$A \longrightarrow Sb \mid bCC$

$C \longrightarrow abb$

$E \longrightarrow aC$

Now since E is not derivable from S E is useless. Eliminating E we get,

$S \longrightarrow aAa$

$A \longrightarrow Sb \mid bCC$

$C \longrightarrow abb$

b) Convert the following grammars to CNF.

1. $S \longrightarrow aB \mid ab$

$A \longrightarrow aAB \mid a$

$B \longrightarrow ABb \mid b$

CNF is of the form

$NT \longrightarrow NTNT$

and $NT \longrightarrow T$

There are no ϵ productions, no Unit pairs and no useless symbols. Replacing all terminals with NT's we get, and adding new pairs we get,

$S \longrightarrow C_aB \mid C_aC_b$

$A \longrightarrow C_aAB \mid a$

$B \longrightarrow ABC_b \mid b$

$C_a \longrightarrow a$, $C_b \longrightarrow b$

$A \longrightarrow C_aAB$

$A \longrightarrow C_aD_1$

$D_1 \longrightarrow AB$

$B \longrightarrow ABC_b$

$B \longrightarrow AD_2$

$D_2 \longrightarrow BC_b$

The final grammar in CNF is,

$S \longrightarrow C_a B \mid C_a C_b$

$A \longrightarrow C_a D_1 \mid a$

$B \longrightarrow A D_2 \mid b$

$C_a \longrightarrow a$, $C_b \longrightarrow b$

$D_1 \longrightarrow AB$, $D_2 \longrightarrow B C_b$

2.

$S \longrightarrow bA \mid aB$

$A \longrightarrow bAA \mid aS \mid a$

$B \longrightarrow aBB \mid bS \mid b$

CNF is of the form $NT \longrightarrow NT\ NT$

$NT \longrightarrow T$

There are no ϵ prods, no unit prod's and no useless symbols. Replacing all terminals with NT's and adding new prod's we get,

$S \longrightarrow C_b A \mid C_a B$

$A \longrightarrow C_b AA \mid C_a S \mid a$

$B \longrightarrow C_a BB \mid C_b S \mid b$

$C_a \longrightarrow a$, $C_b \longrightarrow b$

The prod's $A \longrightarrow C_b AA$ and $B \longrightarrow C_a BB$ are not in CNF.

So,

$A \longrightarrow C_b AA$

$A \longrightarrow C_b D_1$

$D_1 \longrightarrow AA$

$B \longrightarrow C_a BB$

$B \longrightarrow C_a D_2$

$D_2 \longrightarrow BB$

The final grammar in CNF is

$S \longrightarrow C_b A \mid C_a B$

$A \longrightarrow C_b D_1 \mid C_a S \mid a$

$B \longrightarrow C_a D_2 \mid C_b S \mid b$

$C_a \longrightarrow a$, $C_b \longrightarrow b$

$D_1 \longrightarrow AA$, $D_2 \longrightarrow BB$.

3. $S \longrightarrow aSb \mid ab$

There are not ~~use~~ $\in$ prod's, no unit prod's and no useless symbols. So converting the ~~prod's~~ terminals into NT's and adding new prod's we get,

CNF is of the form $NT \longrightarrow NT\ NT$ & $NT \longrightarrow T$

$S \longrightarrow C_a S C_b$

$S \longrightarrow C_a C_b$ ✓

$C_a \longrightarrow a$ ✓

$C_b \longrightarrow b$ ✓

The prod' $C_a S C_b$ is not in CNF. So,

$S \longrightarrow C_a S C_b$

$S \longrightarrow C_a D_1$ ✓

$D_1 \longrightarrow S C_b$ ✓

The final grammar in CNF is;

$S \longrightarrow C_a D_1 \mid C_a C_b$

$D_1 \longrightarrow S C_b$

$C_a \longrightarrow a$, $C_b \longrightarrow b$

LR(0).

7. $S \to aAa \mid aBC$, $A \to aS \mid bD \mid \epsilon$, $B \to aBa \mid C \mid b$,
$C \to abb \mid DD$, $D \to aDa$

ϵ production Elimination : A is nullable, so we substitute ϵ for A in the above grammar, to get additional productions.

$S \to aa$. We now write the grammar by adding this new production and eliminate direct null prods.

$\therefore$ $S \to aAa \mid aBC \mid aa$, $A \to aS \mid bD$, $B \to aBa \mid C \mid b$,
$C \to abb \mid DD$, $D \to aDa$

Unit production Elimination:

UPs	Non UP's
$B \to C$	$S \to aAa \mid aBC \mid aa$
	$A \to aS \mid bD$
	$B \to aBa \mid b$
	$C \to abb \mid DD$
	$D \to aDa$

Since $B \to C \to abb \mid DD$
the grammar is

$S \to aAa \mid aBC \mid aa$, $A \to aS \mid bD$
$B \to aBa \mid b \mid abb \mid DD$, $C \to abb \mid DD$, $D \to aDa$

Useless Symbol Elimination : D is useless since D is not deriving any terminal. Hence we eliminate D.

$S \to aAa \mid aBC \mid aa$, $A \to aS$, $B \to aBa \mid b \mid abb$, $C \to abb$

CNF is: $S \to C_aAC_a \mid C_aBC \mid C_aC_a$, $C_a \to a$

i.e ✓ $S \to C_aD_1$, $D_1 \to AC_a$ ✓
✓ $S \to C_aD_2$, $D_2 \to BC$ ✓
✓ $A \to C_aS$, $B \to C_aBC_a \mid C_aC_bC_b$

i.e ✓$B \to C_aD_3$, ✓$D_3 \to BC_a$, ✓$B \to C_aD_4$, $D_4 \to C_bC_b$
$C \to C_aC_bC_b \Rightarrow$ ✓$C \to C_aD_4$, ~~D_4~~ $C_b \to b$ ✓

Hence final CNF is: $S \to C_aC_a \mid C_aD_1 \mid C_aD_2$, $C_a \to a$, $D_1 \to AC_a$
$D_2 \to BC$, $A \to C_aS$, $B \to C_aD_3 \mid b \mid C_aD_4$, $D_3 \to BC_a$, $D_4 \to C_bC_b$
$C_b \to b$ and $C \to C_aD_4$

Ic) 1. $S \to AA \mid a$, $A \to SS \mid b$.

The grammer is in GNF.

Assume S to be A_1 and A to be A_2. So grammar is $A_1 \to A_2 A_2 \mid a$, $A_2 \to A_1 A_1 \mid b$.

$A_2 \to A_1 A_1 \mid b$ begins with a lowered no variable

we substitute for A_1 from (1)

$$A_2 \to A_2 A_2 A_1 \mid a A_1 \mid b$$

identifying α_i & β_i

$$A_2 \to \underbrace{A_2 A_2 A_1}_{\alpha_i} \mid \underbrace{a A_1}_{\beta_i} \mid \underbrace{b}_{\beta_i}$$

rewriting A_2 productions

$$A_2 \to a A_1 \mid b \mid a A_1 B_2 \mid b B_2 \quad - (3)$$

$$B_2 \to A_2 A_1 \mid A_2 A_1 B_2 \quad - (4)$$

All A_2 productions are in GNF

Substitute for A_2 in (1)

$$A_1 \to a A_1 A_2 \mid b A_2 \mid a A_1 B_2 A_2 \mid b B_2 A_2 \mid a$$

A_1 is now in GNF

Substitute for A_2 in (4)

$B_2 \rightarrow aA_1A_1 \mid bA_1 \mid bA_1B_2A_1 \mid bB_2A_1 \mid aA_1A_1B_2 \mid$
$bA_1B_2 \mid aA_1B_2A_1B_2 \mid bB_2A_1B_1$

B_2 is also in GNF

(Ta) Properties of CFL's (3m)*

Closure Property

1) CFL's are closed under union, concatenation, kleene closure, reversal, substitution, homomorphism & inverse homomorphism.

2) CFL's are not closed under intersection, difference & complementation.

Desicion Property

1) Emptiness, finiteness, & infiniteness & membership of a CFL are all decidable

II b) Pumping lemma for CFL's. Applications of pumping lemma.

Pumping Lemma for CFL's: Let L be a CFL. Then there exists a constant n such that if z is any string in L such that $|z| \geq n$, then we can split z into 5 substrings as follows.

$$z = uvwxy$$

such that i) $|vx| \geq 1$ i.e atleast one of v or x we pump must not be $\in$.

ii) $|vwx| \leq n$ and

iii) $\forall i \geq 0 \quad uv^i wx^i y \in L$

i.e 2 substrings v and x are pumped any no. of times and the resulting string will still be a member of L

Applications:- It is used to show that certain languages are not context free languages.

proof: omit.

c) Using pumping lemma prove that the following languages are not CFL's.

i. $L = \{ ww \mid w \in \{0,1\}^* \}$

Suppose L is context free and a string of L is represented as

$z = \underbrace{010}_{w}\underbrace{010}_{w}$ where $w = 010$

We split this string in 5 substrings as

$z = uvwxy$

such that u is $\in$

y is $\in$

and resulting substring is $z = vwx$

if v and x are pumped i times we get

$v^i w x^i$ which should belong to L if context free.

let $v = 01$; $w = 00$ and $x = 10$

where $|vx| \geq 1$, $|vwx| \leq n$ i.e 6

Now $z = (01)^i 00 (10)^i$ or take $\underbrace{00}_{v}\underbrace{10}_{w}\underbrace{01}_{x}$

$(00)^i 10 (01)^i$

For $i=1 \Rightarrow z = \underbrace{0100}_{w}\underbrace{10}_{w} \in L$ | $i=1$ z⇒ 001001 ∈ L

for $i=2 \Rightarrow z = \underbrace{0101\ 00\ 1010}_{w\qquad\quad w} \in L$ | $i=2$, z⇒ 0000101, 01 ∉ L

Consider another split $v = 0$

$w = 1001$

$x = 0$

Now $z = (0)^i 1001 (0)^i$

for $i=1$ $0\ 10010 \in L$

for $i=2$ $00100100 \notin L$ Not of form ww.

Hence we proved that L is not CFL since there is a contradiction.

Since there is a contradiction we can say that through pumping lemma we proved that the language L is not context free.

2. $L = \{ 0^n 1^n 0^n 1^n \mid n \geq 0 \}$

Suppose L is context free and a string of L is given as $z =$ ~~0101~~ 00110011, ~~where~~ when $n=2$

we split this ~~the~~ into 5 substrings as $z = uvwxy$ such that v, y is $\in$ and resulting substring is $z = vwx$

if v and x are pumped i times we get $v^i w x^i$ which should belong to L if context free

let $v = 00$, $w = 1100$ and $x = 11$

where $|vx| \geq 1$, $|vwx| \leq n$ i.e 8

Now $z = (00)^i 1100 (11)^i$ or $\underset{v}{01}\,\underset{w}{01}\,\underset{x}{01}$

for $i = 1 \Rightarrow z = 00110011 \in L$

for $i = 2 \Rightarrow z = 0000110011111 \notin L$

Is not of the form $0^n 1^n 0^n 1^n$ and hence there is a contradiction. We say that through pumping lemma we have proved that L is not context free.

$01 \notin L$

III

$S \to A_1 A_2 \mid A_2 A_3$

$A_1 \to A_2 A_1 \mid 0$

$A_2 \to A_3 A_3 \mid 1$

$A_3 \to A_1 A_2 \mid 0$

" " "

10010

The given grammer is in CNF

we know construct the table of V_{ij}'s

i \ j	1	2	3	4	5
	1	0	0	1	0
1	A_2 V_{11}	A_1, A_3 V_{21}	A_1, A_3 V_{31}	A_2 V_{41}	A_1, A_3 V_{51}
2	A_1, S V_{12}	A_2 V_{22}	S, A_3 V_{32}	A_1, S V_{42}	
3	$\emptyset$ V_{13}	A_2 V_{23}	A_2 V_{33}		
4	$\emptyset$ V_{14}	S, A_1, A_3 V_{24}			
5	S, A_1, A_3 V_{15}				

To fill up the table we need to follow the algorithm.

Since the last cell V_{15} has the start symbol S we conclude that the given string belongs to the grammer through CYK Algorithm.

Hence, "10010" is a member of the CFG.

j	i	k	$V_{ik} \times V_{i+k, j-k}$	V_{ij}
2	1	1	$V_{11} \times V_{21} = \{A_2\} \times \{A_1, A_3\} = \{A_2A_1\}, \{A_2A_3\}$	A_1, S
	2	1	$V_{21} \times V_{31} = \{A_1, A_3\} \times \{A_1, A_3\} = \{A_1A_1, A_1A_3, A_3A_1, A_3A_3\}$	A_2
	3	1	$V_{31} \times V_{41} = \{A_1, A_3\} \times \{A_2\} = \{A_1A_2, A_3A_2\}$	S, A_3
	4	1	$V_{41} \times V_{51} = \{A_2\} \times \{A_1, A_3\} = \{A_2A_1, A_2A_3\}$	A_1, S
3	1	1	$V_{11} \times V_{22} = \{A_2\} \times \{A_2\} = \{A_2A_2\}$	$\emptyset$
		2	$V_{12} \times V_{31} = \{A_1, S\} \times \{A_1, A_3\} = \{A_1A_1, A_1A_3, SA_1, SA_3\}$	$\emptyset$
	2	1	$V_{21} \times V_{32} = \{A_1, A_3\} \times \{S, A_3\} = \{A_1S, A_1A_3, A_3S, A_3A_3\}$	A_2
		2	$V_{22} \times V_{41} = \{A_2\} \times \{A_2\} = \{A_2A_2\}$	$\emptyset$
	3	1	$V_{31} \times V_{42} = \{A_1, A_3\} \times \{A_1, S\} = \{A_1A_1, A_1S, A_3A_1, A_3S\}$	$\emptyset$
		2	$V_{32} \times V_{51} = \{S, A_3\} \times \{A_1, A_3\} = \{SA_1, SA_3, A_3A_1, A_3A_3\}$	A_2
4	1	1	$V_{11} \times V_{23} = \{A_2\} \times \{A_2\} = \{A_2A_2\}$	$\emptyset$
		2	$V_{12} \times V_{32} = \{A_1, S\} \times \{S, A_3\} = \{A_1S, A_1A_3, SS, SA_3\}$	$\emptyset$
		3	$V_{13} \times V_{41} = \{\emptyset\} \times \{A_2\} = \{A_2\}$	$\emptyset$
	2	1	$V_{21} \times V_{33} = \{A_1, A_3\} \times \{A_2\} = \{A_1A_2, A_3A_2\}$	S, A_3
		2	$V_{22} \times V_{42} = \{A_2\} \times \{A_1, S\} = \{A_2A_1, A_2S\}$	A_1
		3	$V_{23} \times V_{51} = \{A_2\} \times \{A_1, A_3\} = \{A_2A_1, A_2A_3\}$	A_1, S
5	1	1	$V_{11} \times V_{24} = \{A_2\} \times \{S, A_1, A_3\} = \{A_2S, A_2A_1, A_2A_3\}$	A_1, S
		2	$V_{12} \times V_{33} = \{A_1, S\} \times \{A_2\} = \{A_1A_2, SA_2\}$	S, A_3
		3	$V_{13} \times V_{42} = \{\emptyset\} \times \{A_1, S\} = \{A_1, S\}$	$\emptyset$
		4	$V_{14} \times V_{51} = \{\emptyset\} \times \{A_1, A_3\} =$	$\emptyset$

IV a)

DCFL's: Deterministic PDA's accept a family of languages called the deterministic CFL's lying properly between regular sets and CFL's.

Syntax of many programming languages can be described by means of DCFL's. Modern compiler writing systems usually require that the syntax of the language for which they are to produce a compiler be described by a CFG of restricted form. These restricted forms almost only generate DCFLs. The most important of these restricted forms is called LR-grammars.

b) LR(0) grammars: An LR(0) grammar is a restricted type of CFG. This class of grammars is the first in the family collectively called LR-grammars. LR(0) Stands for "left-to-right scan of the input producing a right most derivation and using 0 symbols of lookahead on the input". Every LR(0) grammar is unambiguous.

LR(k) grammars: If we add one symbol of "lookahead" by determining the set of following terminals on which reduction by $A \rightarrow \alpha$ could possibly be performed then we can use DPDAs to recognize the languages of a wider class of grammars. These grammars are called LR(1) grammars, for one symbol of lookahead. All and only deterministic CFL's has LR(1) grammars. This class of grammars has great importance for compiler design since they are broad enough to include the syntax of almost all programming languages. For any k, there are grammars called LR(k), that may be parsed with k symbols of lookahead.

V Is the grammar LR(0)?

1. $A \rightarrow (A) \mid a$

Augment the grammar.

$A' \rightarrow A$

$A \rightarrow (A) \mid a$

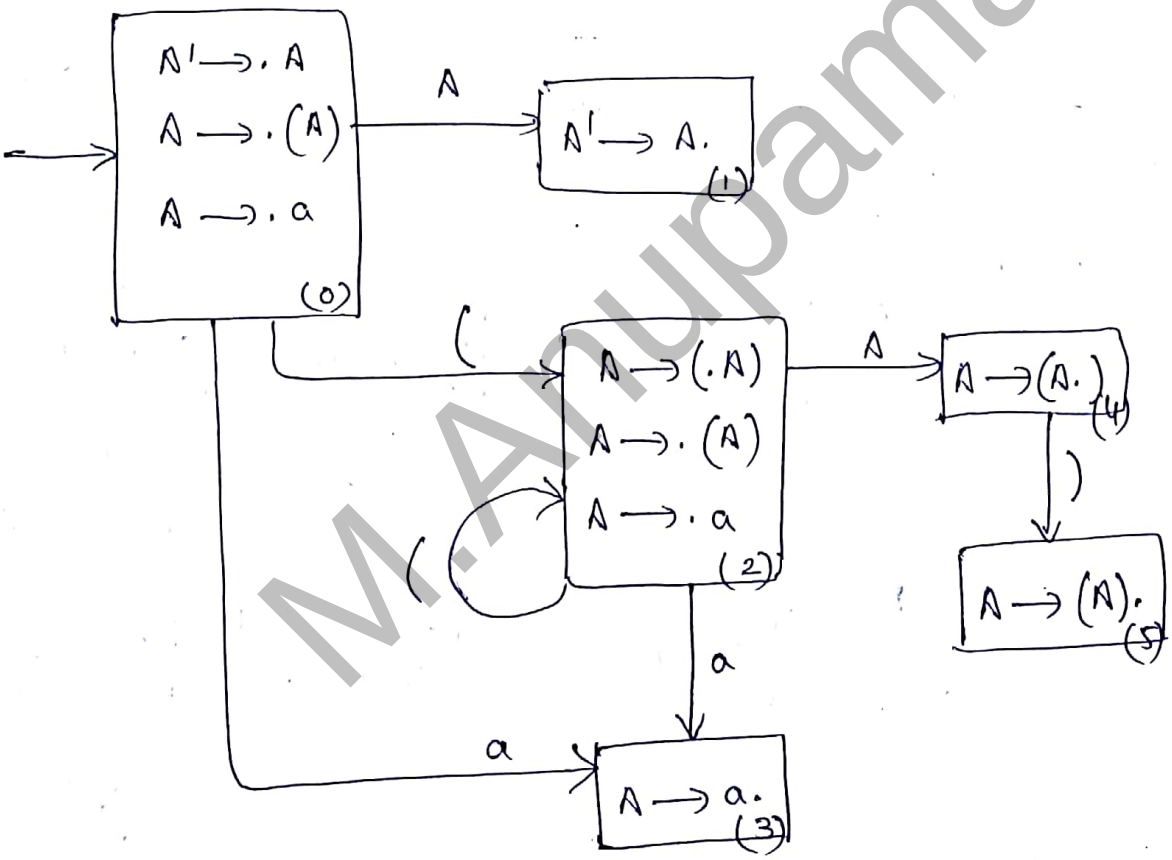
Find the LR(0) items.

$A' \rightarrow .A$, $A' \rightarrow A.$

$A \rightarrow .(A)$, $A \rightarrow (.A)$, $A \rightarrow (A.)$, $A \rightarrow (A).$, $A \rightarrow .a$,

$A \rightarrow a.$

DFA of sets of LR(0) items.



States (1), (3) and (5) have complete items which are all alone, hence we conclude that the grammar is LR(0).

2. $S \rightarrow AaAb \mid BbBa$, $A \rightarrow \epsilon$, $B \rightarrow \epsilon$

Augment the grammar : $S' \rightarrow S$

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$, $B \rightarrow \epsilon$

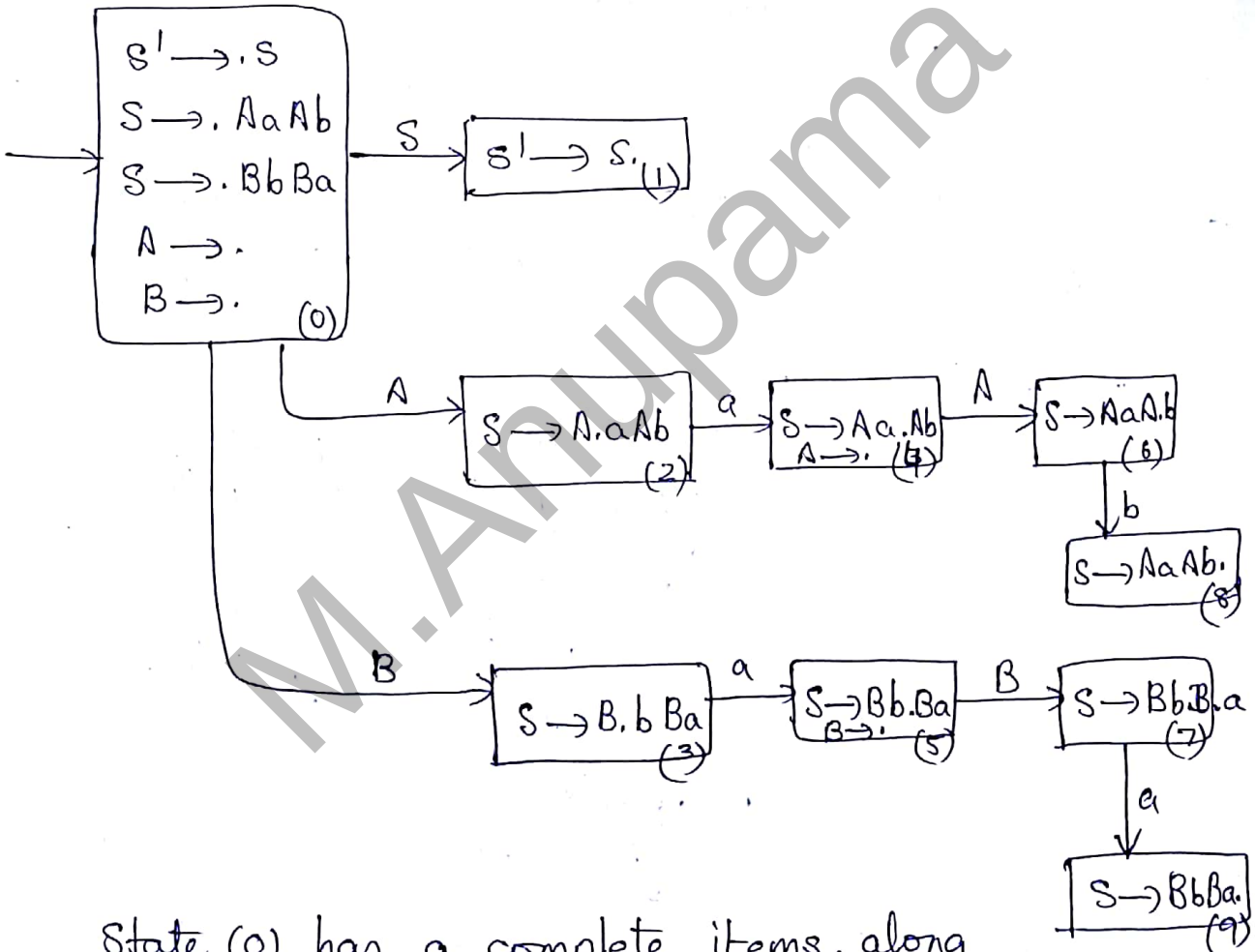
LR(0) items :

$S' \rightarrow .S$, $S' \rightarrow S.$, $S \rightarrow .AaAb$, $S \rightarrow A.aAb$, $S \rightarrow Aa.Ab$,

$S \rightarrow AaA.b$, $S \rightarrow AaAb.$, $S \rightarrow .BbBa$, $S \rightarrow B.bBa$,

$S \rightarrow Bb.Ba$, $S \rightarrow BbB.a$, $S \rightarrow BbBa.$, $A \rightarrow .$, $B \rightarrow .$

DFA of sets of LR(0) items:



State (0) has 2 complete items, along with some incomplete items so it violates the rule for LR(0) — If there is a complete item it should be all alone in any state.

Hence the grammar is not LR(0).

3. $E \rightarrow E*B \mid E+B \mid B$, $B \rightarrow 0 \mid 1$

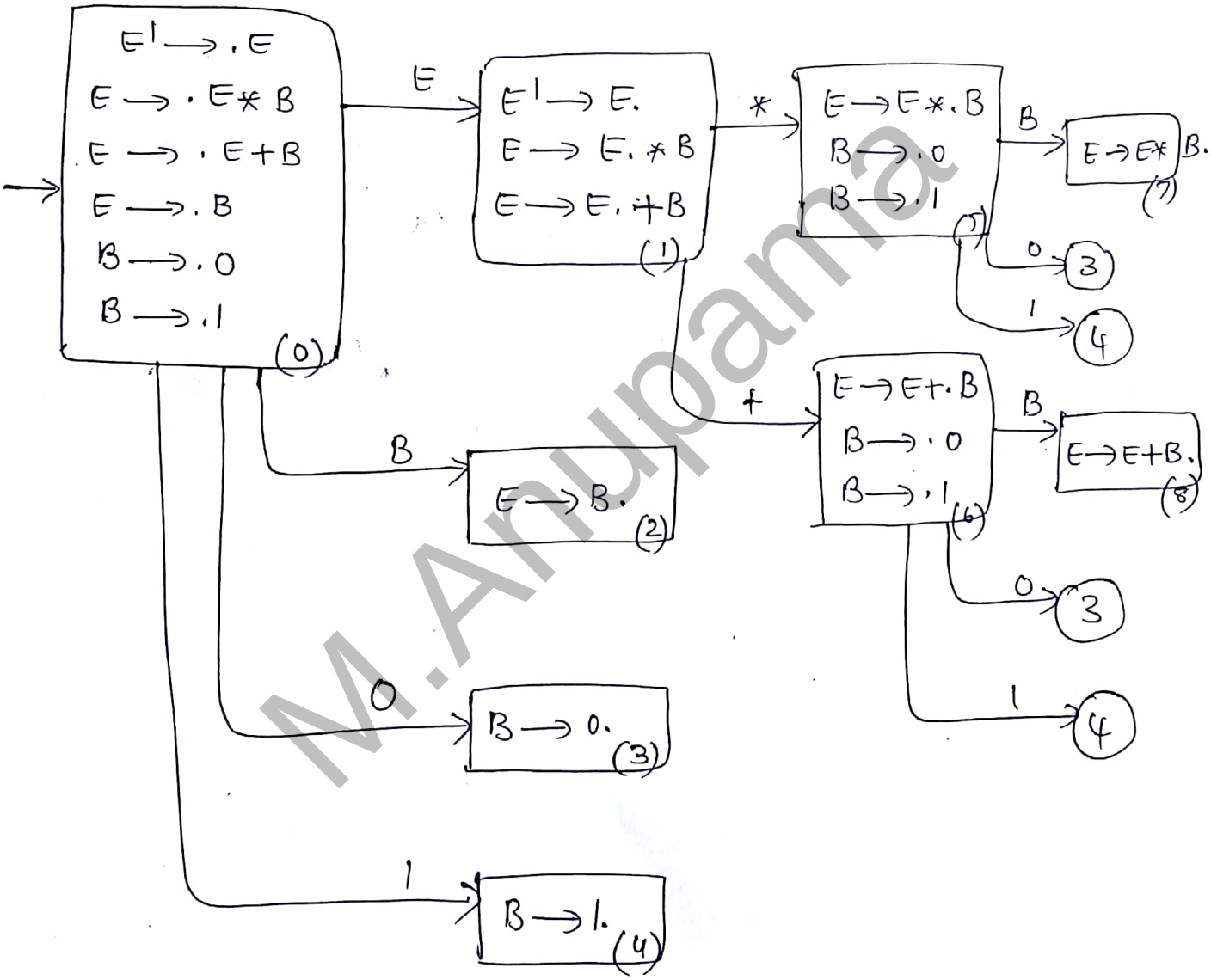
Augment the grammar; $E' \rightarrow E$, $E \rightarrow E*B \mid E+B \mid B$, $B \rightarrow 0 \mid 1$

LR(0) items: $E' \rightarrow .E$, $E' \rightarrow E.$, $E \rightarrow .E*B$, $E \rightarrow E.*B$,

$E \rightarrow E*.B$, $E \rightarrow E*B.$, $E \rightarrow .E+B$, $E \rightarrow E.+B$, $E \rightarrow E+.B$,

$E \rightarrow E+B.$, $E \rightarrow .B$, $E \rightarrow B.$, $B \rightarrow .0$, $B \rightarrow 0.$, $B \rightarrow .1$, $B \rightarrow 1.$

DFA of Sets of LR(0) items:



In State (1) there is a complete item along with other incomplete items so Rule for LR(0) Grammar is violated. Hence the grammar is not LR(0).

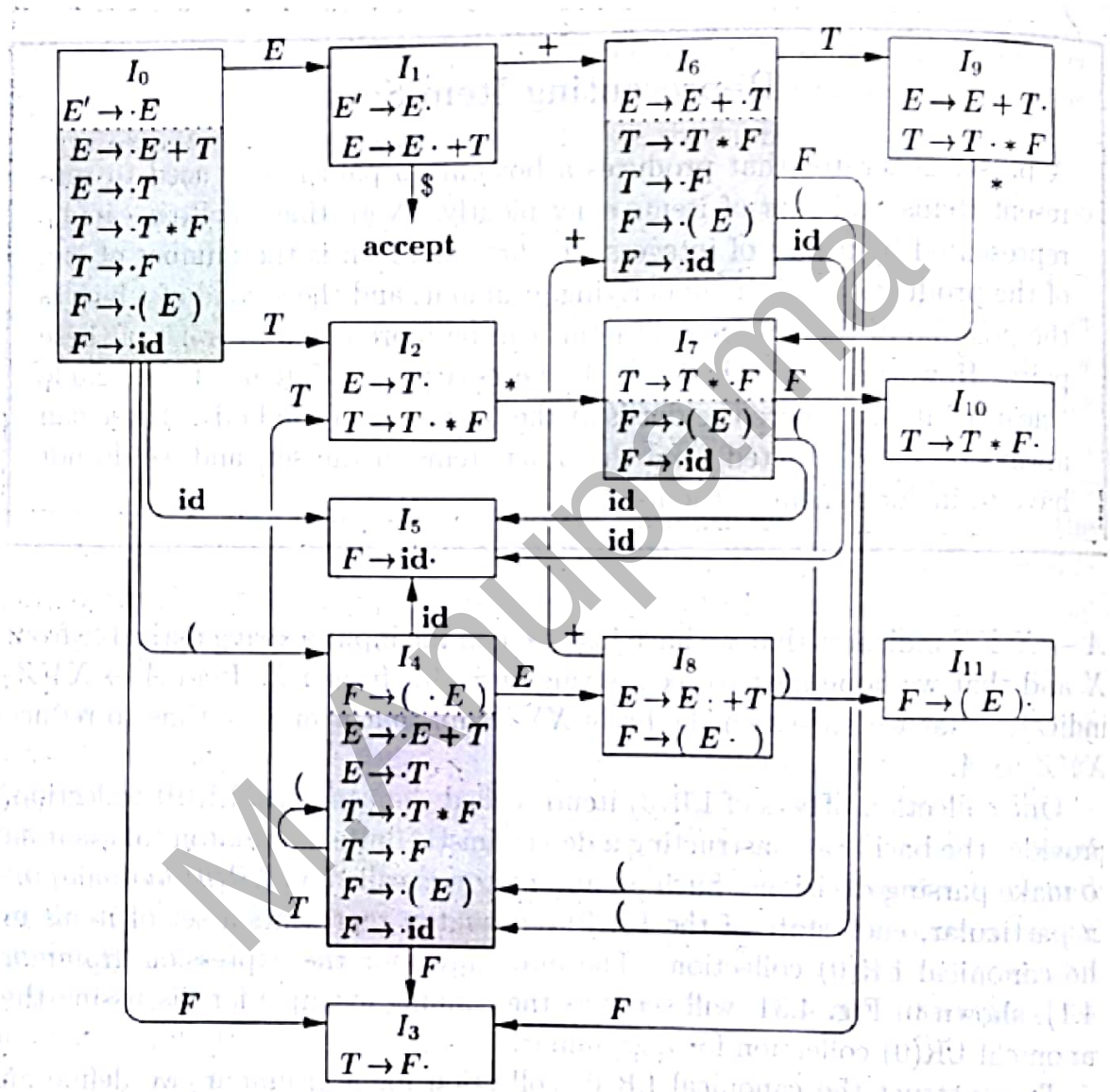
4. $E \rightarrow E+T \mid T$, $T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$

Augmented grammar: $E' \rightarrow E$, $E \rightarrow E+T \mid T$,

$T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$

Write LR(0) items:

Then construct sets of LR(0) items:

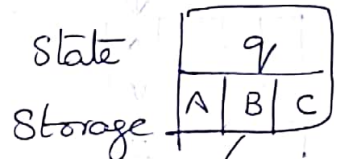


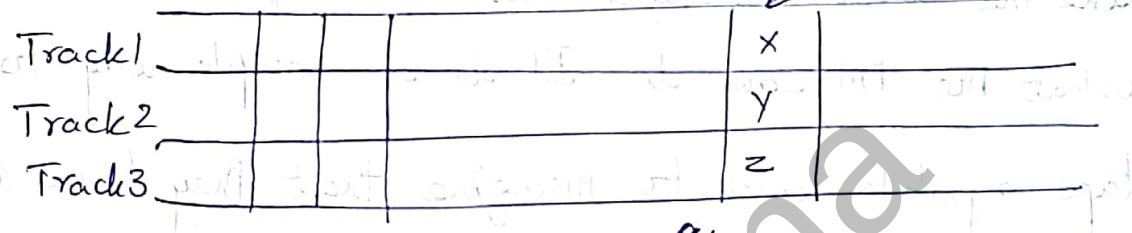
Since in state (1) there is a violation, i.e the complete item is along with an incomplete item and is not all alone we say the grammar is not LR(0).

VI. a) State and explain programming techniques for TM's.

A TM can be used to compute like a conventional computer, i.e it is exactly as powerful as a convential computer, and can perform all sorts of calculations.

1. Storage in the State:-



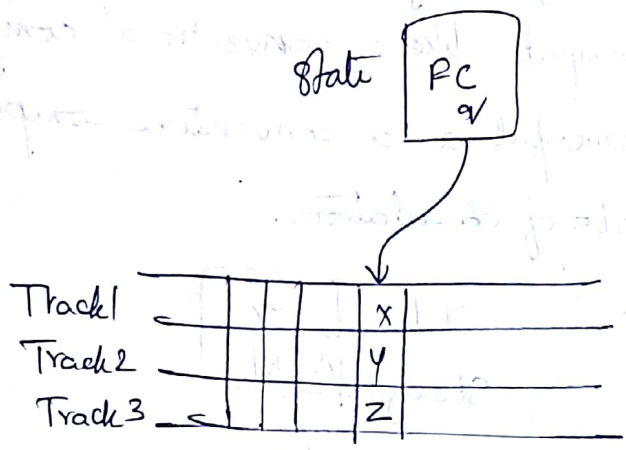


A TM with storage in Finite Control and having multiple tracks.

A finite control is used to represent a position in the program of the TM, but also to hold a finite amount of data, A, B and C. We think of the state as [q, A, B, C]. This allows us to describe transitions in a more systematic way. Eg: TM that accepts the language $01^* + 10^*$ where the TM remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. This does not extend the TM model.

2. Multiple Tracks: The tape of the TM can be thought of as having several tracks. Each track can hold one symbol and the tape alphabet of the TM consists of tuples,

with one component for each track. Thus the cell scanned by the tape head contains the symbols [X, Y, Z]



Like the above method this technique does not extend what the TM can do. It is a simple way to view tape symbols and to imagine that they have an useful structure.

Eg: A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark, like '*', with with we can check off each symbol as we use it.

Eg: TM to recognize $L = \{ wcw \mid w \in (0+1)^+ \}$

We use * to check off symbols & of the first and second groups of 0's and 1's eventually confirming that the string to the left of 'c' is the same as the string to its right.

3. Subroutines: In high level languages use of subroutines build the modularity in the programming development process. The same type of concept can be introduced in the construction of a TM.

A TM subroutine is a set of states that perform some useful process. This set of states includes a start state and another state that is a temporary halt state or return state, which is used to pass control to other set of states that called the subroutine. The call of the subroutine occurs whenever there is a transition to its initial state.

Eg: TM to implement the function multiplication.
i.e our TM will start with 0^m10^n on its tape and will end with 0^{mn} on the tape.

I/p: B0010000B (m=2, n=3) O/P: B000000B

We change the first 0 in the 1st group to B and add n 0's to the last group. As a result we copy n 0's to the end m times, once each time we change a 0 in the first group (to) B. When the 1st group of 0's is completely changed to blanks, then there will be mn 0's in the last group. We then change 10^n1 to blanks and mn 0's are left as output on the tape. The heart of the algorithm is a subroutine which we call 'Copy', which copies n 0's m times to the end.

8.4 Extensions to the Basic Turing Machine

In this section we shall see certain computer models that are related to Turing machines and have the same language-recognizing power as the basic model of a TM with which we have been working. One of these, the multitape Turing machine, is important because it is much easier to see how a multitape TM can simulate real computers (or other kinds of Turing machines), compared with the single-tape model we have been studying. Yet the extra tapes add no power to the model, as far as the ability to accept languages is concerned.

We then consider the nondeterministic Turing machine, an extension of the basic model that is allowed to make any of a finite set of choices of move in a given situation. This extension also makes "programming" Turing machines easier in some circumstances, but adds no language-defining power to the basic model.

8.4.1 Multitape Turing Machines

A multitape TM is as suggested by Fig. 8.16. The device has a finite control (state), and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet. As in the single-tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a member. The set of states includes an initial state and some accepting states. Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.

2. All other cells of all the tapes hold the blank.

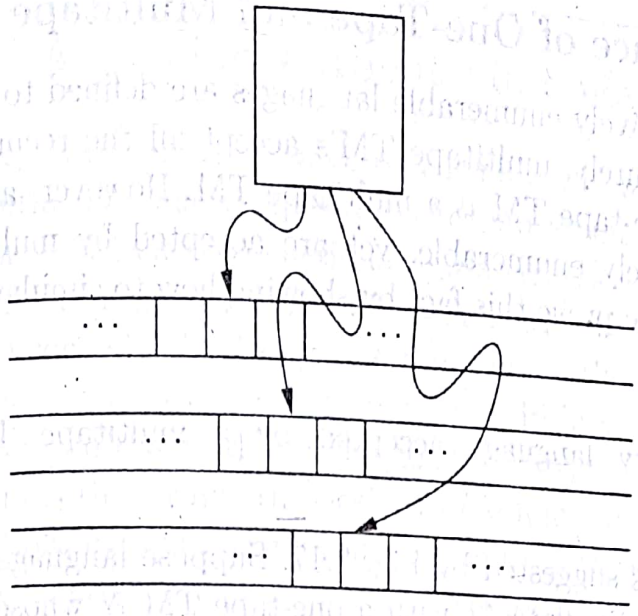


Figure 8.16: A multitape Turing machine

3. The finite control is in the initial state.

4. The head of the first tape is at the left end of the input.

5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes "look" the same.

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

1. The control enters a new state, which could be the same as the previous state.

2. On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.

3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

We shall not give the formal notation of transition rules, whose form is a straightforward generalization of the notation for the one-tape TM, except that directions are now indicated by a choice of L, R, or S. For the one-tape machine, we did not allow the head to remain stationary, so the S option was not present. You should be able to imagine an appropriate notation for instantaneous descriptions of the configuration of a multitape TM; we shall not give this notation formally. Multitape Turing machines, like one-tape TM's, accept by entering an accepting state.

8.4.2 Equivalence of One-Tape and Multitape TM's

Recall that the recursively enumerable languages are defined to be those accepted by a one-tape TM. Surely, multitape TM's accept all the recursively enumerable languages, since a one-tape TM *is* a multitape TM. However, are there languages that are not recursively enumerable, yet are accepted by multitape TM's? The answer is "no," and we prove this fact by showing how to simulate a multitape TM by a one-tape TM.

Theorem 8.9: Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by Fig. 8.17. 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 8.17 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.

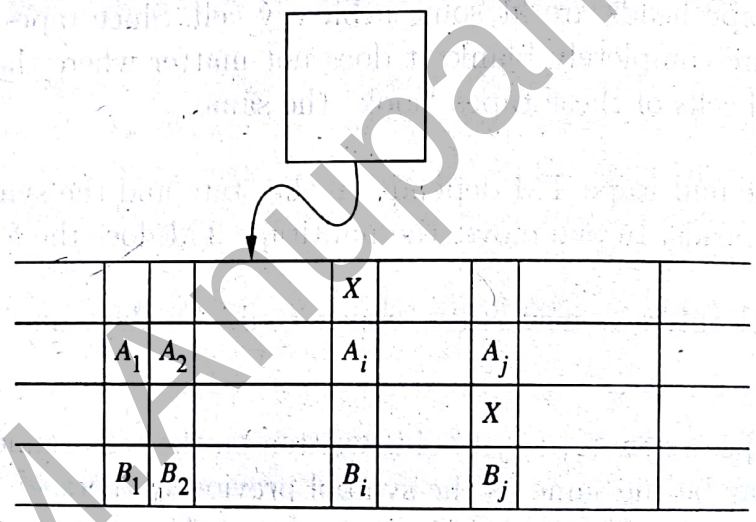


Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

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.

8.4.3 Running Time and the Many-Tapes-to-One Construction

Let us now introduce a concept that will become quite important later: the "time complexity" or "running time" of a Turing machine. we say the *running time* of TM M on input w is the number of steps that M makes before halting. If M doesn't halt on w, then the running time of M on w is infinite. The *time complexity* of TM M is the function $T(n)$ that is the maximum, over all inputs w of length n, of the running time of M on w. For Turing machines that do not halt on all inputs, $T(n)$ may be infinite for some or even all n. However, we shall pay special attention to TM's that do halt on all inputs, and in particular, those that have a polynomial time complexity $T(n)$; Section 10.1 initiates this study.

The construction of Theorem 8.9 seems clumsy. In fact, the constructed one-tape TM may take much more running time than the multitape TM. However, the amounts of time taken by the two Turing machines are commensurate in a weak sense: the one-tape TM takes time that is no more than the square of the time taken by the other. While "squaring" is not a very strong guarantee, it does preserve polynomial running time. We shall see in Chapter 10 that:

a) The difference between polynomial time and higher growth rates in running time is really the divide between what we can solve by computer and what is in practice not solvable.

b) Despite extensive research, the running time needed to solve many problems has not been resolved closer than to within some polynomial. Thus, the question of whether we are using a one-tape or multitape TM to solve the problem is not crucial when we examine the running time needed to solve a particular problem.

The argument that the running times of the one-tape and multitape TM's are within a square of each other is as follows.

Theorem 8.10: The time taken by the one-tape TM N of Theorem 8.9 to simulate n moves of the k-tape TM M is $O(n^2)$.

PROOF: After n moves of M, the tape head markers cannot have separated by more than $2n$ cells. Thus, if M starts at the leftmost marker, it has to move no more than $2n$ cells right, to find all the head markers. It can then make an excursion leftward, changing the contents of the simulated tapes of M, and moving head markers left or right as needed. Doing so requires no more than $2n$ moves left, plus at most $2k$ moves to reverse direction and write a marker X in the cell to the right (in the case that a tape head of M moves right).

Thus, the number of moves by N needed to simulate one of the first n moves is no more than $4n + 2k$. Since k is a constant, independent of the number of moves simulated, this number of moves is $O(n)$. To simulate n moves requires no more than n times this amount, or $O(n^2)$. □

8.4.4 Nondeterministic Turing Machines

A *nondeterministic* Turing machine (*NTM*) differs from the deterministic variety we have been studying by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1),\ (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer. The NTM can choose, at each step, any of the triples to be the next move. It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

The language accepted by an NTM M is defined in the expected manner, in analogy with the other nondeterministic devices, such as NFA's and PDA's, that we have studied. That is, M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state. The existence of other choices that do *not* lead to an accepting state is irrelevant, as it is for the NFA or PDA.

The NTM's accept no languages not accepted by a deterministic TM (or *DTM* if we need to emphasize that it is deterministic). The proof involves showing that for every NTM M_N, we can construct a a DTM M_D that explores the ID's that M_N can reach by any sequence of its choices. If M_D finds one that has an accepting state, then M_D enters an accepting state of its own. M_D must be systematic, putting

new ID's on a queue, rather than a stack, so that after some finite time M_D has simulated all sequences of up to k moves of M_N, for $k = 1, 2, \ldots$.

Theorem 8.11: If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

PROOF: M_D will be designed as a multitape TM, sketched in Fig. 8.18. The first tape of M_D holds a sequence of ID's of M_N, including the state of M_N. One ID of M_N is marked as the "current" ID, whose successor ID's are in the process of being discovered. In Fig. 8.18, the third ID is marked by an x along with the inter-ID separator, which is the $*$. All ID's to the left of the current one have been explored and can be ignored subsequently.

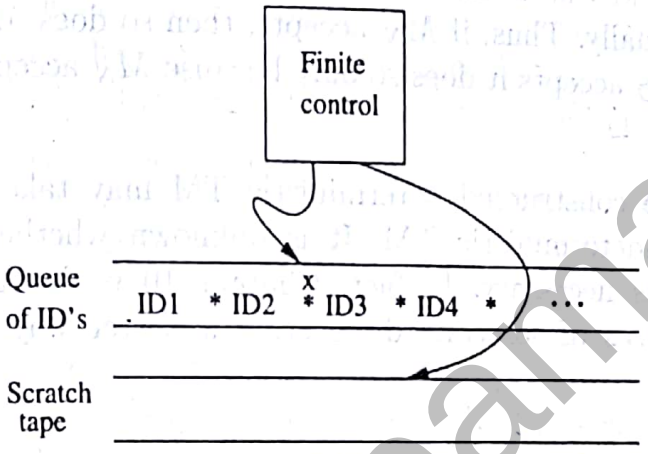


Figure 8.18: Simulation of an NTM by a DTM

To process the current ID, M_D does the following:

1. M_D examines the state and scanned symbol of the current ID. Built into the finite control of M_D is the knowledge of what choices of move M_N has for each state and symbol. If the state in the current ID is accepting, then M_D accepts and simulates M_N no further.

2. However, if the state is not accepting, and the state-symbol combination has k moves, then M_D uses its second tape to copy the ID and then make k copies of that ID at the end of the sequence of ID's on tape 1.

3. M_D modifies each of those k ID's according to a different one of the k choices of move that M_N has from its current ID.

4. M_D returns to the marked, current ID, erases the mark, and moves the mark to the next ID to the right. The cycle then repeats with step (1).

It should be clear that the simulation is accurate, in the sense that M_D will only accept if it finds that M_N can enter an accepting ID. However, we need to confirm that if M_N enters an accepting ID after a sequence of n of its own moves, then M_D will eventually make that ID the current ID and will accept.

Suppose that m is the maximum number of choices M_N has in any configuration. Then there is one initial ID of M_N, at most m ID's that M_N can reach after one move, at most m^2 ID's M_N can reach after two moves, and so on. Thus, after n moves, M_N can reach at most $1 + m + m^2 + \cdots + m^n$ ID's. This number is at most nm^n ID's.

The order in which M_D explores ID's of M_N is "breadth first"; that is, it explores all ID's reachable by 0 moves (i.e., the initial ID), then all ID's reachable by one move, then those reachable by two moves, and so on. In particular, M_D will make current, and consider the successors of, all ID's reachable by up to n moves before considering any ID's that are only reachable by more than n moves.

As a consequence, the accepting ID of M_N will be considered by M_D among the first nm^n ID's that it considers. We only care that M_D considers this ID in some finite time, and this bound is sufficient to assure us that the accepting ID is considered eventually. Thus, if M_N accepts, then so does M_D. Since we already observed that if M_D accepts it does so only because M_N accepts, we conclude that $L(M_N) = L(M_D)$. □

Notice that the constructed deterministic TM may take exponentially more time than the nondeterministic TM. It is unknown whether or not this exponential slowdown is necessary. In fact, Chapter 10 is devoted to this question and the consequences of someone discovering a better way to simulate NTM's deterministically.

X. Design TM's for the following:

2. TM to accept $a^n b^n a^n \mid n \geq 1$

Sample string : aaabbbaaa Δ

XaaYbbZaa Δ

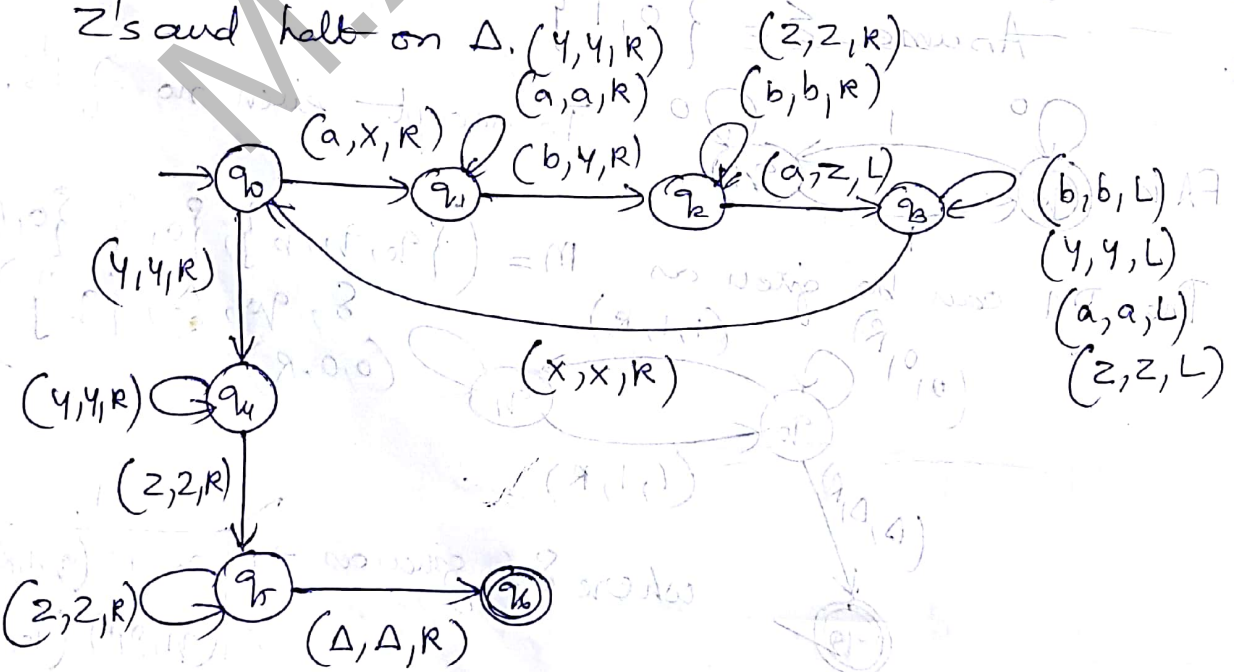
XXaYYbZZa Δ

XXXYYYZZZ Δ

on no more a's

XXXYYYZZZ Δ

Logic: On seeing the 1st 'a' make it an 'X' and move right till a 'b' comes. On seeing the 'b' make it a 'Y' and move right till a 'a' comes. Make the 'a' a 'Z' and move left till X. Now move right and make the second 'a' an X and repeat the process. When no more a's are seen move right over all Y's and Z's and halt on Δ.



The above TM is given as $M = (\{q_0, q_1 \cdots q_6\}, \{a, b\}, \{a, b, X, Y, Z, \Delta\}, \delta, q_0, \Delta, \{q_6\})$ where δ is

given as follows:

	a	b	X	Y	Z	Δ
→ q_0	(q_1, X, R)			(q_4, Y, R)		
q_1	(q_1, a, R)	(q_2, Y, R)	—	(q_1, Y, R)	—	—
q_2	(q_3, Z, L)	(q_2, b, R)	—	—	(q_2, Z, R)	—
q_3	(q_3, a, L)	(q_3, b, L)	(q_0, X, R)	(q_3, Y, L)	(q_3, Z, L)	—
q_4	—	—	—	(q_4, Y, R)	(q_5, Z, R)	—
q_5	—	—	—	—	(q_5, Z, R)	(q_6, Δ, R)
(q_6)	—	—	—	—	—	—

X.

4. TM for Palindrome or ww^R (even) or wcw^R (odd)

where R is reverse, and c is 0 or 1 if $\Sigma = \{0,1\}$

and c = a or b if $\Sigma = \{a,b\}$

The TM for a general palindrome (both even and odd) is given as follows. Assume $\Sigma = \{0,1\}$.

$\underline{0110}$ $\quad$ wcw^R

$w \; w^R$ $\quad$ $001 \mid 100$

0 or 1

logic: Find the first symbol (can be 0 or 1) and make it a blank. Move to the extreme right and when a blank is seen leave it as a blank and move left (last character). If it is the same symbol as the first character replace it with a blank and now move to extreme left blank. Move rt and continue the process. Break when there are no more 0's & 1's.

0110 Δ

Δ Δ Δ

ΔΔ1ΔΔ

ΔΔΔΔΔ

Break

0011100 Δ

Δ01110Δ Δ

ΔΔ111ΔΔΔ

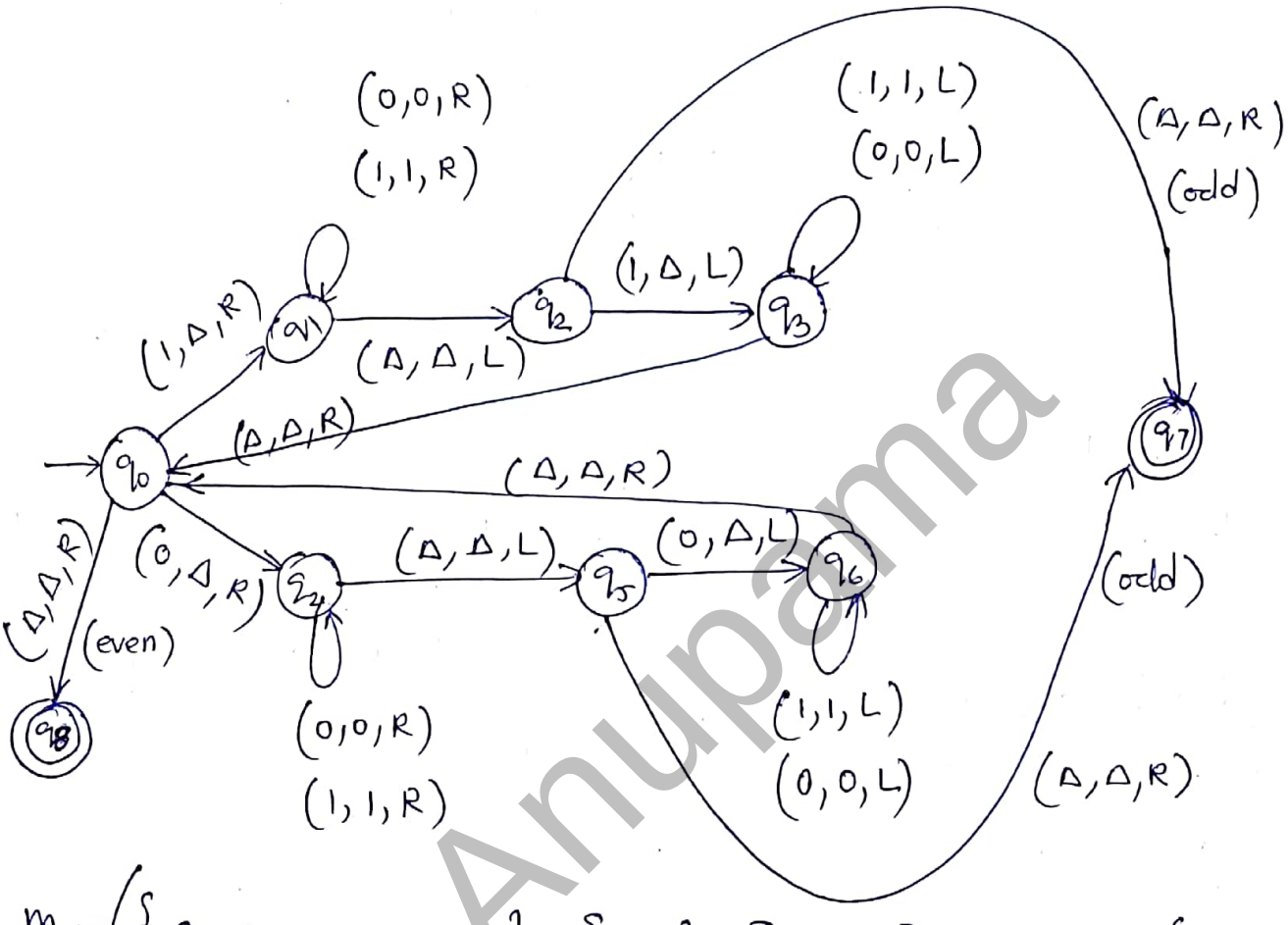
ΔΔΔ1ΔΔΔΔ

ΔΔΔΔΔΔΔΔ

→ Break.

In the ~~given~~ TM if only even is asked then omit transitions from q_2 to q_7 and q_5 to q_7. Only q_8 is the final state. Similarly if only

odd palindrome is asked then omit transition from q_0 to q_8. In this case only q_7 is the final state. For a general palindrome both q_8 and q_7 are final states.



$$M = \left(\{ q_0, q_1, \ldots q_8 \}, \{0,1\}, \{0,1,\Delta\}, \delta, q_0, \Delta, \{q_8, q_7\} \right)$$

and construct the δ table as

	0	1	Δ
→ q_0			
q_1			
q_2			
q_8			

8. TM as a computer of non-ve Integers.

Integers are represented as unary on the tape and an integer $i \geq 0$ is represented by 0^i. If a function has k arguments $i_1, i_2, \ldots i_k$ then these integers are placed on the tape separated by 1's as follows:

$0^{i_1} 1 0^{i_2} 1 0^{i_3} 1 \ldots\ldots 1 0^{i_k} 1$. If a TM halts

on the tape consisting of 0^m for some m then we say $f(i_1, i_2, \cdots i_k) = m$. f is called a total recursive function. If f is computed by a TM then it is called a partial recursive function. If integers are represented as a^k on the tape then a's are separated by b's. like

$a^{i_1} b a^{i_2} b \cdots b a^{i_k} b$.

Eg: 0^5 and 0^3 are represented on tape as | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Δ |

Addition of 2 non -ve integers or Adder function :

Eg: $a^n b a^m = a^{n+m}$

I/p: | a | a | a | b | a | a | Δ | $a^3 b a^2$

O/p: | a | a | a | a | a | Δ | $a^{3+2} = a^5$

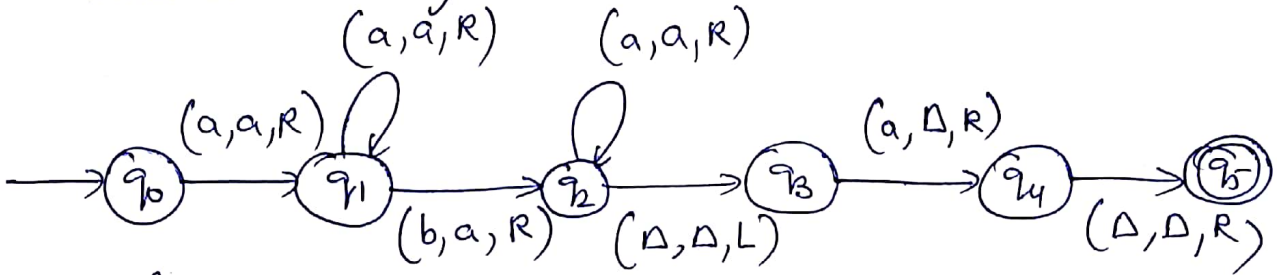
aaabaaΔ

aaaaaaΔ

aaaaaΔΔ

logic: On seeing an 'a' move right. On seeing any no. of a's move right leaving them as a's. On seeing a 'b' make it an 'a' and move right ~~till Δ~~ on all a's leaving them as a's till Δ. Move left and the last a make it a Δ. Move right & halt on Δ.

(i.e since you added one 'a' for b, you are making last one 'a' as Δ.)



$M = (\{q_0, q_1, q_2 \cdots q_5\}, \{a, b\}, \{a, b, \Delta\}, \delta, q_0, \Delta, q_5)$ where δ is given in the table.

	a	b	Δ
→ q_0	(q_1, a, R)	–	–
q_1	(q_1, a, R)	(q_2, a, R)	–
q_2	(q_2, a, R)	–	(q_3, Δ, L)
q_3	(q_4, Δ, R)	–	(q_5, Δ, R)
q_4	–	–	–