THEORY OF AUTOMATA COMPUTATION

Dec - 2014
Paper Code:-CSE-305-F

Note : ***The students have to attempt five questions out of 9 questions. Students have to attempt 1st compusory question and one question from each of four Sections.***

Q.1. Explain the following : **(20)**

(i) FSM

(ii) Closure Properties of Regular Sets,

(iii) PCP problem,

(iv) Unrestricted Grammar.

Ans. (i) FSM : A finite-state machine (FSM) or finite-state automaton (plural: *automata*), or simply a state machine, is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of *states*. The machine is in only one state at a time; the state it is in at any given time is called the *current state*. It can change from one state to another when initiated by a triggering event or condition; this is called a *transition*. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

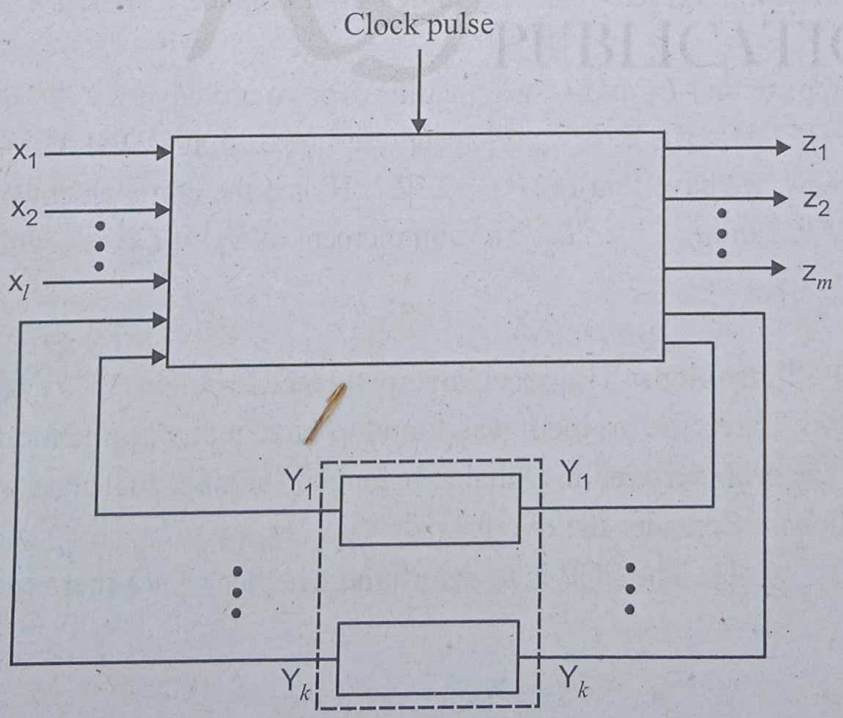


Fig. : Block diagram of a finite state model

The most general model of a sequential circuit has inputs, outputs and internal states. A sequential circuit is referred to as a finite state machine (FSM). A finite state machine is an

abstract model that describes by synchronous sequential machine. Since in a sequential circuit the output depends on the present input as well as on the past inputs, i.e. on the past histories and since a machine might have an infinite varieties of possible histories, it would need an infinite capacity for storing them. Since it is impossible to implement which have infinite storage capabilites, we consider only finite state machines. Finite state machines are sequential circuits whose past histories can affect their future behaviour in only a finite number of ways, i.e. they are machines with a fixed number of states. These machines can distinguish among a finite number of classes of input histories. These classes of input histories are referred to as the internal states of the machine. Every finite state machine therefore contains a finite number of memory devices.

Fig. shows the block diagram of a finite state model. $x_1, x_2, \ldots, x_l$ are inputs. $z_1, z_2, \ldots, z_m$ are outputs $y_1, y_2, \ldots, y_k$ are state variables, and $Y_1, Y_2, \ldots, Y_k$ represent the next state.

Ans. (ii)Closure Properties of Regular Sets :

(1) *Regular languages are closed under union opration and kleene star operation.*

Proof : If L_1 and L_2 are regular, then there are regular expressions r_1, and r_2 denoting the languages L_1 and L_2 respectively.

$(r_1 + r_2)$ and (r_1^*) are negular languages denoting the languages $L_1 \cup L_2$ and L_1^*

Therefore $L_1 \cup L_2$ and L_1^* are regular.

(2) *Regular languages are closed under complement operation and intersection operation.*

Proof : Suppose that L_1 and L_2 are regular over an alphabet Σ.

There is a *DFA* $M = |Q, \Sigma, \delta, q_0, F|$ accepting L_1. Design a *DFA* $M^1 = |Q, \Sigma, \delta, q_0, F^1|$ where $F^1 = Q/F$. Now, we have that $L(M^1) = \Sigma^* | L_1$. Hence the complement of L_1 is regular.

Let $L_1 = \Sigma^* | L_1$ and $L^1{}_2 = \Sigma^* | L_2$. The complement of $L_1{}^1 \cup L_2{}^1$ is regular and is equal to $L_1 \wedge L_2 . L_1 \wedge L_2$ $\bar{v}$ regular.

Ans. (iii) PCP problem : The post Correspondence Problem (PCP) was first introduced by Emil Post in 1946. Later, the problem was found to have many applications in the theory of formal languages. The problem over an alphabet belongs Σ belongs to a class of yes/no problems and is stated as follows : Consider the two lists $x = (x_1 \ldots x_n)$, $y = (y_1 \ldots y_n)$ of nonempty strings over an alphabet $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exist $i_1, \ldots i_m$, where $1 \leq ij \leq n$, such that

$$x_{i_j} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$$

Ans. (iv)Unrestricted Grammar : The unrestricted grammar is defined as

$$G = (V_n, V_t, P, S)$$

where V_n = a finite set of non-terminals

V_t = a finite set of terminals

S = starting non-terminal, $S \in V_n$

and P is set of productions of the following form

$$\alpha \rightarrow \beta$$

where α and β are arbitrary string of grammar symbols with $\alpha \neq \in$. These grammars are known as type-0, phase-structure or unrestricted grammars.

Section-A

Q.2.(i) What do you mean by NFA and DFA? Explain the conversion of NFA to DFA. (10)

Ans. NFA : 5 tuples $(Q, \Sigma, \delta, q_0, F)$

$\theta \rightarrow$ finite non empty set of states

$\Sigma \rightarrow$ finite non empty set input

$\delta \rightarrow$ transition function mapping $\theta \times \Sigma$ into 2^{θ}

$q_0 \rightarrow$ Initial state

$F \rightarrow$ final state.

DFA : DFA is a finite state machine that accepts/rejects finite string of symbols and only produces a unique computation of the automatan for each input string.

DFA is a 5 tuple$(Q, \Sigma, \delta, q_0, F)$

$Q \rightarrow$ finite set of states

$\Sigma \rightarrow$ finite set of inputs.

Transition function $\delta \rightarrow Q * \Sigma \rightarrow Q$

$q_0 \rightarrow$ initial state

$F \rightarrow$ final state

Conversion of NFA to DFA : Let $M_2 = \{Q_2, \Sigma\, q'_0, \delta_2, F_2\}$ be NFA that recognizes the languages L. Then the *DFA* $M = \{Q, \Sigma, q_0, \delta, F\}$ that satisfies the following conditions recognizes L:

$Q = 2^{Q2}$, that is the set of all subject of Q_2.

$q_0 = \{q'_0\}$

$\delta(q, a) = \cup_{p \in q} \delta_2(p, a)$ for each state q in Q and each symbol a in Σ and

$$F = \{q \in Q \mid q \cap F_2 \neq \phi\}$$

To obtain a *DFA* $M = \{Q, \Sigma, \delta, q_0, F\}$ which accepts the same language as given *NFA* $M_2 = \{Q_2, \Sigma, q'_0, \delta_2, F_2\}$ does, we may proceed as follows:

Step 1. Intially $Q = \phi$.

Step 2. Put $\{q'_0\}$ into Q. $\{q'_0\}$ is the initial state of the *DFA* M.

Step 3. Then for each state q in Q do the following : add this new state, add $\delta(q,a)=\cup_{p\in q}\delta_2(p,a)$ to δ, where the δ_2 on the right hand side is that of *NFA* M_2.

Step 4. Repeat step-3 till new states are there to add in Q, if there is new state found to add in Q, the process terminates. All the states of Q that contain accepting states of M_2 are accepting states of M.

Q.2.(ii) What do you mean by Regular Expression? Explain in detail. (10)

Ans. Regular Expression : Language accepted by finite automata are called regular language & these regular language can be described by algebraic expression are called regular expression. Regular expression gives a declarative way to express the strings.

Regular expression are used to generate pattern of string.

A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings called the language of expression.

The set of regular expression is defined by the following rules :

(1) Every letter of Σ can be made into a regular expression, null string, $\in$ itself is a regular expression.

(2) If r_1 and r_2 are regular expression, then

(i) (r_1) (ii) $r_1 r_2$

(iii) $r_1 + r_2$ (iv) r_1^*

(v) r_1^+ are also regular expression.

(3) Nothing else is regular expression.

Building Regular Expression : The algebra of regular expressions follows this path using constants and variables that denote languages, and operators for three operations of union, dot and star. We can describe the regular expression recursively, as follows :

In this definition, we not only describe what are the legal regular expression, but for each regular expression r, we describe the language, it represents, which we denote as $L(r)$. We can consider some facts :

(1) The constants $\in$ (null string) and ϕ (empty set) are regular expression, denoting the languages $\{\in\}$ and ϕ, respectively.

That is, $L(\in) = \{\in\}$, and $L\{\phi\} = \phi$.

(2) If a is any symbol, then a is regular expression. This expression denotes the language $\{a\}$. That is $L(a) = \{a\}$

(3) A variable, usually capitalized and italic such as L is a variable, representing any language.

The Operators of Regular Expression : Following are the operations on languages that operators of regular expression represent :

(1) The union of two languages L_1 and L_2, denoted $L_1 \cup L_2$, is the set of string that are in either L_1 or L_2, or both.

For example, if $L_1 = \{001, 10, 111\}$ and $L_2 = \{\in, 001\}$ then $L_1 \cup L_2 = \{\in, 10, 001, 111\}$.

(2) The concatenation of languages L_1 and L_2 is a set of strings that can be formed by taking any taking any string in L_1 and concatenating it with any strings in L_2. We denote concatenation of languages with a dot.

For example, if $L_1 = \{001, 10, 111\}$ and $L_2 = \{\in, 001\}$, then $L_1, L_2 = \{001, 10, 111, 001001, 10001, 111001\}$.

(3) The closure (or star, or kleene closure) of a language L is denoted by L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (that is same string can be repeated more then once) and concatenating all of them. For example, $L = \{0, 1\}$, then L^* is all strings of 0's and 1's including null string.

More formally, L^* is the infinite union $U_i \geq L^i$, where $i \geq 0$.

(4) The positive closure of a language L is denoted by L^* and represent the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions and concatenating all of them, excluding null string i.e.

$$L^+ = L^* - \in$$

More formally, L^+ is the infinite union $U_i \geq 1\ L^i$, where $i > 0$.

Q.3.(i) Explain the properties and limitations of FSM. **(10)**

Ans. The properties of FSM are as follows :

(i) Periodicity : A FSM does not have a capacity to remember large amount of information. This is because every FSM has only a finite number of states. This finite number of states determines the amount of information (or sequence of information) it can remember. Hence it will repeat some state or states again and again. In other words some sequence of states will be repeated periodically. This is a very important property of FSM.

(ii) State determination : With the help of the knowledge of the initial state of a FSM and the input sequence given to it, one can determine output sequence. So it is always possible to discover some unknown state, in which the FSM resides at a particular instance.

(iii) Impossibility of multiplication : A multiplication operation needs the model to remember two full sequences for the multiplier and the multiplicand. Also the model should be capable of remembering partial sums that are obtained at intermediate stages of multipliction. Therefore, no FSM can multiply two given arbitrarily long numbers.

(iv) Impossibility of plaindrome recognition : Due to the lack of capacity to remember, no FSM can recognize a palindrome. This is because, it can not remember all the sysbols it read until the half way point of the input sequence, in order to match them in reverse order, with the symbols in the second half of the sequence.

(v) Impossibility to check well formedness of parenthesis : Again because of the same reason cited above, no FSM has a capability to check well formedness of parenthesis.

So to solve all such complex jobs, one needs a more capable and powerful model of machine. Such model not only will need finite states but also large amount (possibly infinite) memory to remember longer sequence.

Limitation of FSM :

(i) The defining characteristics of finite automata is that they have only a finite number of states. Hence finite automata can only count a finite number of input scenario.

(ii) Finite automata do not have any auxiliary memory so it works in process & forget mode. It can store anything at particular state. There is no finite automata that recognize these string.

(a) The set of binary strings consisting of an equal no. of 1's & 0's.

(b) Th set of strings over "(" , ")" that have balanced parenthesis.

(c) Finite automata can recognize & process only regular languages.

Q.3.(ii) State and prove Arden's Methods. **(10)**

Ans. Arden's method : Let P and Q be two regular expression over Σ. If P does not contain $\wedge$, then the following equation in R namely.

$$R = Q + RP$$

has a unique solution given by $R = QP^*$

Proof :

$$Q + (QP^*)P = Q(\wedge + P^* P) = QP^*$$

$$Q + RP = Q + (Q + RP)P$$

$$= Q + QP + RP^2$$

$$= Q + QP + QP^2 + \ldots\ldots + QP^i + RP^{i+1}$$

$$= Q(\wedge + P + P^2 + \ldots\ldots + P^i) + RP^{i+1}$$

$$R = Q(\wedge + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \quad \text{For } i \geq 0.$$

Section-B

Q.4.(i) Explain the Pumping Lemma for Regular Sets. Also explain the application of Pumping Lemma. **(10)**

Ans. Pumping Lemma for Regular Sets : Let $M = (Q, \sum, \delta, q_0, F)$ be a finite automation with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz, y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.

A necessary condition for an input string to belong to a regular set. The result is called pumping lemma. It gives a method of pumping many input strings from a given string. It can be used to show that certain sets are not regular.

Application of the Pumping Lemma : The pumping lemma is extremely useful in proving that certain sets are nonregular. The general methodology in its application is an "adversary argument" of the following :

(1) Select the language L, you wish to prove non-regular.

(2) The "adversary" picks n, the constant mentioned in the pumping lemma. Once the adversary has picked n, he may not change it.

(3) Select a string z in L. Your choice may depend implicitly on the value of n chosen.

(4) The adversary breaks z into u, v and w, subject to the constants that $|uv| \leq n$ and $|v| \geq 1$.

(5) You achieve a contradiction to the pumping lemma by showing, for any u, v and w determined by the adversary that there exit i for which uv^i is not in L. It may then concluded and L is not regular. Your selection of i may depend on n, u, v and w.

Q.4.(ii) Explain in detail the Myhill Nerode theorem. **(10)**

Ans. Myhill Nerode theorem : Let $R \subseteq \Sigma^*$. The following three statements are equivalent

(i) L is regular (i.e., there exist some *DFA* that accepts L)

(ii) There exists a Myhill-Nerode relation R_M for L, such that $L = \cup[R_M]$, where $[R_M]$ is the equivalence class of R_M.

(iii) The relation R_L such that $xR_Ly \Leftrightarrow xz \in L$ exactly when $yz \in L$, for $z \in \Sigma^*$, is of finite index.

Proof : (i) $\Rightarrow$ (ii). Assume L is accepted by some *DFA* $M = (Q, \sum, \delta, q_0, F)$. Let R_M be the equivalence relation (i.e., R_M is reflexive, symmetric and transitive) defined by $xR_My \Leftrightarrow \delta(q_0, x) = \delta(q_0, y)$. As, for any z, if $\delta(q_0, x) = \delta(q_0, y)$, then $\delta(q_0, xz) = \delta(q_0, yz)$ so R_M is right invariant. Since the index (the number of elements) of R_M is atmost the number of states in Q, index of R_M is finite. Further, L is the union of those equivalence classes that includes a string x such that $\delta(q_0, x) \in F$, that is, the equivalence classes corresponding to final states.

(ii) $\Rightarrow$ (iii). We show any equivalence relation E satisfying (ii) is contained in R_L.

So that the index of $R_L \geq$ index of E, and hence the indes of R_L is finite. For that, assume xEy. By (ii), since E is right invariant, for each $z \in \sum^*, xz\ E\ yz$. Hence $yz \in L$, iff $xz \in L$. Thus xR_Ly, and hence the equivalence class of x in E is contained in the equivalence class of x in R_L. Hence each equivalence class of E is contained within some equivalence class of R_L.

(iii) $\Rightarrow$ (i). Let xR_Ly. Then for any $z \in \sum^*, xz \in L$ exactly when $yz \in L$. To prove R_L is right invariant, i.e. xwR_Lyw, let Q' be the finite set of equivalence classes of R_L. Let $[x] \in Q'$ be the equivalence class defined by $x \in \sum^*$. Define $\delta'([x], a) = [xa]$ and the definition is well-defined, since, $xR_Ly \Rightarrow xz \in L$ exactly when $yz \in L$. In particular, if $z' = az', xaz'$ is in L exactly when $yaz' \in L$, so xaR_Lya and $[xa] = [ya]$. Let $q_0' = [\varepsilon]$ and $F' = \{[x] : x \in L\}$. Then $M' = (Q', \sum, \delta', q_0', F)$ accepts L, since $\delta'(q_0', x) = [x]$ and hence $x \in L(M') \Leftrightarrow [x] \in F'$. Hence (i) follows.

Q.5.(i) Explain construction of Reduced Forms? Also explain the Removal of useless symbols. **(10)**

Ans. Contruction of Reduced Grammars :

Theorem : If G is a CFG such that $L(G) \neq \phi$, we can find an equivalent grammar G' such that each variable in G' derives some terminal string.

Proof : Let $G = (V_N, \Sigma, P, S)$. We define $G' = (V'_N, \Sigma, G', S)$ as follows :

(a) Construction of V'_N:

We define $W_i \subseteq V_N$ by recursion :

$W_1 = \{A \in V_N)$ there exists a production $A \to w$ where $w \in \Sigma^*\}$. (If $W_1 = \phi$, some variable will remain after the application of any production, and so $L(G) = \phi$.)

$W_{i+1} + W_i \{A \in V_N|$ there exists some production $A \to \alpha$ with $\alpha \in (\Sigma \cup W_i)^*\}$

By the definition of W_i, $W_i \subseteq W_{i+1}$ for all i. As V_N has only a finite number of variables, $W_k = W_{k+1}$ for some $k \leq |V_N|$. Therefore, $W_k = W_{k+j}$ for $j \geq 1$.

We define $V'_N = W_k$.

(b) Construction of P'.

$$P' = \{A \to \alpha | A, \alpha \in (V'_N \cup \Sigma)^*\}$$

We can define $G' = (V'_N, \Sigma, P', S)$. S is in V_N. (We are going to prove that every variable in V_N derives some terminal string. So if $S \in V_N$. $L(G) = \phi$.

But $L(G) \neq \phi$.

Eliminating Use Less Symbols : Here we are going to identified those symbols, which do not play any role in the derivation of any string w in $L(G)$ (these symbols are called use less symbols) and then eliminate the identified production, which contains useless symbols, from the context free grammer.

A symbols Y in a context-free garmmar is useful it and only if :

(a) $Y \Rightarrow w$, where $w \in L(G)$ and w in V_t^*, that is Y leads to a string of terminals. Here Y is said to be "generating".

(b) If there is a derivation $S \Rightarrow \alpha Y \beta \Rightarrow w$, $w \in L(G)$, for same α and β, then Y is said to be reachable.

Surely a symbol that is useful will be both generating and reachable. If we eliminate the symbols that are not generating frist, and then eliminate from the remaining grammar those symbols that are not reachable. Then, after this process, context free grammar will have only userul symbols.

There are reduction of a given grammar G, involves following steps :

(a) Identitied non-generating symbols in given CFG and eliminate those productions which contains non-generating symbols.

(b) Identified non-reachable symbols in grammar and eliminate those productions which contains non-reachable symbols.

Q.5.(ii) What do you mean by Griebach Normal Form (GNF)? (10)

Ans. Griebach Normal Form (GNF) : For every context free language L without $\in$, there exist a grammar in which every production is of the form $A \rightarrow aV$, where 'A' is a variable, 'a' is exactly one terminal and 'V' is the string of none or more variables, clearly $V \in V_n^*$.

"In other words if every prodction of the context free grammar is of the form $A \rightarrow aV/a$, then it is in Greibach Normal Form."

Griebach normal form will be used to construct a push down automata that recognize the language generated by a context free grammar.

To convert a grammar to GNF, we start with a production in which the left side has a higher numbered variable than first variable in the right side and make replacements in right side.

Section-C

Q.6. Explain in detail the Pushdown Machines. Also explain the applications of Pushdown Machines. (20)

Ans. A pushdown automata is a system, which is mathematically defined as follows :

$$P = (Q, \Sigma, \Gamma, \delta, s, F)$$

where

Q : non-empty finite set of states

Σ : non-finite set of input sysbols.

Γ : is finite set of pushdown sysbols.

s : is the initial state, $s \in Q$

F : is the set of final states and $F \subseteq Q$.

δ : it is a transition function or transition relation which maps.

$(Q \times \Sigma^* \times \Gamma) \rightarrow (Q \times \Gamma^*)$

"If it follows all the properties of non determinision then it is said to be non deterministic pushdown automata".

Clearly for the "NPDA"transition relation must behave as follows :

$(Q + (\Sigma \cup \{\in\} \times \Gamma^*) \rightarrow$ finite subset of $(Q \times \Gamma^*)$

"A pushdown automata $P = (Q, \Sigma, \Gamma, \delta, s, F)$ is said to be deterministic if it is an automation as defined in definition of PDA, subject to the restrictions that for every $q \in Q$, $a \in (\Sigma \cup \{\in\})$ and $b \in \Gamma$.

(a) $\delta(q, a, b)$ contains at most one element.

(b) If $\delta(q, \in, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$.

The first of these conditions simple requires that for any given input symbol and any stack top, at most one move can be made. The second condition is that when a $\in$-transition is possible for some configuration, no input consuming alternative is available.

Applications :

(i) Online Transaction processing system

(ii) Tower of itanoi.

(iii) CFL.

Q.7. What do you mean by Turing Machine? Explain the design of Turing Machine. Also explain the Halting Problem of Turing Machine. **(20)**

Ans. Turing Machine : A Turing machine is a very simple machine, but logically speaking, has all the power of any digital computer. A function is computable if it can be computed by a Turning machine.

Formal Definition of Turing Machine : A Turning machine is a six tuple of the form

$$TM = (Q, \Sigma, \Gamma, \delta, q_0, h)$$

where

(i) Q is that finite set of states.

(ii) Σ is the finite set of non-blank information symbols,

(iii) Γ is the set of tape symbols, including the blank symbol #.

(iv) δ is the next-move partial function from

$$(Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R, N\})$$

where $\Gamma = \Sigma \cup \#$, 'L' denotes the tape Head moves to the left adjacent cell, 'R' denote tape Head moves to the right adjacent cell and 'N' denotes Head does not move, that is continuous scanning the same state.

(v) $q_0 \in Q$, is the initial state.

(vi) $h \in Q$ is the 'Halt State', in which the machine stops any further activity.

Turning Machine is a physical computing device that can be represented as a diagram as shown in fig.

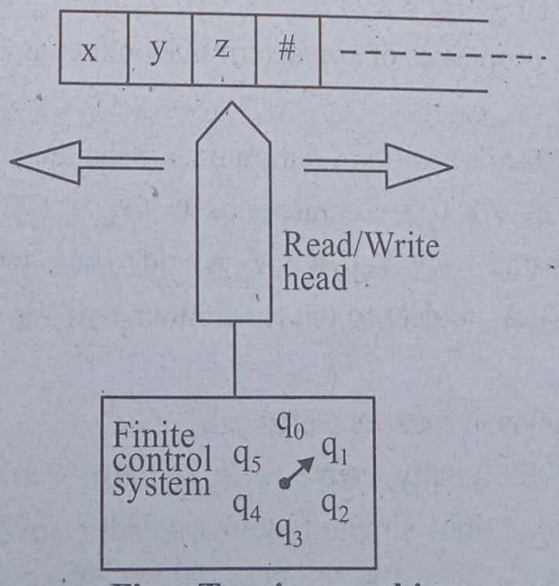


Fig. : Turning machine

According to this view of TM, it consists of :

(i) A **Tape**, with an end on the left but infinite on the right side. The tape is divided into squares or cells, with each cell capable of holding one of the tape symbols including the blank symbol #. At any time, there can be only finitely many cells of the tape that can contain non-blank symbols. The set of tape symbols is denoted by Γ.

As the very first step in the sequence of operations of a TM, the input, as a finite sequence of input symbols is placed in the left-most cells of the tape. The set of the input symbols denoted by Σ, does not contain the blank symbol #. However, during operation of a TM, a cell may contain a tape symbol which is not necessarily an input symbol.

(ii) A **finite control**, which can be in any one of the finite number of states.

The states in TM can be divided in three categories that is :

(a) The *initial state*, the state of the control just at the time when TM starts its operations. The initial state is denoted by q_0.

(b) The *Halt state*, which is the state in which TM stops all further operations. The halt sate is generally denoted by '*h*'. The halt state is distinct from the initial state. Thus a TM has atleast two states.

(iii) A **tape head** is always stationed at one of the tape cells and provides communication for interaction between the tape and the finite control. The head can read or scan the symbol in the cell under it. The symbol is communicated to the finite control. The control taking into consideration the symbol and its current state decides for further course of action including :

(a) The change of the symbol in the cell being scanned and/or

(b) Change of its state and/or

(c) Moving the head to the left or to the right. The control may decide not to move the head.

(iv) The course of action is called move of the Turing Machine.

Halting Problem of Turing Machine : This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Theorem : $HALT_{TM} = \{(M, w) |$ The Turing machine M halts on input $w\}$ is undecidable.

Proof : We assume that $HALT_{TM}$ is decidable, and get a contradiction. Let M_1 be the *TM* such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a *TM* M_2 as follows:

1. For M_2, (M, w) is an input.
2. The *TM* M_1 acts on (M, w).

3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M_1, w), simulate the *TM* M on the input string w until M halds.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w is reached. In the first case (the first alternative of setp 5) M_2 accepts (M, w). In the second case (the second alternative of setp 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $T(M_2) = \{(M, w) | \text{ The Turing machine accepts w}\}$

$= A_{TM}$.

This is a contradiction since A_{TM} is undecidable.

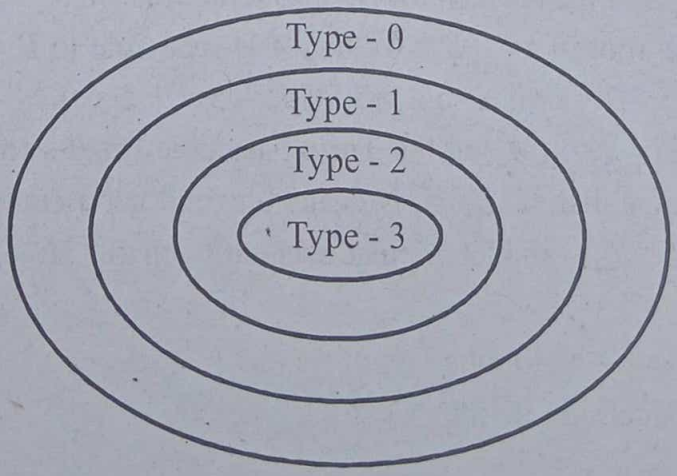
Section-D

Q.8. Explain in detail the chomsky Hierarchies of Grammers. **(20)**

Ans. Chomsky hierarchies of grammars : We can exhibit the relationship between grammars by the Chomsky hierarchy. Chomsky provided an initial classification in to four languages type :

Type – 0	(Unrestricted grammar)
Type – 1	(Context sensitive grammar)
Type – 2	(Context free grammar)
Type – 3	(Regular grammar)

Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages. Type 1 consist of the context-sensitive languages, Type 2 consists of the context-free languages and Type 3 consists of the regular languages. Each language family of type k is a proper subset of the family of type $k - 1$. Following diagram shows the original Chomsky Hierarchy.



3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M_1, w), simulate the *TM* M on the input string w until M halds.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w is reached. In the first case (the first alternative of setp 5) M_2 accepts (M, w). In the second case (the second alternative of setp 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $T(M_2) = \{(M, w) | \text{ The Turing machine accepts w}\}$

$= A_{TM}$.

This is a contradiction since A_{TM} is undecidable.

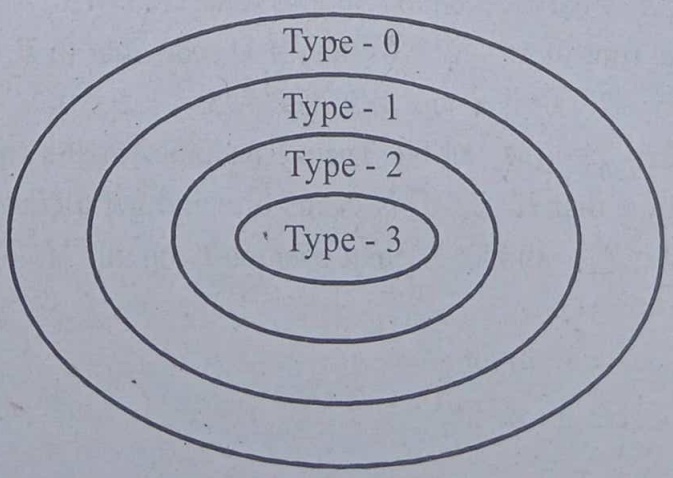
Section-D

Q.8. Explain in detail the chomsky Hierarchies of Grammers. **(20)**

Ans. Chomsky hierarchies of grammars : We can exhibit the relationship between grammars by the Chomsky hierarchy. Chomsky provided an initial classification in to four languages type :

Type – 0	(Unrestricted grammar)
Type – 1	(Context sensitive grammar)
Type – 2	(Context free grammar)
Type – 3	(Regular grammar)

Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages. Type 1 consist of the context-sensitive languages, Type 2 consists of the context-free languages and Type 3 consists of the regular languages. Each language family of type k is a proper subset of the family of type $k - 1$. Following diagram shows the original Chomsky Hierarchy.



Unrestricted (Type -0) Grammars : The unrestricted grammar is defined as

$$G = (V_n, V_t, P, S)$$

where V_n = a finite set of non-terminals

V_t = a finite set of terminals

S = starting non-terminal, $S \in V_n$

and P is set of productions of the following form

$$\alpha \rightarrow \beta$$

where α and β are arbitrary string of grammar symbols with $\alpha \neq \in$. These grammars are known as type-0, phase-structure or unrestricted grammars.

Context Sensitive Grammar : Let $G = (V_n, V_t, P, S)$ be context sentive grammar

where V_n = finite set of non-terminals

V_t = finite set of terminals

S = starting non-terminals $S \in V_n$

and P is the set of fules called productions defined as

$$\alpha \rightarrow \beta$$

where β is at least as long as a that is clearly

$$|\alpha| \leq |\beta|$$

The term "Context-sentive" comes from a normal form for these grammars, where each production is of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ with $\beta \neq \in$. Replacement of variable A by string β is permitted in the "Context" of α_1 and α_2.

Context-Free Grammars : Mathematically context-free grammar is defined as follows :

"A grammar $G = (V_n, V_t, P, S)$ is said to be context-free"

where V_n = A finite set of non-terminals, generally represented by capital letters $A, B, C, D, \ldots$.

V_t = A finite set of terminals, generally represented by small letters, like $a, b, c, d, e, f, \ldots$.

S = Starting non-terminal, called start symbol of the grammar. S belongs to V_n.

P = Set of rules or production in CFG.

G is context-free and all production in P have the form

$$\alpha \rightarrow \beta$$

where $\alpha \in V_n$ and $\beta \in (V_t \cup V_n)^*$

Regular Grammars : A regular grammar may be left linear or right linear.

"If all production of a CFG are of the form $A \rightarrow wB$ or $A \rightarrow w$, where A and B are variables and $w \in V_t^*$, then we say that grammar is right liner".

"If all the production of a CFG are of the form $A \rightarrow Bw$ or $A \rightarrow w$, we call it left linear."

A right or left linear grammar is called a regular grammar.

Q.9. Explain the following :

(i) Computability **(20)**

(ii) Primitive Recursive Functions

Ans. (i) Computability : Computability refers to the ability to solve a problem in an effective manner. It is a key topic of the field of computability theory within mathematical logic and computer science. The computability of a problem is closely linked to the existence of an algorithm to solve the problem.

A computable : In computability theory, a Turing reduction from a problem A to a problem B, named after Alan Turing, is a reduction which solves A, assuming B is already known.

Computable : Expressible as a yes/no question that can be answered in any case by a computer in finite time.

Computable Function : Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

The class of computable functions can be defined in many equivalent models, including

- Turing machine
- μ-recursive functions
- Lambda calculus
- Post machines (Post-Turing machines and tag machines).
- Register machines.

Turning Computable Function : Turing machine can be used to compute functions. We adpot the following policy to input any string to a turing mechine.

The string W is presented in to the form of # W #, that is string, W is surrounded by blanks symbols from both sides and is placed on the left most square of tape, the head of turing machine is positioned at the right most blank symbol which immediately follows the string W.

This is shown by an underscore that is we use an underscore to show the current position of machine head in the tape. A turing machine is said to halt on input 'W' if we can reach to a halting state 'h', after performing some operations, that is if,

$T_m = (Q, \Sigma, \Gamma, \delta, q_0, h)$ is a turing machine then Tm is said to halt to on input W if and only if $(q, \# W \underline{\#})$ yields to $(h, \# \mu \underline{\#})$.

Ans. (ii) Primitive Recursive Functions : function is called partial recursive if it is defined for some of its arguments. Let $f(a_1, a_2, \ldots, a_n)$ is a function and defined on function $g(b_1, b_2, \ldots\ldots, b_m)$, then f is partial function if some elements of f is assigned to almost one element of function g.

A partial function is recursive if

(i) It is an initial function over N, or

(ii) It is obtained by applying recursion or composition or minimalization on initial function over N.

Theorem : Every primitive recursive function is turning computable.

Proof : We know that primitive functions are defined using initial functions with sometimes applying composition and recursion. So if it is possible to design turing machine (Tm) for every initial functions their composition and recursive then we can say that every primitive recursive function is turing computable. Let us see following cases.

Case I : Turing machine for zero function $Z(x) = 0$, for $x \in N$. The Tm for zero function should work as follows :

$$\#\,I\,I\,I\,I\,\#\xrightarrow{Tm}\#\#$$

Here $I\,I\,I\,I$ means that x is 4. So, here we are supported to design an eraser turing mahine:

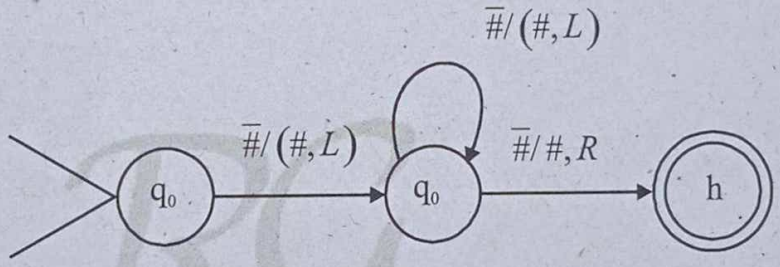


Fig. : Eraser Turing machine for zero function.

Case II : Turing Machine for successor function S(n) = n + 1, for $n \in N$. The Tm for successor function will work as follows :

$$\#\,I\,I\,I\,\#\xrightarrow{Tm}\#\,I\,I\,I\,I\,\#$$

input $\Rightarrow n = 3$ output $\Rightarrow n + 1 = 4$

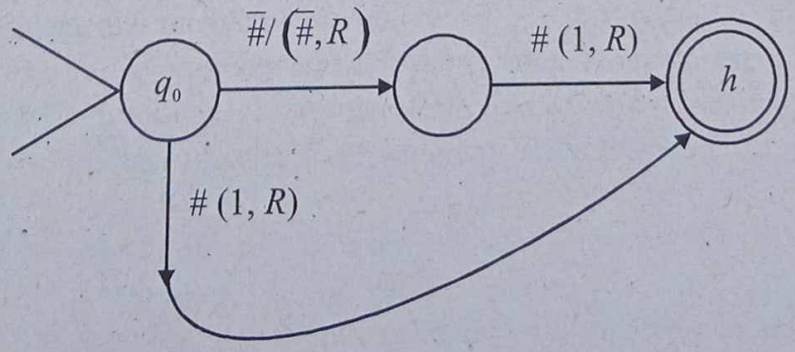


Fig. 2 : Tm for $S(n) = n + 1$

Case III : Turing machine for Projection function

$$p_1^{\,n}(x_1, x_2, \ldots x_n) = x_i \text{ for } x_1, x_2, \ldots x_n \in N.$$

So we can easily design a Turing machine Tm for projection function :

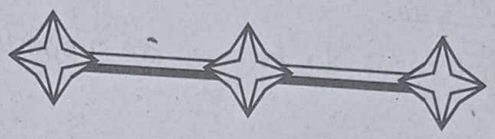
$\# x_1 \# x_2 \# .... \# x_i \# ..... \# x_n \# \xrightarrow{Tm} \# x_i \#$

Case IV : We can also design Turing machine for compostion function by using the concept of combination of Turing machine.

Case V : For recursion, we can design a Tm using multiple calls to a simple Tm designated for initial function with some fixed termination point.

Case VI : We can also combine our both cases 4 and 5 into one to design a Turing machine Tm for compsition and recursion together.

So, our theorem statement is true.



THEORY OF AUTOMATA COMPUTATION

Dec 2015

Paper Code: CSE-305-F

Note: ***Attempt five questions in all, selecting one question from each section. Q. No. 1 is compulsory.***

Q.1. Explain the following :

(i) NDFA

(ii) Removal of useless symbols

(iii) Turing machine

(iv) Computability **(20)**

Ans.(i) NDFA (Nondeterministic finite state machines) : Consider a transition diagram (Fig.).

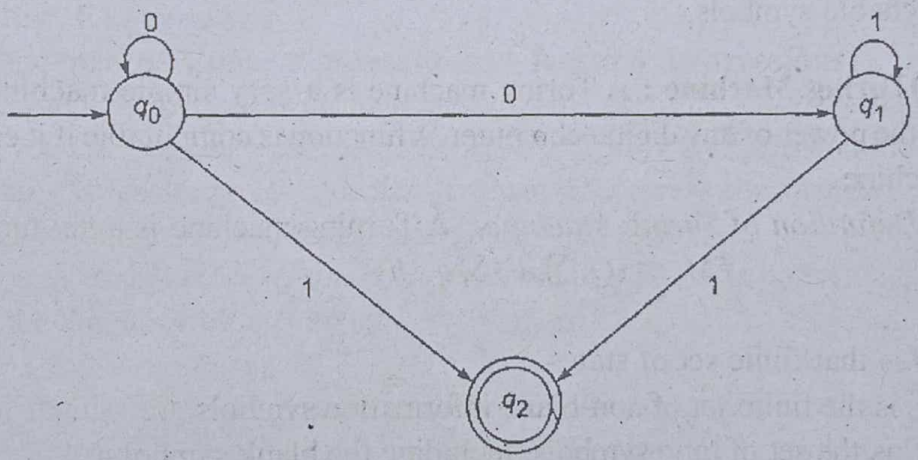


Fig. : Transition system representing nondeterministic automaton

If the automaton is in a state $\{q_0\}$ and the input symbol is 0, what will be the next state? From the figure it is clear that the next state will be either $\{q_0\}$ or $\{q_1\}$. Thus some moves of the machine cannot be determined uniquely by the input symbol and the present state. Such machines are called nonderministic automata, the formal definition of which is now given.

Definition : A nondeterministic finite automaton (NDFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(i) Q is a finite nonempty set of states;

(ii) Σ is a finite nonempty set of inputs;

(iii) δ is the transition function mapping from $Q \times \Sigma$ into 2^Q which is the power set of Q, the set of all subsets of Q;

(iv) $q_0 \in Q$ is the initial state; and

(v) $F \subseteq Q$ is the set of final states.

Ans.(ii)Removal of useless symbols: Here we are going to identified those symbols, which do not play any role in the derivation of any string w in $L(G)$ (these symbols are called use

less symbols) and then eliminate the identified production, which contains useless symbols, from the context free grammer.

A symbols Y in a context-free garmmar is useful it and only if :

(a) $Y \Rightarrow w$, where $w \in L(G)$ and w in V_t^*, that is Y leads to a string of terminals. Here Y is said to be "generating".

(b) If there is a derivation $S \Rightarrow \alpha\, Y\, \beta \Rightarrow w$, $w \in L(G)$, for same α and β, then Y is said to be reachable.

Surely a symbol that is useful will be both generating and reachable. If we eliminate the symbols that are not generating frist, and then eliminate from the remaining grammar those symbols that are not reachable. Then, after this process, context free grammar will have only userul symbols.

There are reduction of a given grammar G, involves following steps :

(a) Identitied non-generating symbols in given CFG and eliminate those productions which contains non-generating symbols.

(b) Identified non-reachable symbols in grammar and eliminate those productions which contains non-reachable symbols.

Ans.(iii)Turing Machine : A Turing machine is a very simple machine, but logically speaking, has all the power of any digital computer. A function is computable if it can be computed by a Turning machine.

Formal Definition of Turing Machine : A Turning machine is a six tuple of the form

$$TM = (Q, \Sigma, \Gamma, \delta, q_0, h)$$

where

(i) Q is that finite set of states.

(ii) Σ is the finite set of non-blank information symbols,

(iii) Γ is the set of tape symbols, including the blank symbol #.

(iv) δ is the next-move partial function from

$$(Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R, N\})$$

where $\Gamma = \Sigma \cup \#$, 'L' denotes the tape Head moves to the left adjacent cell, 'R' denote tape Head moves to the right adjacent cell and 'N' denotes Head does not move, that is continuous scanning the same state.

(v) $q_0 \in Q$, is the initial state.

(vi) $h \in Q$ is the 'Halt State', in which the machine stops any further activity.

Ans.(iv)Computability : Computability refers to the ability to solve a problem in an effective manner. It is a key topic of the field of computability theory within mathematical logic and computer science. The computability of a problem is closely linked to the existence of an algorithm to solve the problem.

A computable : In computability theory, a Turing reduction from a problem A to a problem B, named after Alan Turing, is a reduction which solves A, assuming B is already known.

Computable : Expressible as a yes/no question that can be answered in any case by a computer in finite time.

Computable Function : Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

The class of computable functions can be defined in many equivalent models, including

- Turing machine
- μ-recursive functions
- Lambda calculus
- Post machines (Post-Turing machines and tag machines).
- Register machines.

Unit – I

Q.2. Describe the following :

(i) Regular Expressions

(ii) Equivalence of Finite Automata and Regular expressions **(20)**

Ans.(i) Regular Expression : Language accepted by finite automata are called regular language & these regular language can be described by algebraic expression are called regular expression. Regular expression gives a declarative way to express the strings.

Regular expression are used to generate pattern of string.

A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings called the language of expression.

The set of regular expression is defined by the following rules :

(1) Every letter of Σ can be made into a regular expression, null string, $\in$ itself is a regular expression.

(2) If r_1 and r_2 are regular expression, then

(i) (r_1) (ii) $r_1 r_2$

(iii) $r_1 + r_2$ (iv) r_1^*

(v) r_1^+ are also regular expression.

(3) Nothing else is regular expression.

Building Regular Expression : The algebra of regular expressions follows this path using constants and variables that denote languages, and operators for three operations of union, dot and star. We can describe the regular expression recursively, as follows :

In this definition, we not only describe what are the legal regular expression, but for each regular expression r, we describe the language, it represents, which we denote as $L(r)$. We can consider some facts :

(1) The constants $\in$ (null string) and ϕ (empty set) are regular expression, denoting the languages $\{\in\}$ and ϕ, respectively.

That is, $L(\in) = \{\in\}$, and $L\{\phi\} = \phi$.

(2) If a is any symbol, then a is regular expression. This expression denotes the language $\{a\}$. That is $L(a) = \{a\}$

(3) A variable, usually capitalized and italic such as L is a variable, representing any language.

The Operators of Regular Expression : Following are the operations on languages that operators of regular expression represent :

(1) The union of two languages L_1 and L_2, denoted $L_1 \cup L_2$, is the set of string that are in either L_1 or L_2, or both.

For example, if $L_1 = \{001, 10, 111\}$ and $L_2 = \{\in, 001\}$ then $L_1 \cup L_2 = \{\in, 10, 001, 111\}$.

(2) The concatenation of languages L_1 and L_2 is a set of strings that can be formed by taking any taking any string in L_1 and concatenating it with any strings in L_2. We denote concatenation of languages with a dot.

For example, if $L_1 = \{001, 10, 111\}$ and $L_2 = \{\in, 001\}$, then $L_1, L_2 = \{001, 10, 111, 001001, 10001, 111001\}$.

(3) The closure (or star, or kleene closure) of a language L is denoted by L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (that is same string can be repeated more then once) and concatenating all of them. For example, $L = \{0, 1\}$, then L^* is all strings of 0's and 1's including null string.

More formally, L^* is the infinite union $U_i \geq L^i$, where $i \geq 0$.

(4) The positive closure of a language L is denoted by L^* and represent the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions and concatenating all of them, excluding null string i.e.

$$L^+ = L^* - \in$$

More formally, L^+ is the infinite union $U_i \geq 1\ L^i$, where $i > 0$.

Ans.(ii) Equivalence of Regular expressions and finite automata : Regular expressions and finite automata have equivalent expressive power :

(i) For every regular expression r, there is a corresponding FA that accepts the set of strings generated by r.

(ii) For every FA, M there is a corresponding regular expression that generates the set of strings accepted by M.

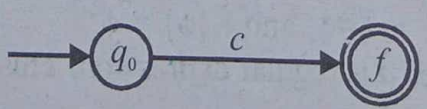
1. An algorithm that, given a regular expression r, produces an FA, M such that L(M) = = L(r).

2. An algorithm that, given an FA, M produces a regular expression, such that L(r) = = L(M).

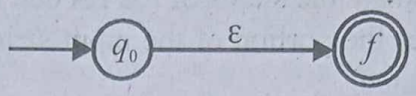
Our construction of FA from regular expressions will allow "epsilon transitions" (a transition from one state to another with epsilon (ε) as the label). Such a transition is always possible, since epsilon (or the empty string) can be said to exist between any two input symbols. We can shown that such epsilon transitions are a notational convenience; for every FA with epsilon transitions there is a corresponding FA without them.

Constructing an FA from an Regular Expression : We begin by showing how to construct an FA for the operands in a regular expression.

(i) If the operand is a character c, then our FA has two states, q_0 (the start state) and f (the final, appcpting state), and a transition form q_0 to f with label c.



(ii) If the operand is epsilon, then our FA has two states, q_0 (the start state) and f (the final, accepting state), and an transition from q_0 to f.

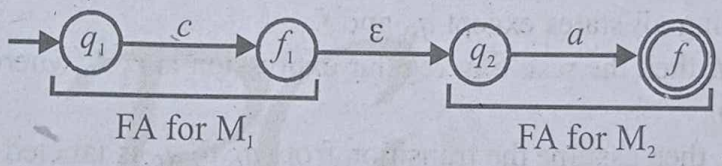


(iii) If the operand is null, then our FA has two states, q_0 (the start state) and f (the final, accepting state), and no transitions.

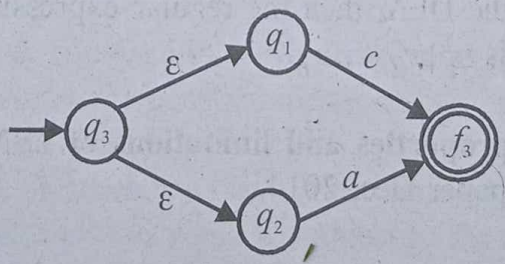
Given FA for r_1 and r_2, we now show how to build an FA for r_1r_2, $r_1|r_2$, and r_1*.

Let M (with start state q_1 and final state f_1) be the machine accepting $L(r_1)$ and B (with start state q_2 and final state f_2) bethe machine accepting $L(r_2)$.

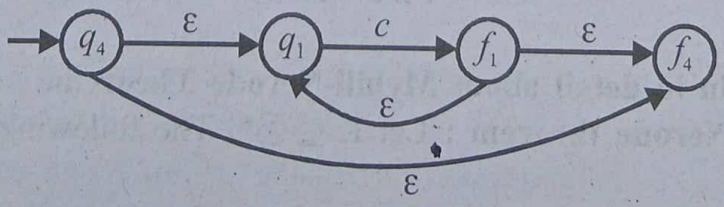
(i) The machine M_3 accepting $L(r_1r_2)$ includes M_1 and M_2, with start state q_1, final state f_2, and an epsilon transition from f_1 to q_2.



(ii) The machine M_3 accepting $L(r_1|r_2)$ includes M_1 and M_2, with a new start state q_3, a new final state f_3, and epsilon transitions from q_3 to q_1 and q_2, and from f_1 and f_2 to f_3.



(iii) The machine M_3 accepting $L(r_1*)$ includes M, with a new start state q_4, a new final state f_4, and epsilon transitions from q_4 to q_1 and f_4, and from f_1 to q_1, and from f_1 to f_4.



Constructing an Regular Expression from an Finite Automata : To construct a regular expression from a DFA (and thereby complete the proof that regular expressions and finite automata have the same expressive power), we replace each state in the DFA one by one with a corresponding regular expression.

Just as we built a small FA for each operator and operand in a regular expression, we will now build a small regular expression for each state in the DFA.

The basic idea is to eliminate the states of the FA one by one, replacing each state with a regulr expression that generates the portion of the input string that labels the transitions into and out of the state being eliminated.

Algorithm for Constructing an RE from an FA :

Given DFA F we construct a regular expression R such that L(M) = = L(r).

We preprocess the FA, turning the labels on transitions into regular expression.

If there is a transition with label {a, b}, then we replace the label with the regular expression $a \mid b$. If there is no transition from a state to itself, we can add one with the label NULL.

For each accepting state f in M, eliminate all states in M except the start state q_0 and f.

To eliminate a state q_1, consider all pairs of states q_2 and q_3 such that there is a transition from q_2 to q_1 with label r_1, a transition from q_1 to q_1 with label r_2 (possibly null, meaning no transition), and a transition from q_1 to q_3 with label r_3. Introduce a transition from q_2 to q_3 with label $r_1 r_2{}^* r_3$. If there is already a transition from q_2 to q_3 with label r_4, then replace that label with $r_4 | r_1\, r_2{}^* r_3$.

After eliminating all states except q_0 and f :

(i) If $q_0 = = f$, then the resulting regular expression is r_1 *, where r_1 is the label on the transition from q_0 to q_0.

(ii) If q_0 ! $= f$, then assume the transition from q_0 to q_0 is labeled r_1, the transition from q_0 to f is labeled r_2, the transition from f to f is labeled r_3, and the transition from f to q_0 is labeled r_4. The resulting expression is $r_1{}^* r_2\ (r_3 \mid r_4 r_1 \mid {}^* r_2)^*$

Let r_i be the regular expression produced by eliminating all the states except q_0 and f_i. If there are n final states in the DFA, then the regular expression that generates the strings accepted by the original DFA is $r_1 \mid r_2 \mid \ldots r_n$.

Q.3.(i) Explain the properties and limitations of FSM. **(10)**

Ans. Refer Q.3(i) of paper Dec. 2014.

Q.3.(ii) State and prove Arden's Method. **(10)**

Ans. Refer Q.3(ii) of paper Dec. 2014.

Unit – II

Q.4.(i) Explain in detail about Myhill-Nerode Theorem. **(10)**

Ans. Myhill Nerode theorem : Let R $\subseteq \Sigma^*$. The following three statements are equivalent

(i) L is regular (i.e., there exist some *DFA* that accepts L)

(ii) There exists a Myhill-Nerode relation R_M for L, such that $L = \cup[R_M]$, where $[R_M]$ is the equivalence class of R_M.

(iii) The relation R_L such that $xR_L y \Leftrightarrow xz \in L$ exactly when $yz \in L$, for $z \in \Sigma^*$, is of finite index.

Proof : (i) $\Rightarrow$ (ii). Assume L is accepted by some DFA $M=(Q,\sum,\delta,q_0,F)$. Let R_M be the equivalence relation (i.e., R_M is reflexive, symmetric and transitive) defined by $xR_My \Leftrightarrow \delta(q_0,x)=\delta(q_0,y)$. As, for any z, if $\delta(q_0,x)=\delta(q_0,y)$, then $\delta(q_0,xz)=\delta(q_0,yz)$ so R_M is right invariant. Since the index (the number of elements) of R_M is atmost the number of states in Q, index of R_M is finite. Further, L is the union of those equivalence classes that includes a string x such that $\delta(q_0,x)\in F$, that is, the equivalence classes corresponding to final states.

(ii) $\Rightarrow$ (iii). We show any equivalence relation E satisfying (ii) is contained in R_L.

So that the index of $R_L \geq$ index of E, and hence the indes of R_L is finite. For that, assume xEy. By (ii), since E is right invariant, for each $z\in\sum^*, xz\ E\ yz$. Hence $yz\in L$, iff $xz\in L$. Thus xR_Ly, and hence the equivalence class of x in E is contained in the equivalence class of x in R_L. Hence each equivalence class of E is contained within some equivalence class of R_L.

(iii) $\Rightarrow$ (i). Let xR_Ly. Then for any $z\in\sum^*, xz\in L$ exactly when $yz\in L$. To prove R_L is right invariant, i.e. xwR_Lyw, let Q' be the finite set of equivalence classes of R_L. Let $[x]\in Q'$ be the equivalence class defined by $x\in\sum^*$. Define $\delta'([x],a)=[xa]$ and the definition is well-defined, since, $xR_Ly \Rightarrow xz\in L$ exactly when $yz\in L$. In particular, if $z'=az', xaz'$ is in L exactly when $yaz'\in L$, so xaR_Lya and $[xa]=[ya]$. Let $q_0'=[\varepsilon]$ and $F'=\{[x]:x\in L\}$. Then $M'=(Q',\sum,\delta',q_0',F)$ accepts L, since $\delta'(q_0',x)=[x]$ and hence $x\in L(M')\Leftrightarrow[x]\in F'$. Hence (i) follows.

Q.4.(ii) Describe the Applications of the Pumping Lemna. **(10)**

Ans. The pumping lemma is extremely useful in proving that certain sets are nonregular. The general methodology in its application is an "adversary argument" of the following :

(1) Select the language L, you wish to prove non-regular.

(2) The "adversary" picks n, the constant mentioned in the pumping lemma. Once the adversary has picked n, he may not change it.

(3) Select a string z in L. Your choice may depend implicitly on the value of n chosen.

(4) The adversary breaks z into u, v and w, subject to the constants that $|uv|\leq n$ and $|v|\geq 1$.

(5) You achieve a contradiction to the pumping lemma by showing, for any u, v and w determined by the adversary that there exit i for which uv^i is not in L. It may then concluded and L is not regular. Your selection of i may depend on n, u, v and w.

Q.5. Explain the following :

(i) Ambiguity Regular Grammar

(ii) Context Free and Context sensitive Grammar. **(20)**

Ans. (i) Ambiguity Regular Grammars : In computer science, an ambiguous grammar is a context-free grammar for which there exists a string that can have more than one leftmost derivation, while an unambiguous grammar is a context-free grammar for which every valid string has a unique leftmost derivation. Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars. Any non-empty language

admits an ambiguous grammar by taking an unambiguous grammar and introducing a duplicate rule or synonym (the only language without ambiguous grammars is the empty language). A language that only admits ambiguous grammars is called an inherently ambiguous language, and there are inherently ambiguous context-free languages. Deterministic context-free grammars are always unambiguous, and are an important subclass of unambiguous CFGs; there are non-deterministic unambiguous CFGs, however.

Let G = (V, P) be a rewriting system and T a proper subset of V. Further, let N denote the set difference V\T and let S be an element of N. We say that G is a *right-linear grammer* with nonterminal N, terminal alphabet T and start symbol S, denoted by

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

if each rule in P has one of the forms

(rl) $A \to x$, $A \to xB$.

where A and B are nonterminals, i.e., elements of N, and x is a terminal string, i.e., an element of T*.

A *left-linear grammer* is defined similarly, except that each rule has one of the forms

(ll) $A \to x$, $A \to Bx$.

A rewriting system is a regular grammar if it is either a right-linear grammar or a left-linear grammar. In a regular grammar, then, either all rules are of the form (rl) or all rules are of the form (ll).

The language generated (or described) by a regular grammar G = (V, T, P, S), denoted by L(G), is the set of terminal strings by the start symbol. In other words,

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w \text{ in } G\}$$

Regular grammars differ from general context-free grammars in that the right-hand of rules can contain only one nonterminal and that this nontermial, when present, must be located either at the extreme right and of the right-hand side in all rules (right-linear grammars) or at the expreme end of the right-hand side in all rules (left-linear grammars). The grammar G_{match}, for example, is not a regular grammar. The presence of the "nonregular" rule $S \to OS1$ has the effect that the grammar generates a nonregular language, L_{match}.

A regular grammar G is *ambiguous* if some sentence in L(G) has two distinct derivations in G from the start symbol S; otherwise G is unambiguous.

Ans. (ii) Type – 1 grammars (*context-sensitive grammars*) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \to \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be non-empty. The rule $S \to \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input).

Type – 2 grammars (*context-free grammars*) generate the context-free languages. These are defined by rules of the form $A \to \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

Unit – III

Q.6. What is push down Machines. Explain in detail about Application of Push down machines. **(20)**

Ans. Refer Q.6 of paper Dec. 2014.

Q.7. What is Turing machine? Explain the design and Halting problem of T.M. **(20)**

Ans. Turing Machine : A Turing machine is a very simple machine, but logically speaking, has all the power of any digital computer. A function is computable if it can be computed by a Turning machine.

Formal Definition of Turing Machine : A Turning machine is a six tuple of the form

$$TM = (Q, \Sigma, \Gamma, \delta, q_0, h)$$

where

(i) Q is that finite set of states.

(ii) Σ is the finite set of non-blank information symbols,

(iii) Γ is the set of tape symbols, including the blank symbol #.

(iv) δ is the next-move partial function from

$$(Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R, N\})$$

where $\Gamma = \Sigma \cup \#$, 'L' denotes the tape Head moves to the left adjacent cell, 'R' denote tape Head moves to the right adjacent cell and 'N' denotes Head does not move, that is continuous scanning the same state.

(v) $q_0 \in Q$, is the initial state.

(vi) $h \in Q$ is the 'Halt State', in which the machine stops any further activity.

Turning Machine is a physical computing device that can be represented as a diagram as shown in fig.

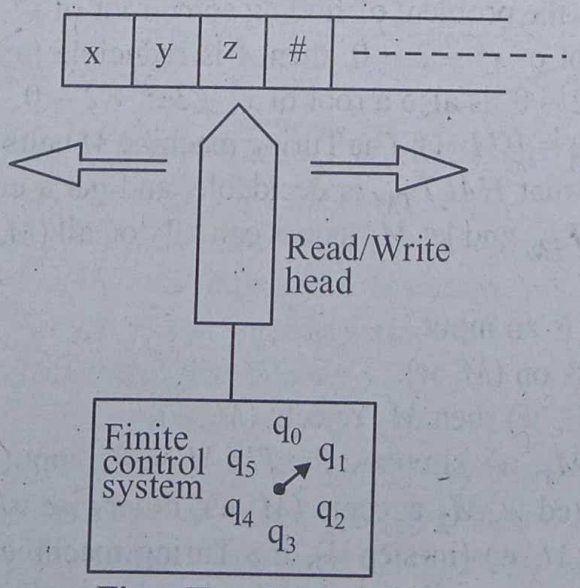


Fig. : Turning machine

According to this view of TM, it consists of :

(i) A **Tape**, with an end on the left but infinite on the right side. The tape is divided into squares or cells, with each cell capable of holding one of the tape symbols including the blank symbol #. At any time, there can be only finitely many cells of the tape that can contain non-blank symbols. The set of tape symbols is denoted by Γ.

As the very first step in the sequence of operations of a TM, the input, as a finite sequence of input symbols is placed in the left-most cells of the tape. The set of the input symbols denoted by Σ, does not contain the blank symbol #. However, during operation of a TM, a cell may contain a tape symbol which is not necessarily an input symbol.

(ii) A **finite control**, which can be in any one of the finite number of states.

The states in TM can be divided in three categories that is :

(a) The *initial state*, the state of the control just at the time when TM starts its operations. The initial state is denoted by q_0.

(b) The *Halt state*, which is the state in which TM stops all further operations. The halt sate is generally denoted by 'h'. The halt state is distinct from the initial state. Thus a TM has atleast two states.

(iii) A **tape head** is always stationed at one of the tape cells and provides communication for interaction between the tape and the finite control. The head can read or scan the symbol in the cell under it. The symbol is communicated to the finite control. The control taking into consideration the symbol and its current state decides for further course of action including :

(a) The change of the symbol in the cell being scanned and/or

(b) Change of its state and/or

(c) Moving the head to the left or to the right. The control may decide not to move the head.

(iv) The course of action is called move of the Turing Machine.

Halting Problem of Turing Machine : This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Theorem : $HALT_{TM} = \{(M, w) |$ The Turing machine M halts on input $w\}$ is undecidable.

Proof : We assume that $HALT_{TM}$ is decidable, and get a contradiction. Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a TM M_2 as follows:

1. For M_2, (M, w) is an input.
2. The TM M_1 acts on (M, w).
3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M_1, w), simulate the TM M on the input string w until M halds.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w is

reached. In the first case (the first alternative of setp 5) M_2 accepts (M, w). In the second case (the second alternative of setp 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $T(M_2) = \{(M, w) |$ The Turing machine accepts w$\}$

$= A_{TM}$

This is a contradiction since A_{TM} is undecidable.

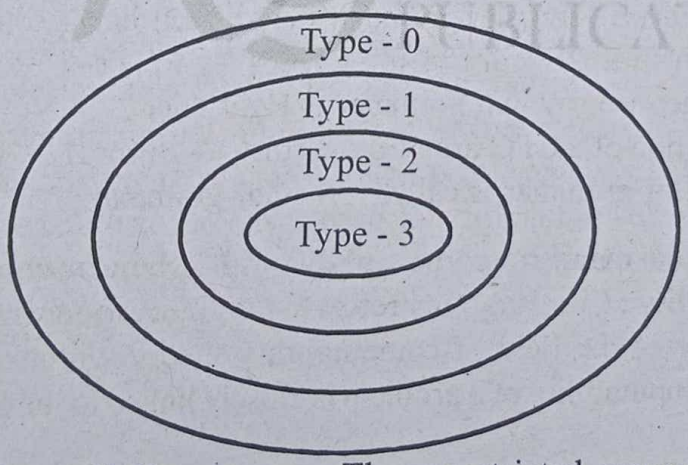
Unit – IV

Q.8. Explain in detail about Chomsky Hierarchies of Lemmas in detail. (20)

Ans. Chomsky hierarchies of grammars : We can exhibit the relationship between grammars by the Chomsky hierarchy. Chomsky provided an initial classification in to four languages type :

Type – 0	(Unrestricted grammar)
Type – 1	(Context sensitive grammar)
Type – 2	(Context free grammar)
Type – 3	(Regular grammar)

Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages. Type 1 consist of the context-sensitive languages, Type 2 consists of the context-free languages and Type 3 consists of the regular languages. Each language family of type k is a proper subset of the family of type $k - 1$. Following diagram shows the original Chomsky Hierarchy.



Unrestricted (Type -0) Grammars : The unrestricted grammar is defined as

$$G = (V_n, V_t, P, S)$$

where V_n = a finite set of non-terminals

V_t = a finite set of terminals

S = starting non-terminal, $S \in V_n$

and P is set of productions of the following form

$$\alpha \rightarrow \beta$$

where α and β are arbitrary string of grammar symbols with $\alpha \neq \in$. These grammars are known as type-0, phase-structure or unrestricted grammars.

Context Sensitive Grammar : Let $G = (V_n, V_t, P, S)$ be context sentive grammar

where V_n = finite set of non-terminals

V_t = finite set of terminals

S = starting non-terminals $S \in V_n$

and P is the set of fules called productions defined as

$$\alpha \rightarrow \beta$$

where β is at least as long as a that is clearly

$$|\alpha| \leq |\beta|$$

The term "Context-sentive" comes from a normal form for these grammars, where each production is of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ with $\beta \neq \in$. Replacement of variable A by string β is permitted in the "Context" of α_1 and α_2.

Context-Free Grammars : Mathematically context-free grammar is defined as follows :

"A grammar $G = (V_n, V_t, P, S)$ is said to be context-free"

where

V_n = A finite set of non-terminals, generally represented by capital letters $A, B, C, D, ...$

V_t = A finite set of terminals, generally represented by small letters, like $a, b, c, d, e, f, ...$

S = Starting non-terminal, called start symbol of the grammar. S belongs to V_n.

P = Set of rules or production in CFG.

G is context-free and all production in P have the form

$$\alpha \rightarrow \beta$$

where

$$\alpha \in V_n \text{ and } \beta \in (V_t \cup V_n)^*$$

Regular Grammars : A regular grammar may be left linear or right linear.

"If all production of a CFG are of the form $A \rightarrow wB$ or $A \rightarrow w$, where A and B are variables and $w \in V_t^*$, then we say that grammar is right liner".

"If all the production of a CFG are of the form $A \rightarrow Bw$ or $A \rightarrow w$, we call it left linear."

A right or left linear grammar is called a regular grammar.

Q.9. What do you mean by computability ? Explain in detail. **(20)**

Ans. Computability : Computability refers to the ability to solve a problem in an effective manner. It is a key topic of the field of computability theory within mathematical logic and computer science. The computability of a problem is closely linked to the existence of an algorithm to solve the problem.

A computable : In computability theory, a Turing reduction from a problem A to a problem B, named after Alan Turing, is a reduction which solves A, assuming B is already known.

Computable : Expressible as a yes/no question that can be answered in any case by a computer in finite time.

Computable Function : Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

The class of computable functions can be defined in many equivalent models, including

- Turing machine
- μ-recursive functions
- Lambda calculus
- Post machines (Post-Turing machines and tag machines).
- Register machines.

Turning Computable Function : Turing machine can be used to compute functions. We adpot the following policy to input any string to a turing mechine.

The string W is presented in to the form of # W #, that is string, W is surrounded by blanks symbols from both sides and is placed on the left most square of tape, the head of turing machine is positioned at the right most blank symbol which immediately follows the string W.

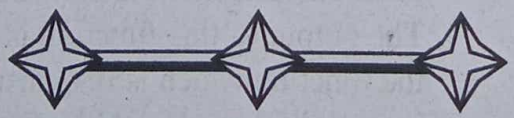
This is shown by an underscore that is we use an underscore to show the current position of machine head in the tape. A turing machine is said to halt on input 'W' if we can reach to a halting state 'h', after performing some operations, that is if,

$T_m = (Q, \Sigma, \Gamma, \delta, q_0, h)$ is a turing machine then Tm is said to halt to on input W if and only if $(q, \# \underline{W} \#)$ yields to $(h, \# \underline{\mu} \#)$.

Multiple track turing machine : In the case of TM, a single tape was used. In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. Hence the only difference between the standard TM and the TM with multiple tracks is the set of tape symbols. In the case of the standard Turning machine, tape symbols are elements of Γ; in the case of TM with multiple track, it is Γ^k. The moves are defined in a similar way.

Subroutines : Subroutines are used in computer languages, when some task has to be repeatedly. We can implement this facility for TMs as well.

First, a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt. For using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of TM.



THEORY OF AUTOMATA COMPUTATION

Dec - 2016
Paper Code:-CSE-305-F

Note : ***Attempt five questions in all, selecting one question from each Section. Question No. 1 is compulsory. All questions carry equal marks.***

Q.1.(a) Explain at least four differences between DFA and NFA. **(4)**

Ans. Differences between DFA and NFA are as follows :

(1) "DFA" stands for "Deterministic Finite Automata", while "NFA" stands for "Nondeterministic Finite Automata."

(2) Both are transition functions of Automata. In DFA the next possible state is distinctly set, while in NFA each pair of state and input symbol can have many possible next states.

(3) NFA can use empty string transition, while DFA cannot use empty string transition.

(4) NFA is easier to construct, while it is more difficult to construct DFA.

(5) Backtracking is allowed in DFA, while in NFA it may or may not be allowed.

(6) DFA requires more space, while NFA requires less space.

(7) While DFA can be understood as one machine and a DFA machine can be constructed for every input and output, But NFA can be understood as several little machines that compute together, and there is no possibility of constructing an NFA machine for every input and output.

Q.1.(b) Explain Moore machine with the help of transition table and also draw transition diagram of the given transition table. **(4)**

Ans. Moore machine : It is the FA in which output is associated with each state. In moore machine every state of this finite machine has a fix output.

Mathematically moore machine is a six-tuple machine and defined as

$$M_0 = (Q, \Sigma, \Delta, \delta, \lambda', q_0)$$

M_0 is the six-tuple moore machine in which

- Q : A nonempty finite set of state of M_0.
- Σ : A nonempty finite set of input symbols.
- Δ : A nonempty finite set of outputs.
- δ : It is transition function which takes two arguments as in finite automata, one is input state and another input symbol. The output of this function is a single state, so clearly δ is the function which is responsible for the transitions in M_0.
- q_0 : is the initial state of M_0 and $q_0 \in Q$.
- λ : is a mapping function which maps Q to Δ, giving the output associated with each state.

Let M_0 be a moore machine and Σ is $a_1, a_2, a_3, \ldots a_n$, $n \geq 0$ then output of M_0 for Σ is $\lambda'(q_0), \lambda'(q_1), \lambda'(q_2) \ldots \lambda'(q_n)$, where $q_0, q_1, q_2, \ldots q_n$ is the sequence of states such that

$$\delta(q_{i-1}, a_i) = q_i \text{ for } 1 \leq i \leq n$$

It is to be noticed that moore machine gives output $\lambda'(q_0)$ for input $\in$, i.e. in case of moore machine length of output sequence is $n + 1$.

Representation of Moore Machine : Moore machine can be represented by transition table as well as transition diagram same as finite automata.

Let M_0 be moore machine having following transition table $\Sigma = \{0, 1\}$

Present State	Next State		Output
	a = 0	a = 1	
$\rightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

$$M_0 = (Q, \Sigma, \Delta, \delta, \lambda', q_0)$$
$$Q = \{q_0, q_1, q_2, q_3\}$$
$$\Sigma = \{0, 1\}$$
$$\Delta = \{0, 1\}$$

$\lambda'(q_0) = 0$, $\lambda'(q_1) = 1$, $\lambda'(q_2) = 0$ and $\lambda'(q_3) = 0$

q_0 is the initial state. There is no concept of final state in moore machine since we are considering output for the each corresponding state.

We can also viewed it by transition diagram as

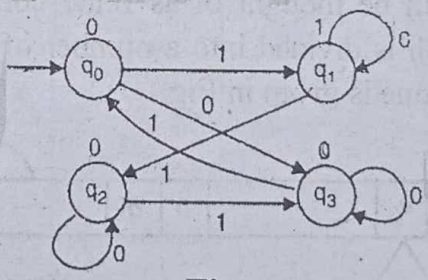


Fig.

By the analysing moore machine transition diagram of Fig.(1). We can easily noticed that it is FA without any final state and output is there for every corresponding state. Output for every is written just above the corresponding state.

Q.1.(c) Briefly explain any two types of normal forms in CFG. **(4)**

Ans. There are two types of normal forms :

(1) Chomsky normal form (CNF)

(2) Greibach normal form (GNF)

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

Definition : A context-free grammer G is in Chomsky normal form if every production is of the form $A \rightarrow a$, or $A \rightarrow BC$, $S \rightarrow \Lambda$ is in G if $A \in L(G)$. 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 productions are $S \rightarrow AB \mid \Lambda$, $A \rightarrow a$, $B \rightarrow b$. Then G is in Chomsky normal form.

Greibach normal form : Greibach normal form (GNF) is another normal form quite useful in some proofs and constructions. A context-free grammer generating the set accepted by a pushdown automation is in Greiback normal form.

Definition : A context-free grammer is in Greibach normal form if every production is of the form $A \to a\alpha$, where $\alpha \in V_N^*$ and $a \in \Sigma(\alpha$ may be $\Lambda)$, and $S \to \Lambda$ 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 $S \to aAB \mid \Lambda$, $A \to bC$, $B \to b$, $C \to c$ is in GNF.

Q.1.(d) Define Turing machine mathematically and also explain its basic structure. **(4)**

Ans. A Turning machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where

(1) Q is a finite nonempty set of states.

(2) Γ is a finite nonempty set of tape symbols.

(3) $b \in \Gamma$ is the blank.

(4) Σ is a nonempty set of input symbols and is a subset of Γ and $b \notin \Sigma$.

(5) δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head; $D = L$ or R according as the movement is to the left or right.

(6) $q_0 \in Q$ is the initial state, and

(7) $F \subseteq Q$ is the set of final states.

The Turning machine can be thought of as finite control connected to a R/W (read/write) head. It has one tape which is divided into a number of cells. The block diagram of the basic model for the Turning machine is given in Fig.

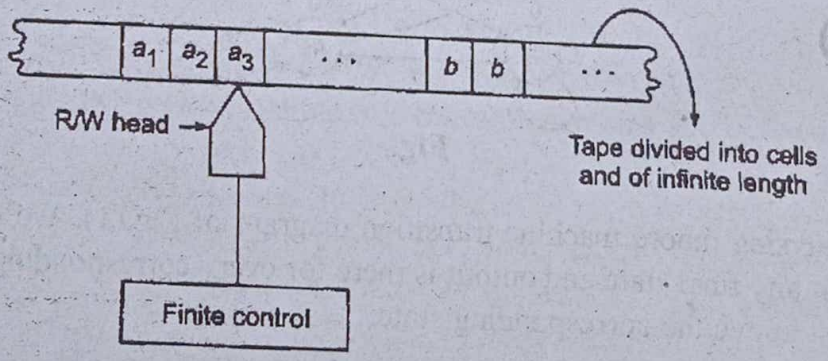


Fig. : Turning machine model

Each cell can store only one symbol. The input to and the output from the finite state automaton are effected by the R/W head which can examine one cell at a time. In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine

(i) a new symbol to be written on the tape in the cell under the R/W head,

(ii) a motion of the R/W head along the tape: either the head moves one cell left (L), or one cell right (R),

(iii) the next state of the automaton, and

(iv) whether to halt or not.

Q.1.(e) What are UNIT productions in CFG ? **(4)**

Ans. The unit productions are the productions in which one non-terminal gives another non-terminal.

For example if

$$X \rightarrow Y$$
$$Y \rightarrow Z \qquad Z \rightarrow X$$

then X, Y and Z are unit productions. To optimize the grammer we need to remove the unit productions.

If $A \Rightarrow B$ is a unit production and $B \rightarrow X_1 X_2 X_3 \ldots X_n$ then while removing $A \rightarrow B$ production we should add a rule $A \rightarrow X_1 X_2 X_3 \ldots X_n$.

SECTION – A

Q.2.(a) M = $(\{q_1, q_2, q_3\}, (0, 1), \delta, \{q_1\}, \{q_3\}$ is a NFA, where δ is given by :

$\delta(q_1, 0) = \{q_2, q_3\}$, $\delta(q_1, 1) = \{q_1\}$

$\delta(q_2, 0) = \{q_1, q_2\}$, $\delta(q_2, 1) = \{\phi\}$

$\delta(q_3, 0) = \{q_2\}$, $\delta(q_3, 1) = \{q_1, q_2\}$

Construct an equivalent DFA. **(8)**

Ans.

States/Σ	0	1
q_1	q_2, q_3	q_1
q_1	q_1, q_2	ϕ
q_3	q_1	q_1, q_2

$$\{(q_2, q_3), 0\} = \{(q_2, 0) \cup (q_3, 0\} = \{q_1, q_2 \cup q_2\} = q_1, q_2$$

$$\{(q_2, q_3), 1\} = \{(q_2, 1)\, u\, (q_3, 1)\} = \{\phi \cup q_1, q_2\} = q_1, q_2$$

$$\{(q_1, q_2), 0\} = \{(q_1, 0) \cup (q_2, 0)\} = \{q_2, q_3\, u\, q_1, q_2\} = q_1, q_2, q_3$$

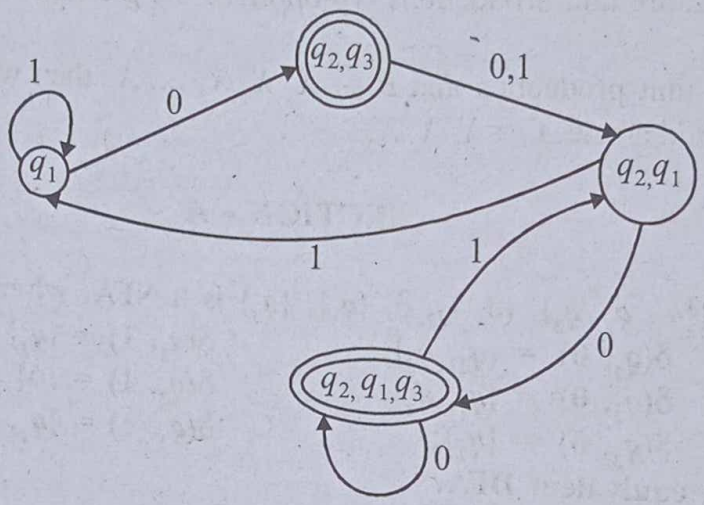
$$\{(q_1, q_2), 1\} = \{(q_1, 1) \cup (q_4, 1)\} = \{q_1 \cup \phi\} = q_1$$

$$\{(q_1, q_2, q_3), 0\} = \{(q_1, 0) \cup (q_2, 0) \cup (q_3, 0)\} = \{q_2, q_3 \cup q_1\, q_2 \cup q_2\} = q_1, q_2, q_3$$

$$\{(q_1, q_2, q_3), 1)\} = \{(q_1, 1) \cup (q_2, 1) \cup (q_3, 1) = q_1 \cup \phi \cup q_1, q_2 = q_1, q_2$$

States/ Σ	0	1
q_1	q_2, q_3	q_1
q_2, q_3	q_1, q_2	q_1, q_2
q_1, q_2	q_1, q_2, q_3	q_1
q_1, q_2, q_3	q_1, q_2, q_3	q_1, q_2

Hence : The DFA for above NDFA is :



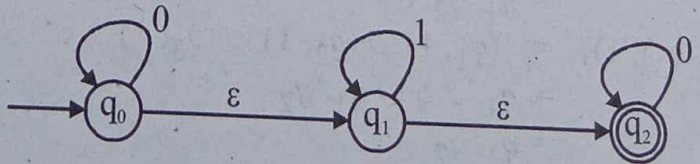
Q.2.(b) Construct a Melay machine equivalent to given Moore machine : (8)

Present State	Next State		Output
	a = 0	a = 1	
→ q_0	q_1	q_2	1
q_1	q_3	q_2	0
q_2	q_2	q_1	1
q_3	q_0	q_3	1

Ans. The corresponding Mealy machine is :

Present state	Next state			
	$a = 0$		$a = 1$	
	State	Output	State	Output
q_0	q_1	0	q_2	1
q_1	q_3	1	q_2	1
q_2	q_2	1	q_1	0
q_3	q_0	1	q_3	1

Q.3.(a) Remove the ε - transition from the given NFA. (10)
(Note : By ε - closure method only)



Ans. Taking $\in$ closure of all

$\in$ closure of $q_0 = \{q_0, q_1, q_2\} = A$

$\in$ closure of $q_1 = \{q_1, q_2\} = B$

$\in$ closure of $q_2 = \{q_2\} = C$

$\Rightarrow$ $* \{(q_0, q_1, q_2), 0\} = \{q_0, q_2\}$

$* \{(q_0, q_1, q_2), 1\} = \{q_1\}$

$\Rightarrow$ $* \{(q_1, q_2), 0\} = \{q_2\}$

$* \{(q_1, q_2), 1\} = \{q_1\}$

$\Rightarrow$ $* \{(q_2), 0\} = \{q_2\}$

$* \{(q_2), 1\} = \phi$

So now are have transactions of all three A, B, C as we have considered above.

Now relating output and A, B, C

$\in$ closure of $\{q_0, q_2\} = \in$ closure of $q_0 = A$

$\in$ closure of $\{q_1\} = B$

$\in$ closure of $\{q_2\} = C$

$\in$ closoure of $\{q_1\} = B$

$\in$ closure of $\{q_2\} = C$

0	1
q_0, q_1, q_2	q_1
q_2	q_1
q_2	ϕ

'O' $\rightarrow$ States are the finite states considered final because consists of 'q_2' state i.e. the final state given in diagram.

	0	1
A	A	B
B	C	B
C	C	ϕ

Q.3.(b) Take an example of Melay and Moore machine each and process any string of at least 4 alphabets from these machines and produce the resulting strings.(6)

Ans. In Mealy machine the output depends on the input states.

Example : Let us consider the example defined by following transition table of Mealy :

Input alphabet = $\{a, b\}$

Output alphabet = $\{0, 1\}$

States = $\{q_0, q_1, q_2, q_3\}$

q_0 is the initial state.

Present state	Input (a)	Output	Input (b)	Output
$\rightarrow q_0$	q_3	0	q_1	1
q_1	q_1	1	q_2	0
q_2	q_2	0	q_3	0
q_3	q_3	0	q_0	0

In Moore machine output depends upon the states of the machine.

Example : Let us consider an example defined by following transition table of Moore machine.

Input alphabet Σ = $\{a, b\}$

Output alphabet D = $\{0, 1\}$

States Q = $\{q_0, q_1, q_2, q_3)$

q_0 is the initial states

$M = \{Q, \Sigma, \Delta, \delta, \lambda, q_0)$

Present state	Next state		Output
	a	b	
q_0	q_1	q_3	1
q_1	q_3	q_1	0
q_2	q_0	q_3	0
q_3	q_3	q_2	1

Q.3.(c) State and prove Arden's theorem.

Ans. Arden's theorem : Let P and Q be two regular expression over Σ. If P does not contain $\wedge$, then the following equation in R namely. **(4)**

$$R = Q + RP$$

has a unique solution given by $R = QP^*$

Proof :

$$Q + (QP^*)P = Q(\wedge + P^* P) = QP^*$$

$$Q + RP = Q + (Q + RP)P$$

$$= Q + QP + RP^2$$

$$= Q + QP + QP^2 + \ldots\ldots + QP^i + RP^{i+1}$$

$$= Q(\wedge + P + P^2 + \ldots\ldots + P^i) + RP^{i+1}$$

$$R = Q(\wedge + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \quad \text{For } i \geq 0.$$

SECTION – B

Q4.(a) Convert the grammer in GNF.

$$S \rightarrow AA \mid a$$
$$A \rightarrow SS \mid b$$

(Note : by taking S as A_1 and A as A_2 method only) **(12)**

Ans. The given grammer is in CNF. S and A are renamed as A_1 and A_2, respectively. So the productions are $A_1 \rightarrow A_1A_2|a$ and $A_2 \rightarrow A_1A_1|b$. As the given grammar has no null productions and is in CNF we need not carry out step 1. So we proceed to step 2.

Step 2. (i) A_1-productions are in the required form. They are $A_1 \rightarrow A_2A_2|a$.

(ii) $A_2 \rightarrow b$ is in required form. Apply Lemma 1 to $A_2 \rightarrow A_1A_1$.

The resulting productions are $A_2 \rightarrow A_2A_2A_1$, $A_2 \rightarrow aA_1$. Thus the A_2-productions are

$$A_2 \rightarrow A_2A_2A_1, \quad A_2 \rightarrow aA_1, \quad A_2 \rightarrow b.$$

Step 3. We have to apply Lemma 2 to A_2-productions as we have $A_2 \rightarrow A_2A_2A_1$. Let Z_2 be the new variable. The resulting productions are

$$A_2 \rightarrow aA_1, \qquad A_2 \rightarrow b$$
$$A_2 \rightarrow aA_1Z_2, \qquad A_2 \rightarrow bZ_2$$
$$Z_2 \rightarrow A_2A_1, \qquad Z_2 \rightarrow A_2A_1Z_2,$$

Step 4. (i) The A_2-productions are $A_2 \rightarrow aA_1|b|aA_1Z_2|bZ_2$.

(ii) Among the A_1-productions we retain $A_1 \rightarrow a$ and eliminate $A_1 \rightarrow A_2A_2$ using Lemma1.The resulting productions are $A_1 \rightarrow aA_1A_2|bA_2$. $A_1 \rightarrow aA_1Z_2A_2|bZ_2A_2$. The set of all (modified) A_1-productions is

$$A_1 \rightarrow a|aA_1A_2|bA_2|aA_1Z_2A_2|bZ_2A_2$$

Step 5. The Z_2-productions to be modified are $Z_2 \rightarrow A_2A_1$, $Z_2 \rightarrow A_2A_1Z_2$.

We apply Lemma 1 and get

$$Z_2 \rightarrow aA_1A_1|bA_1|aA_1Z_2A_1|bZ_2A_1$$
$$Z_2 \rightarrow aA_1A_1Z_2|bA_1Z_2|aA_1Z_2A_1Z_2|bZ_2A_1Z_2$$

Hence the equivalent grammer is

$$G' = (\{A_1, A_2, Z_2\}, \{a, b\}, P_1\ A_1)$$

where P_1 consists of

$$A_1 \rightarrow a|aA_1A_2|bA_2|aA_1Z_2A_1|bZ_2A_2$$
$$A_2 \rightarrow aA_1|b|aA_1Z_2|bZ_2$$
$$Z_2 \rightarrow aA_1A_1|bA_1|aA_1Z_2A_1|bZ_2A_1$$
$$Z_2 \rightarrow aA_1A_1Z_2|bA_1Z_2|aA_1Z_2A_1Z_2|bZ_2A_1Z_2$$

Q.4.(b) Discuss the ambiguity in CFG with the help of example. **(8)**

Ans. Ambiguity in context-free grammars : Sometimes we come across ambiguous sentences in the language we are using. Consider the following sentence in English: "In books

selected information is given." The word 'selected' may refer to books or information. So the sentence may be parsed in two different ways. The same situation may arise in context-free languages. The same terminals string may be the yield of two derivation trees.

Definition : A terminal string $w \in L(G)$ is ambiguous if there exist two or more derivation trees for w (or there exist two or more leftmost derivations of w).

Consider, for example. $G = (\{S\}, \{a, b, +, *\}, P, S)$, where P consists of $S \to S + S|S * S|a|b$. We have two derivation trees for $a + a * b$ given in Fig.

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.

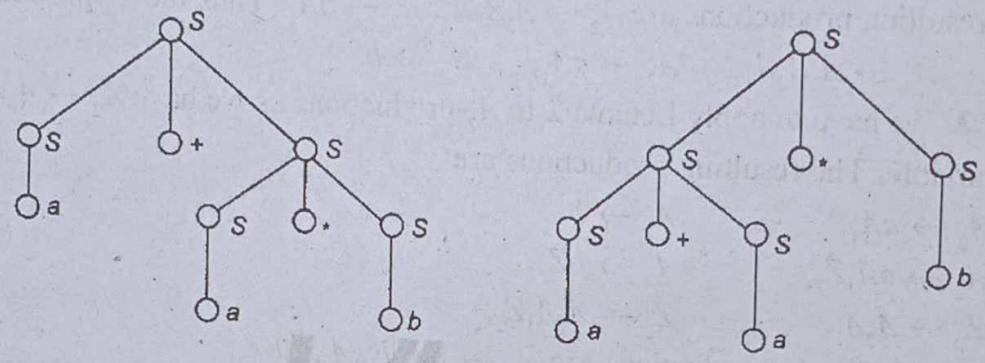


Fig. : Two derivation trees for a + a * b.

Q.5.(a) State the prove pumping lemma for regular languages. **(10)**

Ans. Pumping Lemma : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz$, $y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.

Proof : Let $w = a_1a_2 \ldots a_m, \quad m \geq n$

$\delta(q_0, a_1a_2 \ldots a_i) = q_i$ for $i = 1, 2, \ldots, m$; $Q_1 = \{q_0, q_1, \ldots q_m)$

That is, Q_1 is the sequence of states in the path with path value $w = a_1a_2 \ldots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_j and $q_k (q_j = q_k)$. Then j and k satisfy the condition $0 \leq j < k \leq n$.

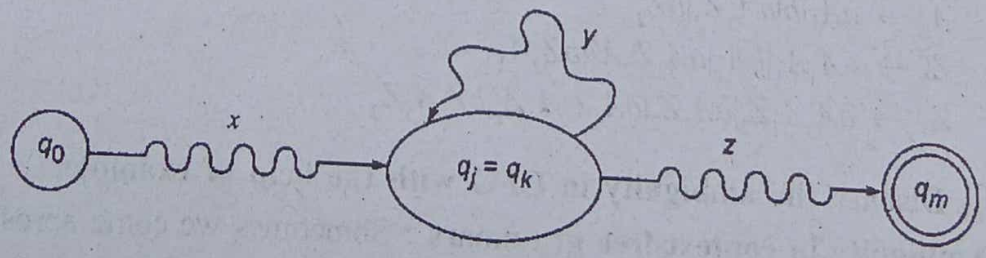


Fig. : String accepted by M.

The string w can be decomposed into three substrings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$ and $a_{k+1} \ldots a_m$. Let x, y, z denote these strings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$, $a_{k+1} \ldots a_m$, respectively. As $k \leq n$, $|xy| \leq n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in Fig.

The automaton M starts from the initial state q_0. On applying the string x, it reaches $q_j(=q_k)$. On applying the string y, it comes back to $q_j(=q_k)$. So after application of y^i for each $i \geq 0$, the automaton is in the same state q_j. On applying z, it reaches q_m, a final state. Hence, $xy^iz \in L$. As every state in Q_1 is obtained by applying an input symbol. $y \neq \Lambda$.

Q.5.(b) Find a reduced grammer equivalent to the grammer G whose productions are : **(10)**

$$\mathbf{S} \rightarrow \mathbf{AB} \mid \mathbf{CA}$$
$$\mathbf{B} \rightarrow \mathbf{Bc} \mid \mathbf{AB}$$
$$\mathbf{A} \rightarrow \mathbf{a}$$
$$\mathbf{C} \rightarrow \mathbf{aB} \mid \mathbf{b}$$

Ans. Step 1 : $W_1 = \{A, C\}$ as $A \rightarrow a$ and $C \rightarrow b$ are productions with a terminal string on R.H.S.

$$W_2 = \{A, C\} \cup \{A_1 | A_1 \rightarrow \alpha \text{ with } \alpha \in (\Sigma \cup \{A, C\})^*\}$$
$$= \{A, C\} \cup \{S\} \text{ as we have } S \rightarrow CA$$
$$W_3 = \{A, C, S\} \cup \{A_1 \mid A_1 \rightarrow \alpha \text{ with } \alpha \in (\Sigma \cup \{S, A, C\})^*\}$$
$$= \{A, C, S\} \cup \phi$$

As $W_3 = W_2$.

$$V'_N = W_2 = \{S, A, C\}$$
$$P' = \{A_1 \rightarrow \alpha \mid A_1, \alpha \in (V'_N \cup \Sigma)^*\}$$
$$= \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}$$

Thus,

$$G_1 = (\{S, A, C\}, (a, b\}, \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}, S)$$

Step 2 : We have to apply Theorem to G_1. Thus,

$$W_1 = \{S\}$$

As we have production $S \rightarrow CA$ and $S \in W_1$, $W_2 = \{S\} \cup \{A, C\}$

As $A \rightarrow a$ and $C \rightarrow b$ are production with $A, C \in W_2$, $W_3 = \{S, A, C, a, b\}$

As $W_3 = V'_N \cup \Sigma$, $P'' = \{S \rightarrow a \mid A_1 \in W_3\} = P'$

Therefore,

$G' = (\{S, A, C\}, (a, b\}, (S \rightarrow CA, A \rightarrow a, C \rightarrow b\}, S)$ is the reduced grammar.

SECTION – C

Q.6.(a) Design a PDA for the language

$$L = \{\omega \in (a, b)^* | \ \omega \text{ has equal number of } a\text{'s and } b\text{'s}\} \quad (10)$$

Ans. We define a pda M as follows :

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

where δ is defined by

$$\delta(q_0, a, Z_0) = \{(q_1, Z_0)\} \quad ...(1)$$
$$\delta(q_0, b, Z_0) = \{(q_0, bZ_0)\} \quad ...(2)$$
$$\delta(q_0, a, b) = \{(q_0, \Lambda)\} \quad ...(3)$$
$$\delta(q_0, b, b) = \{(q_0, bb)\} \quad ...(4)$$
$$\delta(q_1, a, Z_0) = \{(q_1, aZ_0)\} \quad ...(5)$$
$$\delta(q_1, b, Z_0) = \{(q_0, aZ_0)\} \quad ...(6)$$
$$\delta(q_1, a, a) = \{(q_1, aa)\} \quad ...(7)$$
$$\delta(q_1, b, a) = \{(q_1, \Lambda)\} \quad ...(8)$$

The construction can be explained as follows :

If the pda M is in the final state q_1, it means it has seen more a's than b's. On seeing the first a, M changes state (from q_0 to q_1) ((1)). Afterwards it stores the a's in PDS without changing state ((5) and ((7)). It stores the initial b in PDS ((2)) and also the subsequent b's ((4)). The *pda* cancels a in the input string, with the first (topmost) b in PDS ((3)). If all b's are matched with stored a's , and M sees the bottom of PDS, M moves from q_1 to q_0 ((6)). The b's in the input string are cancelled on seeing a in the PDS ((8)).

M is deterministic since δ is not defined for input Λ. The reader is advised to check that q_1 is reached on seeing an input string w in L.

Q.6.(b) Design a Turing Machine to recognise the language

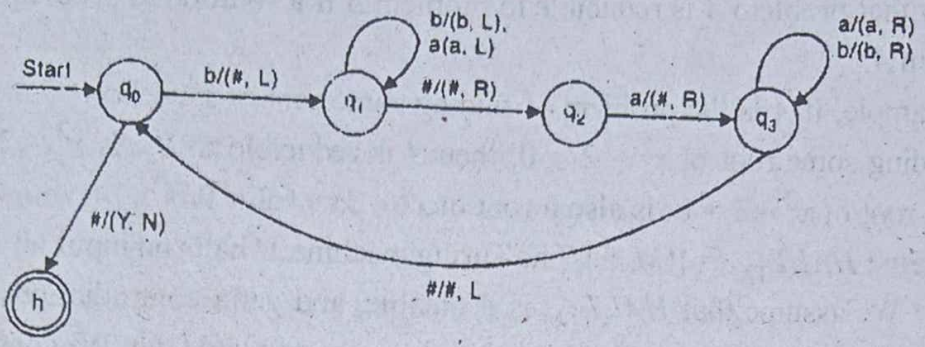
$$L = \{a^n b^n \mid n \geq 1\} \quad (10)$$

Ans. We are going to design a turing machine accepts a set of string in which every string starts with 'a', followed by any number of a's, every string ends in equal number of b's as a's. No 'a' is encountered after first 'b' is read. It will not accepts the strings like $\in$, a, aa, bb, abb, $aabbb$, aab and so on.

Let required turing machine is : $T_m = \{Q, \Sigma, \Gamma, d, q_0, h\}$

$$Q = \{q_0, q_1, q_2, q_3, h\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, \#\}$$

q_0 is initial state and h is halt state.



The moves which are unaddressed, leads towards rejecting states.

Let us pass the string $w = \# a \underline{b} \#$
q_0

$$\begin{aligned} \#abq_0 &\vdash \#aq_1\# \\ &\vdash \#q_1a\# \vdash \#aq_2\# \\ &\vdash \###q_3 \vdash \##q_0\# \\ &\vdash \#Yh\# \end{aligned}$$

Q.7.(a) Design a PDA for the language

$$L = \{ \omega \omega^r \mid \omega \in (a, b)^* \}$$

(i.e. without marker in the middle) **(10)**

Ans. Let PDA be $P = \{Q, \Sigma, \Gamma, \delta, S, F\}$

$Q = \{s, f, f'\}$

$\Sigma = \{a, b\}$

$F = \{f'\}$

and transition relation δ is defined as follows :

(1) $((a, a, \in), (S, a))$

(2) $((s, b, \in), (S, b))$

(3) $((s, \in, \in), (f, \in)) \rightarrow$ (middle of the input string)

(4) $((f, a, a), (f, \in)$

(5) $((f, b, b), (f, \in)$

(6) $((f, \in, \in), (f', \in)$

Clearly whenever the machine is in state 's' it can non-deterministically choose either to push the next input symbol on the stack or to switch to state f without consuming any point. Move 6 is for accepting the string.

Q.7.(b) Discuss the halting problem and PCP problem of turing machines. (10)

Ans. Halting Problem of Turing Machine : This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Theorem : $HALT_{TM} = \{(M, w) |$ The Turing machine M halts on input $w\}$ is undecidable.

Proof : We assume that $HALT_{TM}$ is decidable, and get a contradiction. Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a TM M_2 as follows:

1. For M_2, (M, w) is an input.
2. The TM M_1 acts on (M, w).
3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M_1, w), simulate the TM M on the input string w until M halds.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w is reached. In the first case (the first alternative of setp 5) M_2 accepts (M, w). In the second case (the second alternative of setp 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $T(M_2) = \{(M, w) |$ The Turing machine accepts w$\}$

$= A_{TM}$

This is a contradiction since A_{TM} is undecidable.

PCP problem : The post Correspondence Problem (PCP) was first introduced by Emil Post in 1946. Later, the problem was found to have many applications in the theory of formal languages. The problem over an alphabet belongs Σ belongs to a class of yes/no problems and is stated as follows : Consider the two lists $x = (x_1 \ldots x_n)$, $y = (y_1 \ldots y_n)$ of nonempty strings over an alphabet $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exist $i_1, \ldots i_{m,}$ where $1 \leq ij \leq n$, such that

$$x_{i_j} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$$

SECTION – D

Q.8.(a) What are Primitive resursive functions ? Show that the following function is primitive recursive :

$$r(x, y) = x + y \qquad (10)$$

Ans. Primitive Recursive Functions : A function f of $n + 1$ variables is defined by recursion if there exists a function g of n variables, and a function h of $n + 2$ variables, and f is defined as follows :

$$f(x_1, x_2, ..., x_n, 0) = g(x_1, x_2, ..., x_n) \quad ...(i)$$

$$f(x_1 ... x_n, y + 1) = h(x_1, x_2, ..., x_n, y, f(x_1, x_2, ..., x_n, y)) \quad ...(ii)$$

We may note that f can be evaluated for all arguments $(x_1, x_2, ..., x_n, y)$ by induction on y for fixed $x_1, x_2, ..., x_n$. The process is repeated for every $x_1, x_2, ..., x_n$.

To show that the following function is primitive recursive : $r(x, y) = x + y$

f_1 is a function of two variables. If we want f_1 to be defined by recursion, we need a function g of a single variable and a function h of three variables.

$$f_1(x, 0) = x + 0 = x$$

By comparing $f_1(x, 0)$ with L.H.S. of (i), we see that g can be defined by

$$g(x) = x = U_1^1(x)$$

Also, $f_1(x, y + 1) = x + (y + 1) = (x + y) + 1 = f_1(x, y) + 1$

By comparing $f_1(x, y + 1)$ with L.H.S. of (ii), we have

$$h(x, y, f_1(x, y)) = f_1(x, y) + 1 = S(f_1(x, y)) = S(U_3^3(x, y, f_1(x, y)))$$

Define $h(x, y, z) = S(U_3^3(x, y, z))$. As $g = U_1^1$, it is an initial function. The function h is obtained from the initial functions U_3^3 and S by composition, and by recursion using g and h. Thus f_1 is obtained by applying composition and recursion a finite number of times to initial functions U_1^1, U_3^3 and S. So f_1 is primitive recursive.

Q.8.(b) Show that the CSL's are closed under the following operations : (10)

(i) Union

(ii) Cocatenation

(iii) Intersection

(iv) Substitution

Ans. (i) Union : Suppose L_1 and L_2 are language of the same type, let $G_1 = (V'_n \Sigma', P', S')$ be the grammer from which the language L_1 is generated and $G_2 = (V''_n \Sigma'', P'', S'')$ be the grammer from which the language L_2 generated. The grammer G_1 and G_2 both the of type . So any production in grammer G_1 or G_2 will be either $\alpha \rightarrow \beta$ or $A \rightarrow a$, where $\alpha, \beta, A \in (V'_N \cup V''_N)$ and $\alpha \in \Sigma$.

Let us define a new grammer G_n such that :

$$G_n = (\{V'_N \cup V''_N \cup \{S\}\}, \Sigma' \cup \Sigma'', P_n, S)$$

Where S is a new start symbol. Such that $S \notin V'_N \cup V''_N$.

The set production P_n in G_n is defined as

$$P_n = \{S \rightarrow S_1, S \rightarrow S_2\} \cup \{P' \cup P''\}$$

Now we have to prove that the language generated by G_n i.e., $L(G_n)$ is equivalent to the union of language L_1 and L_2 that is,

$$L(G_n) = L_1 \cup L_2$$

It is string that belongs to language $L_1 \cup L_2$ then the string 'S' can be derived by grammer G_1, such that or, by grammer G_2 such that.

Therefore, $S \xrightarrow[G_n]{} S_1$ and $S_1 \xrightarrow[G_1]{} S$

or $S \xrightarrow[G_n]{} S_2$ and $S_2 \xrightarrow[G_{2_1}]{} S$

It means, $S \in L(G_n)$.

Hence $L_1 \cup L_2 = L(G_n)$.

(b) Substitution : For the proof of $L(G_n) = L_1 \cup L_2$ let us consider a derivation of strings 'S'. The first step is to derive $S \to S_1$ or $S \to S_2$. $S \to S_1$ is derived first then S_1 is changed in subsequent steps. As we have assumed that $V'_N \cap = V''_N \phi$, these steps should involve only variable of V'_N and the production we apply are in P'. Therefore is defined by G_N,

$$S \xrightarrow[G_n]{} S$$

Similarly, if the first step is to derive $S \to S_2$ then

$$S \xrightarrow[G_n]{} S_2 \xrightarrow[G_n]{} S$$

Hence $L(G_n) = L_1 \cup L_2$

Also $L(G_n)$ is a type-2 according as L_2 are of type-2. If $\wedge \in L_1 \cup L_2$, then $L(G_n)$ is of type-3 or type-1 according as L_1 and L_2 are of type-1 or type-3.

Let us suppose $\wedge \in L$, in this case the grammer G_n is defined as,

$$G_n = (\{V'_N \cup V''_N \cup \{S, S''\}\}, \Sigma' \cup \Sigma'', P_n, S')$$

Where,

(i) S' = a new start symbol i.e., $S' \notin V'_N \cup V''_N \cup \{S\}$

(ii) $P_n = P' \cup P'' \cup \{S' \to S, S \to S_1, S \to S_2\}$

Therefore, $L(G_n)$ is of type-1 or type-3 according as L_1 and L_2 are of type-1 or type-3.

Let us suppose now $\wedge \in L_2$, the proof is similar as in case of $\wedge \in L_1$.

(c) Concatenation : Suppose L_1 and L_2 are two language of same type say i. Let the grammer $G_1 = (V''_N, \Sigma', P', S')$ generate the language L_1 and the grammer $G_2 = (V''_N, \Sigma'', P'', S'')$ generates the language L_2. Therefore the grammer G_1 and G_2 will also be of same type i.e., type-i. The connection of language L_1 and L_2 is L_1L_2. Now our goal is to prove that the type of language L_1L_2 is type-i.

Let us construct a new grammer G_c that will generate the language L_1L_2:

$$G_c = (\{V'_N \cup V''_N \cup \{S\}\}, \Sigma' \cup \Sigma'', P_c, S)$$

Where S is the start symbol such that $S \notin V'_N \cup V''$ and $P_c = P' \cup P'' \cup \{S' \to SS''\}$.

If the language gnerated by grammer G_c is $L(G)$ then we have to prove that $L(G_c) = L_1 L_2$ and $w_1 \in L_1$, $w_2 \in L_2$ and $w_c \in L_1L_2$, then $w_1w_2 \in L_1L_2$.

The string w_1 is derived by grammer G_1, such that $S_1 \xrightarrow[G_1]{} w_1$, and the string w_2 is derived by the grammer G_2 such that $S_2 \xrightarrow[G_c]{} S_1S_2$.

Now according to grammer G_c

$$S \xrightarrow[G_c]{} S_1S_2 \text{ and } S_1S_2 \xrightarrow[G_c]{} w_1w_2$$

Hence $L_1L_2 \subseteq (G_c)$.

(d) Intersection : If $w_c \in L(G_c)$, then to generate the string wc the first step is $S \to S_1S_2$. We assume that $V'_N \cap V''_N = \phi$ and the production in grammer G_1 or G_2 involve only noterminals $w_c = w_1w_2$. Therefore $L_1L_2 = L(G_c)$. Also grammer G_c is of type-0 or type-2 according as G_1 and G_2 are of type-2. The above construction is sufficient when G_1 and G_2 are also of type-3 or type-1 with $\wedge \notin L_1 \cup L_2$.

Let G_1 and G_2 be the type-1 or type-3 and $\wedge \notin L_1$, $\wedge \notin L_2$. Let us define language L_A and L_B such that

$$L_A = L_1 - \{\wedge\} \text{ and } L_B = L_2 - \{\wedge\}$$

Then $L_1 L_2$ is defined as,

$$L_1L_2 = \begin{cases} L_AL_B \cup L_B & \text{if } \wedge \in L_1 \text{ but } \wedge \notin L_2 \\ L_AL_B \cup L_A & \text{if } \wedge \in L_2 \text{ but } \wedge \notin L_1 \\ L_AL_B \cup L_A \cup L_B\{\wedge\} & \text{if } \wedge \in L_1.L_2 \end{cases}$$

As we have shown in the previous theorem, that L_{CSL} and L_{RL} are closed under union. So concantation of L_1 and L_2 is of type-1 or type-3 according as L_1 and L_2 are type-1 or type-3.

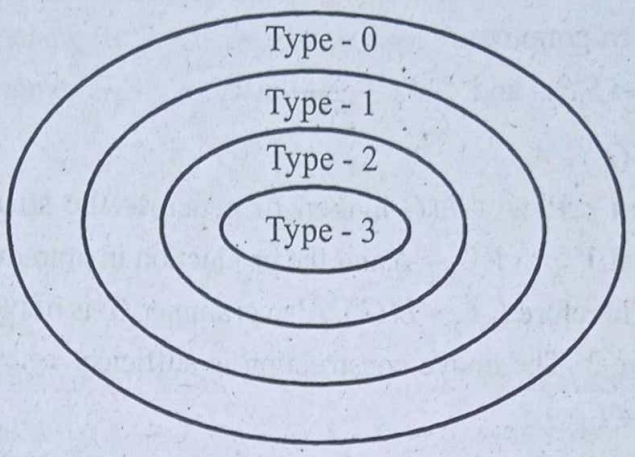
Q.9.(a) Discuss in detail Chomsky hierarchy of grammer and also explain the relation between languages of classes under Chomsky classification with the help of diagram. (10)

Ans. Chomsky hierarchies of grammars : We can exhibit the relationship between grammars by the Chomsky hierarchy. Chomsky provided an initial classification in to four languages type :

Type – 0	(Unrestricted grammar)
Type – 1	(Context sensitive grammar)
Type – 2	(Context free grammar)
Type – 3	(Regular grammar)

Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages. Type 1 consist of the context-sensitive languages, Type 2 consists of the context-free languages and Type 3 consists of the regular languages. Each language family of type k is a proper subset of the family of type $k - 1$. Following diagram shows the original Chomsky Hierarchy.

Relation between classes of languages : The four classes of languages–recursive sets, context sensitive languages, context-free languages and regular sets are often referred to as language of types 0, 1, 2 and 3 respectively.



We can find the relation between all these sets of languages through following theorems:

Theorem 1 : Every context sensitive language is recursive.

Proof : We know that for every recursive language there exists a turning machine for it. But we have not been very specific about how one inputs a grammar into turing machine. Let us imagine that some string of delimiters separating the productions, possibly allowing the production arrow to be tape character as well.

Now let us feed this turning machine and W (input string) into universal turing machine. Because W either halts or crashes on the coded turing machine, this procedure will indeed, lead to a decision about W's membership in the language generated by G. So we prove that there exists a turning machine for every context-sensitive language, which can decide that any arbitrary string W can be derived from the given context-sensitive grammar or not, so we can say that every context-sensitive language is recursive.

Theorem 2 : There is at least one language L that is recursive but not context-sensitive.

Proof : In the previous theorem, we indicate that there was encoding an entire context-sensitive grammar into a single string of symbols.

Let us consider the set of all context-sensitive grammars on $V_i = \{a, b\}$.

Let us assume that each grammar in the set has variable in following form

$$V_n = \{V_0, V_1, V_2, \ldots\}$$

Every context-sensitive grammar is completely specified by its productions. We can think of them as written as a single string.

$$x_1 \rightarrow y_1 \; ; \; x_2 \rightarrow y_2 \; ; \; \ldots, \; x_m \rightarrow y_m$$

We can represent these string in homomorphism as follows :

$$h(a) = 0\,1\,0$$
$$h(b) = 0\,1^2\,0$$
$$h(R) = 1\,1^3\,0$$
$$h(i) = 0\,1^4\,0$$

$$h(V_i) = 0\,1'^{+5}\,0$$

Thus, any context-sensitive grammar can be represented uniquely by a string of the form L (011*0)*). Further more, the representation is invertible in the sense that, given any such string, there is at most one context-sensitive grammar corresponding to it.

Let us introduce a proper ordering on $(0, 1)^+$, so we can write strings in the order w_1, w_2 etc. A given string w_j may not define a context-sensitive grammar; if it does, call the grammar G_j.

Now let us define a language.

$$L = \{W_i : W_i \text{ defines a CSG } G_i \text{ and } W_i \notin L(G_i)\}$$

Q.9.(b) Define the following : **(10)**

(i) Recursive functions

(ii) partial Recursive functions

(iii) Primitive Recursive functions

Ans. (i) Recursive Function : A function is recursive if it can be obtained from the initial functions by a finite number of application of composition, recursion and minimization over regular functions.

(ii) Partial recursive function : A function is partial recursive function if it can be obtained from the initial functions by a finite number of application of composition, recursion and minimization.

Partial recursive functions are turing computable.

For ex. : $f = x/2$ is prtial recursive function over N.

(iii) Primitive recursive function : A total function f over N is called primitive recursive

(i) if it is one of three initial functions.

(ii) if it can be obtained by applying composition & recursion a finite number of time to the set of initial function.

For ex. : $f(x, y) = x^y$ is a primitive recursive function.

THEORY OF AUTOMATA COMPUTATION

Dec - 2017

Paper Code:-CSE-305-F

Note : ***Attempt five questions in all, selecting one question from each Section. Question No. 1 is compulsory. All questions carry equal marks.***

Q.1.(a) Any two difference between DFA and NFA. **(4)**

Ans. Differences between DFA and NFA are as follows :

(1) "DFA" stands for "Deterministic Finite Automata", while "NFA" stands for "Nondeterministic Finite Automata."

(2) Both are transition functions of Automata. In DFA the next possible state is distinctly set, while in NFA each pair of state and input symbol can have many possible next states.

Q.1.(b) Define finite automata with output briefly. **(4)**

Ans. The finite automata have binary output, i.e. either they accept the string or they do not accept the string. This acceptability was decided on the basis of reachability of the final state by the initial state. Now, we remove this restriction and consider the model where the outputs can be chosen from some alphabet. The value of the output function Z(t) in the most general case is a function of the present state q(t) and the present input x(t), i.e.

$$Z(t) = \lambda(q(t), x(t))$$

where λ is called the output function. This generalized model is usually called the Mealy machine. If the output function $Z(t)$ depends only on the present state and is independent of the current input, the output function may be written as

$$Z(t) = \lambda(q(t))$$

This restricted model is called the Moore machine. It is more convenient to use Moore machine in automata theory.

Q.1.(c) Explain closure properties of regular languages. **(4)**

Ans. Closure Properties of Regular Sets :

(1) *Regular languages are closed under union opration and kleene star operation.*

Proof : If L_1 and L_2 are regular, then there are regular expressions r_1, and r_2 denoting the languages L_1 and L_2 respectively.

$(r_1 + r_2)$ and (r_1^*) are negular languages denoting the languages $L_1 \cup L_2$ and L_1^*

Therefore $L_1 \cup L_2$ and L_1^* are regular.

(2) *Regular languages are closed under complement operation and intersection operation.*

Proof : Suppose that L_1 and L_2 are regular over an alphabet Σ.

There is a *DFA* $M = |Q, \Sigma, \delta, q_0, F|$ accepting L_1. Design a *DFA* $M^1 = |Q, \Sigma, \delta, q_0, F^1|$ where $F^1 = Q/F$. Now, we have that $L(M^1) = \Sigma^* | L_1$. Hence the complement of L_1 is regular.

Let $L_1 = \Sigma^* | L_1$ and $L^1{}_2 = \Sigma^* | L_2$. The complement of $L_1{}^1 \cup L_2{}^1$ is regular and is equal to $L_1 \wedge L_2 . L_1 \wedge L_2 \bar{v}$ regular.

Q.1.(d) Give formal definition of a PDA. (4)

Ans. A pushdown automaton consists of

(i) a finite nonempty set of states denoted by Q,

(ii) a finite nonempty set of input symbols denoted by Σ,

(iii) a finite nonempty set of pushdown symbols denoted by Γ,

(iv) a special state called the initial state denoted by q_0,

(v) a special pushdown symbol called the initial symbol on the pushdown store denoted by Z_0,

(vi) a set of final states, a subset of Q denoted by F, and

(vii) a transition function δ from $Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.

Symbolically, a pda is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Q.1.(e) State Halting problem of Turing machine. (4)

Ans. Halting Problem of Turing Machine : This technique is used to prove the undecidability of halting problem of Turing machine.

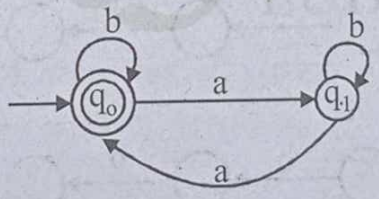
We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Section – A

Q.2.(a) Design a DFA which accepts even number of a's over the alphabet {a,b). (10)

Ans.



String bbaba

$(q_o, \underline{b}baba) \rightarrow (q_o, ba\ ba)$

$(\underline{q}_o, \underline{b}a\ ba) \mapsto (q_o, a\ b\ a)$

$(q_o, a\ ba) \mapsto (q_1, b\ a)$

$(q_1, ba) \mapsto (q_1, a)$

$(q_1, a) \mapsto (q_o)$ (Final statement)

Q.2.(b) Explain the Limitations of Finite Automata. (5)

Ans. Limitations of Finite State Automata :

(1) The predictable nature of deterministic FSAs can be unwanted in some domain such as computer games.

(2) Larger systems implemented using a FSM can be difficult to manage and maintain without a well thought out design. The state transitions can cause a fair degree of "spaghetti-factor" when trying to follow the line of execution.

(3) Not suited to all problem domains, should only be used when a systems behaviour can be decomposed into seprate states with well defined conditions for state transitions. This means that all status, transitions and conditions need to be known up and be well defined.

(4) The conditions for state transitions are redged, meaning they are fixed.

(5) Have no memory of its own.

(6) Travels only from left to right.

Q.2.(c) State and prove Arden's method. **(5)**

Ans. Arden's theorem : Let P and Q be two regular expression over Σ. If P does not contain $\wedge$, then the following equation in R namely.

$$R = Q + RP$$

has a unique solution given by $R = QP^*$

Proof :

$$Q + (QP^*)P = Q(\wedge + P^* P) = QP^*$$

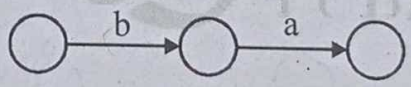
$$\begin{aligned} Q + RP &= Q + (Q + RP)P \\ &= Q + QP + RP^2 \\ &= Q + QP + QP^2 + \ldots\ldots + QP^i + RP^{i+1} \\ &= Q(\wedge + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \end{aligned}$$

$$R = Q(\wedge + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \quad \text{For } i \geq 0.$$

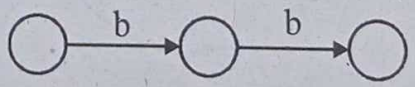
Q.3.(a) Construct a Finite Automata equivalent to the regular expressions : ba + (a + bb)a*b **(10)**

Ans.

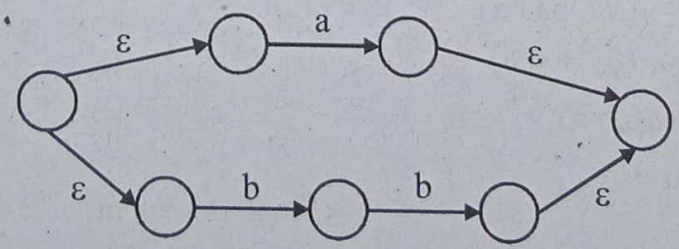
ba



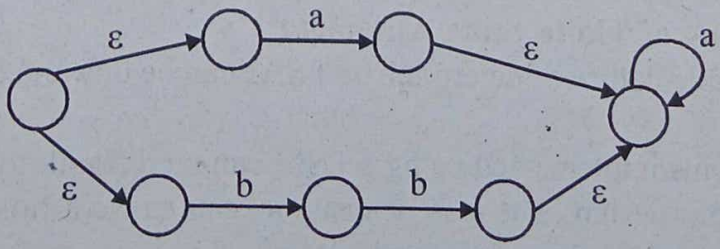
bb



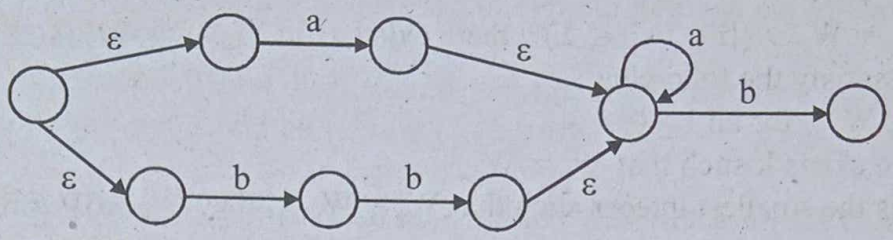
a + bb



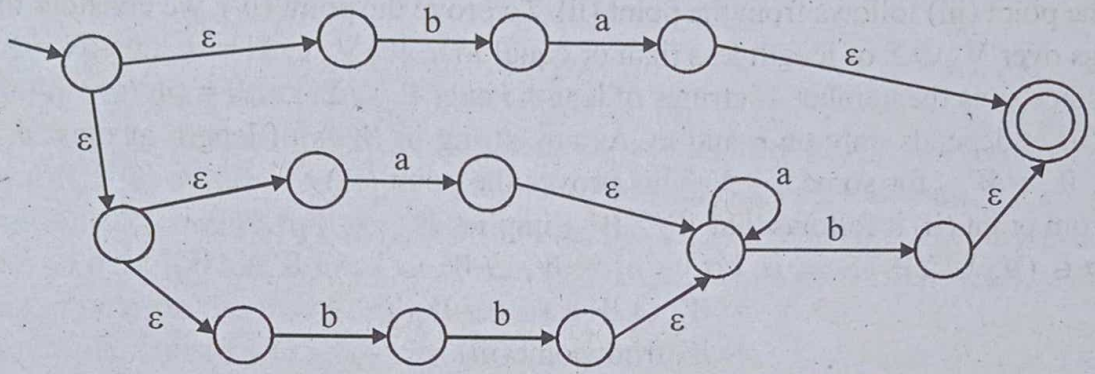
(a + bb)a*

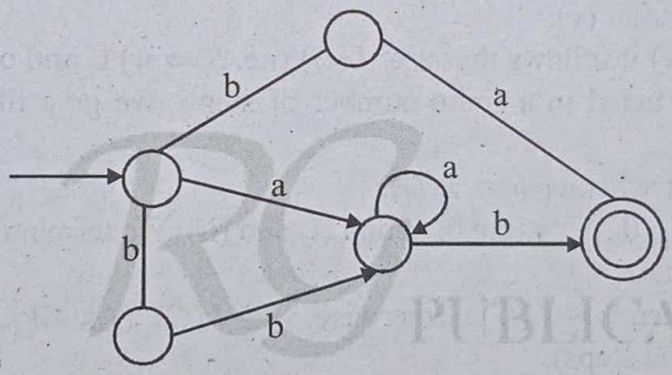


(a + bb)a*b



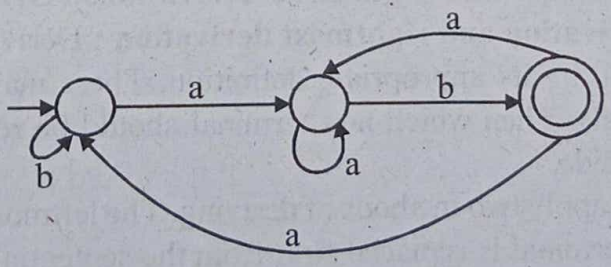
ba + (a + bb)a*b





Q.3.(b) Construct a DFA accepting all strings over {a, b} ending in ab. (10)

Ans.



String accepted - (ab, abab, aaab, bbaab, abaaabab)

Section – B

Q.4.(a) Prove that a context-sensitive language is recursive. (10)

Ans. A context-sensitive language is recursive.

Proof : Let $G = (V_N, \Sigma, P, S)$ and $w \in \Sigma^*$. We have to construct an algorithm to test whether $w \in L(G)$ or not. If $w = \Lambda$, then $w \in L(G)$ iff $S \rightarrow \Lambda$ is in P. As there are only a finite number of productions in P, we have to test whether $S \rightarrow \Lambda$ is in P or not.

Let $| w | = n \geq 1$. The algorithm is based on the construction of a sequence $\{W_i\}$ of subsets of $(V_N \cup \Sigma)^*$. W_i is simply the set of all sentential forms of length less than or equal to n, derivable in at most i steps. The construction is done recursively as follows :

Note that we have replaced first X from left or right in leftmost derivation and XYX the last X i.e. the rightmost symbol.

Actually, we may use leftmost derivation or rightmost derivation we get the same string. The type of derivation does not affect on getting of a string.

Q.5.(a) Define Chomsky normal form. Simplify the following CFG and convert it into CNF : **(10)**

$S \rightarrow ASB \mid \varepsilon$
$A \rightarrow aAS \mid \varepsilon$
$B \rightarrow SbS \mid A \mid bb$

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

Definition : 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 productions are $S \rightarrow AB \mid \Lambda$, $A \rightarrow a$, $B \rightarrow b$. Then G is in Chomsky normal form.

$S \rightarrow A\,S\,B \mid \varepsilon$
$A \rightarrow a\,A\,S \mid \varepsilon$
$B \rightarrow S\,b\,S \mid A \mid bb$

$S \rightarrow \varepsilon$
$A \rightarrow \varepsilon$
$S \rightarrow A\,S\,B \mid S\,B \mid\mid A\,B \mid B$
$A \rightarrow a\,A\,S \mid a\,S \mid a\,A \mid a$
$B \rightarrow S\,b\,s \mid S\,b \mid b\,S \mid b \mid b\,b$

Now production becomes

$S \rightarrow \underline{A}\,S\,B \mid \underline{S\,B} \mid A\,\underline{B} \mid \underline{B}$
$A \rightarrow \underline{a}\,A\,S \mid \underline{a}\,S \mid \underline{a}\,A \mid \underline{a}$
$B \rightarrow S\,b\,S \mid S\,b \mid b\,S \mid b \mid b\,b$
$C_{SB} \rightarrow S\,B$
$C_a \rightarrow a$
$C_b \rightarrow b$ put this production in above production.

Now production becomes :

$S \rightarrow A\,C_{SB} \mid \underline{S\,B} \mid \underline{A\,B} \mid \underline{B}$
$A \rightarrow C_a\,A\,S \mid \underline{C_a}\,S \mid \underline{C_a\,A} \mid a$
$B \rightarrow S\,C_b\,S \mid \underline{S\,C_b} \mid \underline{C_b}\,S \mid b \mid \underline{C_b}\,Cb \mid A$
$C_{SB} \rightarrow S\,B$
$C_a \rightarrow a$
$C_b \rightarrow b$

Put & replace in above equations.

$C_{AS} \rightarrow A\ S$
$C_{CS} \rightarrow C_b\ S$

Now production becomes

$S \rightarrow A\ C_{SB} \mid S\ B \mid A\ B \mid b$
$A \rightarrow C_a\ C_{AS} \mid C_a\ S \mid C_a\ A \mid a$
$B \rightarrow S\ C_{CS} \mid S\ C_b \mid C_b\ S \mid b \mid C_a\ C\ b \mid a$
$C_{SB} \rightarrow S\ B$
$C_a \rightarrow a$
$C_b \rightarrow b$
$C_{AS} \rightarrow A\ S$
$C_{CS} \rightarrow C_b\ S$

Q.5.(b) State and prove pumping lemma for regular languages. Show that the language. **(10)**

$L = \{WW^R | W\ \Sigma(a, b)^*\}$ is not regular.

Ans. Pumping Lemma : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz$, $y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.

Proof : Let $w = a_1a_2 \ldots a_m,\ m \geq n$

$\delta(q_0, a_1a_2 \ldots a_i) = q_i$ for $i = 1, 2, \ldots, m$; $Q_1 = \{q_0, q_1, \ldots q_m\}$

That is, Q_1 is the sequence of states in the path with path value $w = a_1a_2 \ldots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_j and $q_k (q_j = q_k)$. Then j and k satisfy the condition $0 \leq j < k \leq n$.

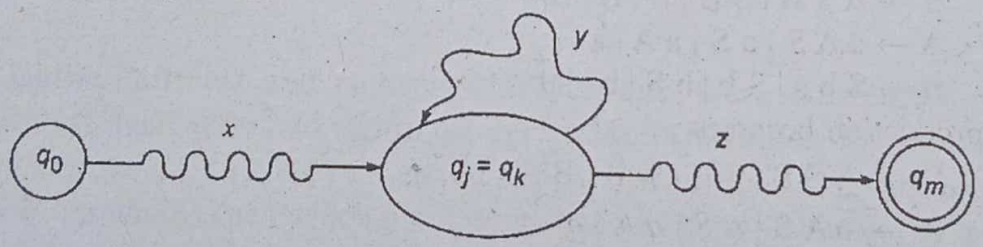


Fig. : String accepted by M.

The string w can be decomposed into three substrings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$ and $a_{k+1} \ldots a_m$. Let x, y, z denote these strings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$, $a_{k+1} \ldots a_m$, respectively. As $k \leq n$, $|xy| \leq n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in Fig.

The automaton M starts from the initial state q_0. On applying the string x, it reaches $q_j (= q_k)$. On applying the string y, it comes back to $q_j (= q_k)$. So after application of y^i for each $i \geq 0$, the automaton is in the same state q_j. On applying z, it reaches q_m, a final state. Hence, $xy^iz \in L$. As every state in Q_1 is obtained by applying an input symbol. $y \neq \Lambda$.

Show that $L = \{ww \mid w \in \{a, b\}^*\}$ is not regular.

Step 1 : Suppose L is regular. Let n be the number of states in the automaton M accepting L.

Step 2 : Let us consider $ww = a^n ba^n b$ in L. $|ww| = 2(n+1) > n$. We can apply pumping lemma to write $ww = xyz$ with $|y| \neq 0$, $|xy| \leq n$.

Step 3 : We want to find i so that $xy^iz \notin L$ for getting a contradiction. The string y can be in only one of the following forms :

Case 1 : y has no b's, i.e. $y = a^k$ for some $k \geq 1$.

Case 2 : y has only one b.

We may note that y cannot have two b's. If so, $|y| \geq n+2$. But $|y| \leq |xy| \leq n$. In Case 1, we can take $i = 0$. Then $xy^0z = xz$ is of the form $a^m ba^n b$, where $m = n - k < n$ (or $a^n ba^m b$). We cannot write xz in the form uu with $u \in \{a, b\}^*$, and so $xz \notin L$. In Case 2 too, we can take $i = 0$. Then $xy^0z = xz$ has only one b (as one b is removed from xyz, b being in y). So $xz \notin L$ as any element in L should have an even number of a's and an even number of b's.

Thus in both the cases we get a contradiction. Therefore, L is not regular.

Section – C

Q.6.(a) Explain the programming techniques involved in Turning machine.(10)

Ans. Programming Techniques involved in Turing Machines :

(1) Turing Machine with Stationary Head : In the definition of a *TM* we defined $\delta(q, a)$ as $\delta(q', y, D)$ where $D = L$ or R. So the head moves to the left or right after reading an input symbol. Suppose, we want to include the option that the head can continue to be in the same cell for some input symbol. Then we define $\delta(q, a)$ as (q', y, S). This means that the TM, on reading the input symbol a, changes the state to q' and writes y in the current cell in place of a and continues to remain in the same cell. In terms of IDs.

$$wqas \vdash wq'yx$$

Of course, this move can be simulated by the standard TM with two moves, namely

$$wqax \vdash wyq''x \vdash wq'yx$$

That is, $\delta(q, a) = (q', y, S)$ is replaced by $\delta(q, a) = (q'', y, R)$ and $\delta(q'', X) = (q', y, L)$ for any tape symbol X.

Thus in this model $\delta(q, a) = (q', y, D)$ where $D = L$, R or S.

(2) Storage in the State : We are using a state, whether it is of a *FA* or pda or *TM*, to 'remember' things. We can use a state to store a symbol as well. So the state becomes a pair (q, a) where q is the state (in the usual sense) and a is the tape symbol stores in (q, a). So the new set of states becomes $Q \times \Gamma$.

(3) Multiple Track Turing Machine : In the case of *TM* defined earlier, a single tape was used. In a multiple track *TM*, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. Hence the only difference between the standard *TM* and the *TM* with multiple tracks is the set of tape symbols. In the case of the standard Turing machine, tape symbols are elements of Γ; in this case of TM with multiple track, it is Γ^k. The moves are defined in a similar way.

(4) Subroutines : We know that subroutines are used in computer languages, when some task has to be done repeatedly. We can implement this facility for *TMs* as well.

First, a *TM* program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt. For using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected

to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of *TM*.

Q.6.(b) Construct PDA M to accept the language having equal number of 0's and 1's. **(10)**

Ans. $L\{0^n, 1^n\}$

$$\text{Strings} = \{01, 0011, 000111, 00001111 \ldots\ldots\}$$

Transition :

$\delta(q_o, 0, z_o) \mapsto (q_1, 0\ z_0)$
$\delta(q_1, 0, 0) \mapsto (q_1, 0\ 0)$
$\delta(q_1, 1, 0) \mapsto (q_2, \varepsilon)$
$\delta(q_2, 1, 0) \mapsto (q_2, \varepsilon)$
$\delta(q_2, \varepsilon, z_0) \mapsto (q_3, \varepsilon)$

State	Input	Stack symbol	Transition
q_0	00001111	Z_0	$\delta(q_o, 0, z_0) \mapsto (q_1, 0\ z_0)$
q_1	0001111	0	$\delta(q_1, 0, 0) \mapsto (q_1, 0\ 0)$
q_1	001111	0	$\delta(q_1, 0, 0) \mapsto (q_1, 0\ 0)$
q_1	01111	0	$\delta(q_1, 0, 0) \mapsto (q_1, 0\ 0)$
q_1	1111	0	$\delta(q_1, 1, 0) \mapsto (q_2, \varepsilon)$
q_2	111	0	$\delta(q_2, 1, 0) \mapsto (q_2, \varepsilon)$
q_2	11	0	$\delta(q_2, 1, 0) \mapsto (q_2, \varepsilon)$
q_2	1	0	$\delta(q_2, 1, 0) \mapsto (q_2, \varepsilon)$
q_2	ε	Z_0	$\delta(q_2, \varepsilon, z_0) \mapsto (q_3, \varepsilon)$

Q.7.(a) Define Turning machine. Design a Turing machine that computes the integer function f defined as follows : **(10)**

***f*(*n*) = 3^n where *n* is integer and *n*≥ 0**

Ans. Turning machine : A Turning machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where

(1) Q is a finite nonempty set of states.
(2) Γ is a finite nonempty set of tape symbols.
(3) $b \in \Gamma$ is the blank.
(4) Σ is a nonempty set of input symbols and is a subset of Γ and $b \notin \Sigma$.
(5) δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head; $D = L$ or R according as the movement is to the left or right.
(6) $q_0 \in Q$ is the initial state, and
(7) $F \subseteq Q$ is the set of final states.

The Turning machine can be thought of as finite control connected to a R/W (read/write) head. It has one tape which is divided into a number of cells. The block diagram of the basic model for the Turning machine is given in Fig.

Each cell can store only one symbol. The input to and the output from the finite state automaton are effected by the R/W head which can examine one cell at a time. In one move, the

machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine

(i) a new symbol to be written on the tape in the cell under the R/W head,

(ii) a motion of the R/W head along the tape: either the head moves one cell left (L), or one cell right (R),

(iii) the next state of the automaton, and

(iv) whether to halt or not.

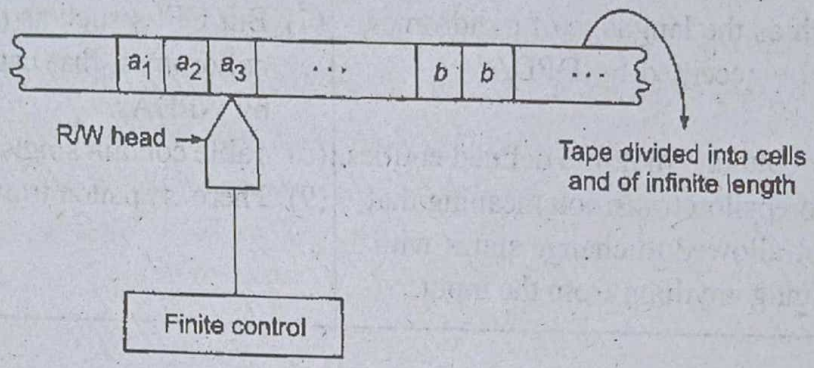


Fig. : Turning machine model

$f(n) = 3^n$ where n is integer and $n \geq 0$

String - {3, 33, 333, 333, ...}

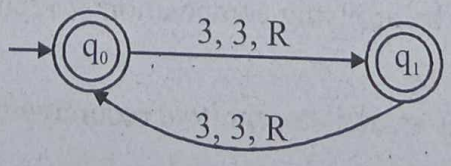
Transition table

	3	β
→ q_0	q_1, 3, R	–
(q_1)	q_0, 3, R	–

$\beta\beta q_0 333\beta\beta \Rightarrow \beta\beta 3 q_1 33\beta\beta \Rightarrow \beta\beta 33 q_0 3\beta\beta \Rightarrow \beta\beta\ 333 q_1 \beta\beta$

Since the turing machine halt in q_1.

The string 111 is accepted.



Q.7.(b) Differentiate between DPDA and NPDA with the help of example. (10)

Ans.

DPDA	NPDA
(1) It is deterministic	(1) It is non-deterministic
(2) DPDA is not generalization of NPDA DPDA	(2) NPDA are a generalization of DPDA
(3) Every DPDA can be simultated by an NPDA	(3) Converse is not true.

(4) DPDAs can simulate more easily with our deterministic computer.	(4) Complex
(5) It is always defined that at for a particular input it will be going to a specific state	(5) It is always defined that at for a particular input it may goto different states.
(6) DPDAs are less powerful than NPDAs.	(6) DPDAs are more powerful than NPDAs.
(7) CFLs, such as the language of palidromes, that can't be accepted by DPDA.	(7) But CFLs such as the language of palidromes, that can be accepted by NPDAs.
(8) Table may contains multiple defined entities.	(8) Table contain single entities.
(9) There is no epsilon transition meaning that you are not allowed to change states with out consuming anything from the input.	(9) There is epsilon transition.

Section – D

Q.8. Explain the following with example : **(10×2=20)**

(a) Partial recursive functions

(b) Primitive recursive functions

Ans. (a) Partial recursive function : A function is partial recursive function if it can be obtained from the initial functions by a finite number of application of composition, recursion and minimization.

Partial recursive functions are turing computable.

For ex. : $f = x/2$ is prtial recursive function over N.

(b) Primitive recursive function : A total function f over N is called primitive recursive

(i) if it is one of three initial functions.

(ii) if it can be obtained by applying composition & recursion a finite number of time to the set of initial function.

For ex. : $f(x, y) = x^y$ is a primitive recursive function.

Q.9. What do you mean by computability ? Explain in detail. **(20)**

Ans. Computability : Computability refers to the ability to solve a problem in an effective manner. It is a key topic of the field of computability theory within mathematical logic and computer science. The computability of a problem is closely linked to the existence of an algorithm to solve the problem.

A computable : In computability theory, a Turing reduction from a problem A to a problem B, named after Alan Turing, is a reduction which solves A, assuming B is already known.

Computable : Expressible as a yes/no question that can be answered in any case by a computer in finite time.

Computable Function : Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

The class of computable functions can be defined in many equivalent models, including

- Turing machine
- μ-recursive functions
- Lambda calculus
- Post machines (Post-Turing machines and tag machines).
- Register machines.

Turning Computable Function : Turing machine can be used to compute functions. We adpot the following policy to input any string to a turing mechine.

The string W is presented in to the form of # W #, that is string, W is surrounded by blanks symbols from both sides and is placed on the left most square of tape, the head of turing machine is positioned at the right most blank symbol which immediately follows the string W.

This is shown by an underscore that is we use an underscore to show the current position of machine head in the tape. A turing machine is said to halt on input 'W' if we can reach to a halting state 'h', after performing some operations, that is if,

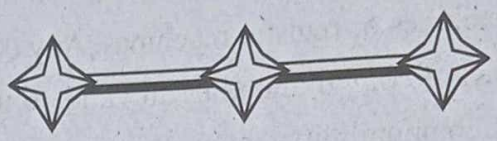
$T_m = (Q, \Sigma, \Gamma, \delta, q_0, h)$ is a turing machine then Tm is said to halt to on input W if and only if $(q, \# \underline{W} \#)$ yields to $(h, \# \underline{\mu} \#)$.

Multiple track turing machine : In the case of TM, a single tape was used. In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. Hence the only difference between the standard TM and the TM with multiple tracks is the set of tape symbols. In the case of the standard Turning machine, tape symbols are elements of Γ; in the case of TM with multiple track, it is Γ^k. The moves are defined in a similar way.

Subroutines : Subroutines are used in computer languages, when some task has to be repeatedly. We can implement this facility for TMs as well.

First, a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt. For using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected

to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of TM.



THEORY OF AUTOMATA AND COMPUTATION

Dec. - 2018

Paper Code:CSE-305-F

Note : ***Attempt five questions in all, selecting one question from each Section. Question No. 1 is compulsory. All questions carry equal marks.***

Q.1.(a) What are limitations of FSM ?

Ans. Limitation of FSM :

(i) The defining characteristics of finite automata is that they have only a finite number of states. Hence finite automata can only count a finite number of input scenario.

(ii) Finite automata do not have any auxiliary memory so it works in process & forget mode. It can store anything at particular state. There is no finite automata that recognize these string.

(iii) The set of binary strings consisting of an equal no. of 1's & 0's.

(iv) Th set of strings over "(" , ")" that have balanced parenthesis.

(v) Finite automata can recognize & process only regular languages.

Q.1.(b) Define DFA and NDFA mathematically.

Ans. DFA : DFA is a finite state machine that accepts/rejects finite string of symbols and only produces a unique computation of the automatan for each input string.

DFA is a 5 tuple$(Q, \Sigma, \delta, q_0, F)$

$Q \rightarrow$ finite set of states

$\Sigma \rightarrow$ finite set of inputs.

Transition function $\delta \rightarrow Q * \Sigma \rightarrow Q$

$q_0 \rightarrow$ initial state

$F \rightarrow$ final state

NDFA : A nondeterministic finite automaton (NDFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(i) Q is a finite nonempty set of states;

(ii) Σ is a finite nonempty set of inputs;

(iii) δ is the transition function mapping from $Q \times \Sigma$ into 2^Q which is the power set of Q, the set of all subsets of Q;

(iv) $q_0 \in Q$ is the initial state; and

(v) $F \subseteq Q$ is the set of final states.

Q.1.(c) Define Decidability.

Ans. A problem with two answers (Yes/No) is decidable if the corresponding language is recursive. In this case, the language L is also called decidable.

A decidable problem is called a solvable problem and an undecidable problem an unsolvable problem by some authors.

Q.1.(d) Define PCP.

Ans. PCP problem : The post Correspondence Problem (PCP) was first introduced by Emil Post in 1946. Later, the problem was found to have many applications in the theory of

formal languages. The problem over an alphabet belongs Σ belongs to a class of yes/no problems and is stated as follows : Consider the two lists $x = (x_1 \ldots x_n)$, $y = (y_1 \ldots y_n)$ of nonempty strings over an alphabet $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exist $i_1, \ldots i_m$, where $1 \leq ij \leq n$, such that

$$x_{i_j} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$$

Q.1.(e) Define Computability.

Ans. Computability : Computability refers to the ability to solve a problem in an effective manner. It is a key topic of the field of computability theory within mathematical logic and computer science. The computability of a problem is closely linked to the existence of an algorithm to solve the problem.

A computable : In computability theory, a Turing reduction from a problem A to a problem B, named after Alan Turing, is a reduction which solves A, assuming B is already known.

Computable : Expressible as a yes/no question that can be answered in any case by a computer in finite time.

Computable Function : Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

The class of computable functions can be defined in many equivalent models, including

- Turing machine
- μ-recursive functions
- Lambda calculus
- Post machines (Post-Turing machines and tag machines).
- Register machines.

Section – A

Q.2.(a) For the following non-deterministic finite automata, make equivalent deterministic finite automata. **(10)**

State/Σ	a	b
$\rightarrow q_0$	q_0, q_1	q_0
q_1	q_2	q_1
q_2	q_3	q_3
q_3	ϕ	q_2

Ans. The deterministic automaton M_1 equivalent to M is defined as follows :

$$M_1 = (2^Q, \{a, b\}, \delta, [q_0], F').$$

where

$$F = \{[q_2], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$$

We start the construction by considering $[q_0]$ first. We get $[q_2]$ and $[q_0, q_1]$. Then we construct δ for $[q_2]$ and $[q_0, q_1]$. $[q_1, q_2]$ is a new state appearing under the input columns. After

constructing δ for $[q_1, q_2]$, we do not get any new states and so we terminate the construction of δ. The state table is given by Table.

Table

State/Σ	*a*	*b*
$[q_0]$	$[q_0, q_1]$	$[q_2]$
$[q_2]$	ϕ	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_0]$	$[q_0, q_1]$

Q.2.(b) Prove that regular sets are closed under concatenation. **(10)**

Ans. Lemma : The class of regular languages is closed under concatenation.

Proof : Prove for arbitrary regular languages L_1 and L_2 that $L_1 . L_2$ is a regular languages.

Let E_1 and E_2 be REGEX accepting L_1 and L_2.

Regex Construction :

We claim the REGEX

$$E = E_1 . E_2$$

accepts $L_1 . L_2$, $L(E_1 . E_2) = L(E_1) . L(E_2)$.

Proof of correctness : Trivial by definition of regular expressions.

$L_1 . L_2$ is regular since there is a REGEX $E_1 . E_2$ accepting this language.

Q.3. What are Mealy/Moore machines ? Explain, can these machines work like one another ? If yes, then explain with an example to convert Moore to Mealy machine. **(20)**

Ans. Mealy machine : A mealy machine is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where all the symbols have the following meaning :

$Q \rightarrow$ finite set of states.

$\Sigma \rightarrow$ i/p alphabet

$\Delta \rightarrow$ o/p alphabet

$\delta \rightarrow$ transition function $\Sigma \times Q$ into Q.

$\lambda \rightarrow$ output function mapping $\Sigma \times Q$ into Δ.

$q_0 \rightarrow$ Initial state

Moore machine : A moore machine is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$

$Q \rightarrow$ finite set of states.

$\Sigma \rightarrow$ input alphabet

$\Delta \rightarrow$ output alphabet

$\delta \rightarrow$ transition function $\Sigma \times Q$ into Q.

$\lambda \rightarrow$ output function mapping Q into Δ.

$q_0 \rightarrow$ Initial state

Yes, these machines can work like one another.

Procedure for transforming a moore machine into a mealy machine : We modify the acceptability of input string by a Moore machine by neglecting the response of the Moore machine to input Λ. We thus define that Mealy Machine *M* and Moore Machine *M'* are equivalent if for all input strings *w*, $bZ_M(w) = Z_{M'}(w)$, where *b* is the output of the Moore machine for its

initial state. We give the following result : Let $M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Moore machine. Then the following procedure may be adopted to construct an equivalent Mealy machine M_2.

Construction : (i) We have to define the output function λ' for the Mealy machine as a function of the present state and the input symbol. We define λ' by $\lambda'(q, a) = \lambda(\delta(q, a))$ for all states q input symbols a.

(ii) The transition function is the same as that of the given Moore machine.

Example : Construct a Mealy Machine which is equivalent to the Moore machine given by Table (1).

Table (1)

Present state	Next state		Output
	$a = 0$	$a = 1$	
$\rightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

We must follow the reverse procedure of converting a Mealy machine into a Moore machine. In the case of the Moore machine, for every input symbol we form the pair consisting of the next state and the corresponding output and reconstruct the table for the Mealy Machine. For example, the states q_3 and q_1 in the next state column should be associated with outputs 0 and 1, respectively. The transition table for the Mealy machine is given by Table (2).

Table (2)

Present state	Next state			
	$a = 0$		$a = 1$	
	state	output	state	output
$\rightarrow q_0$	q_3	0	q_1	0
q_1	q_1	1	q_2	1
q_2	q_2	0	q_3	0
q_3	q_3	0	q_0	0

Section – B

Q.4.(a) What do you mean by Pumping Lemma and applications of pumping Lemma ? Using that concept of pumping lemma prove that $L = \{a^i b^{\,i} \mid i = i^2\}$ is not a CFL. (14)

Ans. Pumping Lemma : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz$, $y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.

Proof : Let $w = a_1 a_2 \ldots a_m$, $m \geq n$

$\delta(q_0, a_1 a_2 \ldots a_i) = q_i$ for $i = 1, 2, \ldots, m$; $Q_1 = \{q_0, q_1, \ldots q_m\}$

That is, Q_1 is the sequence of states in the path with path value $w = a_1 a_2 \ldots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_i and $q_k (q_i = q_k)$. Then i and k satisfy the condition $0 \leq i < k \leq n$.

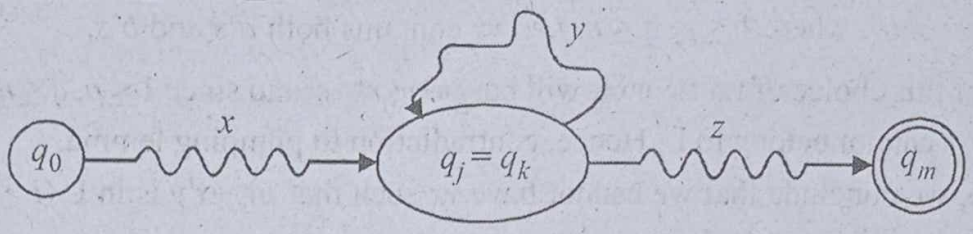


Fig. : String accepted by M.

The string w can be decomposed into three substrings $a_1 a_2 \ldots a_j$, $a_{j-1} \ldots a_k$ and $a_{k-1} \ldots a_m$. Let x, y, z denote these strings $a_1 a_2 \ldots a_j$, $a_{j-1} \ldots a_k$, $a_{k-1} \ldots a_m$, respectively. As $k \leq n$ $|xy| \leq n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in Fig.

The automaton M starts from the initial state q_0. On applying the string x, it reaches $q_j (= q_k)$. On applying the string y, it comes back to $q_j (= q_k)$. So after application of y^i for each $i \geq 0$, the automaton is in the same state q_j. On applying z, it reaches q_m, a final state. Hence, $xy^i z \in L$. As every state in Q_1 is obtained by applying an input symbol. $y \neq \Lambda$.

Application of the Pumping Lemma : The pumping lemma is extremely useful in proving that certain sets are nonregular. The general methodology in its application is an "adversary argument" of the following :

(1) Select the language L, you wish to prove non-regular.

(2) The "adversary" picks n, the constant mentioned in the pumping lemma. Once the adversary has picked n, he may not change it.

(3) Select a string z in L. Your choice may depend implicitly on the value of n chosen.

(4) The adversary breaks z into u, v and w, subject to the constants that $|uv| \leq n$ and $|v| \geq 1$.

(5) You achieve a contradiction to the pumping lemma by showing, for any u, v and w determined by the adversary that there exit i for which $uv^i w$ is not in L. It may then concluded and L is not regular. Your selection of i may depend on n, u, v and w.

Proof : Let n be 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 $1 \le p \le n$, 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) $vx = a^p b^p$, where $1 \le p, q \le n$, *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 $1 \le p, q \le n$, the number of b's is 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 ($i \ge 0$).

Therefore, it is not a CFL.

Q.4.(b) What do you mean by ambiguity ? How do you mean to say a grammar is ambiguous ? **(6)**

Ans. Ambiguity in context-free grammars : Sometimes we come across ambiguous sentences in the language we are using. Consider the following sentence in English: "In books selected information is given." The word 'selected' may refer to books or information. So the sentence may be parsed in two different ways. The same situation may arise in context-free languages. The same terminal string may be the yield of two derivation trees. So there may be two different leftmost derivations of w. This leads to the definition of ambiguous sentences in a context-free language.

Definition : A terminal string $w \in L(G)$ is ambiguous if there exist two or more derivation trees for w (or there exist two or more leftmost derivations of w).

Consider, the example, $G = (\{S\}, \{a, b, +, *\}, P, S)$, where P consists of $S \to S + S \mid S * S \mid a \mid b$. We have two derivation trees for $a + a * b$ given in Fig.

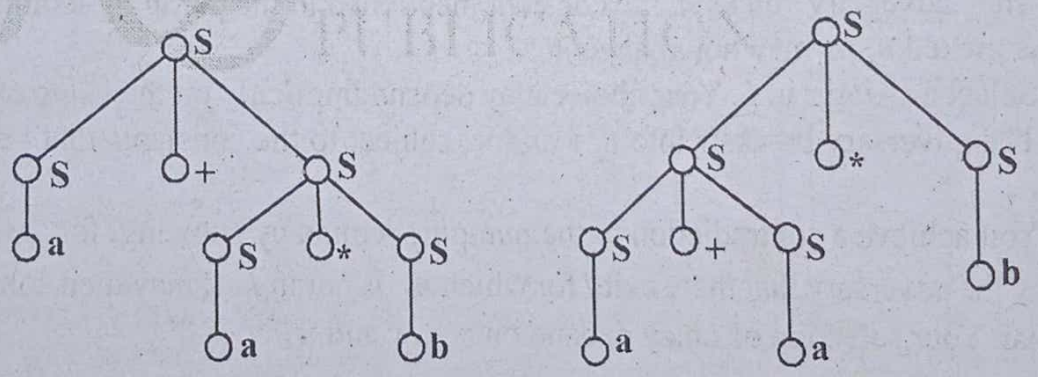


Fig. : Two derivation trees for a + a * b.

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.

Q.5(a) What do you mean by Reduced form of a CFG ? Explain with example.(8)

Ans. Reduced Form : A grammar is said to be in the reduced form if there are productions of the form $w_1 A w_2 \to w_1$, where 'w' does not contain a Non-terminal. This should be applicable for all the Non-terminals that at one point or another Non-terminal, should result in a terminal string.

Consider the following grammar

$S \rightarrow BC$
$S \rightarrow b$
$B \rightarrow b$

Observe that from the start symbol we generate two Non-terminals 'B' and 'C' and out of the 'B' will produce a terminal but we can not eliminate the Non-terminal 'C' from the string such a grammar is not in the reduced form.

Example : Check whether the following grammar is in reduced form.

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

Observe that to terminate the generation process we should use any of the following rules.

$A \rightarrow a$
$C \rightarrow b$

Hence whenever from given Non-terminal we get the Non-terminal 'A' and 'C' we can generate the string only terminals.

Observe that from 'S' we can generate 'CA', hence 'S' will generate a terminal string.

$S \rightarrow CA$
$A \rightarrow a$
$C \rightarrow b.$

Q.5.(b) What are normal forms of CFG ? Explain convert a CFG into CNF.
S → AACD, A → aAb|A, C → aC|a, D → aDa|bDb|A. (12)

Ans. Normal forms for Context-Free Grammars : In a context-free grammer, the R.H.S. of a production can be any string of variables and terminals. When the productions in G satisfy certain restrictions, then G is said to be a 'normal form'. Their are mainly normal forms - the Chomsky normal form (CNF) and the Greibach normal form.

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

Definition : 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 productions are $S \rightarrow AB \mid \Lambda$, $A \rightarrow a$, $B \rightarrow b$. Then G in Chomsky normal form.

Greibach Normal Form : Greibach normal form (GNF) is another normal form quite useful in some proofs and constructions.

Definition : 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 Λ), 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 $S \rightarrow aAB \mid L$, $A \rightarrow bC$, $B \rightarrow b$, $C \rightarrow c$ is in GNF.

Concersion of CFG into CNF : Let G be the grammar with productions

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

Let us go through each step of the conversion to CNF.

(1) Eliminating Λ-productions : The nullable variables are A and D, produces the grammar with productions

$$S \to AACD \mid ACD \mid AAC \mid CD \mid AC \mid C$$
$$A \to aAb \mid ab$$
$$C \to aC \mid a$$
$$D \to aDa \mid bDb \mid aa \mid bb$$

(2) Eliminating unit productions : Here we may simply add the productions

$$S \to aC \mid a$$

and delete $S \to C$.

(3) Restricting the right sides of productions to single terminals or strings of two or more variables : This step yields the productions

$$S \to AACD \mid ACD \mid AAC \mid CD \mid AC \mid X_aC \mid a$$
$$A \to X_aAX_b \mid X_aX_b$$
$$C \to X_aC \mid a$$
$$D \to X_aDX_a \mid X_bDX_b \mid X_aX_a \mid X_bX_b$$
$$X_a \to a$$
$$X_b \to b$$

(4) The final step to CNF : There are six productions whose right sides are too long. Applying our algorithm produces the grammar with productions

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

Section – C

Q.6. Construct a PDA accepting $\{a^n b^m a^n \mid m, n \geq 1\}$ by null store. Construct the corresponding CFG accepting same set. (20)

Ans. The pda A accepting $\{a^n b^m a^n \mid m, n \geq 1\}$ 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

R_1 : $\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$
R_2 : $\delta(q_0, a, a) = \{(q_0, aa)\}$
R_3 : $\delta(q_0, b, a) = \{(q_1, a)\}$
R_4 : $\delta(q_1, b, a) = (\{q_1, a)\}$
R_5 : $\delta(q_1, a, a) = (\{q_1, \Lambda)\}$
R_6 : $\delta(q_1, \Lambda, Z_0) = (\{q_1, \Lambda)\}$

We start storing a's until $a\,b$ occurs (Rules R_1 and R_2). When the current input symbol is b, the state changes, but no change in PDS occurs (Rule R_3). Once all the b's in the input string are exhausted (using Rule R_4), the remaining a's are erased (Rule R_5). Using R_6, Z_0 is erased. So,

$$(q_0, a^n b^m a^n, Z_0) \vdash^* (q_1, \Lambda, Z_0) \vdash (q_1, \Lambda, \Lambda)$$

This means that $a^n b^m a^n \in N(A)$. We can show that

$$N(A) = \{a^n b^m a^n \mid m, n \geq 1\}$$

by using Rules $R_1 - R_6$.

Define $G = (V_N, \{a, b\}, P, S)$, where V_N consists of

$[q_0, Z_0, q_0]$, $[q_1, Z_0, q_0]$, $[q_0, a, q_0]$, $[q_1, a, q_0]$
$[q_0, Z_0, q_1]$, $[q_1, Z_0, q_1]$, $[q_0, a, q_1]$, $[q_1, a, q_1]$

The productions in P are constructed as follows :

The S-productions are

$P_1 : S \to [q_0, Z_0, q_0]$, $P_2 : S \to [q_0, Z_0, q_1]$

$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$ induces

P_3 : $[q_0, Z_0, q_0] \to a[q_0, a, q_0][q_0, Z_0, q_0]$
P_4 : $[q_0, Z_0, q_0] \to a[q_0, a, q_1][q_1, Z_0, q_0]$
P_5 : $[q_0, Z_0, q_1] \to a[q_0, a, q_0][q_0, Z_0, q_1]$
P_6 : $[q_0, Z_0, q_1] \to a[q_0, a, q_1][q_1, Z_0, q_1]$

$\delta(q_0, a, a) = \{(q_1, aa)\}$ yields

P_7 : $[q_0, a, q_0] \to a[q_0, a, q_0][q_0, a, q_0]$
P_8 : $[q_0, a, q_0] \to a[q_0, a, q_1][q_1, a, q_0]$
P_9 : $[q_0, a, q_1] \to a[q_0, a, q_0][q_0, a, q_1]$
P_{10} : $[q_0, a, q_1] \to a[q_0, a, q_1][q_1, a, q_1]$

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

P_{11} : $[q_0, a, q_0] \to b[q_1, a, q_0]$
P_{12} : $[q_0, a, q_1] \to b[q_1, a, q_1]$

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

P_{13} : $[q_1, a, q_0] \to b[q_1, a, q_0]$
P_{14} : $[q_1, a, q_1] \to b[q_1, a, q_1]$

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

P_{15} : $[q_1, a, q_1] \to a$

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

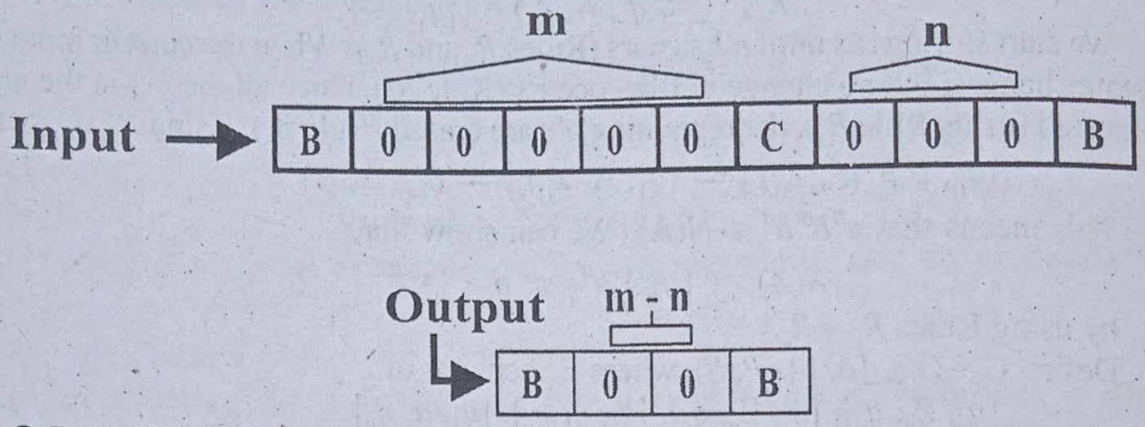
P_{16} : $[q_1, Z_0, q_1] \to \Lambda$.

Q.7.(a) Design a TM for subtraction for $m - n$, where $m > n$ and both $m, n >= 1$. **(10)**

Ans. Step :

- **Step-1 :** If 0 found convert all 0's into 0's and go right then convert C into C and go right.

- **Step-2 :** If X found then convert all X into X and go right or if 0 found then convert 0 into X and go left and go to next step otherwise go to 5th step.
- **Step-3 :** Then convert all X into X and go left then convert C into C and go left
- **Step-4 :** Then convert all 0's into 0's and go left then convert B into B and go right then convert 0 into B and go right and repeat the whole process.
- **Step-5 :** Otherwise if B found convert B into B and go left then convert all X into B and go left then convert C into B and go left and then stop the machine.



Q.7.(b) Write short note on Halting Problem of TM. (10)

Ans. Halting Problem of Turing Machine : This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Theorem : $HALT_{TM} = \{(M, w) |$ The Turing machine M halts on input $w\}$ is undecidable.

Proof : We assume that $HALT_{TM}$ is decidable, and get a contradiction. Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a TM M_2 as follows:

1. For M_2, (M, w) is an input.
2. The TM M_1 acts on (M, w).
3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M_1, w), simulate the TM M on the input string w until M halds.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w reached. In the first case (the first alternative of setp 5) M_2 accepts (M, w). In the second case (the second alternative of setp 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $T(M_2) = \{(M, w) |$ The Turing machine accepts w$\}$

$= A_{TM}$

This is a contradiction since A_{TM} is undecidable.

Section – D

Q.8. What are Primitive Recursive Functions ? Explain and prove that the following functions are primitive recursive : (i) Transpose (ii) Concatenation (iii) Identity. (20)

Ans. Primitive recursive function : A total function f over N is called primitive recursive

(a) if it is one of three initial functions.

(b) if it can be obtained by applying composition & recursion a finite number of time to the set of initial function.

For ex. : $f(x, y) = x^y$ is a primitive recursive function.

(i) Transpose Function : The transpose function can be defined by trans(x) = x^T. Then

$$\begin{aligned} \text{trans}(\Lambda) &= \Lambda \\ \text{trans}(ax) &= \text{concat}(\text{trans}(x), a(x)) \\ \text{trans}(bx) &= \text{concat}(\text{trans}(x), b(x)) \end{aligned}$$

Therefore, trans(x) is primitive recursive.

(ii) Concatenation Function : The concatenation function can be defined by

$$\begin{aligned} \text{concat}(x_1, x_2) &= x_1 x_2 \\ \text{concat}(\Lambda, x_2) &= \text{id}(x_2) \\ \text{concat}(ax_1, x_2) &= \text{cons } a\ (\text{concat}(x_1, x_2)) \\ \text{concat}(bx_1, x_2) &= \text{cons } b\ (\text{concat}(x_1, x_2)) \end{aligned}$$

So concat is defined by recursion using id, cons a and cons b.

Therefore, concat is primitive recursive.

(iii) Identity Function : Let us denote the identity function by id. Then,

$$\begin{aligned} \text{id}(\Lambda) &= \Lambda \\ \text{id}(ax) &= \text{cons } a(x) \\ \text{id}(bx) &= \text{cons } b(x) \end{aligned}$$

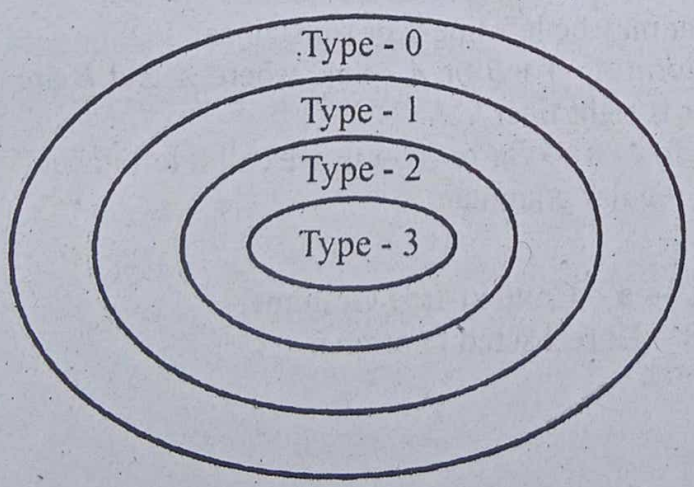
So id is defined by recursion using cons a and cons b. Therefore, the identity function is primitive recursive.

Q.9. Write and briefly explain the characteristics of each class of grammar classified according to Chomsky hierarchy. Determine the type of grammar G :

(i) S → aA, A → aAB, B → b, A → a

(ii) S → aAB, AB → C, A → b, B → AB. **(20)**

Ans. Chomsky hierarchies of grammars : We can exhibit the relationship between grammars by the Chomsky hierarchy. Chomsky provided an initial classification in to four languages type :



Type – 0 (Unrestricted grammar)
Type – 1 (Context sensitive grammar)
Type – 2 (Context free grammar)
Type – 3 (Regular grammar)

Type 0 languages are those generated by unrestricted grammars, that is, the recursively enumerable languages. Type 1 consist of the context-sensitive languages, Type 2 consists of the context-free languages and Type 3 consists of the regular languages. Each language family of type k is a proper subset of the family of type $k - 1$. Following diagram shows the original Chomsky Hierarchy.

Unrestricted (Type -0) Grammars : The unrestricted grammar is defined as

$$G = (V_n, V_t, P, S)$$

where V_n = a finite set of non-terminals
V_t = a finite set of terminals
S = starting non-terminal, $S \in V_n$

and P is set of productions of the following form

$$\alpha \rightarrow \beta$$

where α and β are arbitrary string of grammar symbols with $\alpha \neq \in$. These grammars are known as type-0, phase-structure or unrestricted grammars.

Context Sensitive Grammar : Let $G = (V_n, V_t, P, S)$ be context sentive grammar where V_n = finite set of non-terminals
V_t = finite set of terminals
S = starting non-terminals $S \in V_n$

and P is the set of fules called productions defined as

$$\alpha \rightarrow \beta$$

where β is at least as long as a that is clearly

$$|\alpha| \leq |\beta|$$

The term "Context-sentive" comes from a normal form for these grammars, where each production is of the form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ with $\beta \neq \in$. Replacement of variable A by string β is permitted in the "Context" of α_1 and α_2.

Context-Free Grammars : Mathematically context-free grammar is defined as follows :

"A grammar $G = (V_n, V_t, P, S)$ is said to be context-free"
where V_n = A finite set of non-terminals, generally represented by capital letters $A, B, C, D, \ldots$.
V_t = A finite set of terminals, generally represented by small letters, like $a, b, c, d, e, f, \ldots$.
S = Starting non-terminal, called start symbol of the grammar. S belongs to V_n.
P = Set of rules or production in CFG.

G is context-free and all production in P have the form

$$\alpha \rightarrow \beta$$

where $\alpha \in V_n$ and $\beta \in (V_t \cup V_n)^*$.

Regular Grammars : A regular grammar may be left linear or right linear.

"If all production of a CFG are of the form $A \rightarrow wB$ or $A \rightarrow w$, where A and B are variables and $w \in V_t^*$, then we say that grammar is right liner".

"If all the production of a CFG are of the form $A \rightarrow Bw$ or $A \rightarrow w$, we call it left linear."

A right or left linear grammar is called a regular grammar.

Type of grammar G :

(i) $S \rightarrow aA, A \rightarrow a, A \rightarrow AB, B \rightarrow b, A \rightarrow a$ = Context-free Grammer

(ii) $S \rightarrow aAB, AB \rightarrow c, A \rightarrow b, B \rightarrow AB$ = Unrestricted Grammer