Module 3

Myhill-Nerode Relations and Context Free

MYHILL NERODE THEOREM

Ques 1) Describe the Myhill-Nerode relation and also explain the MNR for regular language.

Ans: Myhill-Nerode relation

Let $R \subseteq \Sigma^*$ be a regular set, and let $M = (Q, \Sigma, \delta, s, F)$ be a DFA for R with no inaccessible states. The automaton M induces an equivalence relation $\equiv M$ on Σ^* defined by:

$$x \equiv_M y \overset{def}{\Leftrightarrow} \hat{\delta}(s, x) = \hat{\delta}(s, y)$$

One can easily show that the relation $\equiv M$ is an equivalence relation; that is, that it is reflexive, symmetric, and transitive. In addition, $\equiv M$ satisfies a few other useful properties:

i) **It is a Right Congruence:** for any $x, y \in \Sigma^*$ and $a \in \Sigma$,

$x \equiv M\, y \Rightarrow xa \equiv M\, ya$

To see this, assume that $x \equiv M\, y$. Then

$\hat{\delta}(s, xa) = \hat{\delta}(\hat{\delta}(s, x), a)$

$= \hat{\delta}(\hat{\delta}(s, y), a)$ by assmption

$= \hat{\delta}(s, ya)$

ii) **It refines R:** for any $x, y \in \Sigma^*$,

$x \equiv M\, y \Rightarrow (x \in R \Leftrightarrow y \in R)$

This is because $\hat{\delta}(s, x) = \hat{\delta}(s, y)$, and this is either an accept or a reject state, so either both x and y are accepted or both are rejected. Another way to say this is that every $\equiv M$-class has either all its elements in R or none of its elements in R; in other words, R is a union of $\equiv M$-classes.

iii) **It is of finite index**; that is, it has only finitely many equivalence classes. This is because there is exactly one equivalence class

$$\{x \in \Sigma^* \mid \hat{\delta}(s, x) = q\}$$

corresponding to each state q of M.

Let us call an equivalence relation $\equiv$ on Σ^* a ***Myhill-Nerode relation*** *for R if it satisfies properties (i), (ii), and (iii), that is, if it is a right congruence of finite index refining R.*

Ques 2) Explain Myhill Nerode theorem. Write the application of MNT.

Or

State Myhill-Nerode theorem. **(2017 [02], 2019[03])**

Or

What is Myhill Nerode Theorem? **(2018 [4.5])**

Ans: Myhill Nerode Theorem

A language L is regular if and only if R_L has a finite number of equivalence classes. Moreover, the number of states is the smallest DFA recognizing L is equal to the number of equivalence classes of R_L. Alternatively the set $L \in Z^*$ is accepted by some FA, and let E_L be the set of equivalence classes of the relation R_L on Σ^*. If E_L is a finite set then a FA accepts L provided that this FA has the fewest states then any FA accepting L. Besides other consequences of this theorem, its implications is that there exists always a unique minimum state DFA for any regular language.

Finding Equivalence of States

Assume DFA $M = (Q, \Sigma, \delta, q_0, F)$ and let r and s are the states $\in$ Q then, states r and s are said to be equivalent if for all strings x of L have:

$\hat{\delta}(r, x) \in F$ and $\hat{\delta}(s, x) \in F$ (for $\forall x \in L$).

States r and s are said to be distinguishable if for at least a string $x \in L$ have:

$\hat{\delta}(r, x) \in F$ and $\hat{\delta}(s, x) \notin F$, or

$\hat{\delta}(r, x) \notin F$ and $\hat{\delta}(s, x) \in F$, or

$\hat{\delta}(r, x) \notin F$ and $\hat{\delta}(s, x) \notin F$;

(where F is the set of accepting state/s).

The equivalence of states have been categorized into various classes which is depend upon up to what extents of symbols of string x they behaves in similar nature (returns on the set F). These equivalence classes are:

1) **0-Equivalence Class:** Test the equivalence relation such as rR_0s for the string of length 0 (ignore the meaningless case of length less than 0).

It means we are talking about the string containing symbol $\in$ ($|\in| = 0$) only. If both $\delta(r, \in) \in F$ and $\delta(s, \in) \in F$, then states r and s are said to be 0-equivalent states, and partition of states is 0-equivalent classes.

2) **1-Equivalence Class:** Similarly test the equivalence relation rR_1s for the string of length one or zero (i.e., length $\leq$ 1). Assume language L is define over Σ, where Σ = {10, 1} then test the relation rR_1s over symbols 0, 1, and $\in$ only and make partition of states.

3) **k-Equivalence Class:** Let states r and $s \in$ Q then r and s are k-equivalent states if and only if no string of length $\leq$ k can distinguish them. It says that, for all strings of length k or smaller upto zero (because symbol $\in$ has length 0) transition of states r and s reaches to final state.

We say that equivalence relation R_K exists between states r and s (pR_Kq). So, to distinguish the states on these bases and make partition, this partition of states is called k-equivalence classes.

Steps of Myhill Nerode Theorem

Step I: Build a two-dimensional matrix labelled by the states of the given DFA at the left and bottom side. The major diagonal and the upper triangular parts are shown as dashes.

Step II: One of the three symbols, X, x, or 0 are put in the locations where there is no dash.

1) Mark X at p, q in the lower triangular part such that p is the final state and Q is the non-final state.

2) Mark distinguished pair combination of the non-final states. If there are n number of non-final states, there are nC_2 number of distinguished pairs.

Take a pair (p, q) and final (r, s), such that $r = \delta$ (p, a) and $s = \delta$(q, a). If in the place of (r, s) there is X or x, in the place of (p, q), there will be x.

3) If (r, s) is neither X nor x, then (p, a) is 0.

4) Repeat (2) and (3) for final states also.

StepIII: The combination of states where there is 0, they are the states of the minimized machine.

Application of MNT

1) The Myhill-Nerode theorem is used to prove that a certain language is regular or not.

2) It can be also used to find the minimal number of states in a Deterministic Finite Automata(DFA).

Ques 3) Minimize the following finite automata by the Myhill-Nerode theorem.

Present State	Next State I/P = a	Next State I/P = b
→A	B	F
B	A	F
C	G	A
D	H	B
E	A	G
F	H	C
G	A	D
H	A	C

Here F, G, H are final states

Ans:

Step I: Divide the states of the DFA into two subsets: final (F) and non-final (Q – F).

F = {E, F, G}, Q – F = {A, B, C, D}

Make a two-dimensional matrix (**Figure 3.1**) labeled at the left and bottom by the states of the DFA

A	–	–	–	–	–	–	–	–
B		–	–	–	–	–	–	–
C			–	–	–	–	–	–
D				–	–	–	–	–
E					–	–	–	–
F						–	–	–
G							–	–
H								–
	A	B	C	D	E	F	G	H

Figure: 3.1

Step II:

1) The following combinations are the combination of the beginning and final states. (A, E), (A, F),(A, G), (B, E), (B, F), (B, G), (C, E), (C, F), (C, G), (D, E), (D, F), (D, G) Put X in these combinations of states. The modified matrix is given in **figure 3.2**.

B							
C							
D							
E							
F	X	X	X	X	X		
G	X	X	X	X	X		
H	X	X	X	X	X		
	A	B	C	D	E	F	G

Figure: 3.2

2) The pair combination of non-final states are (A, B), (A, C), (A, D), (A, E), (B, C), (B, D), (B, E), (C, D), (C, E) and (D, E).

$r = \delta(A, a) \rightarrow B$ $s = \delta(B, a) \rightarrow A$, in the place of (A, B), there is neither X nor x. So, in the place of (A, B), there will be 0.

Similarly,

(r, s) = δ((A, C), a) → (B, G) (there is X). In the place of (A, C), there will be x.

(r, s) = δ((A, D), a) → (B, H) (there is X). In the place of (A, D), there will be x.

(r, s) = δ((A, E), a) → (B, A) (there is neither X nor x). In the place of (A, E), there will be 0.

(r, s) = δ((B, C), a) → (A, G) (there is X). In the place of (B, C), there will be x.

(r, s) = δ((B, D), a) → (A, H) (there is X.) In the place of (B, D), there will be x.

(r, s) = δ((B, E), a) → (A, A) (there is neither X nor x, only dash). In the place of (B, E), there will be 0.

(r, s) = δ((C, D), a) → (G, H) (there is neither X nor x). In the place of (C, D), there will be 0.

(r, s) = δ((C, E), a) → (G, A) (there is X). In the place of (C, E), there will be x.

(r, s) = δ((D, E), a) → (H, A) (there is X). In the place of (D, E), there will be x.

3) The pair of combinations of final states are (F, G), (F, H), and (G, H)

(r, s) = δ((F, G), a) → (A, H) (there is X). In the place of (F, G), there will be x.

(r, s) = δ((F, H), a) → (H, A) (there is X). In the place of (F, H), there will be x.

(r, s) = δ((G, H), a) → (A, A) (there is neither X nor x, there is only dash). In the place of (G, H), there will be 0.

The modified table is given in **figure 3.3**.

B	0						
C	x	x					
D	x	x	0				
E	0	0	x	x			
F	X	X	X	X	X		
G	X	X	X	X	X	x	
H	X	X	X	X	X	x	0
	A	B	C	D	E	F	G

Figure: 3.3

The combination of entries 0 are the states of the modified machine. The states of the minimized machine are (A, B), (A, E), (B, E), (C, D), (G, H), i.e., (A, B, E), (C, D), (G, H), and (F) (As F is a final state of the machine, it is left in the state combinations).

(A, B, E) for input 'a' gives the output (A, B, A) and for input 'b' gives the output (F, F, G), where (F, F) belongs to one set and (G) belongs to another set. So, it will be divided into (A, B), (E).

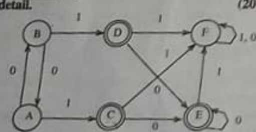
The states of the minimized machines are (A, B), (E), (C, D), (G, H), and (F). Let us name them as S_1, S_2, S_3, S_4, and S_5. For the minimized machine M'.

$Q = \{S_1, S_2, S_3, S_4, S_5\}$
$\Sigma = \{a, b\}$

State Table (transitional function δ)

Present State	Next State	
	a	b
S_1	S_1	S_1
S_2	S_3	S_4
S_3	S_4	S_3
S_4	S_3	S_3
S_5	S_4	S_5

Ques 4) Minimise the following DFA by table filling method using Myhill-Nerode theorem describing the steps in detail. **(2017 [07])**



Ans: Steps for minimisation of DFA by table filling method using myhill-nerode theorem –

1) Draw a table for all pairs of states (P,Q)
2) Marks all pairs where P∈F and Q∉F
3) If there are any Unmarked pairs (P,Q) such that [δ(P,x), δ(Q,x)] is marked, then mark [P,Q] where 'x' is an input symbol repeat this until no more marketing can be made
4) Combine all the Unmarked pairs and make them a single state in the minimized DFA

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

$(B,A) - \begin{matrix}\delta(B,0)=A \\ \delta(A,0)=B\end{matrix}\Big\}\begin{matrix}\delta(B,1)=D \\ \delta(A,1)=C\end{matrix}\Big\}$

$(D,C) - \begin{matrix}\delta(D,0)=E \\ \delta(C,0)=E\end{matrix}\Big\}\begin{matrix}\delta(D,1)=F \\ \delta(C,1)=F\end{matrix}\Big\}$

$(E,C) - \begin{matrix}\delta(E,0)=E \\ \delta(C,0)=E\end{matrix}\Big\}\begin{matrix}\delta(E,1)=F \\ \delta(C,1)=F\end{matrix}\Big\}$

$(E,D) - \begin{matrix}\delta(E,0)=E \\ \delta(D,0)=E\end{matrix}\Big\}\begin{matrix}\delta(E,1)=F \\ \delta(D,1)=F\end{matrix}\Big\}$

$(F,A) - \begin{matrix}\delta(F,0)=F \\ \delta(A,0)=B\end{matrix}\Big\}\begin{matrix}\delta(F,1)=F \\ \delta(A,1)=C\end{matrix}\Big\}$

$(F,B) = \begin{matrix}\delta(F,0=F) \\ \delta(B,0)=A\end{matrix}\Big\}$

	A	B	C	D	E	F	
A							
B							(D, C)
C	✓	✓					
D	✓	✓					(E,C) -
E	✓	✓					
F	✓	✓	✓	✓	✓		(E)

Now the unmarked states are shown below
(A, B) (D, C) (E, C) (E, D)

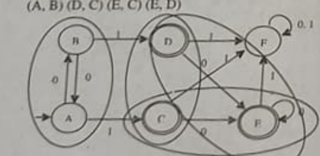


Figure 3.4

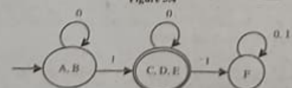


Figure 3.5

Ques 5) Let the language L = {x ∈ (0, 1)*, the second symbol from the right is a 1, then the corresponding regular expression is (0 + 1)*.1.(0 + 1) and the possible DFA that accepts given regular expression is shown in figure 3.6:

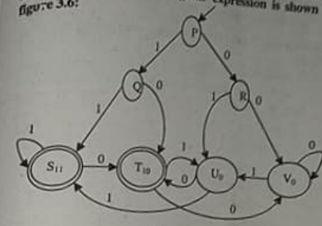


Figure 3.6

Now, is this a minimum states DFA M? If not, then construct the minimum states DFA.

Where, M = ({P, Q, R, S, T, U, V}, {0, 1}, δ, {P}, {S, T})

Ans: Find Equivalence Classes

1) **0-Equivalence Classes:** Test the transition of states over the symbol ∈ and make groups of the states which are equivalent and which are distinguishable.

Since, $PR_0Q \Rightarrow \delta(P, \epsilon) = P(\notin F)$ and $\delta(Q, \epsilon) = Q(\notin F)$ so they are distinguishable.

And, $PR_0R \Rightarrow \delta(P, \epsilon) = P(\notin F)$ and $\delta(R, \epsilon) = R(\notin F)$ so they are distinguishable.

And, $PR_0S \Rightarrow \delta(P, \epsilon) = P(\notin F)$ and $\delta(S, \epsilon) = S(\in F)$ so they are distinguishable.

Similarly test all other states with each other we find states P, Q, R, U, and V are distinguishable so, they form a group:
{P, Q, R, U, V}

Only states S and T are equivalence, because $SR_0T \Rightarrow \delta(S, \epsilon) = S(\in F)$ and $\delta(T, \epsilon) = T(\in F)$ so they are equivalence and form another group {S, T}

Hence, 0-equivalence classes are,
{P, Q, R, U, V} and {S, T}

2) **1-Equivalence Classes:** Test the equivalence relation between states of groups for the strings of length less then or equal to one (i.e., for the symbols ∈, 0, and 1).

First take-up the group of states (P, Q, R, U, V) and test for equivalence.

Symbol ∈ is not able to distinguish between states of above group so, test equivalence with respect to another symbols 0 and 1.

$PR_1Q \Rightarrow \delta(P, 0/1) = R/Q(\notin F)$ and $\delta(Q, 0/1) = T/S(\in F)$ so they are distinguishable.

And, $PR_1R \Rightarrow \delta(P, 0/1) \notin F$ and $\delta(R, 0/1) \in F$, so they are distinguishable.

And, $PR_1S \Rightarrow \delta(P, 0/1) \notin F$ and $\delta(S, 0/1) \in F$, so they are distinguishable.

And, $PR_1U \Rightarrow \delta(P, 0/1) \notin F$ and $\delta(U, 0/1) \in F$, so they are distinguishable.

And, $PR_1V \Rightarrow \delta(P, 0/1) \notin F$ and $\delta(V, 0/1) \in F$, so they are distinguishable.

Similarly test for other pairs of states in this group, we find states Q and U are equivalent because $QR_1U \Rightarrow \delta(Q, 0/1) \in F$ and $\delta(U, 0/1) \in F$, so they places in other group.

Test equivalence relation for other group of states {S, T}, we see that
$SR_1T \Rightarrow \delta(S, 0/1) \in F$ and $\delta(T, 0/1) \notin F$, since they are distinguishable so they will not be in the same group.

Hence 1-equivalence classes are,
{P, R, V}, {Q, U}, {S} and {T}
(this is a refinement of 0-equivalence class)

3) **2-Equivalence Classes:** Now test the equivalence relation for the strings ε, 0, 1, 00, 01, 10, 11 (all strings of length ≤ 2). Since we form the groups of states w.r.t strings ε, 0 and 1 so further it cannot be distinguish. Now test the equivalence relation between states of the groups only the over remaining strings. We say it xy, where x and y are either 0 or 1.

Since,
$PR_2R \Rightarrow \delta(P, 00/01) \notin F$ and $\delta(R, 00/01) \notin F$, so they are distinguishable.

and, $PR_1V \Rightarrow \delta(P, 00/01) \notin F$ and $\delta(V, 00/01) \notin F$, so they are distinguishable.

and, $RR_1V \Rightarrow \delta(R, 00/01) \notin F$ and $\delta(V, 00/01) \notin F$, so they are distinguishable.

Since, we could not find any equivalent of states in this group so there is no split in the group

Next, test for equivalence in other group {Q, U}
$QR_1U \Rightarrow \delta(Q, 00/01) \notin F$ and $\delta(U, 00/01) \notin F$, so they are distinguishable.

So there is no split in this group also.

Since, group {S} and {T} contains single state so further there is no split in these groups.

Hence, 2-equivalence classes are,
{P, R, V}, {Q, U}, {S} and {T}

That is similar to 1-equivalence classes and since there is no further refinement. So, process to find equivalence classes terminate.

Hence, equivalent states are ⇒ {P, R, V}, {Q, U}, {S} and {T}

Therefore, minimum state DFA has just above 4-states.

Transition nature of groups is given by the transitions of all the states in the group that return on the same group or some other group. Thus, we obtain the transition table shown in **table below**:

State	Input Symbol	
	0	1
{P, R, V}	Return on same group {P, R, V}	Return on group {Q, U}
→ {Q, U}	Return on group {T}	Return on group {S}
● {T}	Return on group {V}	Return on group {U}
{S}	Return on group {T}	Return on group {S}

And the minimum states DFA is shown in **figure 3.7**. [whose language is also the language expressed by the regular expression $(0 + 1)^* 1(0 + 1)$]

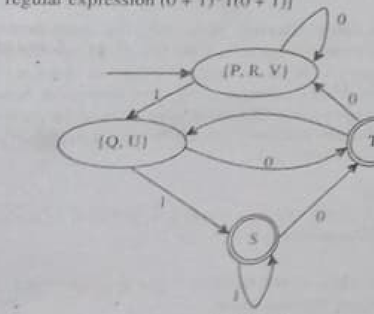


Figure 3.7

CONTEXT FREE GRAMMAR

Ques 6) Define context free grammar.
Or
What is type-2 grammar?

Ans: Context Free Grammar (CFG)/(Type-2 Grammar)

A context-free grammar (CFG) is a grammar which naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap. Context Free Grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

A context-free grammar is given as follows:

1) An alphabet Σ of **terminal symbols**, also called the object alphabet;
2) An alphabet N of **non-terminal symbols**, the elements of which are also referred to as auxiliary characters, placeholders or, variables, where $N \cap \Sigma = \phi$;
3) A special non-terminal symbol $S \in N$ called the **start symbol**;
4) A finite set of **production rules**, that is strings of the form $R \to \Gamma$ where, $R \in N$ is a non-terminal symbol and $\Gamma \in (\Sigma \cup N)^*$ is an arbitrary string of terminal and non-terminal symbols, which can be read as 'R can be replaced by Γ'.

For example, let G be the grammar = (W, Σ, R, S)

Where,

$W = \{S, A, N, V, P\} \cup \Sigma$,
$\Sigma = \{$Jim, big, green, cheese, ate$\}$,
$R = \{P \to N$,
$P \to AP$,
$S \to PV P$,
$A \to$ big,
$A \to$ green,
$N \to$ cheese,
$N \to$ Jim,
$V \to$ ate$\}$

Here, G is designed to be a grammar for a part of English; S stands for sentence, A for adjective, N for noun, V for verb, and P for phrase.

Following are some strings in L(G):

Jim ate cheese
big Jim ate green cheese
big cheese ate Jim

Unfortunately, the following are also strings in L(G):

big cheese ate green big green big cheese
green Jim ate green big Jim

Ques 7) Discuss the CFG representation of Context Free languages.

Ans: CFG representation of Context Free languages

CFG stands for context-free grammar. It is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

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

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

For example, let us construct CFG for the language having any number of a's over the set $\Sigma = \{a\}$

As we know the regular expression for the above language is r.e. $= a^*$

Production rule for the Regular expression is as follows:

$S \to aS$ Rule 1
$S \to \epsilon$ Rule 2

Now if we want to derive a string "aaaaaa", we can start with start symbols.

S
aS
aaS rule 1
aaaS rule 1
aaaaS rule 1
aaaaaS rule 1
aaaaaaS rule 1
aaaaaaϵ rule 2
aaaaaa

The r.e. $= a^*$ can generate a set of string $\{\epsilon, a, aa, aaa, ...\}$. We can have a null string because S is a start symbol and rule 2 gives $S \to \epsilon$.

Ques 8) Construct a context-free grammar G generating all integers (with sign).

Ans: Let $G = (V_N, \Sigma, P, S)$

Where, $V_N = \{S, \langle sign \rangle, \langle digit \rangle, \langle Integer \rangle\}$
$\Sigma = \{0, 1, 2, 3, ..., 9, +, -\}$

P consists of $S \to \langle sign \rangle \langle integer \rangle$, $\langle sign \rangle \to + \mid -$,
$\langle integer \rangle \to \langle digit \rangle \langle integer \rangle \mid \langle digit \rangle$
$\langle digit \rangle \to 0 \mid 1 \mid 2 \mid ... \mid 9$

L(G) = the set of all integers. **For example**, the derivation of –17 can be obtained as follows:

$S \Rightarrow \langle sign \rangle \langle integer \rangle \Rightarrow - \langle integer \rangle$
$\Rightarrow - \langle digit \rangle \langle integer \rangle \Rightarrow -1 \langle integer \rangle \Rightarrow -1 \langle digit \rangle$
$\Rightarrow -17$

Ques 9) Define CFG for the following languages over the alphabets {a, b} (2017 [09])

i) $L = \{a^{m+n} b^m c^n, m>0\}$
ii) **L contains all odd length string only**
iii) $L = \{0^n 1^n 2^n \; n>0\}$

Ans:

i) $L = \{a^{m+n} b^m, C^n, n, m > 0\}$
$= a^m a^n b^m c^n/n, m > 0$
$= a^m b^m a^n c^n/n, m > 0$
$S \to AB$
$A \to aAb/ab$
$B \to bBC/bC$

ii) Here is a context-free grammar for L = {w | the length of w is odd}:
$S \to 0T \mid 1T$
$T \to 0S \mid 1S \mid \epsilon$

Note that S and T generate words with even and odd length, respectively.

iii) $L = \{0^n 1^n 2^n, n > 0\}$
$S \to AB$
$A \to 0A1/01$
$B \to 2B/2$

Ques 10) Consider the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, a, b\}$, $\Sigma = \{a, b\}$, and R consists of the rules $S \to aSb$ and $S \to \epsilon$. Derive a string aabb from the given CFG.

Ans: A possible derivation is
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

Here the first two steps used the rule $S \to aSb$, and the last used the rule $S \to \epsilon$. In fact, it is not hard to see that $L(G) = \{a^n b^n : n \geq 0\}$. Hence some context-free languages are not regular.

Ques 11) Let $G = (V, \Sigma, R, E)$ where V, Σ, and R are as follows:
$V = \{+, *, (,), id, T, F, E\}$,
$\Sigma = \{+, *, (,), id\}$,

$R = \{E \to E + T$, ... (R1)
$E \to T$, ...
(R2)
$T \to T * F$ ......
(R3)
$T \to F$ ......
(R4)
$F \to (E)$, ...
(R5)
$F \to id\}$ ......
(R6)

The symbols E, T, and F are abbreviations for expression, term, and factor, respectively.

Derive the string (id * id + id) * (id + id) by using the given rules.

Ans: The grammar G generates the string (id * id + id) * (id + id) by the following derivation:

E	$\Rightarrow T$	by Rule R2
	$\Rightarrow T * F$	by Rule R3
	$\Rightarrow T * (E)$	by Rule R5
	$\Rightarrow T * (E + T)$	by Rule R1
	$\Rightarrow T * (T + T)$	by Rule R2
	$\Rightarrow T * (F + T)$	by Rule R4
	$\Rightarrow T * (id + T)$	by Rule R6
	$\Rightarrow T * (id + F)$	by Rule R4
	$\Rightarrow T * (id + id)$	by Rule R6
	$\Rightarrow F * (id + id)$	by Rule R4
	$\Rightarrow (E) * (id + id)$	by Rule R5
	$\Rightarrow (E + T) * (id + id)$	by Rule R1
	$\Rightarrow (E + F) * (id + id)$	by Rule R4
	$\Rightarrow (E + id) * (id + id)$	by Rule R6

$\Rightarrow$ (T + id) * (id + id)

$\Rightarrow$ (T * F + id) * (id + id) by Rule R2

$\Rightarrow$ (F * F + id) * (id + id) by Rule R3

$\Rightarrow$ (F * id + id) * (id + id) by Rule R4

$\Rightarrow$ (id * id + id) * (id + id) by Rule R6

by Rule R6

Ques 12) Prove that the grammar G = ({S}, {a, b}, S, P), with productions:

S → aSa,

S → bSb,

S → λ,

is context-free but not regular.

Ans: A typical derivation in this grammar is:

$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$

This, and similar derivations, make it clear that

$L(G) = \{ww^R : w \in \{a, b\}^*\}$

The language is context-free, but it is not regular.

Ques 13) Construct the CFG for the union of the languages $0^n 1^n$ and $a^n b^n$ for n>0. (2018 [4.5])

Ans: $L_1 = 0^n 1^n$, $L_2 = a^n b^n$

For language L_1, Converting into CFG, we get Grammar,

$S_1 \rightarrow 0S_1 1/\epsilon$

For L_2,

$S_2 \rightarrow aS_2 b/\epsilon$

So grammar for L (Union of L_1 and L_2) is:

$S \rightarrow S_1/S_2$

Ques 14) Describe the derivation tree.

Or

What is the meaning of parse tree? Explain with example.

Or

What is a Derivation Tree? (2018 [03])

Ans: Derivation/Parse Trees

Grammar can be represented using trees. Such trees representing derivations are called **derivation trees**.

A derivation tree (also called a parse tree) for a G = (V_N, Σ, P, S) is a tree satisfying the following:

1) Every vertex has a label which is a variable or terminal or Λ.
2) The root has label S.
3) The label of an internal vertex is a variable.
4) If the vertices $n_1, n_2, \ldots, n_k$ written with labels $X_1, X_2, \ldots, X_k$ are the sons of vertex n with label A, then $A \rightarrow X_1 X_2 \ldots X_k$ is a production in P.
5) A vertex n is a leaf if its label is a ∈ Σ or Λ; n is the only son of its father if its label is Λ.

For example, let G = ({S, A}, {a, b}, P, S), where P consists of S → aAS | a | SS, A → SbA | ba. **Figure 3.8** is an example of a derivation tree.

Note: Vertices 4-6 are sons of 3 written from the left, and S→aAS is in P. Vertices 7 and 8 are sons of 5 written from the left, and A→ ba is a production in P. Vertex 5 is an internal vertex and its label is A, which is a variable.

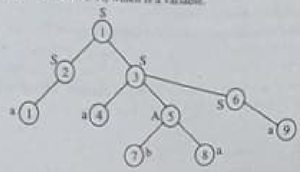


Figure 3.8: Derivation Tree

In **figure 3.8**, e.g., the sons of the root are 2 and 3 ordered from the left. So, the son of 2, viz., 10, is to the left of any son of 3. The sons of 3 ordered from the left are 4-5-6. The vertices at level 2 in the left-to-right ordering are 10-4-5-6. 4 is to the left of 5. The sons of 5 ordered from the left are 7-8. So 4 is to the left of 7. Similarly, 8 is to the left of 9. Thus the order of the leaves from the left is 10-4-7-8-9.

Ques 15) Consider G whose productions are S → aAS | a, A→SbA| SS | ba. Show that $S \overset{*}{\Rightarrow}$ aabbaa and construct a derivation tree whose yield is aabbaa.

Ans: $S \Rightarrow aAS \Rightarrow aSbAS$

$\Rightarrow aabAS \Rightarrow a^2bbaS \Rightarrow a^2b^2a^2$ .....(1)

Hence, $S \overset{*}{\Rightarrow} a^2b^2a^2$. The derivation tree is given in **figure 3.9**.

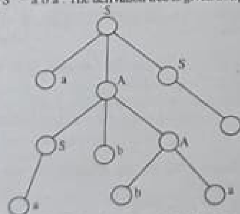


Figure 3.9: Derivation Tree with yield aabbaa

Consider G of the **example.** We have seen that $S \overset{*}{\Rightarrow} a^2b^2a^2$, and (1) gives a derivation of $a^2b^2a^2$.

Another derivation of $a^2b^2a^2$ is

$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$ .....(2)

Yet another derivation of $a^2b^2a^2$ is

$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aSbAa \Rightarrow aabAa \Rightarrow aabbaa$ .....(3)

In derivation (1), whenever we replace a variable X using a production, there are no variables to the left of X. In derivation (2), there are no variables to the right of X. But in (3), no such conditions are satisfied.

Ques 16) Let G be the grammar S → 0B | 1A, A→ 0 | 0S | 1AA, B → 1 | 1S | 0BB. For the string 00110101, find:

i) Leftmost derivation,

ii) Rightmost derivation, and

iii) Derivation tree.

Ans:

i) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011S$

$\Rightarrow 0^21^20B \Rightarrow 0^21^201S \Rightarrow 0^21^2010B \Rightarrow 0^21^20101$

ii) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1S \Rightarrow 00B10B$

$\Rightarrow 0^2B101S \Rightarrow 0^2B1010B \Rightarrow 0^2B10101$

$\Rightarrow 0^2110101.$

iii) The derivation tree is given in **figure 3.10**.

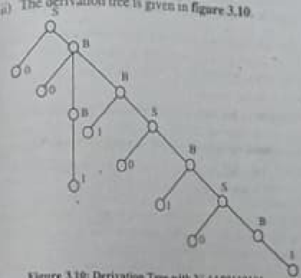


Figure 3.10: Derivation Tree with Yield 00110101

Ques 17) When is a grammar said to be ambiguous?

Or

Explain the ambiguous grammar with an example.

Ans: Ambiguity / Ambiguous Grammars

A grammar G is said to be ambiguous if there is some word in L(G) generated by more than one leftmost derivation or rightmost derivation. In other words a grammar G is said to be ambiguous if there is some word in L(G) has atleast two derivation trees.

Definition

A terminal string w ∈ L(G) is ambiguous if there exists two or more derivation trees for w (or there exist two or more leftmost derivation of w). A context-free grammar G is ambiguous if there exists some w ∈ L(G), which is ambiguous.

For example, consider, e.g., G = ({S}, {a, b, +, *}, P, S), where P consists of S → S + S | S * S | a | b, following are the possible ambiguous derivation trees of G. We have two derivation trees for a + a * b given in **figure 3.11**.

The leftmost derivations of a + a * b induced by the two derivation trees are,

$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$

$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$

Therefore, a + a * b is ambiguous.

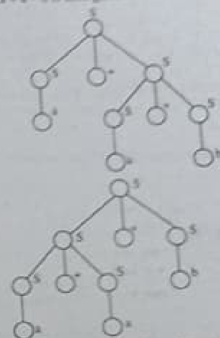


Figure 3.11: Derivation Trees for a + a * b

Ques 18) If G is the grammar S → SbS | a, show that G is ambiguous.

Ans: To prove that G is ambiguous, we have to find a w ∈ L(G), which is ambiguous. Consider w = abababa ∈ L(G). Then we get two derivation trees for w (**figure 3.12**). Thus, G is ambiguous.

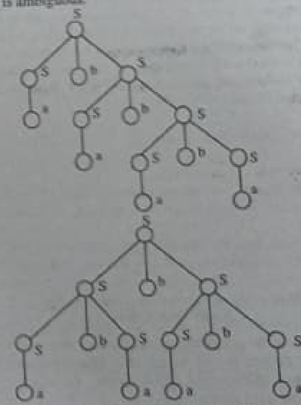


Figure 3.12: Derivation Tree for abababa

Ques 19) Is the grammar {E→E+E | E-E | id} ambiguous? Why? (2018 [03])

Ans: Given Grammar ⇒ $E \rightarrow E + E \mid E - E \mid id$

Derivation tree for the given grammar is,

Leftmost Derivation Tree

⇒ id + id – id

Right Most Derivation Tree

⇒ id + id – id

The given grammar is ambiguous since there exist more than one derivation tree.

Ques 20) How one can remove an ambiguity from CFG? Give an example.

Ans: Removal of Ambiguity from Context-Free Grammars

To remove ambiguity from CFG's, means to remove unnecessary states of a finite automation. However, the surprising fact is, that there is no algorithm whatsoever that can even tell us whether a CFG is ambiguous in the first place. Moreover, there are context-free languages that have nothing but ambiguous CFG's; for these languages, removal of ambiguity is impossible.

1) $E \rightarrow I$
2) $E \rightarrow E + E$
3) $E \rightarrow E * E$
4) $E \rightarrow (E)$
5) $I \rightarrow a$
6) $I \rightarrow b$
7) $I \rightarrow Ia$
8) $I \rightarrow Ib$
9) $I \rightarrow I0$
10) $I \rightarrow I1$

Figure 3.13: A Context-free Grammar for Simple Expressions

Fortunately, the situation in practice is not so grim. For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity.

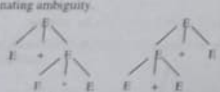


Figure 3.14: Two Parse Trees with Same Yield

First, let us note that there are two causes of ambiguity in the grammar of **figure 3.13**.

1) The precedence of operators is not respected. While **figure 3.14 (a)** properly groups the * before the + operator, **figure 3.14 (b)** is also a valid parse tree and groups the + ahead of the *. We need to force only the structure of **figure 3.14 (a)** to be legal in an unambiguous grammar.

2) A sequence of identical operators can group either from the left or from the right. **For example**, if the *'s in **figure 3.14** were replaced by +'s, we would see two different parse trees for the string E + E + E.

Since addition and multiplication are associative, it does not matter whether we group from the left or the right, but to eliminate ambiguity, we must pick one.

The conventional approach is to insist on grouping from the left, so the structure of **figure 3.14 (b)** is the only correct grouping of two + signs.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of "binding strength."

Specifically:

1) A 'factor' is an expression that cannot be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
   a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.

2) A 'term' is an expression that cannot be broken by the + operator. **For example**, the term a * b can be "broken" if we use left associativity and place a1* to its left. That is, a1 * a * b is grouped (a1 * a) * b, which breaks apart the a * b. However, placing an additive term, such as a1+, to its left or +a1 to its right cannot break a*b. The proper grouping of a1+ a*b is a1 + (a * b), and the proper grouping of a * b + a1 is (a * b) + a1.

3) An 'expression' will, henceforth, refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +. Thus, an expression is a sum of one or more terms.

$I \rightarrow a|b|Ia|Ib|I0|I1$
$F \rightarrow I|(E)$
$T \rightarrow F|T*F$
$E \rightarrow T|E+T$

Figure 3.15: Unambiguous Expression Grammar

Ques 21) Give an example for removing the ambiguity from CFG.

Ans: Figure 3.15 shows an unambiguous grammar that generates the same language as the grammar of **figure 3.14**. F, T, and E as the variables whose languages are the factors, terms, and expressions, as defined above. **For example**, this grammar allows only one parse tree for the string a + a * a; it is shown in **figure 3.16**.

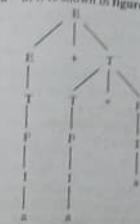


Figure 3.16: Sole Parse Tree for a + a

The fact that this grammar is unambiguous may be far from obvious. Here are the key observations that explain why no string in the language can have two different parse trees:

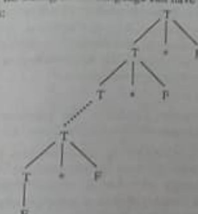


Figure 3.17: Form of All Parse Trees for a Term

1) Any string derived from T, a term, must be a sequence of one or more factors, connected by *'s. A factor, as we have defined it, and as follows from the productions for F in **figure 3.16**, is either a single identifier or any parenthesized expression.

2) Because of the form of the two productions for T, the only parse tree for a sequence of factors is the one that breaks $f_1 * f_2 * ... * f_n$, for n > 1 into a term $f_1 * f_2 * ... * f_{n-1}$ and a factor f_n.

The reason is that F cannot derive expressions like $f_{i-1} * f_i$ without introducing parentheses around them.

Thus, it is not possible that when using the production $T \rightarrow T * F$, the F derives anything but the last of the factors. That is, the parse tree for a term can only look like **figure 3.17**.

3) Likewise, an expression is a sequence of terms connected by +.

When we use the production $E \rightarrow E + T$ to derive $t_1 + t_2 + ... + t_n$, the T must derive only t_n, and the E in the body derives $t_1 + t_2 + ... + t_{n-1}$.

The reason, again, is that T cannot derive the sum of two or more terms without putting parentheses around them.

Ques 22) Whether the following grammar is ambiguous? (2019[03])

E->E+E|E*E|I
I->0|1|a|b

Ans: We have two derivation trees for a + a * b given in **figure 3.18**.

The leftmost derivations of a + a * b induced by the two derivation trees are,

$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * b$
$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * b$

Therefore, a + a * b is ambiguous.

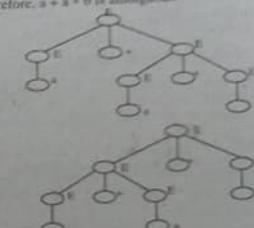


Figure 3.18: Derivation Trees for a + a * b

Ques 23) Write the various means of simplifying the grammar.

Or

What do you mean by useless symbol in a grammar? Show the elimination of useless symbols with an example. (2017 [03])

Ans: Means of Simplifying the Grammar

The means of simplification are as follows:

1) **Remove All Null Productions:** A production is said to be null production if it derives a null string (∈). **For example,** production $A \rightarrow \in$ is a null production where, A is a non-terminal and ∈ is a variable/terminal.

All non-terminals that derive the string ∈ in one/more steps of derivation are called nullable non-terminals of a grammar viz:

i) If $X \xrightarrow{*} \in$ then X is nullable.

ii) If $A \to \epsilon$ is a production then $A \Rightarrow \epsilon$, so A is nullable.

iii) If $A \to B$ and $B \to \epsilon$ are the productions then $A \Rightarrow B \Rightarrow \epsilon$ and $B \Rightarrow \epsilon$, so A and B are nullable.

iv) If $A \to BC$ and $B \to \epsilon$ and $C \to \epsilon$ are the productions then all non-terminals A, B and C are nullable.

By eliminating the null productions it is likely to increase the number of productions in the grammar. (The ambiguity characteristics of the grammar remain unaltered.)

Lemma: Let $G = (V_N, V_T, S, P)$ be a CFG that allows null production/s ($A \to \epsilon$) then the language $L(G) - \{\epsilon\}$ can be generated from a equivalent grammar $G' = (V_N', V_T', S', P')$ such that G' has no null production.

So, Grammar G' has a new production

$S_{new} \to S \mid \epsilon$ followed by all productions derives from S as usual.

Constructive Proof

Assume a grammar G has the productions

$S \to a\,b \mid AB \mid A$
$A \to B \mid a$
$B \to S \mid b \mid \epsilon$

Then remove all null productions from G.

Observe the null production/s of the grammar G. Remove all null productions such that resultant grammar generates the similar language excluding string ϵ.

i) We find the production $B \to \epsilon$ is a null production (nullable B) so remove it form G.

(–) $B \to \epsilon$
(+) $S \to A$
(+) $A \to \epsilon$

So, add two new productions in the grammar because, all the productions that derive including symbol B are manipulated according to following possibilities.

If we use the definition $B \to \epsilon$ then $S \to A\,B$ becomes $S \to A$ and $A \to B$ becomes $A \to \epsilon$.

Otherwise, production $S \to A\,B$ and $A \to B$ remains in the grammar for using next production definitions $B \to S$ and $B \to b$.

The new set of productions are

$S \to a\,b \mid AB \mid A \mid A$ remove duplicate productions, i.e.

So, Grammar becomes

$S \to a\,b \mid AB \mid A$
$A \to B \mid a \mid \epsilon$
$B \to S \mid b$

ii) Next null production is $A \to \epsilon$ (A is nullable), remove it from the grammar, i.e.

(–) $A \to \to \epsilon$
(+) $S \to \epsilon$ ($S \to \epsilon$ can be derive from $S \to A$ when $A \to \epsilon$)
(+) $S \to B$ ($S \to B$ can be derive from $S \to A\,B$ when $A \to \epsilon$)

Thus, new set of productions are

$S \to ab \mid AB \mid A \mid B \mid \epsilon$
$A \to B \mid a$
$B \to S \mid b$

iii) $S \to \epsilon$ is a nullable production (S is nullable), remove it from G, i.e.

(–) $S \to \epsilon$

(+) $B \to \epsilon$ ($B \to \epsilon$ can be derive from $B \to S$ when $S \to \epsilon$)

Thus, new set of productions are,

$S \to ab \mid AB \mid A \mid B$
$A \to B \mid a$
$B \to \epsilon \mid b$

Again symbol B becomes nullable so we find a cycle of occurrence of nullable symbols that never terminate. Hence the sequential removal of nullables might not free the grammar from null production. Therefore, we search for alternate method of elimination of null production/s.

2) **Remove All Useless Productions:** A production is said to be useless if there is no way to reach to that production in the grammar. So, a production $\alpha \to \beta$ is useless if and only if the non-terminal symbol α is non-reachable from any deriving non-terminal in the grammar such that for all productions

$\gamma \to \lambda$ then $\lambda \neq \alpha$

Then α is the useless symbol of the grammar. A symbol is again useless if it is non-terminative, i.e.

For example, $A \to aA \mid bA$. In this production there is no way to come out from the definition of A or there is no recovery derivation is defined from A. Since, A is non-terminative so A is a useless symbol and simultaneously this production is a useless production for the grammar. So, during simplification of the grammar we remove all useless production/s and also useless symbol/s.

3) **Eliminate the Unit Productions:** A production of form $X \to Y$ (where X and $Y \in V_N$) is a unit production. These productions may be useful or may not be useful (useless). If derivation of unit productions terminated on terminals then it is useful like as, whenever, $X \to Y$ is a unit production and $Y \overset{*}{\Rightarrow} \alpha \in (V_T)^*$, then we can add the production $X \to \alpha$ (after removing unit production $X \to Y$) in the set P and whenever, $X \to Y$ and $Y \to Z$ are unit productions and $Z \overset{*}{\Rightarrow} \alpha \in (V_T)^*$, then we can add the production $X \to \alpha$ (after removing unit productions $X \to Y$ and $Y \to Z$) in the set P. A cycle of unit productions is the case of all useless unit productions. **For example,** $A \to B$, $B \to C$, $C \to A$ are useless productions.

4) **Remove All Useless Symbols:** There is another approach to eliminate the useless symbols. We may start to search the useful symbols. The useful symbols are reachable symbols and active non-terminals.

Useful Symbol

A symbol X is useful if it occurs in the derivation of terminals from starting symbol S, i.e.,

$S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} x\ (\in V_T^*)$ for some α and β

[or, $S \overset{1}{\Rightarrow} x \in (V_T)^*$ then $S \to x$ is the useful production]

Active Non-Terminal

Symbol A is active non-terminal if it generates the terminal string, i.e.,

$A \overset{*}{\Rightarrow} x\ (\in V_T)^*$

Reachable Symbol

If there exist the derivation $S \overset{*}{\Rightarrow} \alpha\, X\, \beta$ (for some α and β) then symbol X is said to be reachable symbol. Start symbol is always considered as a reachable symbol. Now we discuss the Algorithm to find useful symbol:

Step 1: Find active non-terminals and drop rest of the non-active non-terminals from the grammar.

Step 2: Find reachable symbols and remove non-reachable symbols from the grammar.

Ques 24) A grammar G has the production

$S \to aXYZ \mid ab$
$X \to aAb \mid A \mid a$
$Y \to bBa \mid B \mid b$
$Z \to a \mid aA \mid XY$
$A \to \epsilon \mid a \mid aA$
$B \to \epsilon \mid b \mid bB$

Simplify the grammar (by removing all null productions).

Ans: Find nullable symbols that are

i) B [$\therefore B \Rightarrow \epsilon$]
ii) A [$\therefore A \Rightarrow \epsilon$]
iii) Y [$\therefore Y \Rightarrow B \Rightarrow \epsilon$]
iv) X [$\therefore X \Rightarrow A \Rightarrow \epsilon$]
v) Z [$\therefore Z \Rightarrow XY \Rightarrow \epsilon Y \Rightarrow \epsilon\epsilon \Rightarrow \epsilon$]

So, {B, A, Y, X, Z} are nullables.

After dropping the nullables add following productions, i.e.

(+) $X \to a\,b$ [$X \to aAb$ when $A \to \epsilon$ is removed]
(+) $Y \to b\,a$ [$Y \to bBa$ when $B \to \epsilon$ is removed]
(+) $Z \to X$ [$Z \to XY$ when $Y \to \epsilon$ is removed]
(+) $Z \to Y$ [$Z \to XY$ when $X \to \epsilon$ is removed]
(+) $S \to a\,Y\,Z$ [$S \to aXYZ$ when $X \to \epsilon$ is removed]
(+) $S \to a\,X\,Z$ [$S \to aXYZ$ when $Y \to \epsilon$ is removed]
(+) $S \to a\,X\,Y$ [$S \to aXYZ$ when $Z \to \epsilon$ is removed]

Also production

(+) $Z \to a$ [$Z \to aA$ when $A \to \epsilon$] but this is a repeatative production so there is no need to add further into the grammar. Hence the new grammar G' has following productions:

$S \to aXYZ \mid ab \mid aYZ \mid aXZ \mid aXY$
$X \to aAb \mid A \mid a \mid ab$
$Y \to bBa \mid B \mid b \mid ba$
$Z \to a \mid aA \mid XY$
$A \to a \mid aA$
$B \to b \mid bB$

Ques 25) Consider the grammar

$S \to S + T \mid T$
$T \to T * F \mid F$
$F \to (S) \mid a$

Remove unit productions from the grammar.

Ans:

i) Remove the unit production $S \to T$, thus we can add following productions i.e.,
a) $T \Rightarrow T \Rightarrow F$, so add production $S \to T * F$
b) $T \Rightarrow F \Rightarrow a$, so add production $S \to a$
c) $T \Rightarrow F \Rightarrow (S)$, so add production $S \to (S)$

ii) Remove the unit production $T \to F$, thus we can add following productions i.e.,
a) $F \Rightarrow (S)$, so add production $T \to (S)$
b) $F \Rightarrow a$, so add production $T \to a$

iii) No other production is the unit production. Hence grammar left with following productions which are free from unit productions

$S \to S + T \mid T * F \mid a \mid (S)$
$T \to T * F \mid (S) \mid a$
$F \to (S) \mid a$

Ques 26) A grammar G is given, find an equivalent grammar with no useless symbols.

$S \to AB \mid AC$
$A \to aAb \mid bAa \mid a$
$B \to bbA \mid aaB \mid AB$
$C \to abCa \mid aDb$
$D \to bD \mid aC$

Ans: Since grammar G uses the non-terminals {S, A, B, C, D} and terminals {a, b}. We test the non-terminals and find the active non-terminals and drop non-active ones, i.e.,

i) $D \Rightarrow bD$ or $D \Rightarrow aC$ never terminates on terminals so non-active;
ii) $C \Rightarrow abCa$ or $C \Rightarrow aDb \Rightarrow$ never terminates on terminals so non-active;
iii) $B \Rightarrow bbA \Rightarrow bba$; so active non-terminal;
iv) $A \Rightarrow a$; so active non-terminal;
v) $S \Rightarrow AB \Rightarrow$ terminated on terminals because both A and B are active; so active one.

Since, {S, A, B} are only active non-terminals so the productions defined and using these symbols are:

$S \to AB$
$A \to aAb \mid bAa \mid a$
$B \to bbA \mid aaB \mid AB$

Now, we test the reachability of the symbols used in grammar G.

i) Symbols A and B are reachable because $S \Rightarrow AB$.
ii) Symbol a and b are also reachable because $S \Rightarrow AB \Rightarrow aAbB$ and so on.

Hence, all symbols used in the previous simplified step of grammar {S, A, B, a, b} are reachable symbols. So, the simplified grammar with no useless symbols is

$S \rightarrow AB$
$A \rightarrow aAb \mid bAa \mid a$
$B \rightarrow bbA \mid aaB \mid AB$

Ques 27) What is normal form? Also explain the various normal forms.

Or

Define Chomsky normal form with an example.

Or

Explain the Greibach normal form.

Ans: Normal Forms

A normal form is a specific format for productions, because it is specific, it is easier to analyze, and it must allow all context-free languages to be generated by a grammar in that normal form. Normal form grammars are easy to handle and are useful in proving results.

Types of Normal Forms

1) **Chomsky Normal Form(CNF):** In the Chomsky normal form, we have restrictions on the length of R.H.S. and the nature of symbols in the R.H.S. of productions.

A context-free grammar G is in Chomsky normal form if every production is of the form $A \rightarrow a$, or $A \rightarrow BC$, and $S \rightarrow \Lambda$ is in G if $\Lambda \in L(G)$. When Λ is in L(G), we assume that S does not appear on the R.H.S. of any production.

For example, consider G whose production are $S \rightarrow AB \mid \Lambda$, $A \rightarrow a$, $B \rightarrow b$. Then G is in Chomsky normal form. For a grammar in CNF, the derivation tree has the following property: Every node has atmost two descendants – either two internal vertices or a single leaf.

2) **Greibach Normal Form(GNF):** A context-free grammar is in Greibach normal form if every production is of the form $A \rightarrow a\alpha$, where $\alpha \in V_N^*$ and $a \in \Sigma$ (α may be $\in$), and $S \rightarrow \Lambda$ is in G if $\Lambda \in L(G)$. When $\Lambda \in L(G)$, we assume that S does not appear on the R.H.S. of any production.

For example, G given by $S \rightarrow aAB \mid \Lambda$, $A \rightarrow bC$, $B \rightarrow b$, $C \rightarrow c$ is in GNF.

Ques 28) Prove that for every context-free grammar, there is an equivalent grammar G_2 in Chomsky normal form.

Or

Write the steps of reduction to Chomsky normal form.

Ans: Reduction to Chomsky normal form

Step 1: Elimination of Null Productions and Unit Productions: Let the grammar thus obtained be $G = (V_N, \Sigma, P, S)$.

Step 2: Elimination of Terminals on R.H.S: We define $G_1 = (V'_N, \Sigma, P_1, S)$, where P_1 and V'_N are constructed as follows:

All the productions in P of the form $A \rightarrow a$ or $A \rightarrow BC$ are included in P_1. All the variables in V_N are included in V'_N. Consider $A \rightarrow X_1X_2 \ldots X_n$ with some terminal on R.H.S. If X_i is a terminal, say a_i, add a new variable C_{a_i} to V'_N and $C_{a_i} \rightarrow a_i$ to P_1. In production $A \rightarrow X_1X_2 \ldots X_n$, every terminal on R.H.S. is replaced by the corresponding new variable and the variables on the R.H.S. are retained. The resulting production is added to P_1. Thus we get $G_1 = (V'_N, \Sigma, P_1, S)$.

Step 3: Restricting the Number of Variables on R.H.S: For any production in P_1 the R.H.S. consists of either a single terminal (or Λ in $S \rightarrow \Lambda$) or two or more variables.

We define $G_2 = (V''_N, \Sigma, P_2, S)$ as follows:
All productions in P_1 are added to P_2 if they are in the required form. All the variables in V'_N are added to V''_N. Consider $A \rightarrow A_1A_2 \ldots A_m$, where $m \geq 3$. We introduce new productions $A \rightarrow A_1C_1$, $C_1 \rightarrow A_2C_2$, ..., $C_{m-2} \rightarrow A_{m-1}A_m$ and new variables $C_1, C_2, \ldots, C_{m-2}$. These are added to P'' and V''_N, respectively. Thus, we get G_2 in Chomsky normal form.

Ques 29) Find a grammar in Chomsky normal form equivalent to $S \rightarrow aAbB$, $A \rightarrow aA \mid a$, $B \rightarrow bB \mid b$.

Ans: As there are no unit productions or null productions, so we omit step 1. We proceed to step 2.

Step 1: Let $G_1 = (V'_N, \{a, b\}, P_1, S)$, where P_1 and V'_N are constructed as follows:

i) $A \rightarrow a$, $B \rightarrow b$ are added to P_1.
ii) $S \rightarrow aAbB$, $A \rightarrow aA$, $B \rightarrow bB$ yield $S \rightarrow C_aAC_bB$, $A \rightarrow C_aA$, $B \rightarrow C_bB$, $C_a \rightarrow a$, $C_b \rightarrow b$
$V'_N = \{S, A, B, C_a, C_b\}$

Step 2: P_1 consists of $S \rightarrow C_aAC_bB$, $A \rightarrow C_aA$, $B \rightarrow C_bB$, $C_a \rightarrow a$, $C_b \rightarrow b$, $A \rightarrow a$, $B \rightarrow b$.
$S \rightarrow C_aAC_bB$ is replaced by $S \rightarrow C_aC_1$, $C_1 \rightarrow AC_2$, $C_2 \rightarrow C_bB$.

The remaining productions in P_1 are added to P_2. Let $G_2 = (\{S, A, B, C_a, C_b, C_1, C_2\}, \{a, b\}, P_2, S)$.

Where, P_2 consists of $S \rightarrow C_aC_1$, $C_1 \rightarrow AC_2 \rightarrow C_2 \rightarrow C_bB$, $A \rightarrow C_aA$, $B \rightarrow C_bB$, $C_a \rightarrow a$, $C_b \rightarrow b$, $A \rightarrow a$, and $B \rightarrow b$.

G_2 is in CNF and equivalent to the given grammar.

Ques 30) How one can convert a CFG into Chomsky normal form?

Or

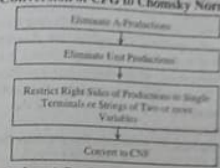
Write the steps of conversion of CFG into Chomsky normal form.

Ans: Converting a CFG to Chomsky Normal Form

Let G be the grammar with productions

$S \rightarrow AACD$
$A \rightarrow aAb \mid \Lambda$
$C \rightarrow aC \mid a$
$D \rightarrow aDa \mid bDb \mid \Lambda$

Steps in Conversion of CFG to Chomsky Normal Form



1) **Eliminating Λ-Productions:** The nullable variables are A and D, and produces the grammar with productions
$S \rightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid C$
$A \rightarrow aAb \mid ab$
$C \rightarrow aC \mid a$
$D \rightarrow aDa \mid bDb \mid aa \mid bb$

2) **Eliminating Unit Productions:** Here we may simply add the productions
$S \rightarrow aC \mid a$
and delete $S \rightarrow C$

3) **Restricting the Right Sides of Productions to Single Terminals or Strings of Two or more Variables:** This step yields the productions
$S \rightarrow AACD \mid ACD \mid AAC \mid CD \mid AC \mid X_aC \mid a$
$A \rightarrow X_aAX_b \mid X_aX_b$
$C \rightarrow X_aC \mid a$
$D \rightarrow X_aDX_a \mid X_bDX_b \mid X_aX_a \mid X_bX_b$
$X_a \rightarrow a$
$X_b \rightarrow b$

4) **Final Step to CNF:** There are six productions whose right sides are too long. Applying our algorithm produces the grammar with productions

S	$\rightarrow AT_1$ $T_1 \rightarrow AT_2$	$T_2 \rightarrow CD$
S	$\rightarrow AU_1$	$U_1 \rightarrow CD$
S	$\rightarrow AV_1$	$V_1 \rightarrow AC$
S	$\rightarrow CD \mid AC \mid X_aC \mid a$	
A	$\rightarrow X_aW_1$	$W_1 \rightarrow AX_b$
A	$\rightarrow X_aX_b$	
C	$\rightarrow X_aC \mid a$	
D	$\rightarrow X_aY_1$	$Y_1 \rightarrow DX_a$
D	$\rightarrow X_bZ_1$	$Z_1 \rightarrow DX_b$
D	$\rightarrow X_aX_a \mid X_bX_b$	
X_a	$\rightarrow a$	
$X_b \rightarrow b$		

Ques 31) Convert the grammar {$S \rightarrow AaCb \mid ABa$, $A \rightarrow bAa \mid a$, $B \rightarrow BaB \mid b$, $C \rightarrow \epsilon$} to Chomsky normal form. (2018 [4.5])

Ans:
$S \rightarrow AaCb/ABa$
$A \rightarrow bAa/a$
$B \rightarrow BaB/b$
$C \rightarrow \epsilon$

Step 1: In production rule $S \rightarrow AaCb/ABa$, $A \rightarrow bAa/a$ and $B \rightarrow BaB$, terminals a and b exists along with non-terminals. Removing them from RHS

$S \rightarrow AX_aCY_b/ABX_a$
$A \rightarrow Y_bAX_a/a$
$B \rightarrow BX_aB$
$C \rightarrow \epsilon$
$X_a \rightarrow a$
$Y_b \rightarrow b$

Step 2: Production Rule $S \rightarrow AX_aCY_b/ABX_a$, $A \rightarrow Y_bAX_a$ and $B \rightarrow BX_aB$ contain more than two symbols, removing them from grammar yields

$S \rightarrow PCY_b/PY_a$
$A \rightarrow Y_bP/a$
$B \rightarrow QB/b$
$C \rightarrow \epsilon$
$X_a \rightarrow a$
$Y_b \rightarrow b$
$P \rightarrow AX_a$
$Q \rightarrow BX_a$

Repeating Step 2 for $S \rightarrow PCY_b$,
$S \rightarrow PR/PY_a$
$A \rightarrow Y_bP/a$
$B \rightarrow QB/b$
$C \rightarrow \epsilon$
$X_a \rightarrow a$
$Y_b \rightarrow b$
$P \rightarrow AX_a$
$Q \rightarrow BX_a$
$R \rightarrow CY_b$

This is the required Chomsky normal form of the given grammar.

Ques 32) Convert the following grammar to Chomsky Normal Form: (2019[03])
S->0S0|1S1|e

Ans: We have the following grammar
$S \rightarrow 0S0 \mid 1S1 \mid \epsilon$ Or $S \rightarrow aSa \mid bSb \mid \epsilon$

A grammar is said to be in Chomsky Normal Form (CNF):
1) If each terminal produce exactly one non-terminal
2) If each terminal produces exactly two terminals.

Step 1) Remove Empty/Null Production

$S \to 0S0 \mid 1S1 \mid 00 \mid 11$ Or

$S \to aSa \mid bSb \mid aa \mid bb$

Step 2) Remove Unit Production

If there is any capital letter on right side change it small and odd new rule. There is no such case.

Step 3) Convert all mix terms into capitals (Solid non-terminals)

$S \to ASA \mid BSB \mid AA \mid BB$

$A \to a, B \to b$

Step 4) Reduce the length of each non-terminals upto 2 and add new rule.

$S \to AR \mid BT \mid AA \mid BB$

$R \to SA, T \to SB, A \to a, B \to b$

Ques 33) Let $G = (V_N, \Sigma, P, S)$ be a context-free grammar. Let the set of A-productions be $A \to A\alpha_1 \mid \ldots A\alpha_r \mid \beta_1 \mid \ldots \mid \beta_s$ (β's do not start with A). Let Z be a new variable. Let $G_1 = (V_N \cup \{Z\}, \Sigma, P_1, S)$, where P_1 is defined as follows:

1) **The set of A-productions in P_1 are $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_s$**
   $A \to \beta_1 Z \mid \beta_2 Z \mid \ldots \mid \beta_s Z$.
2) **The set of Z-productions in P_1 are $Z \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_r$**
   $Z \to \alpha_1 Z \mid \alpha_2 Z \ldots \mid \alpha_r Z$.
3) **The productions for the other variables are as in P. Then G_1 is a CFG and equivalent to G.**

Ans: We prove the theorem when $\Lambda \notin L$ and then extend the construction to L having Λ.

Case 1: Construction of G (when $\Lambda \notin L$)

Step 1: We eliminate null productions and then construct a grammar G in Chomsky normal form generating L. We rename the variables as $A_1, A_2, \ldots, A_n$ with $S = A_1$. We write G as $(\{A_1, A_2, \ldots, A_n\}, \Sigma, P, A_1)$.

Step 2: To get the productions in the form $A_i \to a\gamma$ or $A_i \to A_j\gamma$, where $j > i$, convert the A_i-productions ($i = 1, 2, \ldots, n-1$) to the form $A_i \to A_j\gamma$ such that $j > i$. Prove that such modification is possible by induction on i.

Consider A_1-productions. If we have some A_1-productions of the form $A_1 \to A_1\gamma$, where $j > i$, convert, the A_i-production ($i = 1, 2, \ldots, n-1$) to the form $A_i \to A_j\gamma$ then we can apply **lemmas** to get rid of such productions. We get a new variable, say Z_1, and A_1-productions of the form $A_1 \to a$ or $A_1 \to A_j\gamma'$, where $j > 1$. Thus there is basis for induction.

Assume we have modified A_1-productions, A_2-productions ... A_i-productions.

Consider A_{i+1}-productions. Productions of the form $A_{i+1} \to a\gamma$ required no modification. Consider the first symbol (this will be a variable) on the R.H.S. of the remaining A_{i+1}-productions.

Let t be the smallest index among the indices of such symbols (variables). If $t > i + 1$, there is nothing to prove. Otherwise, apply induction hypothesis to A_t-productions for $t \le i$. So any A_t-production is of the form $A_t \to A_j\gamma$, where $j > t$ or $A_t \to a\gamma'$.

Now we can apply **lemmas** to A_{i+1}-production whose R.H.S. starts with A_t. The resulting A_{i+1}-productions are of the form $A_{i+1} \to A_j\gamma$, where $j > t$ (or $A_{i+1} \to a\gamma'$).

We repeat the above construction by finding t for the new set of A_{i+1}-productions. Ultimately, the A_{i+1}-productions are converted to the form $A_{i+1} \to A_j\gamma$, where $j \ge i+1$ or $A_{i+1} \to a\gamma'$. Productions of the form $A_{i+1} \to A_{i+1}\gamma$ can be modified. Thus, we have converted A_{i+1}-productions to the required form.

By the principle of induction, the construction can be carried out for $i = 1, 2, \ldots, n$. Thus for $i = 1, 2, \ldots, n-1$, any A_i-production is of form $A_i \to A_j\gamma$, where $j > i$ or $A_i \to a\gamma'$. Any A_n-production is of the form $A_n \to A_n\gamma$ or $A_n \to a\gamma'$.

Step 3: Convert A_n-production to the form $A_n \to a\gamma$. Here, productions of the form $A_n \to A_n\gamma$ are eliminated using **lemmas**. The resulting A_n-productions are of the form $A_n \to a\gamma$.

Step 4: Modify A_i-productions to the form $A_i \to a\gamma$ for $i = 1, 2, \ldots, n-1$. At the end of step 3, the A_n-productions are of the form $A_n \to a\gamma$. The A_{n-1}-prouctions are of the form $A_{n-1} \to a\gamma'$ or $A_{n-1} \to A_n\gamma$.

By applying **lemmas**, we eliminate productions of the form $A_{n-1} \to A_n\gamma$. The resulting A_{n-1}-productions are in the required form. We repeat the construction by considering $A_{n-2}, A_{n-3}, \ldots, A_1$.

Step 5: Modify Z_i-productions. (We take it as Z_i, when we apply the lemma for A_i-productions.) The Z_i-productions are of the form $Z_i \to \alpha Z_i$ or $Z_i \to \alpha$ (where α is obtained from $A_i \to A_i\alpha$), and hence of the form $Z_i \to a\gamma$ or $Z_i \to A_k\gamma$ for some k.

At the end of step 4, the R.H.S. of any A_k-produciton starts with a terminal. So we can apply **lemmas** to eliminate $Z_i \to A_k\gamma$. Thus at the end of step 5, we get an equivalent grammar G_1 in GNF.

It is easy to see that G_1 is in GNF. We start with G in CNF. In G any A-production is of the form $A \to a$ or $A \to AB$ or $A \to CD$.

When we apply **lemmas** in step 2, we get new productions of the form $A \to a\alpha$ or $A \to \beta$, where $\alpha \in V_N^*$ and $\beta \in V_N^*$ and $a \in \Sigma$. In steps 3-5, the productions are modified to the form $A \to a\alpha$ or $Z \to a'\alpha'$, where $a, a' \in \Sigma$ and $\alpha, \alpha' \in V_N^*$.

Case 2: Construction of G when $\Lambda \in L$. By the previous construction we get $G' = (V_N', \Sigma, P_1, S)$ in GNF such that $L(G') = L - \{\Lambda\}$. Define a new grammar G_1, as

$G_1 = (V_N' \cup \{S'\}, \Sigma, P_1 \cup \{S' \to S, S' \to \Lambda\}, S')$

$S' \to S$ can be eliminated. As S-productions are in the required form, S'-productions are also in the required form. So $L(G) = L(G_1)$ and G_1 is in GNF.

Ques 34) Find the Greibach normal form grammar equivalent to the following CFG: (2019[4.5])

$S \to AB$

$A \to BS \mid 1$

$B \to SA \mid 0$

Ans: Rearranging by an order:

$A_1 \to A_2A_3, A_2 \to A_3A_1 \mid 1, A_3 \to A_1A_2 \mid 0$

Again we have:

$A_1 \to A_2A_3, A_2 \to A_3A_1 \mid 1, A_3 \to A_2A_3A_2 \mid 0$

Rewriting we get:

$A_1 \to A_2A_3, A_2 \to A_3A_1 \mid 1$

$A_3 \to \underbrace{A_3 A_1 A_3 A_2}_{\alpha} \mid \underbrace{1 A_3 A_2}_{\beta_1} \mid \underbrace{0}_{\beta_2}$

Or

$A_1 \to A_2A_3, A_2 \to A_3A_1 \mid 1, A_3 \to 1 A_3 A_2 \mid 0$

$A_3 \to 1 A_3 A_2 Z \mid 0Z, Z \to A_1 A_3 A_2 Z \to A_1 A_3 A_2 Z$

Now we change A_1 rule to GNF

$A_1 \to A_2A_3, A_2 \to 1A_3 A_2 A_1 \mid 0A_1 \mid 1A_3A_2ZA_1 \mid 0ZA_1 \mid 1$

$Z \to A_1 A_3 A_2 Z \to A_1 A_3 A_2 Z$

Applying lemma, we get:

$A_1 \to 1A_3 A_2 A_1 A_3 \mid 0A_1A_3 \mid 1A_3A_2ZA_1A_3 \mid 0ZA_1A_3 \mid 1A_3$

$A_2 \to 1A_3 A_2 A_1 \mid 0A_1 \mid 1A_3A_2ZA_1 \mid 0ZA_1 \mid 1$

$Z \to A_1 A_3 A_2 Z \to A_1 A_3 A_2 Z$

Now Changing Z → GNF

$A_1 \to 1A_3 A_2 A_1 A_3 \mid 0A_1A_3 \mid 1A_3A_2ZA_1A_3 \mid 0ZA_1A_3 \mid 1A_3$

$A_2 \to 1A_3 A_2 A_1 \mid 0A_1 \mid 1A_3A_2ZA_1 \mid 0ZA_1 \mid 1$

$A_3 \to 1 A_3 A_2 \mid 0, A_3 \to 1 A_3 A_2 Z \mid 0Z$

$Z \to 1A_3\ A_2\ A_1A_3\ A_3\ A_2 \mid 0A_1A_3\ A_3\ A_2 \mid 1A_3A_2ZA_1A_3A_3A_2 \mid 0ZA_1A_3A_3A_2 \mid 1A_3A_3A_2$

$Z \to 1A_3\ A_2\ A_1A_3\ A_3\ A_2Z \mid 0A_1A_3\ A_3\ A_2Z \mid 1A_3A_2ZA_1A_3A_3A_2Z \mid 0ZA_1A_3A_3A_2Z \mid 1A_3A_3A_2Z$

Ques 35) Find a grammar in GNF equivalent to the grammar $E \to E + T \mid T, T \to T * F \mid F, F \to (E) \mid a$.

Ans: Step1: We first eliminate unit productions. Hence

$W_0(E) = \{E\}$,

$W_1(E) = \{E\} \cup \{T\} = \{E, T\}$

$W_2(E) = \{E, T\} \cup \{F\} = \{E, T, F\}$

So, $W(E) = \{E, T, F\}$

So, $W_0(T) = \{T\}, W_1(T) = \{T\} \cup \{F\} = \{T, F\}$

Thus,

$W(T) = \{T, F\}$

$W_0(F) = \{F\}, \quad W_1(F) = \{F\} = W(F)$

The equivalent grammar without unit productions is, therefore, $G_1 = (V_N, \Sigma, P_1, S)$, where P_1 consists of

1) $E \to E + T \mid T * F \mid (E) \mid a$
2) $T \to T * F \mid (E) \mid a$, and
3) $F \to (E) \mid a$

We apply step 2 of reduction to CNF. We introduce new variables A, B, C corresponding to +, *,). The modified productions are:

1) $E \to EAT \mid TBF \mid (EC \mid a$
2) $T \to TBF \mid (EC \mid a$
3) $F \to (EC \mid a$
4) $A \to +, B \to *, C \to)$

The variables A, B, C, F, T and E are re-named as $A_1, A_2, A_3, A_4, A_5, A_6$. Then the productions become

$A_1 \to +, A_2 \to *, A_3 \to), A_4 \to (A_6A_3 \mid a$ ......(1)

$A_5 \to A_5A_2A_4 \mid (A_6A_3 \mid a$

$A_6 \to A_6A_1A_5 \mid A_5A_2A_4 \mid (A_6A_3 \mid a$

Step 2: We have to modify only A_5- and A_6-productions. $A_5 \to A_5A_2A_4$ can be modified. The resulting productions are:

$A_5 \to (A_6A_3 \mid a, \quad A_5 \to (A_6A_3Z_5 \mid aZ_5)$ ......(2)

$Z_5 \to A_2A_4 \mid A_2A_4Z_5$

$A_6 \to A_5A_2A_4$ can be modified. The resulting productions are

$A_6 \to A_6A_1A_5A_4 \mid aA_2A_4 \mid (A_6A_3Z_5A_2A_4 \mid aZ_5A_2A_4$

$A_6 \to A_6A_3 \mid a$ are in the proper form.

Step 3: $A_6 \to A_6A_1A_5$ can be modified.

The resulting productions give all the A_6-productions:

$A_6 \to (A_6A_3A_2A_4 \mid aA_2A_4 \mid (A_6A_3Z_5A_2A_4$

$A_6 \to aZ_5A_2A_4 \mid (A_6A_3) \mid a$ ......(3)

$A_6 \to (A_6A_3A_2A_4Z_6 \mid aA_2A_4Z_6 \mid (A_6A_3Z_5A_2A_4Z_6$

$A_6 \to aZ_5A_2A_4Z_6 \mid (A_6A_3Z_6 \mid aZ_6$ ......(4)

$Z_6 \to A_1A_5 \mid A_1A_5Z_6$

Step 4: This step is not necessary as A_i-productions for $i = 5, 4, 3, 2, 1$ are in the required form.

Step 5: The Z_5-productions are $Z_5 \to A_2A_4 \mid A_2A_4Z_5$. These can be modified as

$Z_5 \to * A_4 \mid * A_4Z_5$ ......(5)

The Z_6-productions are $Z_6 \to A_1A_5 \mid A_1A_5Z_6$. These can be modified as:

$Z_6 \to + A_5 \mid + A_5Z_6$ ......(6)

The required grammar in GNF is given by (1)-(6).

Ques 36) Prove that E $\{ww \mid w \in \{0, 1\}^*\}$ is not a CFL.

Or

Consider L = $\{ww \mid w \in \{0, 1\}^*\}$. Prove L is not a CFL. **(2018 [05])**

Ans: Suppose E is context-free. Let p be the pumping length.

1) Consider $z = 0^p 1^p 0^p 1^p \in L$.
2) Since $|z| > p$, there are u, v, w, x, y such that $z = uvwxy$, $|vwx| \le p$, $|vx| > 0$ and $uv^i wx^i y \in L$ for all $i \ge 0$.
3) vwx must straddle the midpoint of z.
   i) Suppose vwx is only in the first half. Then in uv^2wx^2y the second half starts with 1. Thus, it is not of the form ww.
   ii) Case when vwx is only in the second half. Then in uv^2wx^2y the first half ends in a 0. Thus, it is not of the form ww.
   iii) Suppose vwx straddles the middle. Then uv^0wx^0y must be of the form $0^p 1^i 0^j 1^p$, where either i or j is not p. Thus $ux^0wx^0y \notin E$.

Ques 37) Find a CFG without $\in$-productions equivalent to the grammar defined by: **(2019[4.5])**
$S \to ABaC$, $A \to BC$, $B \to b/\in$, $C \to D/\in$, $D \to d$

Ans: The grammar is not in minimized format because there are there are three unit productions $A \to C$, $A \to B$ and $C \to D$. by removing the unit productions from the grammar, the minimized grammar will become:

$S \to ABaC | AaC | ABa | Aa | BaC | aC | Ba | a,$

$A \to BC | d | b,\ B \to b,\ C \to d,\ D \to d,$

Again in the grammar there is a non-reachable symbol D. By removing the non-reachable symbol the minimized grammar will be:

$S \to ABaC | AaC | ABa | Aa | BaC | aC | Ba | a,$

$A \to BC | d | b,\ B \to d,\ C \to d$

Ques 38) Give a CFG for the language N(M) where M = ({p,q,r}, {0, 1}, {Z, X_0} δ, q_0, Z, r) and δ is given by δ(p, ∈, X_0) = {(q, ZX_0)}, δ(q, ∈, X_0) = {(r, ∈)}, δ(q, 1, Z) = {(q, ZZ)}, δ(q, 0, Z) = {(q, ∈)}. **(2019[4.5])**

Ans: The language is the set of strings with some number of 1's followed by one more 0, that is , $\{1^n 0^{n+1} \mid n >= 0\}$. This set is exactly those if-else violation that consist of a block of it's followed by a block of else's.

The language is evidently a CFL, generated by the grammar with productions $S \to 1S0|0$.

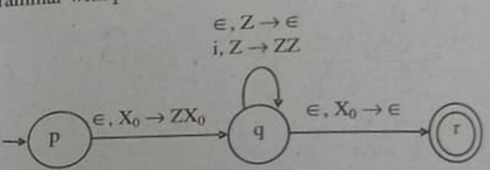


Figure 3.19

Module 4

More on Context-Free Languages

PUSHDOWN AUTOMATA (PDA)

Ques 1) What is a pushdown automaton?

Or

What do you mean by PDA? How it is different from finite state machine?

Ans: Pushdown Automaton (PDA)

Pushdown automata (PDA) are a way to represent the language class called context free languages. Pushdown automata are abstract devices defined in theory of automata. They access a potentially unlimited amount of memory in form of a stack.

A finite automaton cannot accept a language having strings of the form $a^n b^n$ because a finite automaton has to remember the number of a's in the string. For this, a finite automaton will require an infinite number of states. To overcome this problem an additional auxiliary memory in the form of stack is included. In stack the stored elements are processed in last in first out (LIFO) fashion.

To accept strings of the form $a^n b^n$ given by language L, a's are pushed (the insert operation in stack is called push and deletion is called pop) to the stack and when 'b' encounters the topmost 'a' from stack is deleted. Thus the matching of number of a's and b's is accomplished. This type of additional arrangement in a finite automaton is called **pushdown automaton** as shown in **figure 4.1.**

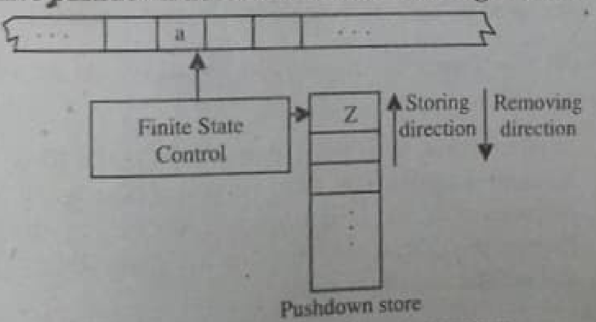


Figure 4.1: Model of Pushdown Automaton

Mathematical Definition: A pushdown automata is defined by seven tuple,

$$A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Where,

1) A finite non-empty set of states denoted by Q.
2) A finite non-empty set of input symbols denoted by Σ.
3) A finite non-empty set of pushdown symbols denoted by Γ.
4) The transition function δ from $Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.
5) A special state called the initial state denoted by q_0 such that $q_0 \in Q$.
6) A special pushdown symbol called the initial symbol on the pushdown store denoted by Z_0, such that $Z_0 \in \Gamma$.
7) The set of final states, denoted by F such that $F \subseteq Q$.

The arguments of δ are current state of the control unit, the current input symbol and current symbol on top of the stack. The result is a set of pairs (q, x) where q is the next state and x is a string which put on top of the stack in place of the single symbol there before. Note that second argument of δ may be Λ indicating that a move that does not consume an input symbol possible. We will call such a move a Λ-transition. δ is defined so that it needs a stack symbol no move is possible if stack is empty. Finally the requirement that the range of δ be finite subset is necessary because $Q \times \Gamma^*$ is an infinite set and therefore has infinite subsets.

For example, suppose the set of transition rules in PDA contains,

$$\delta(q_1, a, b) = \{(q_2, a)\} \quad \text{......(1)}$$

$$\delta(q_1, b, a) = \{(q_2, \Lambda)\} \quad \text{......(2)}$$

According to rule (1), if at any time the PDA is in state q_1, the input symbol read is a, the symbol on top of the stack is b, then the PDA goes into state q_2 and a replaces b on the top of the stack.

According to rule(2), if the PDA is in state q1, the input symbol read is b, the symbol on top of the stack is a then PDA goes into state q1 and the symbol a is removed from the stack.

Graphical Notations of PDA

The list of δ facts is not too easy to follow. Sometimes, a diagram, generalizing the transition diagram of a finite automation, will make aspects of the behavior of a given PDA clearer. We shall therefore introduce and subsequently use a transition diagram for PDA's in which:

1) **Nodes:** The nodes correspond to the states of the PDA.
2) **Arrow:** An arrow labeled Start indicates the start state, and doubly circled states are accepting(final state), as for finite automata.
3) **Arcs:** The arcs correspond to transitions of the PDA in the following sense. An arc labeled a, X/α from state q to state p means that $\delta(q, a, X)$ contains the pair (p, α), perhaps among other pairs. That is, the arc

... what input is used, and also gives the old and new tops of the stack.

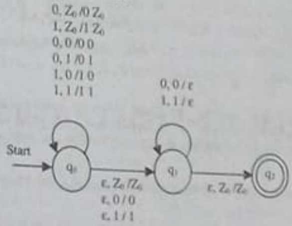


Figure 4.2: Representing a PDA as a Generalized Transition Diagram

The only thing that the diagram does not tell us is which stack symbol is the start symbol. Conventionally, it is Z_0, unless we indicate otherwise.

Difference between PDA and FS Machine

Pushdown automata differ from finite state machines in two ways:

1) They can use the top of the stack to decide which transition to take.
2) They can manipulate the stack as part of performing a transition.

Pushdown automata choose a transition by indexing a table by **input signal**, **current state**, and the **symbol** at the top of the stack. This means that those three parameters completely determine the transition path that is chosen. Finite state machines just look at the input signal and the current state: they have no stack to work with. Pushdown automata add the stack as a parameter for choice.

Ques 2) Give an example of PDA that recognizes the language $\{a^i b^j c^k \mid i, j, k \geq 0$ and $i = j$ or $i = k\}$.

Ans: Informally the PDA for this language works by first reading and pushing the a's. When the a's are done the machine has all of them on the stack so that it can match them with either the b's or the c's. This maneuver is a bit tricky because the machine does not know in advance whether to match the a's with the b's or the c's. Non-determinism comes in handy here.

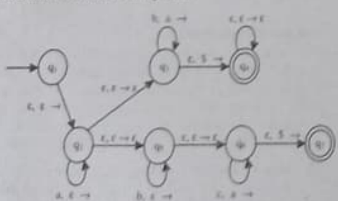


Figure 4.3: State Diagram for PDA M_1 that Recognizes $\{a^i b^j c^k \mid i, j, k \geq 0$ and $i = j$ or $i = k\}$

Using its non-determinism, the PDA can guess whether to match the a's with the b's or with the c's, as shown in the figure 4.3. Think of the machine as having two branches of its non-determinism, one for each possible guess. If either of them match, that branch accepts and the entire machine accepts.

Ques 3) Describe the various approaches that are used by PDA to accept any language.

Or

How PDA accept the language by final state and by empty stack?

Or

Explain the different methods by which a PDA accepts a language. **(2017 [03])**

Or

Which are the methods to accept a string in a PDA? Whether both type of PDAs can define the same language. Justify your answer. **(2019[03])**

Ans: Languages of PDA

A language can be accepted by pushdown automata using two approaches:

1) **Input is consumed and PDA is in a final state (Acceptance by "Final State"):** The PDA accepts its input by consuming it and finally it enters in the final state. PDA has final states like a non-deterministic finite automaton and has also the additional structure, viz., PDS. So we can define acceptance of input strings by PDA in terms of final states or in terms of PDS.

   Let $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The set accepted by PDA by final state is defined by:

   $T(A) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \overset{*}{\vdash} (q_f, \Lambda, \alpha)$ for some $q_f \in F$ and $\alpha \in \Gamma^*\}$

2) **Input is consumed and stack is empty (Acceptance by "Empty Stack"):** On reading the input string from initial configuration for some PDA, the stack of PDA becomes empty.

   Let $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a pda. The set $N(A)$ accepted by null store(or empty sotre)is defined by,

   $N(A) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \overset{*}{\vdash} (q_f, \Lambda, \Lambda)$ for some $q \in Q\}$

   In other words, w is in $N(A)$ if A is in initial ID(q_0, w, Z_0) and empties the PDS after processing all the symbols of w. So in defining $N(A)$, we consider the change brought about on PDS by application of w, and not the transition states.

Ques 4) Construct a PDA A accepting $L = \{wcw^T \mid w \in \{a, b\}^*\}$ by final state.

Ans: Consider the pda $A = (\{q_0, q_1, q_f\}, \{a, b, c\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_f\})$.

Let $wcw^T \in L$. Write $w = a_1 a_2 \ldots a_n$, where each a_i is either a or b. Then we have:

$(q_0, a_1 a_2 \ldots a_n c w^T, Z_0)$

$\overset{*}{\vdash} (q_0, cw^T, a_n a_{n-1} \ldots a_1 Z_0)$ by Rules (5)-(7)

$\vdash (q_1, a_1 a_2 \ldots a_n, a_n a_{n-1} \ldots a_1 Z_0)$ by Rule (8)

$\overset{*}{\vdash} (q_1, \Lambda, Z_0)$

$\vdash (q_f, \Lambda, Z_0)$ by Rule (9) by Rule (10)

Therefore, $wcw^T \in T(A)$, i.e., $L \subseteq T(A)$.

To prove the reverse inclusion, it is enough to show that $L^c \subseteq T(A)^c$. Let $x \in L^c$.

Case 1: x does not have the symbol c. In this case the pda never makes a transition to q_1. So the pda cannot make a transition to q_f as we cannot apply Rule (10). Thus $x \in T(A)^c$.

Case 2: $x = w_1 c w_2$, $w_2 \neq w_1^T$

$(q_0, w_1 c w_2, Z_0)$

$\overset{*}{\vdash} (q_0, cw_2, w_1^T Z_0) \vdash (q_1, w_2, w_1^T Z_0)$

As $w_2 \neq w_1^T$, the pda cannot reach an ID of the form (q_1, Λ, Z_0). So we cannot apply (6.10). Therefore, $x \in T(A)^c$.

Thus we have proved $L^c \subseteq T(A)^c$.

Ques 5) Construct a PDA for the language $L = \{w\,c\,w^R / w \in \{a, b\}^*\}$

Ans: Since language L is a context free language so an equivalent PDA can be constructed. Also assume L = N(M), where assume PDA M will be,

$M = (\{q_1, q_2\}, \{a, b, c\}, \{Z_0, A, B\}, \delta, q_1, Z_0, \phi)$

Where δ are as fallow:

$\delta(q_1, a, Z_0) \rightarrow \{(q_1, AZ_0)\}$ [If the first input alphabet is a, then symbol A will be pushed into the stack];

$\delta(q_1, b, Z_0) \rightarrow \{(q_1, BZ_0)\}$ [If the first input alphabet is b, then symbol B will be pushed into the stack];

$\delta(q_1, c, Z_0) \rightarrow \{(q_2, Z_0)\}$ [If input alphabet is c then stack remains unchanged];

$\delta(q_1, a, A) \rightarrow \{(q_1, AA)\}$ [For the next occurrence of a's, stack symbols A's will be pumped];

$\delta(q_1, b, A) \rightarrow \{(q_1, BA)\}$ [For the next occurrence of b's stack symbols B's will be pumped];

$\delta(q_1, c, A) \rightarrow \{(q_2, A)\}$ [For the next occurrence of c, stack remains unchanged];

Similarly,

$\delta(q_1, a, B) \rightarrow \{(q_1, AB)\}$;
$\delta(q_1, b, B) \rightarrow \{(q_1, BB)\}$;
$\delta(q_1, c, B) \rightarrow \{(q_2, B)\}$.

For the matching of substring lies left side of symbol c with the substring lies right side of c, following are the moves:

$\delta(q_2, a, A) \rightarrow \{(q_2, \epsilon)\}$
$\delta(q_2, b, B) \rightarrow \{(q_2, \epsilon)\}$

Finally,

$\delta(q_2, \epsilon, Z_0) \rightarrow \{(q_2, \epsilon)\}$; **Accepted**

[While other moves like $\delta(q_2, a, Z_0) \rightarrow \phi$ and $\delta(q_2, b, Z_0) \rightarrow \phi$ where state φ signifies that automaton crashes if it reaches to this state.]

Hence, PDA will be a deterministic PDA.

To verify that above moves are correct we assume a string w = abacaba, then trace the transitions over w, i.e.,

(q_1, abacaba, Z_0) ⊢ (q_1, bacaba, AZ_0) ⊢ (q_1, acaba, BAZ_0) ⊢ (q_1, caba, $ABAZ_0$) ⊢ (q_2, aba, $ABAZ_0$) ⊢ (q_2, ba, BAZ_0) ⊢ (q_2, a, AZ_0) ⊢ (q_2, ε, Z_0) ⊢ (q_2, ε, ε) [**Accepted**]

Ques 6) Design a Push Down Automata for the language $L = \{a^n b^{2n} \mid n > 0\}$. Trace your PDA with n = 3. **(2017 [09])**

Or

Design a PDA to accept the language $\{0^{2n} 1^n \mid n \geq 1\}$. **(2019[4.5])**

Ans: In the language set L each string is in the form $a^n b^{2n}$ where $n \geq 1$. Number of 'b' double to number of 'a'. At the time of traversing a single 'a' from the input tape two Z_i as stack symbol will be pushed to the stack. These two Z will be popped at the time of traversing two 'b' [Single 'a' is equal to two 'b']

The PDA will be as follows:

$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$q_0 = \{q_0\}$
$Z = \{Z\}$
$F = \{q_2\}$.

Accepted strings are {ε, abb, aabbbb, ...}

The transitional function will be:

$\delta(q_0, a, Z) = (q_1, aaZ)$
$\delta(q_1, a, a) = (q_1, aaa)$
$\delta(q_1, b, a) = (q_2, \epsilon)$
$\delta(q_2, b, a) = (q_2, \epsilon)$
$\delta(q_2, \epsilon, Z) = (q_0, Z)$

Note: q_0 is the final state, means stack should be empty on q_0. It means if we push something on this state it will not leave the stack empty, so we have to move to the new non-final state.

Tracing PDA for n = 3.

L =	a	a	a	b	b	b	b	b	b	ε

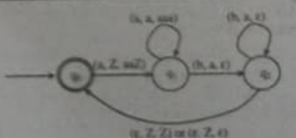


Figure 4.4

$(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \epsilon, \alpha)$ by definition of P_F

$(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha)$ by (2)

$(q_0, w, Z_0 \perp) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha \perp)$ by Theorem

$(q'_0, w, \perp) \underset{P_N}{\vdash} (q_0, w, Z_0 \perp) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha \perp)$ by (1)

$(q'_0, w, \perp) \underset{P_N}{\vdash} (q_0, w, Z_0 \perp) \overset{*}{\underset{P_N}{\vdash}} (q, \epsilon, \alpha \perp) \overset{*}{\underset{P_N}{\vdash}} (q_e, \epsilon, \epsilon)$ by (3) and (4)

These are all same class. We have already shown that (2) and (3) are same. It turns out to the easiest next to show that (1) and (3) are the same, thus implying the equivalence of all three as shown in **figure 4.7**.

Figure 4.7: Organization of Constructions showing Equivalence of Three Way's of defining CFL's

CFG and PDA have a strong relationship. We can construct a PDA from given CFG. Similarly we can obtain CFG from given PDA.

CFG Corresponding to Given PDA

Let, $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, q_f)$ is a PDA there exists CFG G which is accepted by PDA P.

The G can be defined as

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

Where,

S = start symbol,
T = set of terminals,
V = set of non-terminals

Algorithm for getting Production Rules of CFG

1) If q_0 is start state in PDA and q_n is final state of PDA then $[q_0\ Zq_n]$ becomes start state of CFG. Here Z represents the stack symbol.

2) The production rule for the ID of the from $\delta(q_i, a, Z_i) = (q_{i+1}, Z_1Z_2)$ can be obtained as

$\delta(q_iZ_iq_{i+1}) \to a(q_{i+1}Z_1q_{i+1})(q_kZ_2q_{k+1})$

Where, q_{i+1}, q_k represents the intermediate states, Z_i, Z_1, Z_2 are stack symbols and a is input symbol.

3) The production rule for the ID of the form $\delta(q_i, a, Z_0) = (q_{i+1}, \epsilon)$ can be converted as $(q_iZ_0q_{i+1}) \to a$

PDA Corresponding to Given CFG

PDA and CFL are strongly related to each other. The steps to convert given CFG to PDA are as follows:

Step 1: Convert the given CFG to Chomsky's normal form.

Step 2: The PDA should start by pushing start symbol onto the stack. To derive further production rules for the start symbol, we immediately perform the pop operation as is as shown in **figure 4.8**.

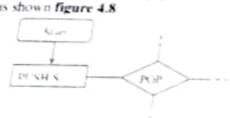


Figure 4.8

Step 3: If the production is of the form $S \to AB$, we push A and B onto the stack in reverse order. (Since we get after popping the reverse order is straight

∵ reverse, reverse = straight)

So $S \to AB$

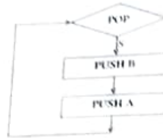


Figure 4.9

Step 4: If the production is of the form:

$S \to a$ we design as:

Figure 4.10

This means replacing a non-terminal s by a.

Step 5: Finally when the complete input is read from the tape, we encounter with Δ. Hence by popping Δ we get ensured with the fact that stack is also empty. This can be designed as:

Figure 4.11

Ques 18) Evaluate CFG from PDA Accepting Simple Palindromes

Ans: Let the language $L = \{xcx^r \mid x \in \{a, b\}^*\}$

Transition Table

Move Number	State	Input	Stack Symbol	Move(s)
1	q_0	a	Z_0	(q_0, AZ_0)
2	q_0	b	Z_0	(q_0, BZ_0)
3	q_0	a	A	(q_0, AA)
4	q_0	b	A	(q_0, BA)
5	q_0	a	B	(q_0, AB)
6	q_0	b	B	(q_0, BB)
7	q_0	c	Z_0	(q_1, Z_0)
8	q_0	c	A	(q_1, A)
9	q_0	c	B	(q_1, B)
10	q_1	a	A	(q_1, Λ)
11	q_1	b	B	(q_1, Λ)
12	q_1	Λ	Z_0	(q_1, Λ)
		(All other combinations)		None

In the grammar $G = (V, \Sigma, S, P)$, V contains S as well as every object of the form $[p, X, q]$, where X is a stack symbol and p and q can each be either q_0 or q_1. Productions of the following types are contained in P:

(0)	S	$\to [q_0, Z_0, q]$
(1)	$[q_0, Z_0, q]$	[illegible]
(2)	$[q_0, Z_0, q]$	[illegible]
(3)	$[q_0, A, q]$	[illegible]
(4)	$[q_0, A, q]$	[illegible]
(5)	$[q_0, B, q]$	[illegible]
(6)	$[q_0, B, q]$	[illegible]
(7)	$[q_0, Z_0, q]$	[illegible]
(8)	$[q_0, A, q]$	[illegible]
(9)	$[q_0, B, q]$	[illegible]
(10)	$[q_1, A, q_1]$	$\to a$
(11)	$[q_1, B, q_1]$	$\to b$
(12)	$[q_1, Z_0, q_1]$	$\to \Lambda$

Allowing all combinations of p and q gives 35 productions in all.

Consider the string bacab. The PDA accepts it by the sequence of moves:

$(q_0, bacab, Z_0) \vdash (q_0, acab, BZ_0)$
$\vdash (q_0, cab, ABZ_0)$
$\vdash (q_1, ab, ABZ_0)$
$\vdash (q_1, b, BZ_0)$
$\vdash (q_1, \Lambda, Z_0)$
$\vdash (q_1, \Lambda, \Lambda)$

The corresponding leftmost derivation in the grammar is

$S \Rightarrow [q_0, Z_0, q_1]$
$\Rightarrow b\,[q_0, B, q_1][q_1, Z_0, q_1]$
$\Rightarrow ba\,[q_0, A, q_1][q_1, B, q_1][q_1, Z_0, q_1]$
$\Rightarrow bac\,[q_1, A, q_1][q_1, B, q_1][q_1, Z_0, q_1]$
$\Rightarrow baca\,[q_1, B, q_1][q_1, Z_0, q_1]$
$\Rightarrow bacab\,[q_1, Z_0, q_1]$
$\Rightarrow bacab$

From the sequence of PDA moves, it may look as though there are several choices of leftmost derivation.

Since the PDA ends-up in state q_1, it is clear that q should be q_1. Similarly, it may seem as if the second step could be:

$[q_0, Z_0, q_1] \Rightarrow b\,[q_0, B, q_0][q_0, Z_0, q_1]$

However, the sequence of PDA moves that starts in q_0 and eliminates B from the stack ends with the PDA in state q_1, not q_0. In fact, because every move to state q_0 adds to the stack, the variable $[q_0, B, q_0]$ in this grammar is useless. No string of terminals can be derived from it.

Ques 19) Construct a PDA accepting $\{a^mb^ma^n \mid m, n \geq 1\}$ by null store. Construct the corresponding context-free grammar accepting the same set.

Ans: The PDA A accepting $\{a^mb^ma^n \mid m, n \geq 1\}$ is defined as follows:

$A = (\{q_0, q_1\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, \Phi)$

Where δ is defined by

[illegible]

This is a modification of δ.

[illegible]

By using Rules

[illegible]

The productions in P are constructed as follows:

The S-productions are

[illegible]

$\delta(q_0, b, a) = \{(q_1, \Lambda)\}$ gives

$P: [q_0, a, q_1] \to b$

$\delta(q_1, b, a) = \{(q_1, \Lambda)\}$ yields

$P: [q_1, a, q_1] \to b$

$\delta(q_1, a, a) = \{(q_1, \Lambda)\}$ gives

$P: [q_1, a, q_1] \to a$

$\delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$ yields

$P: [q_1, Z_0, q_1] \to \Lambda$

: When the number of states is a large number, it is neither ...ssary nor advisable to write all the productions. We ...struct productions involving those variables appearing in ...e sentential form. Using the constructions, we can simplify ... grammar further.

Ques 20) Construct PDA for the given CFG

S → a S b

S → a b

Ans: Now we will first convert this CFG to CNF first.

This is basically the language of $a^n b^n$ where $n \geq 1$.

$S \rightarrow a\,S\,b$ $S \rightarrow a\,b$

⇒

$S \rightarrow AR_1$ $S \rightarrow AR_2$ $R_1 \rightarrow SR_2$ $R_2 \rightarrow b$ $A \rightarrow a$

The simulation for input aabb can be,

Input	Stack	Action
aabbΔ	Δ	Push S
aabbΔ	S Δ	POP S
aabbΔ	Δ	Push R_1
aabbΔ	R_1 Δ	Push A
aabbΔ	AR_1 Δ	Read a
abbΔ	R_1 Δ	POP
abbΔ	Δ	Push R_2

Input	Stack	Action
abbΔ	R_2 Δ	Push S
abbΔ	S R_2 Δ	POP
abbΔ	R_2 Δ	Push R_2
abbΔ	R_2R_2 Δ	Push A
abbΔ	A R_2R_2 Δ	POP
abbΔ	R_2R_2 Δ	Read a
bbΔ	R_2R_2 Δ	POP
bbΔ	R_2 Δ	Read b
bΔ	R_2 Δ	POP
bΔ	Δ	Read b
Δ	Δ	POP
Δ	-	Read Δ
-	-	Accept

The ID for this graphical PDA is

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

Now place string w on input tape.

$\delta(q, w, S) = (q, AR_1)$

$\delta(q, w, S) = (q, AR_2)$

$\delta(q, w, R_1) = (q, SR_2)$

$\delta(q, b, R_2) = \delta(q, \varepsilon)$

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

Consider now input aabb for simulation.

Initially

$(q, \varepsilon, Z_0) \vdash (q, SZ_0)$

The start symbol is pushed onto the stack. The string aabb is placed on input tape.

$(q, aabb, S) \vdash (q, aabb, AR_1)$

$\vdash (q, abb, R_1)$

$\vdash (q, abb, SR_2)$

$\vdash (q, abb, AR_2 R_2)$

$\vdash (q, bb, R_2 R_2)$

$\vdash (q, b, R_2)$

$\vdash (q, \varepsilon)$

Accept state.

Thus, input is accepted by PDA.

Ques 21) Construct a PDA equivalent to the following grammar:

S → aSa | bSb | a

Ans: The grammar is not in GNF, therefore, first we obtain an equivalent grammar in GNF, which is:

S → aSA | bSB | a

A → a

B → b

Therefore, the PDA equivalent to the given grammar is:

$M = (\{q\}, \{a, b\}, \{S, A, B\}, \delta, q, S, \phi)$

Where,

$\delta(q, a, S) = \{(q, SA), (q, \epsilon)\}$, because of S → aSA, and S → a

$\delta(q, b, S) = \{(q, SB)\}$, because of S → bSB

$\delta(q, a, A) = \{(q, \epsilon)\}$, because of A → a

$\delta(q, b, B) = \{(q, \epsilon)\}$, because of B → b.

Ques 22) Which Normal Form representation of CFG will you prefer in converting CFG to NPDA? Why? (2017 [03])

Ans: Chomsky Normal Form can be used in converting CFG to NPDA.

The Chomsky's Normal form can be defined as

Non terminal → Non terminal. Non terminal

Non terminal → terminal

The given CFG should be converted in the above format then we can say that the grammar is in CNF. Before converting the grammar in to CNF it should be in reduced form. That means remove all the useless symbols, ε productions and unit productions from it. Thus this reduced grammar can be then converted to CNF.

Ques 23) Construct a PDA equivalent to the following grammar:

E → +EE | *EE | id

Ans: The PDA equivalent to the given grammar is:

$M = (\{q\}, \{+, *, id\}, \{E\}, \delta, q, E, \phi)$

Where,

$\delta(q, +, E) = \{(q, EE)\}$, because of E → +EE

$\delta(q, *, E) = \{(q, EE)\}$, because of E → *EE

$\delta(q, id, E) = \{(q, \epsilon)\}$, because of E → id

Ques 24) Show the equivalence between NDPA and CFG.

Or

Show that NPDA and CFG are equivalent. (2018 [06])

Ans: Equivalence between NPDA and CFG

We have shown that every CFL is accepted by some NPDA. Now we show conversely that NPDAs accept only CFLs. Thus NPDAs and CFGs are equivalent in expressive power. We will do this in two steps by showing that:

1) L is recognized by a NPDA implies L is described by a CFG.
2) L is described by a CFG implies L is recognized by a NPDA.

We describe the algorithm to convert a CFG G into an equivalent NPDA M.

1) Start with a CFG G = (V, T, S, P).
2) Create an NPDA $M = (Q, \Sigma, \Gamma, \delta, q_s, Z, F)$ with three states such that $Q = \{q_0, q_1, q_2\}$, $q_s = q_0$; $F = \{q_2\}$, Σ = T, and $\Gamma = V \cup T \cup \{Z\}$.
3) Create a starting transition, $\delta(q_0, \lambda, Z) = (q_1, SZ)$.
4) Create an ending transition, $\delta(q_1, \lambda, Z) = (q_2, Z)$.
5) For each terminal a ∈ T, create the transition $\delta(q_1, a, a) = (q_1, \lambda)$.
6) For each production A → BCD ∈ P, create the transition $\delta(q_1, \lambda, A) = (q_1, BCD)$.

We describe the algorithm to convert an NPDA M into a CFG G.

1) Start with an NPDA $M = (Q, \Sigma, \Gamma, \delta, q_s, Z, F)$.
2) Modify M to have one final state q_f
3) Modify M such that each transition pops exactly one symbol and pushes either zero or two symbols.
4) Create a CFG G = (V, T, S, P) with $V = \{(q_a A q_b) | q_a, q_b \in Q$ and $A \in \Gamma\}$ and with the start variable equal to $(q_s Z q_f)$.
5) For each transition $\delta(q_a, a, A) = (q_b, \lambda)$, generate one production $(q_a, A, q_b) \rightarrow a$.
6) For each transition $\delta(q_a, a, A) = (q_b, BC)$, generate the productions $(q_a A q_x) \rightarrow a(q_b B q_y)(q_y C q_x)$ for all q_x and q_y.

PUMPING LEMMA FOR CFL

Ques 25) State pumping lemma for Context Free Languages. (2017 [10])

Or

State pumping lemma for CFL. (2018 [03])

Ans: Pumping Lemma for Context-Free Languages (CFL)

Let L be an infinite context-free language. Then there exists some positive integer m such that any w ∈ L with | w | ≥ m can be decomposed as

w = uvxyz, ...(1)

With

| vxy | ≤ m, ...(2)

and

| vy | ≥ 1, ...(3)

Such that

$uv^ixy^iz \in L$. ...(4)

For all i = 0, 1, 2, ... This is known as the pumping lemma for context-free languages.

Ques 26) What is the application of pumping lemma for CFL? Explain with an example.

Or

Mention one application of Pumping Lemma. (2018 [02])

Ans: Application of Pumping Lemma

Pumping lemma for CFL is used to prove that certain sets are not context free. Every CFL fulfils some general properties. But if a set of language fulfils all the properties of Pumping Lemma for CFL, it cannot be said that the language is context free.

But the reverse, i.e., if a language breaks the properties it can be said confirm that the language is not context free.

For example, the language $L = \{a^ib^ic^i \mid i \geq 1\}$ is not a CFL. It is proved as below. Let n be a constant of pumping lemma. Select $Z = a^n b^n c^n$. This ensures that Z is in L and |Z| ≥ n. If we write Z = uvwxy, then the possible choices of vx satisfying the conditions

$1 \leq |vx| \leq n$ and $|vwx| \leq n$ are

1) $vx = a^p$, where $1 \leq p \leq n$, i.e., vx contains only a's.

   For this choice of vx, uv^0wx^0y will be $a^{n-p}b^nc^n$ and since $1 \leq p \leq n$, uv^0wx^0y contains less number of a's than b's and c's. Hence, uv^0wx^0y cannot belong to L. Hence, contradiction to pumping lemma.

2) $vx = b^p$, where $1 \leq p \leq n$, i.e., vx contains only b's.

   For this choice of vx, uv^0wx^0y will be $a^nb^{n-p}c^n$ and since $1 \leq p \leq n$, uv^0wx^0y contains less number of b's than a's and c's. Hence, uv^0wx^0y cannot belong to L. Hence, contradiction to pumping lemma.

3) $vx = c^p$, where $1 \leq p \leq n$, i.e., vx contains only c's.

   For this choice of vx, uv^0wx^0y will be $a^nb^nc^{n-p}$ and since $1 \leq p \leq n$, uv^0wx^0y contains less number of c's than a's and b's. Hence, uv^0wx^0y cannot belong to L. Hence, contradiction to pumping lemma.

4) $vx = a^pb^q$, where $1 \leq p, q \leq n$, i.e., vx contains both a's and b's.

   For this choice of vx, uv^0wx^0y will be $a^{n-p}b^{n-q}c^n$ and since $1 \leq p, q \leq n$, uv^0wx^0y contains less number of a's and b's than c's. Hence, uv^0wx^0y cannot belong to L. Hence, contradiction to pumping lemma.

5) $vx = b^pc^q$, where $1 \leq p, q \leq n$, i.e., vx contains both b's and c's.

   For this choice of vx, uv^0wx^0y will be $a^nb^{n-p}c^{n-q}$ and since $1 \leq p, q \leq n$, uv^0wx^0y contains less number of b's and c's than a's. Hence uv^0wx^0y cannot belong to L. Hence, contradiction to pumping lemma. We find that vx cannot contain both a's and c's because there are n number of b's between rightmost a and leftmost c, and hence if we take vx to contain both a's and c's then condition |vwx| ≤ n is not satisfied.

Hence, we conclude that we cannot have vx such that uv^iwx^iy is in L(i ≥ 0). Therefore, it is not a CFL.

Ques 27) Prove that $L = \{a^i b^j \mid j = i^2\}$ is not a CFL.

Ans: Let n be a constant of pumping lemma.

Select $Z = a^n b^{n^2}$. This ensures that Z is in L and $|Z| \geq n$.

If we write Z = uvwxy, then the possible choices of vx satisfying the conditions:

$1 \leq |vx| \leq n$ and $|vwx| \leq n$ are:

1) $vx = a^p$, where $1 \leq p \leq n$, i.e., vx contains only a's.

For this choice of vx, uv^2wx^2y will be $a^{n+p}b^{n^2}$ and since $1 \leq p \leq n$, uv^2wx^2y contains number of a's between n + 1 and n + n = 2n, whereas number of b's equal to square of n, hence the number of b's is not the square of number of a's. Therefore uv^2wx^2y cannot belong to L. Hence, contradiction to pumping lemma.

2) $vx = b^p$, where $1 \leq p \leq n$, i.e., vx contains only b's.

For this choice of vx, uv^2wx^2y will be $a^n b^{n^2+p}$ and since $1 \leq p \leq n$, number of b's in uv^2wx^2y will not be the square of number of a's. Hence uv^2wx^2y cannot belong to L. Hence, contradiction to pumping lemma.

3) $vx = a^p b^q$, where $1 \leq p, q \leq n$, i.e., vx contains both a's and b's.

For this choice of vx, uv^2wx^2y will be $a^{n+p}b^{n^2+q}$ and since $1 \leq p, q \leq n$, the number of b's in uv^2wx^2y will not be the square of the number of a's even if p = q. Hence, uv^2wx^2y cannot belong to L. Hence, contradiction to pumping lemma.

Hence, we conclude that we cannot have vx such that uv^iwx^iy is in L ($i \geq 0$). Therefore, it is not a CFL.

Ques 28) Prove that the language $L = \{a^{2^i} / i \geq 1\}$ is not context free.

Ans:

Step 1: Assume that the language set L is CFL. Let n be a natural number obtained by using pumping lemma.

Step 2: Let $z = a^{2^i}$ /$i \geq 1$. So $|z| = 2^i$. Let $2^i > n$.

According to pumping lemma for CFL we can write z = uvwxy, where $|vx| \geq 1$ and $|vwx| \leq n$.

Step 3: The string z contains only 'a', so v and x will be also a string of only 'a'. Let $v = a^p$ and $x = a^q$, where $(p + q) \geq 1$. Since $n \geq 0$ and $uvwxy = 2^i$, so $|uv^nwx^ny| = |uvwxy| + |v^{n-1}x^{n-1}| = 2^i + (p + q)(n-1)$. As $uv^nwx^ny \in L$, $|uv^nwx^ny|$ is also a power of 2, say 2^j.

$(p + q)(n-1) = 2^j - 2^i \Rightarrow (p + q)(n-1) + 2^i = 2^j$

$\Rightarrow (p + q)2^{i+1} + 2^i = 2^j$

$\Rightarrow 2^i(2(p + q) + 1) = 2^j$

Here, (p + q) may be even or odd, but 2(p + q) is always even. Whereas 2(p + q) + 1 is odd, which cannot be a power of 2. Thus L is not context free.

Ques 29) Show that the language $L = \{a^{n!} : n \geq 0\}$ is not context-free.

Ans: Given the opponent's choice for m, we pick $w = a^{m!}$. Obviously, whatever the decomposition is, it must be of the form $v = a^k$, $y = a^l$. Then $w_0 = uxz$ has length $m! - (k + l)$. This string is in L only if

$m! - (k + l) = j!$

For some j. But this is impossible, since with $k + l \leq m$,

$m! - (k + l) > (m - 1)!$.

Therefore, the language is not context-free.

Ques 30) Show that the language $L = \{0^n1^n2^n \mid n \geq 1\}$ is not context-free.

Ans: Let L be the language $\{0^n1^n2^n \mid n \geq 1\}$. That is, L consists of all strings in $0^+1^+2^+$ with an equal number of each symbol, e.g., 012, 001122, and so on. Suppose L were context-free. Then there is an integer n given to us by the pumping lemma. Let us pick $z = 0^n1^n2^n$.

Suppose the "adversary" breaks z as z = uvwxy, where $|vwx| \leq n$ and v and x are not both ε. Then one knows that vwx cannot involve both 0's and 2's, since the last 0 and the first 2 are separated by n + 1 positions. One shall prove that L contains some string known not to be in L, thus contradicting the assumption that L is a CFL. The cases are as follows:

1) vwx has no 2's. The vx consists of only 0's and 1's, and has atleast one of these symbols. Then uwy, which would have to be in L by the pumping lemma, has n 2's, but has fewer than n 0's, or fewer than n 1's, or both. It therefore does not belong in L, and one conclude L is not a CFL in this case.
2) vwx has no 0's. Similarly, uwy has n 0's, but fewer 1's or fewer 2's. It therefore is not in L.

Whichever case holds, one conclude that L has a string he/she knows not to be in L. This contradiction allows us to conclude that our assumption was wrong: L is not a CFL.

Ques 31) Show that the language $L = \{a^ib^ic^i \mid i \geq 0\}$ is not context-free.

Or

Check whether $L = \{a^ib^ic^i \mid i > 0\}$ belong CFL or not.

(2019[06])

Ans: The language $L \{a^ib^ic^i \mid i \geq 0\}$ is not context-free. Assume L is context-free. By theorem, the string $z = a^kb^kc^k$, where k is the number specified by the pumping lemma, can be decomposed into substrings uvwxy that satisfy the repetition properties.

Consider the possibilities for the substrings v and x. If either of these contains more than one type of terminal symbol, then uv^2wx^2y contains a b preceding an a or a c preceding a b. In either case, the resulting string is not in L. By the observation, v and x must be substrings of one of a^k, b^k, or c^k. Since at most one of the strings v and x is null, uv^2wx^2y increases the number of atleast one, may be two, but not all three types of terminal symbols. This implies that $uv^2wx^2y \notin L$.

Thus, there is no decomposition of $a^kb^kc^k$ satisfying the conditions of the pumping lemma; consequently, L is not context-free.

Ques 32) Define the closure properties of CFLs.

Ans: Closure Properties of CFLs

The context free languages are closed under some operation means after performing that particular operation on those CFLs the resultant language is context free language. These properties are as below:

1) The context free languages are closed under union.
2) The context free languages are closed under concatenation.
3) The context free languages are closed under kleen closure.
4) The context free languages are not closed under intersection.
5) The context free languages are not closed under complement.

Ques 33) What are the decision problems related with the type 3 formalism?

Ans: Decision Problems of Type 3 Formalism / Decision Problems of CFLs

A problem is said to be decidable if there is an algorithm to solve that problem. For a given CFG G = (V, T, P, S), there exists an algorithm for deciding a problem whether it is a decidable or not.

The following CFLs are decidable:

1) **Emptiness of CFL is Decidable:** Assume the given language does not contain ε. Find the reduced grammar of the given grammar.
   i) If the reduced grammar vanishes then the given language is empty.
   ii) Draw a graph with the productions of the reduced grammar. If the graph contains cycles, the given grammar generates infinite language, otherwise it generates finite language.

2) **Finiteness Test of CFL:** A language L generated from a given CFG is finite if there are no cycles in the directed graph generated from the production rules of the given CFG. The longest string generated by the grammar is determined by the derivation from the start symbol.

   The number of vertices of the directed graph is the same as the number of non-terminals in the grammar. If there is a production rule S → AB, the directed graph is as shown in **figure 4.12**.

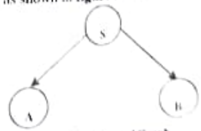


Figure 4.12: Directed Graph

3) **Infiniteness Test of CFL:** A language L generated from a given CFG is infinite if there is atleast one cycle in the directed graph generated from the production rules of the given CFG.

4) **Membership Problem for CFL:** Given a CFL L and a string w, one wishes to have a procedure to test whether w is in L or not. One can approach for the result directly using derivation trees.

   That if |w| = n and the grammar G generating L is in CNF, then the derivation tree for w uses exactly 2n − 1 nodes labelled by variables of G. The number of possible trees for w and node labellings leads to an exponential time algorithm. There is another efficient algorithm which uses the idea of "dynamic programming". The algorithm is known as CYK (**Cocke, Younger, and Kasami**) algorithm which is an $O(n^3)$ algorithm.

Ques 34) *Prove* that there is an algorithm to decide whether a given CFG generates an infinite language or a finite language.

Ans: The proof will be by constructive algorithm. One shall show that there exists such a procedure by presenting one. If any word in the language is long enough to apply the pumping lemma to, one can produce an infinite sequence of new words in the language.

If the language is infinite, then there must be some words long enough so that the pumping lemma applies to them. Therefore, the language of a CFG is infinite if and only if the pumping lemma can be applied.

The essence of the pumping lemma was to find a self-embedded non-terminal X. We shall also how to tell whether a particular non-terminal is self-embedded, but first one should also note that the pumping lemma will work only if the non-terminal that he/she pumps is involved in the derivation of any words in the language.

Without the algorithm, one could be building larger and larger trees, none of which are truly derivation trees.

For example, in the CFG

$S \to aX \mid b$

$X \to XXb$

The non-terminal X is certainly self-embedded, but the language is finite nonetheless.

So, the algorithm is as follows:

Step 1: Use the algorithm to determine which non-terminals are useless. Eliminate all productions involving them.

Step 2: Use the following algorithm to test each of the remaining non-terminals, in turn, to see whether they are self-embedded. When a self-embedded one is discovered, stop.

To test **X**

1) Change all X's on the left side of productions into the Russian letter ж, but leave all X's on the right side of production alone.
2) Paint all X's blue.

3) If Y is any non-terminal that is the left side of any production with some blue on the right side, then paint all Y's blue.
4) Repeat step 2(3) until nothing new is painted blue.
5) If X is blue, then X is self-embedded; if not, it is not.

Step 3: If any non-terminal left in the grammar after step 1 is self-embedded, the language generated is infinite. If not, then the language is finite.

Ques 35) Consider the following grammar and find whether it generates empty or finite or infinite language.

$S \rightarrow AB$
$S \rightarrow BC/a$
$B \rightarrow Cc/b$
$C \rightarrow a$

Ans: The reduced grammar for this grammar is:

$S \rightarrow AB$
$A \rightarrow BC/a$
$B \rightarrow Cc/b$
$C \rightarrow a$

As the grammar does not vanish, it generates non-empty language. It can be shown in **figure 4.13.**

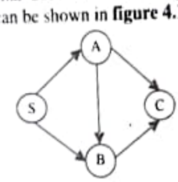


Figure 4.13: Decidable Algorithm Graph

As the graph contains no cycles, the language generated by the given grammar is finite.

Ques 36) Verify whether the languages generated by the following grammar are finite or not. If finite, find the longest string generated by the grammar:

1) **$S \rightarrow AB$**
$A \rightarrow BC$
$B \rightarrow C$
$C \rightarrow a$

2) **$S \rightarrow AB$**
$A \rightarrow B$
$B \rightarrow SC/a$
$C \rightarrow AB/b$

Ans:

1) The grammar is not in CNF. Removing the unit productions, the grammar becomes:

$S \rightarrow AB$
$A \rightarrow BC$
$B \rightarrow a$
$C \rightarrow a$

Now the grammar is in CNF. The directed graph for the grammar is shown in **figure 4.14**:

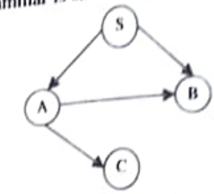


Figure 4.14: Directed Graph

The graph does not contain any loop. So, the language generated by the CFG is finite. The derivation from the grammar is $S \rightarrow AB \rightarrow BCB \rightarrow aaa$.

Thus, the length of the longest string is 3.

2) In the grammar, there is a unit production $A \rightarrow B$. By removing the unit production, the grammar becomes:

$S \rightarrow AB$
$A \rightarrow SC/a$
$B \rightarrow SC/a$
$C \rightarrow AB/b$

The grammar is in CNF. The non-terminal transitional graph for the grammar is shown in **figure 4.15**:

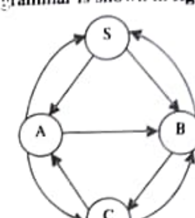


Figure 4.15

The graph contains loops. So, the language generated by the CFG is infinite.

Ques 37) Given a CFL L, there exists an algorithm to test whether L is empty, finite or infinite.

Ans: To test whether L is empty, one can see whether the start symbol S of the CFG $G = (N, T, S, P)$ which generates L is useful or not. If S is a useful symbol, then $L \neq \phi$. To see whether the given CFL L is infinite, one has the following discussion.

By pumping lemma for CFL, if L contains a word of length t, with $|t| > k$ for a constant k (pumping length), then clearly L is infinite. Conversely, if L is infinite it satisfies the conditions of the pumping lemma, otherwise L is finite. Hence, one has to test whether L contains a word of length greater than k.

Module 5

Context Sensitive Languages, Turing Machines

CONTEXT-SENSITIVE LANGUAGES

Ques 1) Define context-sensitive grammar. *Or*
What is a Context Sensitive Grammar (CSG)? (2017 [02])

Ans: Context-Sensitive Grammar
Type 1 grammar is called context-sensitive grammar. The context sensitive grammars are more general than context free grammars and less so than unrestricted grammar. The corresponding model of computation called as linear bounded automata lie between push-down automata and Turing machine. Formally we can say that, a context sensitive grammar (CSG) is an unrestricted grammar in which every production has the form,

$\alpha \to \beta$ with $|\beta| \geq |\alpha|$

A context sensitive language (CSL) is a language that can be generated by such a grammar. A language is a context sensitive 1F and only 1F it can be generated by a grammar in which every production has the form.

$\alpha A\beta \to \alpha x\beta$

Where α, β and x are strings of variables and/or terminals, with x not null, and A is a variable. Such a production allow A to be replaced by x, depending on the context. A context sensitive grammar (CSG), in which production contain the non-contracting production rule.

Non-Contracting Production Rule
A production $\alpha \to \beta$ satisfying $|\beta| \geq |\alpha|$ is known as non-contracting production rule.

For example,

$w_1 \to w_2 \to w_3 \to \ldots \to w_n$

$|w_1| \leq |w_2| \leq |w_3| \leq \ldots |w_n|$ are non-decreasing in length.

For example, consider a grammar $G = (\{S, A\}, \{a\}, P, S)$ where P is given as

$S \to aA$
$aA \to aaA$
$aA \to aa$

All the production of this grammar follows the non-contracting production rule. So, it is a context sensitive grammar and the language

$(L = a^n \mid n > 1 \text{ or } n \geq 2)$

Ques 2) Design a CSG to accept the language $L=\{0^n1^n2^n \mid n > 0\}$. (2017 [04])

Ans: A grammar is context sensitive if all rules are of one of two forms.

$S \to \epsilon$ and S never appears on the right-hand side of a rule.

Or

$\alpha \to \beta$ where $|\alpha| \leq |\beta|$

In particular, a derivation is never "collapsing" i.e. sentential forms in the derivation sequence never decrease in length. A language is a Context Sensitive Language (CSL) if it is generated by a context-sensitive grammar

The $0^n1^n2^n$ language is context sensitive. The grammar above is technically not context sensitive, but we can make it so by "removing ε-production" as follows

$S' \to S \mid \epsilon$
$S \to 0SBC \mid 0BC$
$CB \to BC$
$0B \to 01$
$1B \to 11$
$1C \to 12$
$2C \to 22$

Ques 3) Discuss the model of linear bounded automata. **Or**
Define Linear Bound Automata. (2017 [04])

Ans: Model of Linear Bounded Automata
Linear Bounded automata is important because of the two reasons first, is the set of context sensitive languages is accepted by the model and the second is the with respect to the Turing machine, in linear bounded automata the infinite storage is restricted in size, but not in accessibility to the storage. It is called as a linear bounded automata because a linear function is used to bound the length of the tape. Formally we can define the linear bounded automata as:

A linear bounded automata is a 9-tuple given as

$M = (Q, \Sigma, \Gamma, \delta, q_0, b, ¢, \$, F)$

Where

Q	is a finite non-empty set of states
Σ	is a finite non-empty set of i/p symbol.
Γ	is a finite non-empty set of tape symbol.
$b \in \Gamma$	is a blank symbol
q_0	is the initial state, i.e., $q_0 \in Q$
$F \subseteq Q$	is he final state
δ	is the transition function mapping the state of finite automaton and tape symbols to states, tape symbols and movement of the head, i.e., $Q \times \Gamma \to Q \times \Gamma \times (L, R)$
¢	is the left end marker which is entered in the left-most cell of the i/p tape and prevents the R/W head from getting of the left end of the tape.
\$	is the right end marker which is entered in the right most cell of the tape and prevent the R/W head from getting of the right end of tape.

The model of linear bounded automata is given in the figure 5.1 as:

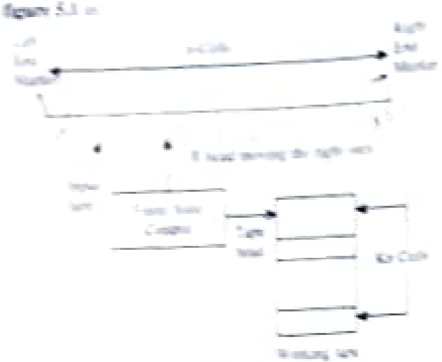


Figure 5.1

A linear bounded automata accepts a string w if, after starting at the initial state with R/W head reading the left end-marker, M halts over the right end marker in a final state, otherwise w is rejected. A linear bounded automata is same as non-deterministic turing machine except in the following respect:

1. It has two extra tape symbols, left end marker and right-end marker which can be assumed not to be the elements of Γ.
2. Here, there are two tapes, one is called input tape and the other is working tape. In the input tape head never prints and never moves left. On the working tape the head can modify the contents in any way without any restriction.

Here, in LBA (Linear bounded automata) an ID is given by [illegible]

where [illegible] is some integer between [illegible] and [illegible].

The transition of ID is similar except that K changes to K − 1 if the tape head moves to the left and to K + 1 if the head moves to the right. An linear bounded automata (LBA) accepts the null-string. The null string can be represented either by the absence of input string or by [illegible] a blank tape.

TURING MACHINE (TM)

Ques 4: Define the Turing machine and also write its formal definition. Or
Discuss the standard Turing machine.

Ans: Standard Turing Machine

A Turing machine is the simplest form of a computer. The concept was invented by **Alan Turing** in 1936. This was the first computer invented (on paper only). Turing Machine (TM) is a simple mathematical model of a general purpose computer. In other words, Turing machine models the computing power of a computer (i.e., the Turing machine is capable of performing any calculation which can be performed by any computing machine.

A Turing machine is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules. A Turing machine is written [illegible] automaton, but there are important differences [illegible]

1. A Turing machine can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take immediate effect.

Definition

A Turing Machine (TM) is 5-tuple

$M = (Q, \Sigma, \Gamma, q_0, B, \delta, F)$

Where, Q = finite set of states
Γ = finite set of allowable tape symbols
B = symbol of Γ, is the blank
Σ = subset of Γ not including B, is the set of input symbols
δ = next move function, a mapping from Q × Γ to Q × Γ × {L, R, H} (possibly undefined at certain points)
$q_0 \in Q$ is the start state
$F \subseteq Q$ is the set of final states.

For elements [illegible] ∈ Q, X, Y ∈ Γ and D ∈ {R, L, H} we interpret the formula

[illegible]

to mean that when M is in state [illegible] and the current symbol scanning is X, then the machine replaces X by Y, changes the state [illegible] and either moves the tape head one cell right or moves it one cell left or halts depending on whether D is R, L or H respectively.

A Turing machine M = (Q, Σ, Γ, q_0, B, δ, F) computes as follows: Initially M receives its input [illegible] ∈ Σ* on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols). The head starts on the leftmost square of the tape.

Ques 5: Write the basic features of Turing machine.

Ans: Basic Features of Turing Machine

1. The machine "moves" in discrete steps 0, 1, 2, [illegible] which we may call moments.
2. The "working part" of the machine is a doubly infinite tape, divided into squares numbered by the integers. Each square contains one of the bits 0, 1. At any moment all but a finite number of the bits are 0.
3. At each moment the machine is in one of a finite number of states

[illegible]

State [illegible] is both the initial and the final, or halting, state. The computation is complete if and when the machine re-enters this state. Both the action that the machine takes at each moment, and the new state that it moves into, depend entirely on 2 factors:

i) The state [illegible] of the machine
ii) The bit [illegible] on square 0 of the tape.

It follows that a machine T can be described by a single map

[illegible]

Where A denotes the set of possible actions.

Ques 6: Explain the abstract view of Turing machine.

Ans: Abstract View of a TM

Assume that at time t = 0 Machine M is in state [illegible] (initial state) and its tape head pointing to its left most cell. Tape cells contain the string of input symbols from over Σ and the remaining cells hold the blank symbols that is shown by B. Machine has a read-write tape head T that scans one cell at a time. Assume that after reading the symbol [illegible] machine reaches to state [illegible] and simultaneously tape head replaces the input symbol by another tape symbol let it be [illegible] and move to right. Now tape head pointed to the second cell containing the input symbol [illegible]. In this way M scans all cells that contain the input symbol and left the tape symbols that will be the possible form of output generated by the Turing Machine M. The abstract view of Turing machine is shown in figure 5.2 (a) and (b).

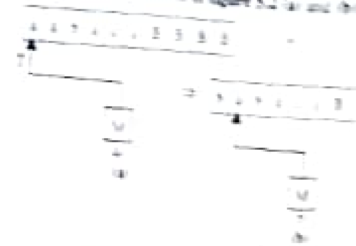


Figure 5.2: Abstract view of Turing

Ques 7: Discuss the robustness of Turing machine.

Ans: Robustness of Turing Machine

Alternative definitions of Turing machines abound, including versions with multiple tapes or with non-determinism. They are called variants of the Turing machine model. The original model and its reasonable variants all have the same power – they recognize the same class of languages. We call this invariance to certain changes in the definition robustness. Both finite automata and pushdown automata are somewhat robust models, but Turing machines have an astonishing degree of robustness.

To illustrate the robustness of the Turing machine model, let's vary the type of transition function permitted. The transition functions forces the head to move to the left or right after each step; the head may not simply stay put. Suppose that we had allowed the Turing machine the ability to stay put. The transition function would then have the form δ: Q × Γ → Q × Γ × {L, R, S}. Might this feature allow Turing machines to recognize additional languages, thus adding to the power of the model? Of course not, because we can convert any TM with the "stay put" feature to one that does not have it. We do so by replacing each stay put transition with two transitions: one that moves to the right and the second back to the left.

Turing machines do not get fundamentally more (or less) powerful if we:

1. Allow multiple tapes
2. Allow multi-dimensional tapes
3. Allow tapes that are bounded on one side
4. Allow non determinism
5. Restrict the alphabets of tape [illegible]

Moreover, every other (realistic) mathematical model of computation conceived so far can be "simulated" with a Turing machine and hence is not more powerful.

Ques 8: Describe the various methods of representing the Turing machine. Or
Explain instantaneous description (ID) for Turing machine. Or
How transition table and transition diagram are used to represent Turing machine?
Or
What is the instantaneous description for a Turing machine? Explain with an example. (2018 (05))

Ans: Representation of Turing Machines

There are three methods of representing turing machine as:

1. **Instantaneous Description (ID):** An Instantaneous Description (ID) of the Turing machine M is denoted by [illegible]. Here [illegible] ∈ Q is the current state of M, [illegible] is the string in Γ* that is the contents of the tape upto the rightmost non-blank symbol or the symbol to the left of the head, whichever is rightmost.

TM Moves

The δ [illegible] induces a change in ID of the Turing machine. Such a change in ID is called a move. If [illegible] Let the input string to be processed is [illegible] and the present symbol under R/W head is [illegible]. So the ID before processing the symbol [illegible] will be:

[illegible]

So after processing [illegible] the ID will be

[illegible]

This is represented as

[illegible]

If [illegible] then the change of ID is represented by

[illegible]

Thus the [illegible] defines a relation among IDs. So the symbol [illegible] denotes the reflexive transitive closure of the relation [illegible]. So if [illegible] then we can split this as [illegible] for some IDs [illegible].

For example, consider the Turing machine shown in figure 5.3. Obtain its ID.

Figure 5.3

The present symbol under R/W head is [illegible] and the present state is [illegible]. So [illegible] is written to the right of [illegible]. The non-blank symbols to the left of [illegible] is a string [illegible] which is written to the left of [illegible].

The sequence of non-blank symbols to the right of a_i … Thus the ID is

Figure 5.4

2) **Transition Table:** In case of TM, the transition table consists of a column to represent the states of TM and one column for each of the tape symbols. If $\delta(q, a) = (\gamma, \alpha, \beta)$ we write (γ, α, β) under the a column and q^{th} row. So any time we get (γ, α, β) in the transition table, it means that α is written in the current state table. **For example,** for the TM described in **table 5.1**, the computation sequence for the input string 00. Such a computation sequence can be shown in terms of the contents of the tape and the current state. So, we get following sequence:

q_0 00B ⊢ 0 q_1 0B ⊢ 00 q_1 B ⊢ 0 q_2 01 ⊢ q_3 001 ⊢ q_3 B 001 ⊢ B q_4 001 ⊢ BB q_0 01 ⊢ BB 0 q_1 1 ⊢ BB 01 q_1 B

⊢ BB 010 q_2 ⊢ BB 01 q_2 00 ⊢ BB 0 q_2 100 ⊢ BB q_3 0100

⊢ B q_3 B 0100 ⊢ BB q_4 0100 ⊢ BBB q_0 100 ⊢ BBB 1 q_1 00

⊢ BBB 10 q_1 0 ⊢ BBB 100 q_1 B ⊢ BBB 1000 q_2 B

⊢ BBB 100 q_2 00 ⊢ BBB 10 q_2 000 ⊢ BBB 1 q_2 0000

⊢ BBB q_3 10000 ⊢ BB q_3 B 10000 ⊢ BBB q_4 10000 ⊢ BBBB q_5 0000

Following table shows a typical transition table. Note B stand for a blank symbol.

Table 5.1: Transition table for a TM

State	0	1	B
→ q_0	(q_2, 1, L)	(q_1, 0, R)	
q_1	(q_1, B, R)	(q_1, 0, L)	(q_2, 1, L)
q_2		(q_3, B, R)	(q_4, B, R)
q_3	(q_3, 0, R)	(q_3, 0, R)	(q_4, 1, R)
q_4	(q_5, 0, L)		

3) **Transition Diagram (Transition Graph):** TM can be represented using transition diagram also, the vertices denote the states. Directed edges are used to represent transition of states. Each label on the edge is of the form (α, β, γ).

Where α, β ∈ Γ and γ ∈ {L, R}.

So when there is a directed edge from state q_i to the state q_j with the label (α, β, γ), it means that

$\delta(q_i, \alpha) = (q_j, \beta, \gamma)$

Suppose while processing an input string, the TM is in **state q_i**. Also let α be the symbol under the **read/write head. Then for the** above transition, the symbol α will be replaced by β, the tape head will either move to left or right depending on γ and it will enter into the state q_j.

For example, consider the Turing machine given by the following table:

Table 5.2: Transition Table

Present State	Input-Tape Symbols	
	1	#
→ q_0	1Rq_0	1Rq_1
q_1	1Rq_1	#Lq_2
q_2	#Lq_3	–
q_3	1Lq_3	#RH
H	–	–

As the transition table shows that q_0 is the initial state and H is the halt state. The transition diagram is given by **figure 5.5**.

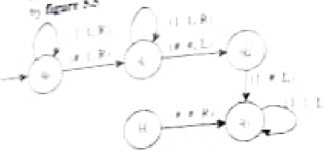


Figure 5.5: Transition Diagram

Ques 9) Write a note on language of a Turing machine.

Ans: Language of a Turing Machine

Let M be a Turing machine defined as M = (Q, Σ, Γ, B, q_0, δ, F) then language of M be L(M), where,

$L(M) = \{x \in \Sigma^* \mid q_0 x \vdash^* \alpha_1 p \alpha_2$ and $p \in F\}$ Where $\alpha_1, \alpha_2 \in \Gamma^*$

It means that the string x is in the language of machine if from starting state q_0 machine M reaches to the state p (final state) after scanning the complete string x (whatever the tape symbols it left α_1 and α_2).

The language of the Turing machine is the **recursive enumerable language**. The acceptance that is commonly used for Turing machines is the **acceptance by halting**. It means Turing machine halts if there is no move define further in that instance of problem state. Alternatively one can always assume that a Turing machine always halts when it is at an accepting state. In fact a class of languages whole Turing machine halts, regardless of whether or not it reaches an accepting state are called recursive languages. The other class of languages consists of these recursive enumerable languages that are not accepted by any Turing machine with the guarantee of halting. These languages are accepted in an inconvenient way.

Ques 10) Explain the term "Turing machines as language acceptors". Or Describe the language accepted by a Turing machine.

Ans: Turing Machines (TM) as Language Acceptors

The TM is started on a tape containing a string w ∈ Σ* at the beginning of the tape and blanks B after it. A TM

accepts a string w when it enters a final state in F. If the string is not accepted, the TM may or may not stop. It can be shown that the class of languages accepted by TM is the same as the class of languages generated by unrestricted grammars. A string w is written on the tape, with blanks filling out the unused portions. The machine is started in the initial state q_0 with the read-write head positioned on the left most symbol of w. If, after a sequence of moves, the TM enters a final state and halts, then w is considered to be accepted.

Definition

Let M = (Q, Σ, Γ, δ, q_0, ⊔, F) be a Turing machine.

The language accepted by M is

$L(M) = \{w \in \Sigma^* \mid q_0 w \vdash^* x_1 q_f x_2$ for some $q_f \in F$, $x_1, x_2 \in \Gamma^*\}$

This definition tells us that the machine will halt when w ∈ L (M).

When w is not in L(M) then the machine can halt in a non final state or it can enter an infinite loop and never halt.

For example, for Σ = {a, b}, we design a TM that accepts

$L = \{a^n b^n \mid n \geq 1\}$

Intuitively, the problem is solved in the following fashion. Starting at the leftmost a, we check it off by replacing it with some x.

aaabbb ⊢ xaabbb

We travel to the right to find the leftmost b, which we check off by replacing it with y.

xaaybb

We go left again to the leftmost a.

xaaybb

Traveling back and forth we match each a with a corresponding b.

Ques 11) How does a Turing machine differ from PDA and FSA? (2018 [05])

Ans: Difference Between Turing Machine, PDA and FSA

Device	Input	By Default Nature	Read/Write Data Structure	Type of Grammar	String/Language
FSA	Separate input	Deterministic	None	Regular (type 3)	a^n
PDA	Separate input	Non-deterministic	LIFO stack	Context-free (type 2)	$a^n b^n$
TM	No Separate input	Deterministic (but it also allows crashes)	1-way infinite tape, 1-cell access each step	Unrestricted (type 0)	$a^n b^n c^n$

Ques 12) Show that normal single tape Turing machine can perform computations performed by multi-tape Turing machine (informal explanation is sufficient). (2018 [05])

Ans: Two-Tape, Single-Head Turing Machines

A two-tape Turing machine is identical to the standard (single-tape) version, except that there are now two infinite tapes, which we might call tape A and tape B – **see Figure 5.6**. The input is placed on tape A before the computation begins, and tape B is initially filled with blanks. The machine has a single read-write head, which can be positioned over any single cell on both tapes. **For example,** in **Figure 5.6** the head is currently positioned at cell 3 on both tapes.

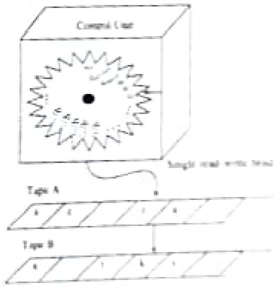


Figure 5.6: Two-Tape, Single Head Turing Machine

It is not possible for the head to be positioned simultaneously at different cells on each tape – this feature will be added soon, when we have multiple heads. The two-tape, single-head machine can read, erase, or write the current cell on both tapes in a single step. **For example,** given the configuration in **figure 5.6**, the machine could erase the pair (z, k) at cell 3 and write (·, ·) instead. The transition function depends on the values in the current cell on both tapes, of course.

A multitape TM can be simulated by a single tape TM.

Proof: Let M = (K, Σ, Γ, δ, q_0, F) be a k-tape turing machine. It can be simulated by a single tape TM M' having 2k tracks. Odd numbered tracks contain the contents of M's tapes.

To simulate a move of the multitape TM M, the single tape TM M' makes two sweeps, one from left to right and another from right to left. It starts on the leftmost cell which contains a X in one of the even tracks. While moving from left to right, when it encounters a X, it stores the symbol above it in its finite control. It keeps a counter as one of the component of the state to check whether it has read the symbols from all the tapes. After the left to right move is over, depending on the move of M determined by the symbols read, the single tape TM M' makes a right to left move changing the corresponding symbols on the odd tracks and positioning the X's properly in the even numbered tracks. To simulate one move of M, M' roughly takes (it may be slightly more depending on the left or right shift of X) a number of steps equal to twice the distance between the leftmost and the rightmost cells containing a X in an even numbered track. When M' starts all X's will be one cell. After i moves the distance between the leftmost and rightmost X can be at most 2i. Hence to simulate n moves of M, M' roughly takes $\sum_{i=1}^{n} 2(2i) = \sum_{i=1}^{n} 4i = O(n^2)$ steps. If M reaches a final state, M' accepts and halts.

Ques 13) Design a Turing machine that copies strings of 1's. More precisely, find a machine that performs the computation.

$$q_0 w \vdash^* q_f ww,$$

for any $w \in \{1\}^+$.

Ans: To solve the problem, we implement the following intuitive process:

1) Replace every 1 by an x.
2) Find the rightmost x and replace it with 1.
3) Travel to the right end of the current nonblank region and create a 1 there.
4) Repeat Steps 2 and 3 until there are no more x's.

The solution is shown in the transition graph in **figure 5.7**. It may be a little hard to see at first that the solution is correct, so let us trace the program with the simple string 11. The computation performe in this case is

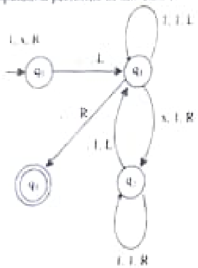


Figure 5.7

$q_0 11 \vdash xq_0 1 \vdash xxq_0 \vdash xq_1 x$
$\vdash 1q_2 \cdot xq_1 11 \vdash q_1 x11$
$1q_2 11 \vdash 11q_2 1 \vdash 111q_2$
$\vdash 11q_1 11 \vdash 1q_1 111$
$\vdash q_1 1111 \vdash q_1 B1111 \vdash q_f 1111$

Ques 14) Describe the process of designing Turing machine.

Ans: Designing Turing Machine

Construct a turing machine that accept the language $\{0^n1^n \mid n \geq 1\}$

First we can start by listing down some valid and invalid input strings. The valid inputs are 01, 00001111 etc. and invalid inputs are ε, 001, 0010 etc. The basic logic is to change 0 to X and 1 to Y in an alternating fashion until all 0s and 1s are matched. The numbers of states depends upon the problem and are introduced as required. In the present problem, we require a total of 5 states q_0 through q_4 where q_0 is the start state and q_4 is the accepting state. The input alphabet is {0, 1} and the tape symbols are {0, 1, X, Y, B}.

Thus the turing machine can be represented as

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1, X, Y, B\}, \delta, q_0, B, q_4)$$

And δ is given in the table below:

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, X, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

A Turing machine to accept $\{0^n1^n \mid n \geq 1\}$

Analysing the Transitions

The transitions are numbered T1, T2, ..., T10 for explanatory purpose. The transitions can be exercised using input strings like 0011 (valid), 0010 (Invalid).

T1 : $\delta(q_0, 0)$

At first the machine is in state q_0 and the head scans the first 0 from left. We have a 0 so rewrite it with X, change the state to q_1 and move to right starting the search for a 1.

T2 : $\delta(q_1, 0)$

Now the head my encounter a 0 but we rewrite it, and move to right and countinue search for 1.

T3 : $\delta(q_1, Y)$

The head may encounter a Y but we rewrite it, and move to right and countinue search for 1.

T4 : $\delta(q_1, 1)$

The head encounters a 1 so replace it with Y, change the state to q_2 and start moving to left in search for the leftmost 0 (comes after the rightmost X).

T5 : $\delta(q_2, Y)$

The head may encounter a Y but we rewrite it, and move to left and continue search for the leftmost 0 (comes after the rightmost X).

T6 : $\delta(q_2, 0)$

Now the head may encounter a 0 but we rewrite it (because it may not be the leftmost 0), and move to left and continue search for the rightmost X.

T7 : $\delta(q_2, X)$

We encounter rightmost X, so rewrite it, and move to right and change the state to q0. Now the machine is at q_0, the machine may follow the path from T1 if it encounters a 0. Otherwise continue down.

T8 : $\delta(q_0, Y)$

Instead if the machine encounters a Y, Change to state q_3 and move right seaching for a blank B.

T9 : $\delta(q_3, Y)$

When q_3 encounter a Y, just rewrite it and move right in search for B.

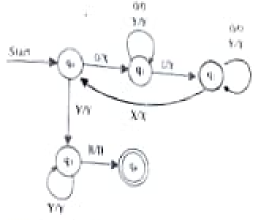
T10 : $\delta(q_3, B)$

When q_3 encounters B then machine change to state q4 and the string is accepted.

If M encounters any other transitions other than the above ones, then the input string is in the wrong from and M dies without accepting the string. At any time, the string on the tape can be represented using the regular expression $0^*X^*1^*Y^*$.

The entire sequence of moves for the input 0011 can be represented as

$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash$
$Xq_2 0Y1 \vdash q_2 X0Y1 \vdash Xq_0 0Y1 \vdash$
$XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash$
$Xq_2 XYY \vdash XXq_0 YY \vdash XXYq_3 Y \vdash$
$XXYYq_3 B \vdash XXYYBq_4 B$



For invalid input 0010 the moves are

$q_0 0010 \vdash Xq_1 010 \vdash X0q_1 10 \vdash$
$Xq_2 0Y0 \vdash q_2 X0Y0 \vdash Xq_0 0Y0 \vdash$
$XXq_1 Y0 \vdash XXYq_1 0 \vdash XXY0q_1 B$

Transition Diagrams for Turing Machines

A turing machine can be represented using a state transition diagram like a PDA. The nodes will represent the differnet states. An arc from state q_0 to q_1 will be labeled as 0/X → which represents the transition function $\delta(q_0, 0) = (q_1, X, R)$, the arrow represents the direction (right) the head moves.

Another arc from q_2 to q_2 (self loop) will be labeled Y/Y ← which represents the transition function $\delta(q_2, Y) = (q_2, Y, L)$, the arrow represents the direction (left) the head moves. The start state is represented by word "Start" and an arrow entering that state. Accepting states are represented using double circles. Transition diagram for a TM that accepts strings of the form 0^n1^n.

Ques 15) Design Turing machines for the following languages:

i) The set of strings with an equal number of 0's and 1's.

ii) $\{a^nb^nc^n \mid n \geq 1\}$

iii) $\{ww^R \mid$ W is any string of 0's and 1's$\}$

Or

Construct a Turing machine that recognises the language $L = \{a^nb^nc^n \mid n > 0\}$ (2017 [10])

Ans:

i) The set of strings with an equal number of 0's and 1's.

$M = (\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_f\})$ where δ is as follows:

State	Symbol 0	1	B	X	Y
q_0	(q_2, X, R)	(q_1, X, R)	(q_f, B, R)		(q_0, Y, R)
q_1	(q_3, Y, L)	$(q_1, 1, R)$			(q_1, Y, R)
q_2	$(q_2, 0, R)$	(q_3, Y, L)			(q_2, Y, R)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$		(q_0, X, R)	(q_3, Y, L)
q_f					

In explanation, the TM makes repeated excursions back and forth along the tape. The symbols X and Y are used to replace 0's and 1's that have been cancelled one against another. The difference is that an X guarantees that there are no unmatched 0's and 1's to its left (so the head never moves left of an X), while a Y may have 0's and 1's to its left. Initially in state q_0, the TM picks up a 0 or 1, remembering it in its state (q_1 = found a 1, q_2 = found a 0), and cancels what it found with an X. As an exception, if the TM sees the blank in state q_0, then all 0's and 1's have matched, so the input is accepted by going to state q_f.

In state q_1, the TM moves right, looking for a 0. If it finds it, the 0 is replaced by Y, and the TM enters state q_3 to move left an look for an X. Similarly, state q_2 looks for a 1 to match against a 0. In state q_3, the TM moves left until it finds the rightmost X. At that point, it enters state q_0 again, moving right over Y's until it finds a 0, 1 or blank, and the cycle begins again.

ii) $\{a^nb^nc^n \mid n \geq 1\}$

$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_f\}, \{a, b, c\}, \{a, b, c, X, Y, Z\}, \delta, q_0, B, \{q_f\})$ where δ is as follows:

State	Symbol a	b	c	X	Y	Z	B
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_2, b, R)	(q_3, Z, L)	–	–	(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_f, B, R)
q_f	–	–	–	–	–	–	–

The above TM is the 3 character version of the same machine given in the as example.

iii) $\{ww^R \mid$ w is any string of 0's and 1's$\}$

$M = (\{q_{start}, q_0, q_1, q_{1a}, q_{0a}, q_{back}, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_{start}, B, \{q_f\})$

Where δ is as follows:

State	Symbol		
q_{start}	(q_0, B, R)	(q_1, B, R)	(q_f, B, R)
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_{0a}, B, L)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_{1a}, B, L)
q_{0a}	(q_{back}, B, L)	–	–
q_{1a}		(q_{back}, B, L)	
q_{back}	$(q_{back}, 0, L)$	$(q_{back}, 1, L)$	(q_{start}, B, R)
q_f	–	–	–

Ques 16) Design a TM M to accept the language $L = \{0^n 1^n : n \geq 1\}$.

Ans: The general idea is as follows.
Initially the tape of M contains 0^n1^n followed by infinity of blanks. Repeatedly M replaces the left most 0 by X, moves right to the left most 1 replaces it by Y, moves left to find the right most X, then moves one cell right to the left most 0 and repeats the cycle. While searching for a 1 if M finds a blank, then M halts without accepting. If, after changing a 1 to a Y, M finds no more 0's then M checks whether there is any other 1's left out, if none, then it accepts.

Let $M = (Q, \Sigma, \Gamma, q_0, B, \delta, F)$ be a TM

Where, $Q = \{q_0, q_1, q_2, q_3, q_4\}$

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

$\Gamma = \{0, 1, X, Y, B\}$, q_0 is the starting state

$F = \{q_4\}$

q_0 is used immediately prior to each replacement of a left most 0 by an X. q_1 is used to search right, skipping 0's and Y's until it finds the leftmost 1 and replaces 1 by Y and the machine enters the state q_2. q_2 is used to search left for an X and if it finds then it enters state q_0 moving right to the leftmost 0, as it changes state q_3 is used to scan over Y's and check that no 1's remain. If the Y's are followed by B, then the machine enters the accepting state q_4, otherwise the string is rejected.

The action δ of M is defined as follows:

$\delta(q_0, 0) = (q_1, X, R)$ $\quad$ $\delta(q_2, X) = (q_0, X, R)$
$\delta(q_0, Y) = (q_3, Y, R)$ $\quad$ $\delta(q_2, Y) = (q_2, Y, L)$
$\delta(q_1, 0) = (q_1, 0, R)$ $\quad$ $\delta(q_3, Y) = (q_3, Y, R)$
$\delta(q_1, 1) = (q_2, Y, L)$ $\quad$ $\delta(q_3, B) = (q_4, B, H)$
$\delta(q_1, Y) = (q_1, Y, R)$
$\delta(q_2, 0) = (q_2, 0, L)$

The computation of M on the input string 0011 is as follows:

$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1$
$\vdash q_2 X0Y1 \vdash Xq_0 0Y1 \vdash XXq_1 Y1$
$\vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY$
$\vdash XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3$
$\vdash XXYYBq_4$

Ques 17) Design a TM to accept the language pal of palindromes over {a, b}.

Ans: Let $M = \{Q, \Sigma, \Gamma, q_0, B, \delta, F\}$ be a TM.

Where,

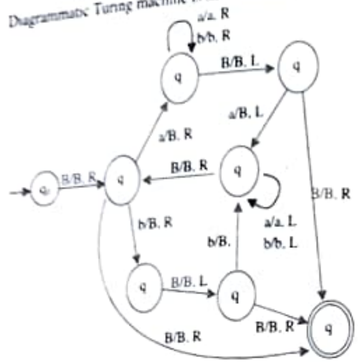
$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, $\Sigma = \{a, b\}$,
$\Gamma = \{a, b, B\}$

There are three different kinds of input strings:
1) Non-palindrome.
2) Even-length palindrome.
3) Odd-length palindrome.

The action δ is defined as follows:

$\delta(q_0, B) = (q_1, B, R)$ $\quad$ $\delta(q_2, a) = (q_2, a, R)$
$\delta(q_1, a) = (q_2, B, R)$ $\quad$ $\delta(q_2, b) = (q_2, b, R)$
$\delta(q_1, b) = (q_3, B, R)$ $\quad$ $\delta(q_2, B) = (q_4, B, L)$
$\delta(q_1, B) = (q_7, B, L)$ $\quad$ $\delta(q_4, a) = (q_6, B, L)$
$\delta(q_3, a) = (q_3, a, R)$ $\quad$ $\delta(q_6, a) = (q_6, a, L)$
$\delta(q_3, b) = (q_3, b, R)$ $\quad$ $\delta(q_6, b) = (q_6, b, L)$
$\delta(q_3, B) = (q_5, B, L)$ $\quad$ $\delta(q_6, B) = (q_1, B, R)$
$\delta(q_5, B) = (q_7, B, R)$ $\quad$ $\delta(q_4, B) = (q_7, B, R)$
$\delta(q_5, b) = (q_6, B, L)$

Diagrammatic Turing machine is shown below:



Even Palindrome

$q_0Baa \vdash Bq_1aa \vdash BBq_2a \vdash BBaq_2B \vdash BBq_4a \vdash Bq_6BB \vdash BBq_1B \vdash BBBq_7B$ (accepted)

Odd Palindrome

$q_0Baba \vdash Bq_1aba \vdash BBq_2ba \vdash BBbq_2a \vdash BBbaq_2B \vdash BBbq_4a \vdash BBq_6bB \vdash Bq_6Bb \vdash BBq_1b \vdash BBBq_3B \vdash BBq_5B \vdash BBBq_7$ (accepted)

Consider a non-palindrome say abaa

$q_0Babaa \vdash Bq_1abaa \vdash BBq_2baa \vdash BBbq_2aa \vdash BBbaq_2a \vdash BBbaaq_2B \vdash BBbaq_4a \vdash BBbq_6aB \vdash BBq_6ba \vdash Bq_6Bba \vdash BBq_1ba \vdash BBBq_3a \vdash BBBaq_3B \vdash BBBq_5a$ (reject)

Ques 18) Design a TM Accepting $\{ss \mid s \in \{a, b\}^*\}$

Ans: The idea behind the TM will be to separate the processing into two parts:
1) Finding the middle of the string, and making it easier for the TM to distinguish the symbols in the second half from those in the first half.
2) Comparing the two halves.

We accomplish the first task by working our way in from both ends simultaneously, changing symbols to their uppercase versions as we go.

This means that our tape alphabet will include A and B in addition to the input symbols a and b. There are two ways that an input string can be rejected. If its length is odd, the TM will discover this in the first phase. If the string has even length but a symbol in the first half fails to match the corresponding symbol in the second half, the TM will reject the string during the second phase.

Figure 5.8: Turing Machine to Accept $\{ss \mid s \in \{a, b\}^*\}$

In **figure 5.8** we trace it for three strings: two that illustrate both ways the TM can reject the input, and one that is in the language.

$(q_0, \Delta aba) \vdash (q_1, \Delta aba) \vdash (q_2, \Delta Aba) \vdash (q_2, \Delta Aba\Delta)$
$\vdash (q_3, \Delta Aba) \vdash (q_4, \Delta AbA) \vdash (q_4, \Delta AbA)$
$\vdash (q_1, \Delta AbA) \vdash (q_2, \Delta ABA) \vdash (q_3, \Delta ABA)$
$\vdash (h_r, \Delta ABA)$ (reject)

$(q_0, \Delta abaa) \vdash (q_1, \Delta abaa) \vdash (q_2, \Delta Abaa) \vdash (q_2, \Delta Abaa\Delta)$
$\vdash (q_3, \Delta Abaa) \vdash (q_4, \Delta AbaA) \vdash (q_4, \Delta AbaA)$
$\vdash (q_4, \Delta AbaA) \vdash (q_1, \Delta AbaA) \vdash (q_2, \Delta ABaA)$
$\vdash (q_2, \Delta ABaA) \vdash (q_3, \Delta ABaA) \vdash (q_4, \Delta ABAA)$
$(q_5, \Delta ABAA) \vdash (q_5, \Delta AbAA) \vdash (q_5, \Delta abAA)$
(first phase completed)
$\vdash (q_6, \Delta abAA) \vdash (q_8, \Delta AbAA) \vdash (q_8, \Delta AbAA)$
$\vdash (q_9, \Delta AbaA) \vdash (q_6, \Delta AbaA) \vdash (q_6, \Delta AbaA)$
$\vdash (q_7, \Delta ABAA) \vdash (q_7, \Delta ABAA) \vdash (h_r, \Delta ABAA\Delta)$ (reject)

$(q_0, \Delta abab) \vdash^*$
(same as previous case, up to 3rd-from-last move)
$\vdash (q_6, \Delta AbAB) \vdash (q_7, \Delta ABAB) \vdash (q_7, \Delta ABAB)$
$\vdash (q_7, \Delta ABA\Delta) \vdash (q_6, \Delta AB) \vdash (q_6, \Delta ABA)$
$\vdash (h_a, \Delta ABA)$ (accept)

Ques 19) Design a Turing machine M to recognize the language $\{1^n 2^n 3^n \mid n \geq 1\}$.

Ans: Before designing the required turing machine M, let us evolve a procedure for processing the input string 112233. After processing, we require the ID to be of the form bbbbbbq. The processing is done by using five steps.

Step 1: q_1 is the initial state. The R/W head scans the leftmost 1, replaces 1 by b, and moves to the right. M enters q_2.

Step 2: On scanning the leftmost 2, the R/W head replaces 2 by b and moves to the right. M enters q_3.

Step 3: On scanning the leftmost 3, the R/W head replaces 3 by b, and moves to the right. M enters q_4.

Step 4: After scanning the rightmost 3, the R/W heads moves to the left until it finds the leftmost 1. As a result, the leftmost 1, 2 and 3 are replaced by b.

Step 5: Steps 1-4 are repeated until all 1's, 2's and 3's are replaced by blanks. The change of IDs due to processing of 112233 is given as

$q_1 112233 \vdash bq_2 12233 \vdash b1q_2 2233 \vdash b1bq_3 233$
$\vdash b1b2q_3 33$
$\vdash b1b2bq_4 3 \vdash b1b2q_5 b3 \vdash b1bq_5 2b3 \vdash b1q_5 b2b3$
$\vdash bq_5 1b2b3$
$\vdash q_5 b1b2b3 \vdash bq_1 1b2b3 \vdash bbq_2 b2b3 \vdash bbbq_2 2b3$
$\vdash bbbbq_3 b3 \vdash bbbbbq_3 3 \vdash bbbbbbq_4 b \vdash bbbbbq_7 bb$

Thus, $q_1 112233 \overset{*}{\vdash} q_7 bbbbbb$

As q_7 is an accepting state, the input string 112233 is accepted.

Now we can construct the transition **table** for M. It is given in **table 5.3**.

Transition Table 5.3

Present State	Input Tape Symbol			
	1	2	3	b
→ q_1	bRq_2			bRq_1
q_2	$1Rq_2$	bRq_3		bRq_2
q_3		$2Rq_3$	bRq_4	bRq_3
q_4			$3Lq_5$	bLq_7
q_5	$1Lq_6$	$2Lq_5$		bLq_5
q_6	$1Lq_6$			bRq_2
(q_7)				

It can be seen from the table that strings other than those of the form $0^n1^n2^n$ are not **accepted**. It is advisable to compute the computation sequence for strings like 1223, 1123, 1233 and then see that these strings are rejected by M.

Ques 20) Design a TM which can multiply two positive integers.

Ans: The input (m, n) m, n being given, the positive integers are represented by 0^m10^n. M starts with 0^m10^n in its tape. At the end of the computation, 0^{mn} (mn in unary representation) surrounded by b's is obtained as the output. The major steps in the construction are as follows:

Step 1) 0^m10^n1 is placed on the tape (the output will be written after the rightmost 1).

Step 2) The leftmost 0 is erased.

Step 3) A block of n 0's copied onto the right end.

Step 4) Steps 2 and 3 are repeated m times and 10^m10^{mn} is obtained on the tape.

Step 5) The prefix 0^m1 of 10^m10^{mn} is erased, leaving the product mn as the output.

For every 0 in 0^m, 0^n is added onto the right end. This requires repetition of step 3. We define a subroutine called COPY for step 3. For the subroutine COPY, the initial state is q_1 and the final state is q_5. δ is given by the transition table (see **table 5.4**).

Table 5.4: Transition Table for Subroutine COPY

State	Tape Symbol			
	0	1	2	b
q_1	$q_2 2R$	$q_4 1L$	–	–
q_2	$q_2 0R$	$q_2 1R$	–	$q_3 0L$
q_3	$q_3 0L$	$q_3 1L$	$q_1 2R$	–
q_4	–	$q_5 1R$	$q_4 0L$	–
q_5	–	–	–	–

The Turing machine M has the initial stage q_0. The initial ID for M is $q_0 0^m10^n1$. On seeing 0, the following moves take place (q_6 is a state of M): $q_0 0^m10^n1 \vdash bq_6 0^{m-1}10^n1 \overset{*}{\vdash} b0^{m-1}q_6 10^n1 \vdash b0^{m-1}1q_1 0^n1$. q_1 is the initial state of COPY. The TM M_1 performs the subroutine COPY. The following moves take place for M_1: $q_1 0^n1 \vdash 2q_2 0^{n-1}1 \overset{*}{\vdash} 20^{n-1}1q_2 b \vdash 20^{n-1}q_3 10 \overset{*}{\vdash} 2q_1 0^{n-1}10$

After exhausting 0's q_1 encounters **1. M_1** moves to state q_4. All 2's are converted back to 0's and M_1 halts in q_5. The TM M picks up the computation by starting from q_5. The q_0 and q_6 are the states of M. Additional states are created to check whether each 0 in 0^m gives rise to 0^n at the end of the rightmost 1 in the input string. Once this is over, M erases 10^n1 and finds 0^{mn} in the input tape.

M can be defined by

$M = (\{q_0, q_1, \ldots, q_{12}\}, \{0, 1\}, \{0, 1, 2, b\}, \delta, q_0, b, \{q_{12}\})$

Where δ is defined by **table 5.5**:

Table 5.5: Transition Table

	0	1	2	b
q_0	$q_6 bR$	–	–	–
q_6	$q_6 0R$	$q_1 1R$	–	–
q_5	$q_7 0L$	–	–	–
q_7	–	$q_8 1L$	–	–
q_8	$q_9 0L$	–	–	$q_{10} bR$
q_9	$q_9 0L$	–	–	$q_0 bR$
q_{10}	–	$q_{11} bR$	–	–
q_{11}	$q_{11} bR$	$q_{12} bR$	–	–

Thus M performs multiplication of two numbers in unary representation.

Ques 21) Design a TM that copies strings of 1's.

Ans: We design a TM so that we have ww after copying w ∈ {1}*. Define M by

$M = (\{q_0, q_1, q_2, q_3\}, \{1\}, \{1, b\}, \delta, q_0, b, \{q_3\})$

Where δ is defined by **table 5.6**:

Table 5.6: Transition Table

Present State	Tape Symbol		
	1	b	a
q_0	$q_0 aR$	$q_1 bL$	–
q_1	$q_1 1L$	$q_3 bR$	$q_2 1R$
q_2	$q_2 1R$	$q_1 1L$	–
q_3	–	–	–

The procedure is simple. M replaces every 1 by the symbol a. Then M replaces the rightmost a by 1. It goes to the right end of the string and writes a 1 there. Thus M has added a 1 for the rightmost 1 in the input string w. This process can be repeated. M reaches q_1 after replacing all 1's by a's and reading the blank at the end of the input string. After replacing a by 1, M reaches q_2. M reaches q_3 at the end of the process and halts. If $w = 1^n$, than we have 1^{2n} at the end of the computation. A sample computation is given below:

$$\begin{aligned} q_0 11 &\vdash aq_0 1 \vdash aaq_0 b \vdash aq_1 a \\ &\vdash a1q_2 b \vdash aq_1 11 \vdash q_1 a11 \\ &\vdash 1q_2 11 \vdash 11q_2 1 \vdash 111q_2 b \\ &\vdash 11q_1 11 \vdash 1q_1 111 \\ &\vdash q_1 1111 \vdash q_1 b1111 \vdash q_3 1111 \end{aligned}$$

Ques 22) Construct a TM to accept the set L of all strings over {0,1} ending with 010.

Ans: L is certainly a regular set and hence a deterministic automaton is sufficient to recognise L. **Figure 5.9** gives a DFA accepting L.

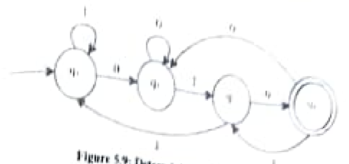


Figure 5.9: Deterministic Finite Automata (DFA)

Converting this DFA to a TM is simple. In a DFA M, the move is always to the right. So the TM's move will always be to the right. Also M reads the input symbol and changes state. So the TM M_1 does the same; it reads an input symbol, does not change the symbol and changes state. At the end of the computation, the TM sees the first blank b

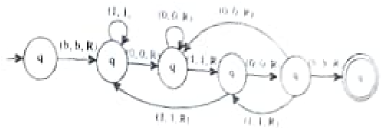


Figure 5.10: Turing Machine (TM)

and changes to its final state. The initial ID of M_1 is $q_0 w$. By defining $\delta(q_0, b) = (q_1, b, R)$, M_1 reaches the initial state of M. M_1 can be described by **figure 5.10**.

Note: q_5 is the unique final state of M_1. By comparing **figures 5.9** and **5.10** it is easy to see that strings of L are accepted by M_1.

Ques 23) Design a Turing machine that reads binary strings and counts the number of 1's in the sequence. The output is 0 if the number of 1's in the string is even and 1 if the number of 1's is odd.

Or

Design a Turing machine that determines whether the binary input string is of odd parity or not? (2018 [05])

Ans: The actions that should be carried out are:

1) Search for 1's to the right.
2) The 0's in the sequence are read but ignored.
3) Maintain a count of whether the number of 1's is odd or even.
4) After a blank (#) is found, write the output to the right of the input string.

The set of states is as follows:

q0 searches to the right a 1, indicates the number of 1's so far is even.

q1searches to the right for a 1, indicates the number of 1's so far is odd.

The Turing machine is represented by the following state diagram:

TM = ({q0, q1, [illegible]}, {0, 1}, {0, 1, #}, δ, q0, {p}) where δ is given by

Q \ Σ	0	1	#
q_0	(q_0, 0, R)	(q_1, 1, R)	(p, 0, R)
q_1	(q_1, 0, R)	(q_0, 1, R)	(p, 1, R)

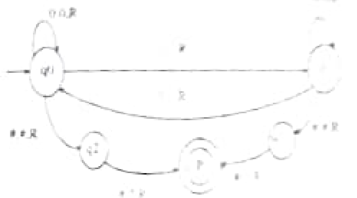
Processing of the string "01#" is as follows:

$q_0 01\# \to 0q_0 1\# \to 01q_1 \# \to 011p$

Processing of the string "1011010#" is as follows:

$q_0 1011010\# \to 1q_1 011010\# \to 10q_1 11010\# \to$
$101q_1 1010\# \to 1011q_0 010\# \to 10110q_0 10\# \to$
$101101q_1 0\# \to 1011010q_1\# \to 10110101p$

Modify the above Turing machine so that the output is written to the right of the blank symbol and then followed by another blank symbol. That is, after processing the input string "01#" the tape should have the string "01#1#" and after processing "1011010#" the tape should have "1011010#0#". The extra action that should be carried out is to search the blank symbol and write the output to the right. The modified Turing machine is represented by the following state diagram:



Formally

TM = ({q0, q1, q2, q3, p}, {0, 1}, {0, 1, #}, δ, q0, {p}) where δ is given by

Q \ Σ	0	1	#
q_0	[illegible]	[illegible]	[illegible]
q_1	[illegible]	[illegible]	[illegible]
q_2	[illegible]	[illegible]	[illegible]
q_3	[illegible]	[illegible]	[illegible]

Processing of the string "01#" is as follows:

[illegible]

Processing of the string "1011010#" is as follows:

[illegible]

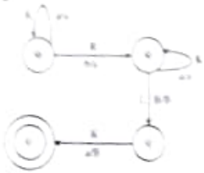
Ques 24) Design a Turing machine that accepts a^nb^m where n>0 and m>n. (2018 [05])

Ans: The Turing machine will have $a^n b^m$ on its tape initially. Therefore it will start with the leftmost a, go on scanning a's and moving right keeping them as it is, till it get B(blank), it replaces this b by a, and then again go on scanning a's and moving right keeping them as it is. When it gets a B(blank), it moves left, and replaces the rightmost a by B and halts. Therefore, the moves of the Turing machine are

	a	b	B
q_0	(q_1, a, R)	(q_0, a, R)	–
q_1	(q_1, a, R)	–	(q_2, B, L)
q_2	(q_3, B, R)		–
q_3		–	

Therefore, the turing machine $M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, b, B\}, \delta, q_0, B, \{q_3\})$, where δ is given above.

The transition diagram corresponding to the above table is shown in **figure** below.



Ques 25) Design the Turing machine to recognise the language: $\{0^n 1^n 0^n \mid n \geq 1\}$. (2019[10])

Ans: First replace a 0 from front by X, and then keep moving right till you find a 1 and replace this 1 by Y. Again, keep moving right till you find a 2, replace it by Z and move left. Now keep moving left till you find a X. When you find it, move a right, then follow the same procedure as above.

A condition comes when you find a X immediately followed by a Y. At this point we keep moving right and keep on checking that all 1's and 2's have been converted to Y and Z. If not then string is not accepted. If we reach $ then string is accepted.

Step 1: Replace 0 by X and move right, Go to state Q1.

Step 2: Replace 0 by 0 and move right, Remain on same state. Replace Y by Y and move right, Remain on same state. Replace 1 by Y and move right, go to state Q2.

Step 3: Replace 1 by 1 and move right. Remain on same state.

Replace Z by Z and move right. Remain on same state.

Replace 0 by Z and move right, go to state Q3.

Step 4: Replace 1 by 1 and move left. Remain on same state. Replace 0 by 0 and move left. Remain on same state.

Replace Z by Z and move left. Remain on same state.

Replace Y by Y and move left. Remain on same state.

Replace X by X and move right, go to state Q0.

Step 5: If symbol is Y replace it by Y and move right and Go to state Q4.

Else go to step 1.

Step 6: Replace Z by Z and move right. Remain on same state. Replace Y by Y and move right. Remain on same state. If symbol is $ replace it by $ and move left. STRING IS ACCEPTED. GO TO FINAL STATE Q5.

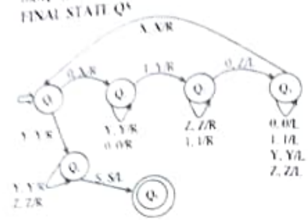


Figure 5.11

Ques 26) Discuss the universal Turing machine.

Or

Discuss about Universal Turing Machines. (2017 [05])

Or

Write a note on Universal Turing Machines. (2019[2.5])

Ans: Universal Turing Machine

A universal Turing machine is designed to simulate the computations of an arbitrary Turing machine M. To do so, the input to the universal machine must contain a representation of the machine M and the string w to be processed by M. For simplicity, we will assume that M is a standard Turing machine that accepts by halting.

The action of a universal machine U is depicted by:

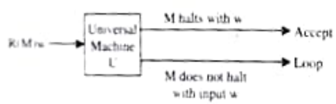


Figure 5.12

Where R(M) is the representation of the machine M. The output labelled loop indicates that the computation of U does not terminate. If M halts and accepts input w, U does the same. If M does not halt with w, neither does U. The machine U is called universal since the computation of any Turing machine M can be simulated by U.

A Universal Turing Machine (UTM) is a Turing machine that can be fed as input a string composed of two parts:

1) An encoded Turing machine T followed by a marker.
2) A string that is to be run through T.

The UTM runs T on the string. If T rejects x then the UTM rejects it. If T accepts then the UTM accepts, and if T traps into a loop then the UTM also does the same. Typically implementations involve atleast three tapes, one on which the UTM copies the encoding of T and another on which it copies the string to be fed to T. The third tape is used to keep track of which state T is in. The UTM operates by looking up a transition for T's current state using the current symbol of the input string for T, then carrying out that transition.

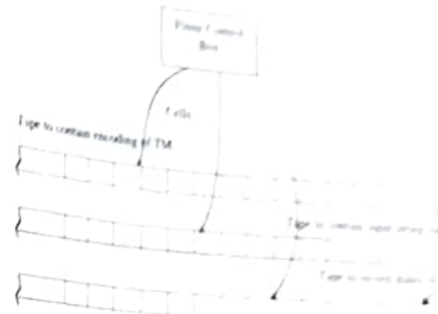


Figure 5.13: Universal Turing Machine with Three Tapes

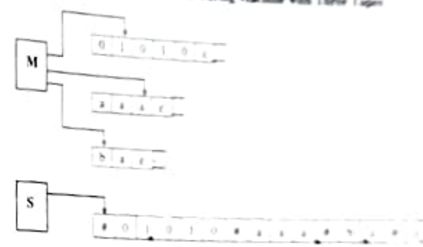


Figure 5.14: Representing Three Tapes with One

One way to encode a TM is to encode all of its states, tape symbols and directions. The simplest way to do this is with a unary encoding:

1) **Encoding States:** We shall assume that Turing machine has only one final state. We also adopt the convention that start state and final states will always be represented by q_1 and q_2 respectively. All other states shall be represented by $q_3, q_4, \ldots, q_k$ for some k. Then q_1 will be represented by 1, q_2 by 11, q_3 by 111 and so on.

2) **Encoding Tape Symbols:** We can encode tape symbols as a = 1, b = 11, B = 111, and so on.

3) **Encoding Directions:** We shall encode L = 1, R = 11, S = 111.

4) **Encoding Transitions:** We shall encode the transitions, $\delta(q_i, a) = (q_j, x, D)$, of a TM as a sequence of 5-tuples (old state, symbol read, new state, symbol to write, direction) each component of which will be separated by a 0. **For example,** to encode the 5-tuple (q_1, a_1, q_2, a_2, L) in unary, we could use the sequence 10110111011101.

5) **Encoding Turing Machine:** We can encode a Turing machine as a binary string consisting of all of encoded transitions separated by 00.

Language called **universal language** L_u is set of strings in the form of pair (T, x) such that $x \in L(T)$. The machine describing this language is universal Turing machine.

For example, consider the Turing machine described by transition **table 5.7**

Table 5.7

State	Input	Move $(\delta q_i$ X)	Encoding of Transitions
q_1	a	[illegible]	[illegible]
q_1	b	[illegible]	[illegible]
q_2	a	[illegible]	[illegible]
q_3	a	[illegible]	[illegible]

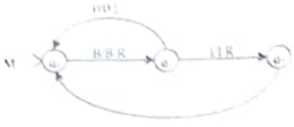
A binary string equivalent to this Turing machine will be [illegible]

The computation of the Turing machine halts for the null string and strings that begin with 1 and does not terminate for strings beginning with 0.

The encoded transitions of M are given in the following table:

Transition	Encoding
$\delta(q_1, B) = [q_2, B, R]$	[illegible]
$\delta(q_2, 0) = [q_2, 0, L]$	[illegible]
$\delta(q_2, 1) = [q_3, 1, R]$	[illegible]
$\delta(q_3, 1) = [q_2, 1, L]$	[illegible]

The machine M is represented by the string
000101110110111011001010101010010110111011011011
00111011010110100



Ques 27) How does the Universal Turing machine simulate other Turing machines? **(2018 [05])**

Ans: Universal Turing Machine Simulates Other Turing Machines

UTM is able to simulate all other TMs, that's what the universal part of the name means. Since that simulation is also known as software, it involves inputs only. We can assume that the state set and alphabet of the machine to be simulated are subsets of the natural numbers ("State 1", "State 2", "State 3", etc.). Even with just two non-blank characters, the universal machine can represent all those integers as binary strings. Note, though, that the universal machine has a fixed number of states, which makes computing the transition function a little tricky. What we would like to do is write some instructions that implement a big switch statement of the form "If the state is ss and the character under the head is xx, then move to state s's', write character x'x' and move the head in direction dd." So – and I think this may be the root of your question – how do we calculate the transition function if we do not even have enough states in the universal machine to store the transition function's input?

One way is to store the transition function as a binary tree. Suppose all the simulated machine has 2q2q states and 2t2t tape symbols. Store the transition function as a binary tree of depth q+tq+t where, at the first qq levels, you go left or right according to whether the next bit of the simulated state is a one or a zero and the next tt levels are the same but for the successive bits of the simulated tape character. Now, your universal machine can walk backwards and forwards on its tape, checking the next bit of the state/character, remembering that bit in its own states, moving back to the tree, putting a marker at the correct node, and so on. This gets somewhat easier if you let your universal machine have multiple tapes but then you still have to show that your multitape machine is equivalent to a single tape machine.

HALTING PROBLEM

Ques 28) Discuss the halting problem of Turing problem.
Or
Write short note on Halting Problem of Turing. (2019[3.5])

Ans: Halting Problem of Turing Machines

One begins with some problems that have historical significance and that at the same time give us a starting point for developing later results. The best-known of these is the Turing machine halting problem. Simply stated, the problem is: "Given the description of a Turing machine M and an input w, does M, when started in the initial configuration qow, perform a computation that eventually halts?" Using an abbreviated way of talking about the problem, one asks whether M applied to w, or simply (M, w), halts or does not halt. The domain of this problem is to be taken as the set of all Turing machines and all w; i.e., one is looking for a single Turing machine that, given the description of an arbitrary M and w, will predict whether or not the computation of M applied to w will halt."

One cannot find the answer by simulating the action of M on w, say by performing it on a universal Turing machine, because there is no limit on the length of the computation. If M enters and infinite loop, then no matter how long one waits, he/she can never be sure that M is in fact in a loop. It may simply be a case of a very long computation. What one needs is an algorithm that can determine the correct answer for any M and w by performing some analysis on the machine's description and the input.

One might reasonably ask whether any problems can be shown to be incomputable. More common terms for such problems – those known to be insolvable by any computer problems – are intractable or undecidable. In the same 1936 paper, **Alan Turing** also provided an answer to this question by introducing (and proving) that there are in fact problems that cannot be computed by a universal computing machine. The problem that he proved undecidable, using proof techniques almost identical to those developed for similar problems in the 1880s, is now known as the halting problem.

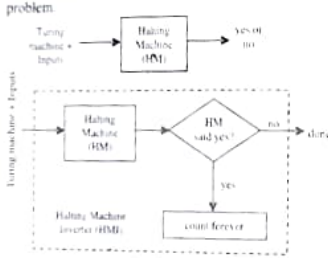


Figure 5.15

The halting problem is easy to state and easy to prove undecidable. The problem is this – given a Turing machine and an input to the Turing machine, does the Turing machine finish computing in a finite number of steps (a finite amount of time)? In order to solve the problem, an answer, either yes or no, must be given in a finite amount of time regardless of the machine or input in question. Clearly some machines never finish. **For example**, one can write a Turing machine that counts upwards starting from one. To see that no Turing machine can solve the halting problem, one begins by assuming that such a machine exists, and then show that its existence is self contradictory. One calls the machine the "Halting machine (HM)". HM is a machine that operates on another machine and its inputs to produce a yes or no answer in finite time: either the machine in question finishes in finite time (HM returns "yes"), or it does not (HM returns "no"). **Figure 5.15** illustrates HM's operation. From HM, one construct a second machine that he/she calls the HM Inverter(HMI). This machine inverts the sense of the answer given by HM. In particular, the inputs are fed directly into a copy of HM, and if HM answers "yes", HMI enters an infinite loop. If HM answers "no", HMI halts. A diagram appears to the right.

The inconsistency can now be seen by asking HM whether HMI halts when given itself as an input (repeatedly), as shown in **figure 5.16**. Two copies of HM are thus being asked the same question. One copy is the rightmost in the figure below and the second is embedded in the HMI machine that one is using as the input to the rightmost HM. As the two copies of HM operate on the same input (HMI operating on HMI), they should return the same answer – a Turing machine either halts on an input, or it does not; they are deterministic.

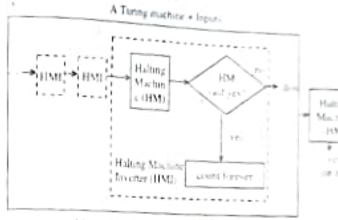


Figure 5.16: Halting Machine's Operation

Consider that the rightmost HM tells us that HMI operating on itself halts. Then the copy of HM in HMI (when HMI executes on itself, with itself as an input) must also say "yes". But this answer implies that HMI does not halt, so the answer should have been no.

Alternatively, one can assume that the rightmost HM says that HMI operating on itself does not halt. Again, the copy of HM in HMI must give the same answer. But in this case HMI halts, again contradicting assumption. Since neither answer is consistent, no consistent answer can be given, and the original assumption that HM exists is incorrect. Thus, no Turing machine can solve the halting problem.

Ques 29) Prove that there is a Turing machine K with alphabet {1} that has an unsolvable halting problem.

Ans: Take for the set U, some recursively enumerable set that is not recursive. Let K be the corresponding Turing machine. Thus K accepts a string of ones if and only if its length belongs to U. Hence, x ∈ U if and only if K eventually halts when started with the configuration

$B 1^{(x)}$
$\uparrow$
q_1

Thus, if there were an algorithm for solving the halting problem for K, it could be used to test a given number x for membership in U. [illegible] algorithm is impossible. Our present result gives a fixed Turing machine whose halting problem is unsolvable. Actually, this result could also have been easily obtained from a universal Turing machine. Next, we show how the unsolvability of the halting problem can be used to obtain another unsolvable problem concerning Turing machines. We begin with a Turing machine K with alphabet {1} that has an unsolvable halting problem.

Let the states of K be $q_1, \ldots, q_n$. We will construct a Turing machine by $\bar{K}$ by adjoining to the quadruples of K the following quadruples:

$q_i B B q_{n+1}$

for i = 1, 2, ..., k for which no quadruple of K begins q_i B and q_i 1 1 q_{n+1}

for i = 1, 2, ..., k when no quadruple of K begins q_i 1.

Thus, K eventually halts beginning with a given configuration if and only if $\bar{K}$ eventually is in state q_{n+1}.

Ques 30) Explain why Halting problem is unsolvable problem. **(2018 [05])**

Ans: Halting Problem is Unsolvable Problem

One of well-known unsolvable problems is the halting problem. It asks the following question:

Given an arbitrary Turing machine M over alphabet = {a, b}, and an arbitrary string w over {a, b}, does M halt when it is given w as an input?

It can be shown that the halting problem is not decidable, hence unsolvable.

Theorem: The halting problem is undecidable.

Proof (by M. L. Minsky): This is going to be proven by "proof by contradiction". Suppose that the halting problem is decidable. Then there is a Turing machine T that solves the halting problem. That is, given a description of a Turing machine M (over the alphabet ∑) and a string w, T writes "yes" if M halts on w and "no" if M does not halt on w, and then T halts.

Turing Machine T

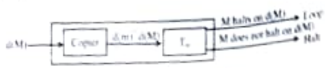
We are now going to construct a new Turing machine T_c [illegible]

[illegible]

Turing Machine T_c

Next using T_c we are going to construct another Turing machine T_d as follows:

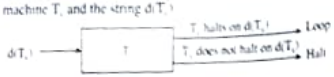
T_d takes as input a description of a Turing machine M, denoted by d(M), copies it to obtain the string d(M)*d(M), where * is a symbol that separates the two copies of d(M) and then supplies d(M)*d(M) to the Turing machine T.



Turing Machine T_1

Let us now see what T_1 does when a string describing T_1 itself is given to it.

When T_1 gets the input $d(T_1)$, it makes a copy, constructs the string $d(T_1)*d(T_1)$ and gives it to the modified T. Thus the modified T is given a description of Turing machine T_1 and the string $d(T_1)$.



Turing Machine T_1 on input $d(T_1)$

The way T was modified the modified T is going to go into an infinite loop if T_1 halts on $d(T_1)$ and halts if T_1 does not halt on $d(T_1)$. Thus T_1 goes into an infinite loop if T_1 halts on $d(T_1)$ and it halts if T_1 does not halt on $d(T_1)$. This is a contradiction. This contradiction has been deduced from our assumption that there is a Turing machine that solves the halting problem. Hence that assumption must be wrong. Hence there is no Turing machine that solves the halting problem.

RECURSIVE LANGUAGES AND RECURSIVELY ENUMERABLE

Ques 31) What is enumeration machine?

Ans: Enumeration Machine

An enumeration machine is a modification of a Turing machine. It has a finite control and two tapes, a read/write work tape and a write-only output tape. The work tape head can move in either direction and can read and write any element of Γ. The output tape head moves right one cell when it writes a symbol, and it can only write symbols in Σ. There is no input and no accept or reject state. The machine starts in its start state with both tapes blank. It moves according to its transition function like a TM, occasionally writing symbols on the output tape as determined by the transition function. At some point it may enter a special enumeration state, which is just a distinguished state of its finite control.

When that happens, the string currently written on the output tape is said to be enumerated. The output tape is then automatically erased and the output head moved back to the beginning of the tape (the work tape is left intact), and the machine continues from that point. The machine runs forever. The set L(E) is defined to be the set of all strings in Σ* that are ever enumerated by the enumeration machine E. The machine might never enter its enumeration state, in which case L(E) = ∅, or it might enumerate infinitely many strings. The same string may be enumerated more than once. Enumeration machines and Turing machines are equivalent in computational power.

Ques 32) What are recursively enumerable languages? Also define the recursive language. Or

Define recursively enumerable languages and recursive language with their properties. *Or*

Write a note on Recursive Enumerable Languages.

Or **(2017 [05])**

What is a Recursive language? Give an example. (2018 [05])

Ans: Recursively Enumerable Languages

The language L ∈ Σ is called recursively enumerable language if there is a Turing Machine T that accepts every word in L and either rejects or loops for every word in the complement of L.

Recursive Language

A language L over input set Σ is called **recursive** if there is a TM that accepts every word in L and rejects every word in L'. A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language.

For example, $L=\{a^nb^nc^n|n>=1\}$ is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^nb^nc^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as Turing decidable languages. From these two languages, following observations are made:

1) Every recursive language is recursively enumerable language.
2) Not all recursively languages are recursive. That means every TM which accepts the recursively enumerable language must have some words for which it loops forever.

Properties of Recursive and Recursively Enumerable Languages

The language accepted by FM is called a regular language; the language accepted by PDA is a context free language. But TM defines two classes of languages such as recursively enumerable language and recursive language. When a string belonging to Σ runs on TM then it results in three states:

1) Acceptance of string.
2) Loops for ever on receiving string, and
3) Rejects the given string.

Now we can define recursively enumerable language and recursive language.

Ques 33) Write the properties of recursive languages and recursively enumerable languages. (2019[03])

Ans: Closure Properties of Recursive Languages

1) **Union:** If L1 and If L2 are two recursive languages, their union L1 ∪ L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1 ∪ L2.
2) **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive. **For example,**

$L_1=\{a^nb^nc^n|n>=0\}$
$L_2=\{d^me^mf^m|m>=0\}$
$L_3=L_1.L_2$
$=\{a^nb^nc^nd^me^mf^m|m>=0 \text{ and } n>=0\}$ is also recursive

L_1 says n no. of a's followed by n no. of b's followed by n no. of c's. L_2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.

3) **Kleene Closure:** If L_1 is recursive, its kleene closure L1* will also be recursive. **For example,**

$L_1=\{a^nb^nc^n|n>=0\}$
$L_1^*=\{a^nb^nc^n|n>=0\}^*$ is also recursive

4) **Intersection and Complement:** If L_1 and L_2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. **For example,**

$L_1=\{a^nb^nc^nd^m|n>=0 \text{ and } m>=0\}$
$L_2=\{a^nb^nc^nd^n|n>=0 \text{ and } m>=0\}$
$L_3=L_1 \cap L_2$
$=\{a^nb^nc^nd^n|n>=0\}$ will be recursive

L_1 says n no. of a's followed by n no. of b's followed by n no. of c's and then any no. of d's. L_2 says any no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. Their intersection says n no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. So it can be decided by turing machine, hence recursive. Similarly, complement of recursive language L_1 which is Σ^*-L_1, will also be recursive.

Ques 34) Prove that if a language L and its complement L' both are RE then L is a recursive language.

Ans: Consider a Turing machine M made-up of two Turing machines M1 and M2. The machine M2 is complement of machine M1. We can also denote that L(M) = L(M1) and L(M2). Both M1 and M2 are simulated in parallel by machine M. Machine M is a two tape TM, which can be made one tape TM for the ease of simulation. This one tape then will consist of tape of machine M1 and machine M2. The states of M consists of all the states of machine M1 and all the states of machine M2. The machine M made-up of M1 and M2 is as shown below:

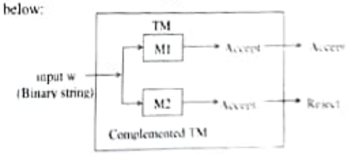


Figure 5.17

If the input w of language L is given to M then M1 will eventually accept and therefore M will accept L and halt. If w is not in L, then it is in L'. So M2 will accept and therefore M will half without accepting.

Thus on all inputs, M halts. Thus L(M) is exactly L. Since M always halts we can conclude that L(M) mean L is a recursive language. Thus a recursive language can be recursively enumerable but a recursively enumerable language is not necessarily be recursive.

Ques 35) What do you mean by enumerating a language?

Ans: Enumerating a Language

Let T be a k-tape TM (k ≥ 1) and L ⊆ Σ*. We say T enumerates L if it operates so that the following conditions are satisfied:

1) The tape head on the first tape never moves to the left and no non-blank symbol printed on tape 1 is subsequently modified or erased.
2) For every x ∈ L, there is some point in the operation of T when tape 1 has contents

$x_1\#x_2\# \ldots \#x_n\#x\#$

for some n ≥ 0, where the strings $x_1, x_2, \ldots, x_n, x$ are distinct elements of L. If L is finite, then nothing is printed after the # following the last element of L.

Ques 36) Discuss the Chomsky hierarchy of grammar

Or

Explain Chomsky's Hierarchy of Languages.

Or **(2017 [06])**

Explain Chomsky hierarchy and corresponding type0, type1, type2 and type3 formalism. (2018 [05])

Or

Write the Chomsky Hierarchy of languages. (2019[03])

Ans: Chomsky Hierarchy/Classification of Grammar

Since productions or rules provides the basis to the grammar. A rule may be represented by α → β. There are several possibilities of the selection of the term α and β in the rule. On the basis of these selections of α and β we classify the productions. Therefore, there are several restrictions in the production α → β on which grammars are classified.

1) **Restriction 1:** For the grammar G, α must contain atleast one non-terminal.

i.e., $\alpha \in (V_N \cup V_T)^* V_N (V_N \cup V_T)^*$ and $\beta \in (V_N \cup V_T)^*$

Let $V_T = \{a, b, c\}$ and $V_N = \{S, A, B, C\}$ then applying this restriction following are only valid productions.

i) aABac → abBC

(Here α (left side of production) is aABac, which contains two non-terminals A and B and β (right side of production) is abBC and both α and β ∈ $(V_N \cup V_T)^*$

ii) But ab → ABac is not a valid production, because on its left side there is not a single non-terminal symbol.

iii) If ∈ V_N then, Aab → ε is a valid rule. [where ε is a null string]

iv) But ε → ab is not a valid production because ε ∉ V_N. Therefore α ≠ ε.

The grammar defined under above restriction is called **Phase Structured Grammar** or **Type-0 Grammar** or **Recursive Enumerable Grammar** and the language generated by this grammar is called **Phase Structured Language** or **Type-0 Language** or **Recursive Enumerable Language**. The automata accept such language is **Turing Machine**.

2) **Restriction 2 (Followed by Restriction 1)**: For the grammar G, if $\alpha \rightarrow \beta$ is a rule then alongwith the restriction 1 such that α contains atleast a non-terminal it follows another restriction, $|\beta| \geq |\alpha|$, or length of right side derived symbols is not less than the length of left side derivatives symbols.

i.e. $\alpha \in (V_N \cup V_T)^* V_N (V_N \cup V_T)^*$ and $\beta \in (V_N \cup V_T)^*$ and also $|\alpha| \leq |\beta|$ and $\beta \neq \epsilon$

Some of the valid productions are:

i) From the previous example of production, i.e., aABac → abBC is not a valid production here, because |abBC| (2) < |aABac| (...), on left side there should atleast five symbols (terminals/non-terminals) but it fulfills restriction 1.

ii) A production of type bABc → babbc is certainly a valid production. On left side it contains 2 non-terminals and the length of derived symbols (5) is greater than length of derivatives symbols (4).

iii) Aab → ε is not a valid production. Because, |Aab| (3) > (0) or length of derived symbol (0) is not greater than or equal to the length of derivatives symbols (3).

The grammars follow restriction 2 along with restrictions 1 is called **Context Sensitive Grammar** or **Type-1 Grammar** or **Length Increasing Grammar** and its language is called **Context Sensitive Language** or **Type-1 Language** or **Length Increasing Language**.

3) **Restriction 3 (followed Restriction 2 + Restriction 1)**: If the grammar G has the production $\alpha \rightarrow \beta$ then besides, restriction 1 and restriction 2, there is another restriction, i.e., α **must be** a single non-terminal symbol. Some of the valid productions are:

i) A → ab: there is only a single derivative symbol that is A, and restriction 1 and 2 also fulfill.

ii) B → aABc

iii) A → ε is not also a valid production.

So the grammar that is bounded under these restrictions (1, 2 and 3) is **Context Free Grammar** or **Type-2 Grammar** and so the language is **Context Free Language** or **Type-2 Language**. The automaton accepts such type of language is **Push down Automata**.

4) **Restriction 4 (Followed Restriction 3 + Restriction 2 + Restriction 1)**: In the grammar G, if $\alpha \rightarrow \beta$ is a production then, besides above restrictions (1, 2, 3) restriction 4 says, β must be a single terminal symbol or a terminal followed by a non-terminal symbol. Such as, followings are the valid type productions:

i) A → b, where $b \in V_T$

ii) A → bC, a terminal symbol b is followed by symbol $C \in V_N$.

The grammar fulfill above restrictions (1,2,3,4) is **Regular Grammar** or **Type-3 Grammar** and the language generated by G is **Regular Language** or **Type-3 Language**. As we say earlier if grammar G (V_T, V_N, S, P) then language generated by grammar G is L (G) where,

$L(G) = \{x \in V_T^* / S \Rightarrow x\}$

From starting symbol S and by using the production/s ($\in$ P) we reaches to the string x (that is formed over set of terminal symbol/s) in finite steps.

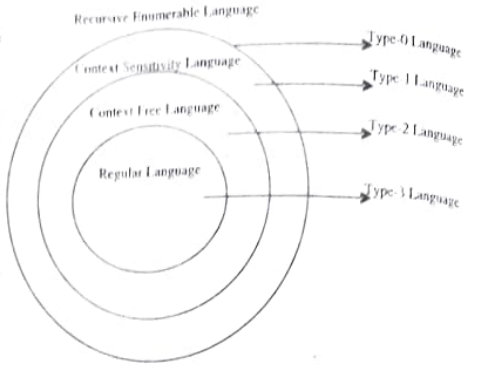


Figure 5.18

So the classification of grammars can be heirarchly arranged which is shown in **figure 5.18**. This arrangement is known as Chomesky's hierarchy. Alternatively we say that:

1) A type-3 language has the property of, type-2 language, type-1 language and type-0 language.

2) A type-2 language has the property of type-1 language and type-0 language.

A type-1 language has the property of type-0 language.

Ques 37) Prepare a table indicating the automata and grammars for the languages in the Chomsky Hierarchy. **(2019[04])**

Ans: Chomsky Hierarchy of Languages in Tabular Form

Table 5.8 shows the Chomsky's hierarchy of grammars:

Table 5.8: Chomsky's Hierarchy of Grammars

Type of Grammar	Form	Language it Defines	Corresponding automation
Type 0 or Unrestricted Grammar	$\alpha \rightarrow \beta$ No restrictions	Recursively Enumerable	Turing Machine
Type 1 or Context Sensitive Grammar	$\alpha \rightarrow \beta$ $\lvert\alpha\rvert \leq \lvert\beta\rvert$	Context Sensitive	Linear Bounded Automation
Type 2 or Context Free Grammar	$\alpha \rightarrow \beta$, $\lvert\alpha\rvert \leq \lvert\beta\rvert$, $\lvert\alpha\rvert = 1$	Context free	Pushdown Automation
Type 3 or Regular Grammar	$\alpha \rightarrow \beta$, $\alpha = \{V\}$ and $\beta = V\{T\}^*$ or $\{T\}^*V$ or T^*	Regular	Finite Automation