

B.Tech.

Fifth Semester Examination

Theory of Automata Computation (CSE-305-F)

Note : Attempt any five questions.

Q. 1. (a) Define Chomsky Hierarchy of Languages.

10

Ans. In the definition of a grammar (V_N, Σ, P, S) , V_N & Σ are sets of symbols and $S \notin V_N$. So if we want to classify grammars, we have to do it only by considering the form of productions. Chomsky classified the grammars into four types in terms of productions (types 0-3).

A types 0 Grammar is any phrase structure grammar without any restrictions.

To define the other types we need a definition.

In a production of the form $\phi A \psi \rightarrow \phi \alpha \psi$, where A is a variable, ϕ is called the left context ψ , the right context, and $\phi \alpha \psi$ the replacement string.

Example (a) In $ab\Delta bcd \rightarrow ab\Delta Bbcd$, ab is the left context, bcd is the right context, $\alpha = AB$.

(b) In $A\sqsubset \rightarrow A$, A is the left context, \sqsubset is the right context $\alpha = \sqsubset$. The production, simply erases C when the left context is A and right context is \sqsubset .

A production of the form $\phi A \psi \rightarrow \phi \alpha \psi$ is called a type 1 production if $\alpha \neq \sqsubset$. In type 1 production erasing of A is not permitted.

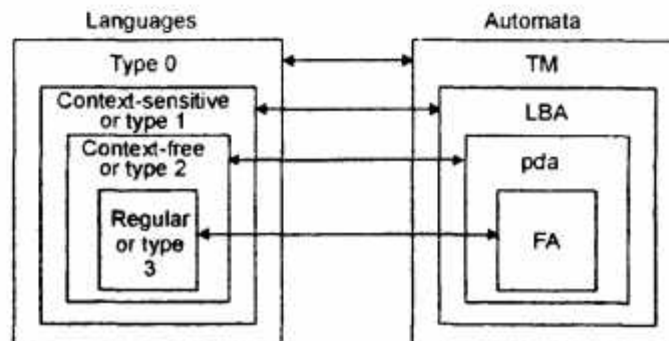
Example : $a\Delta bcD \rightarrow abc\Delta bcD$ is a type 1 production a, bcD are the left context & right context, respectively. A is replaced by $bcD \neq \sqsubset$.

Definition : A grammar is called type 1 or context-sensitive or context-dependent if all its productions are type 1 productions. The production $S \rightarrow \sqsubset$ is also allowed in a type 1 grammar, but in this case S does not appear on the right hand side of any production.

Definition : A type production is a production of the form $A \rightarrow \alpha$, where $A \in V_N$ and $\alpha \in (V_N \cup V_\Sigma)^*$. In other words, the L.H.S. has no left context or right context. For example, $S \rightarrow Aa$, $A \rightarrow a$, $B \rightarrow abc$, $A \rightarrow \sqsubset$ are type 2 productions.

Definition : A grammar is called a type 2 grammar if it contains only type 2 productions. It is also called a context-free grammar.

Definition : A production of the form $A \rightarrow \alpha$ or $A \rightarrow aB$, where $A, B \in V_N$ & $a \in \Sigma$, is called a type 3 production.



Definition : A grammar is called a type 3 or regular grammar if all its productions are type 3 productions. A production $S \rightarrow \Lambda$ is allowed in type 3 grammar, but in this case S does not appear on the right hand side of any production.

Q. 1. (b) Prove that : There is a recursive language that is not context sensitive.

Ans. Proof : Consider the set of all context-sensitive grammars on $T = \{a, b\}$. We can use a convention in which each grammar has a variable set of the form

$$V = \{V_0, V_1, V_2, \dots\}$$

Every context sensitive grammar is completely specified by its production, we can write as a single string.

$$\{x_1 \rightarrow y_1; x_2 \rightarrow y_2; \dots; x_m \rightarrow y_m\}$$

This string we now apply the homomorphism

$$h(a) = 010$$

$$h(b) = 01^20$$

$$h(\rightarrow) = 01^30$$

$$h(;) = 01^40$$

$$h(v_i) = 01^{i+5}0$$

Thus, any context-sensitive grammar can be represented uniquely by a string from $L((011^*0)^*)$. Furthermore, 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 . Next, we define a language L by $L = \{w_j : w_j \text{ defines a context-sensitive grammar } G_j \text{ and } w_j \in L(G_j)\}$. L is well defined and is in fact recursive. To see this, we construct a membership algorithm. Given w_i , we check it to see if it defines a context-sensitive grammar G_i . If not, then $w_i \notin L$. If the string does define a grammar, then $L(G_i)$ is recursive.

But L is not context-sensitive. If it were, there would exist some w_j such that $L = L(G_j)$. If we assume that $w_j \in L(G_j)$, then by definition w_j is not in L . But $L = L(G_j)$, so we have a contradiction. Conversely, if we assume that $w_j \notin L(G_j)$, then by definition $w_j \in L$ and we have another contradiction. We must therefore conclude that L is not context-sensitive.

Q. 2. Describe the context free and context sensitive Grammars. Also explain Grebaich Normal form.

Ans. Context free language is the most important aspect of formal language theory as it applies to program many features that can be described elegantly by means of context-free languages. Context free language has important applications in the design of programming languages as well as in the construction of efficient compilers.

Context Free Grammars : The productions in a regular grammar are restricted in two ways : the left side must be a single variable, while the right side has a special form. To create grammars that are

more powerful, we must relax some of these restrictions. By retaining the restriction on the left side, but permitting anything on the right, we get context-free grammars.

Definition : A grammar $G = (V, T, S, P)$ is said to be context free if all productions in P have the form $A \rightarrow x$

Where $A \in V$ and $x \in (V \cup T)^*$

A language L is said to be context-free if and only if there is a context free Grammars G such that $L = L(G)$

Here V = Variables (Non-Terminal)

T = Terminals

S = Start symbol

P = Productions

Context-Sensitive Grammar :

A grammar $G = (V, T, S, P)$ is said to be context-sensitive if all productions in P have the form

$$x \rightarrow y$$

Where $x \in (V \cup T)^+$ and $y \in (V \cup T)^*$

& $x \geq y$

A language L is said to be context-sensitive if and only if there is a context-sensitive Grammars G such that $L = L(G)$

Here V = Variables (Non-Terminals)

T = Terminals

S = Start symbol

P = Production

Greibach Normal Form : Greibach normal form is a useful grammatical form, we put restrictions not on the length of the right sides of a production, but on the position in which terminals and variables can appear. Greibach normal form are a little complicated and not very transparent. Similarly constructing a grammar in Greibach normal form equivalent to a given context-free grammar is tedious. Greibach normal form has many theoretical and practical consequences.

Definition : A context-free grammar is said to be in Greibach normal form if all productions have the form

$$A \rightarrow ax$$

Where $a \in T$ and $x \in V^*$

For every context-free grammar G with $\lambda \notin L(G)$ there exists an equivalent grammar \hat{G} in Greibach normal form.

Example : The grammar

$$S \rightarrow AB$$

$$S \rightarrow aA \mid bB \mid b$$

$$B \rightarrow b$$

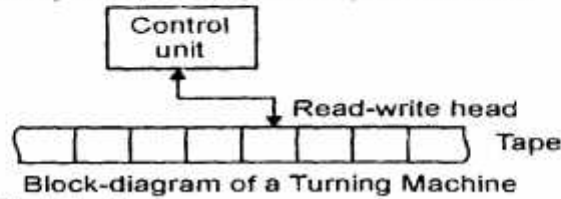
is not in Greibach normal form. Using substitution method, we immediately get the equivalent grammar.

$$S \rightarrow aA\{ \mid bBB \mid bB,$$

$$\begin{aligned} A &\rightarrow aA \mid bB \mid b, \\ B &\rightarrow b \end{aligned}$$

Q. 3. (a) What is Turing machine ? Explain. Also write advantages of TM over FSM.

Ans. A Turing machine is an automation whose temporary storage is a tape. This is divided into cells, each of which is capable of holding one symbol. Associated with the tape is a read-write head that can travel right or left on the tape and that can read and write a single symbol on each move. Turing machine will have neither an input file nor any special output mechanism.



A Turing machine M is defined by

$$M = \{Q, \Sigma, \Gamma, \delta, q_0, f\}$$

Where

Q is the set of internal states,

Σ is the input alphabet,

Γ is a finite set of symbols called the tape alphabet,

δ is the transition function

$\square \in \pi \Gamma$ is a special symbol called the blank,

$q_0 \in Q$ is the initial state,

$f \subseteq Q$ is the set of final states.

In a Turing machine, we assume that $\Sigma \subseteq \Gamma - \{\square\}$, that is, that the input alphabet is a subset of the tape alphabet, not including the blank.

The transition function δ is defined as

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

δ is a partial function on $Q \times \Gamma$; its interpretation gives the principle by which a Turing machine operates. The argument of δ are the current state of the control unit and the current tape symbol being read. The result is a new state of the control unit, a new tape symbol, which replaces the old one, and a move symbol, L or R. The move symbol indicates whether the read-write head moves left or right one cell after the new symbol has been written on the tape.

A Turing machine is an automaton whose temporary storage is tape. But in FSM, there is no way for storing the values.

In Turing machine (read-write head) λ can move either left or right according to information. But there is such no facility for moving the read-write head in any direction.

Q. 3. (b) Design a TM to accept the language :

$$L = \{WCM/W \text{ in } (a + b)^+\}$$

Ans. $(a + b)^+$ represents the set of all string of a's and b's at any position excluding null.

$L = \{abcbab, bacba, aabcaab, babcbab, baacbaa, \dots\}$

We take any string

abcbab

TM = $(\{q, q_1\}, \{a, b\}, z, \delta, q, \square, q_1)$

Designing of a T.M.

$$\delta(q, a) = (q_1, x, R)$$

$$\delta(q, b) = (q, b_1, R)$$

$$\delta(q, c) = (q, c, R)$$

$$\delta(q, a) = (q, x, L)$$

$$\delta(q, c) = (q, c, L)$$

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

$$\delta(q, x) = (q, x, R)$$

$$\delta(q, b) = (q, y, R)$$

$$\delta(q, b) = (q, y, L)$$

$$\delta(q_1, x) = (q, x, L)$$

$$\delta(q, y) = (q, y, R)$$

$$\delta(q, \square) = (q_1, \square, R)$$

Final state is q_1 .

↓ ↓
 abcbab | - xbcab | - xbcab | -
 x, y, c, b यदि में स्पेश बढाकर ↓ को ठीक करें

xbcbab | - xbcab | -
 xycxb | - xycxb | -
 xycxy | - xycxy | -
 xycxy | - xycxy
 | - xycxy | - xycxy
 x y c x y □

This is the configuration of a string

Q. 4. (a) What is minimization Algorithm ?

Ans. Two states q_1 and q_2 are equivalent (denoted by $q_1 \equiv q_2$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states, or both of them are non-final states for all $x \in \Sigma^*$.

- Two states q_1 & q_2 are k -equivalent ($k \geq 0$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both non-final states for all strings x of length k or less. In particular, any two final states are 0-equivalent and any two non-final states are also 0-equivalent.

Minimization Algorithm :

Step 1 : (Construction of π_0). By definition of 0-equivalence, $\pi_0 = \{Q_1^0, Q_2^0\}$, where Q_1^0 is the set of all final states and $Q_2^0 = Q - Q_1^0$.

Step 2 : (Construction of π_{k+1} from π_k). Let Q_i^k be any subset in π_k . If q_1 and q_2 are in Q_i^k , they are $(k+1)$ -equivalent provided $\delta(q_1, a)$ and $\delta(q_2, a)$ are k -equivalent.

Find out whether $\delta(q_1, a)$ and $\delta(q_2, a)$ are in the same equivalence class in π_k for every $a \in \Sigma$. If so, q_1 and q_2 are $(k+1)$ -equivalent. In this way, Q_i^k is further divided into $(k+1)$ equivalence classes. Repeat this for every Q_i^k in π_k to get all the elements of π_{k+1} .

Step 3 : Construct π_n for $n = 1, 2, \dots$ until $\pi_n = \pi_{n+1}$

Step 4 : (Construction of minimum automaton). For the required minimum state automaton, the states are the equivalence classes obtained in Step 3, i.e. the element of π_n . The state table is obtained by replacing a state q by the corresponding equivalence class $[q]$.

Q. 4. (b) State the Myhill-Nerode theorem and its applications.

Ans. Myhill-Nerode theorem is used to eliminate useless states from a DFA.

For theorem, let us see some basics first. An equivalence relation as being true or false for a specific pair of string x and y . Thus $x R y$ is true for some set of pairs x and y ; we will use a relation R such that $x R y \Leftrightarrow y R x$, x has a relation of y if and only if y has the same relation to x . This is known as symmetric $x R y$ & $y R z \Rightarrow x R z$. This is known as transitive

$x R x$ is true. This is known as reflexive.

The notation R_L means an equivalence relation (R) over the language L . The notation R_M means an equivalence relation R over machine M . We know for every regular language L there is a machine M that exactly accepts the strings in L .

R_L is defined $x R_L y \Leftrightarrow$ for all z in Σ^*

$$(xz \text{ in } L \Leftrightarrow yz \text{ in } L)$$

R_M is defined $x R_M y \Leftrightarrow x R_M yz$ for z in Σ^*

$$\begin{aligned} \text{In other words, } \delta(q_0, xz) &= \delta(\delta(q_0, x), z) \\ &= \delta(\delta(q_0, y), z) \\ &= \delta(q_0, yz) \end{aligned}$$

For x, y & z strings in Σ^* .

" R_M divides the set Σ^* into equivalence classes, one class for each state reachable in that particular state of M from the starting state q_0 . To get R_L from this we have to consider only the final reachable state of M ."

Statement (Applications) of the Myhill-Nerode Theorem :

1. The set L , a subset of Σ^* is accepted by a DFA. (We know this means L is regular language).
2. L is the union of some of the equivalence classes of a right invariant (with respect to concatenation) equivalence relation of finite index.
3. Let equivalence relation R_L be defined by : $x R_L y$ if and only if for all z in Σ^* , xz is in L exactly, when yz is in L . Then R_L is of finite index.

Q. 4. (c) What is Pumping Lemma ? Discuss the application of Pumping Lemma with examples.

Ans. When we give a necessary condition for an input string to belong to a regular set. The result is

called pumping lemma as it gives a method of pumping (generating) many input strings from a given string. As pumping lemma gives a necessary condition, it can be used to show that certain sets are not regular.

Pumping Lemma : Let L be an infinite regular language. Then there exists some positive integer m such that any $w \in L$ with $|w| \geq m$ can be decomposed as

$$w = xyz,$$

With

$$|xy| \leq m, \text{ and } |y| \geq 1,$$

Such that, $w_i = xy^i z$ is also in L for all $i = 0, 1, 2, \dots$

Every sufficient long string in L can be broken into three parts in such a way that an arbitrary number of repetitions of the middle part yields another string in L . We say that the middle string is "pumped" hence the term pumping lemma for this result.

Application of Pumping Lemma : This theorem can be used to prove that certain sets are not regular. We now give the steps needed for proving that a given set is not regular.

Step 1 : Assume L is regular. Let n be the number of states in the corresponding F_A (finite automaton).

Step 2 : Choose a string w such that $|w| \geq n$, use pumping lemma to write $w = xyz$, with $|xy| \leq n$ and $|y| > 0$.

Step 3 : Find a suitable integer i such that $xy^i z \notin L$. This contradicts our assumption. Hence L is not regular.

Example : Show that $L = \{a^p \mid P \text{ is a prime}\}$ is not regular.

Solution :

Step 1 : We suppose L is regular and get a contradiction. Let n be the number of states in the FA accepting L .

Step 2 : Let p be a prime number greater than n . Let $w = a^p$. By pumping lemma, we can be written as $w = xyz$, with $|xy| \leq n$ and $|y| > 0$. x, y, z are simply strings of a 's. So, $y = a^m$ for some $m \geq 1$ (and $\leq n$).

Step 3 : Let $i = p + 1$. Then $|xy^i z| = |xyz| + |y^{i-1}| = p + (i-1)m = p + pm$.

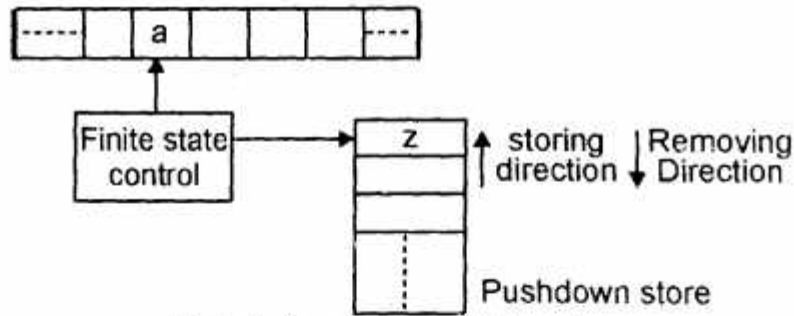
By pumping lemma, $xy^i z \in L$. But $|xy^i z| = p + pm = p(1+m)$ and $p(1+m)$ is not a prime. So $xy^i z \notin L$. This is a contradiction. Thus L is not regular.

Q. 5. Explain Push down Automata and their applications in detail.

Ans. A finite automaton cannot accept L , i.e. strings of the form $a^n b^n$ as it has to remember the number of a 's in a string and so it will require infinite number of states. This difficulty can be avoided by adding an auxiliary memory in the form of a 'stack'. The a 's in the given string are added to the stack. When the symbol b is encountered in the input string an a is removed from the stack. Thus the matching of number of a 's and numbers of b 's is accomplished. This type of arrangement where a finite automaton has a stack leads to the generation of a pushdown automaton.

Definition : A pushdown automaton consists of seven types

$$A = \{Q, \Sigma, \delta, q_0, z_0, F\}$$



Model of push down Automaton

- (i) a finite non empty set of states denoted by Q ,
- (ii) A finite non-empty set of input symbols denoted by Σ .
- (iii) A finite nonempty set of pushdown symbol denoted by z .
- (iv) q_0 is initial state.
- (v) z_0 is the starting symbol of stack.
- (vi) F is the set of final states
- (vii) The transition function δ from $Q \times (\Sigma \cup \{ \wedge \}) \times \tau$ to the set of finite subsets of $Q \times z^*$

Application of Pushdown Automaton :

- (1) If L is a context-free language, then we can construct a pda A accepting L by empty store, i.e., $= N(A)$.

Proof : We construct A by making use of productions in G .

Step 1 : (Construction of A). Let $L = L(G)$, Where $G = (V_N, T, P, S)$ is a context-free grammar.

We construct a pda A as

$$A = (\{q\}, \Sigma, V_N \cup \Sigma, \delta, q, \delta, \phi)$$

Where δ is defined by the following rules :

$$R_1 : \delta(q, \wedge, A) = \{(q, \alpha) \mid A \rightarrow \alpha \text{ is in } P\}$$

$$R_2 : \delta(q, a, a) = \{(q, \wedge)\} \text{ for every } a \text{ in } \Sigma.$$

Example : Construct a pda A equivalent to the following context-free grammar :

$$S \rightarrow 0BB, B \rightarrow 0S \mid 1S \mid 0$$

Test whether 010^4 is $N(A)$.

Solution : Define pda A as follows :

$$A = (\{q\}, \{0, 1\}, \{S, B, 0, 1\}, \delta, q, S, \phi)$$

$$R_1 : \delta(q, \wedge, S) = \{(q, 0BB)\}$$

$$R_2 : \delta(q, \wedge, B) = \{(q, 0S), (q, 1S), (q, 0)\}$$

$$R_3 : \delta(q, 0, 0) = \{(q, \wedge)\}$$

$$R_4 : \delta(q, 1, 1) = \{(q, \wedge)\}$$

$$(q, 010^4, S) \vdash (q, 010^4, 0BB) \text{ by Rule } R_1$$

$$\vdash (q, 10^{10}, BB) \text{ by Rule } R_3$$

$| - (q, 10^{10}, 1SB)$ by Rule R_2

$| - (q, 0^4, SB)$ by Rule R_4

$| - (q, 0^4, 0BBB)$ by Rule R_1

$| - (q, 0^3, BBB)$ by Rule R_3

$| \dot{-} (q, 0^3, 000)$ by Rule R_2

$| \dot{-} (q, \wedge, \wedge,)$ by Rule R_3

Thus $010^4 \in N(A)$

(2) If $A = (Q, \Sigma, z, \delta, q_0, z_0, F)$ is a pda, then there exists a context-free grammar G such that $L(G) = N(A)$.

Proof : We first give the construction of G and then prove that $N(A) = L(G)$.

Step 1 : (Construction of G). We define $G = (V_N, \Sigma, P, S)$.

Where

$$V_N = \{S\} \cup \{[q, z, q'] \mid q, q' \in Q, z \in \Sigma^*\}$$

The productions in P are induced by moves of pda as follows :

R_1 : S-productions are given by $S \rightarrow [q_0, z_0, q]$ for every q in Q .

R_2 : Each move erasing a pushdown symbol given by $(q', \wedge) \in \delta(q, a, z)$ induces the production $[q, z, q'] \rightarrow a$.

R_3 : Each move not erasing a pushdown symbol given by $(q_1, z_1 z_2 \dots z_m) \in \delta(q, a, z)$ induces many production of the form,

$$[q, z, q'] \rightarrow a[q_1, z_1][q_2, z_2][q_3, z_3] \dots [q_m, z_m, q']$$

Where each of the states q', q_2, \dots, q_m can be any state in Q . Each move yields many productions because of R_3 .

Example : Construct a context free grammar G which accepts $N(A)$, where

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

δ is given by $\delta(q_0, b, z_0) = \{q_0, z, z_0\}$

$$\delta(q_0, \wedge, z_0) = \{(q_0, \wedge)\}$$

$$\delta(q_0, b, z) = \{(q_0, z z)\}$$

$$\delta(q_0, a, z) = \{(q_1, z)\}$$

$$\delta(q_1, a, z) = \{(q_1, \wedge)\}$$

$$\delta(q_1, a, z_0) = \{(q_0, z_0)\}$$

Solution : Let $G = (V_N, \{a, b\}, P, S)$

Where V_N consists of $S, [q_0, z_0, q_0], [q_0, z_0, q_1], [q_0, z, q_0]$

$$[q_0, z, q_1], [q_1, z_0, q_0], [q_1, z_0, q_1], [q_1, z, q_0], [q_1, z, q_1]$$

The productions are

$$P_1 : S \rightarrow [q_0, z_0, q_0]$$

$$P_2 : S \rightarrow [q_0, z_0, q_1]$$

$$\delta(q_0, b, z_0) = \{(q_0, z z_0)\} \text{ yields}$$

$$P_3 : [q_0, z_0, q_0] \rightarrow b[q_0, z, q_0][q_0, z_0, q_0]$$

$$P_4 : [q_0, z_0, q_0] \rightarrow b[q_0, z, q_1][q_1, z_0, q_0]$$

$$P_5 : [q_0, z_0, q_1] \rightarrow b[q_0, z, q_0][q_0, z_0, q_1]$$

$$P_6 : [q_0, z_0, q_1] \rightarrow b[q_0, z, q_1][q_1, z_0, q_1]$$

$$\delta(q_0, \wedge, z_0) = \{(q_0, \wedge)\} \text{ gives}$$

$$P_7 : [q_0, z_0, q_0] \rightarrow \wedge$$

$$\delta(q_0, b, z) = \{(q_0, z z)\} \text{ gives}$$

$$P_8 : [q_0, z, q_0] \rightarrow b[q_0, z, q_0][q_0, z, q_0]$$

$$P_9 : [q_0, z, q_0] \rightarrow b[q_0, z, q_1][q_1, z, q_0]$$

$$P_{10} : [q_0, z, q_1] \rightarrow b[q_0, z, q_0][q_0, z, q_1]$$

$$P_{11} : [q_0, z, q_1] \rightarrow b[q_0, z, q_1][q_1, z, q_1]$$

$$\delta(q_0, a, z) = \{(q_1, z)\} \text{ yields}$$

$$P_{12} : [q_0, z, q_0] \rightarrow a[q_1, z, q_0]$$

$$P_{13} : [q_0, z, q_1] \rightarrow a[q_1, z, q_1]$$

$$\delta(q_1, b, z) = \{(q_1, \wedge)\} \text{ gives}$$

$$P_{14} : [q_1, z, q_1] \rightarrow b$$

$$\delta(q_1, a, z_0) = \{(q_0, z_0)\} \text{ gives}$$

$$P_{15} : [q_1, z_0, q_0] \rightarrow a[q_0, z_0, q_1]$$

$$P_{16} : [q_1, z_0, q_1] \rightarrow a[q_0, z_0, q_1]$$

$P_1 - P_{16}$ give the productions in P .

Q. 6. The following grammar generates prefix expressions with operations x and y and binary operators $+$, $-$, and $*$,

$$E \rightarrow +EE/*EE/-EE/x/y$$

(a) Find left most and right most derivations and a derivation tree for the string

$$+* - xy xy$$

(b) Prove that this grammar is unambiguous.

$$\text{Ans. } E \Rightarrow +EE/*EE/-EE \mid x \mid y$$

Infix :

$$E \Rightarrow E + E$$

$$E \Rightarrow E * E$$

$$E \Rightarrow E - E$$

$$E \Rightarrow x/y$$

(a) Left most derivation for the string

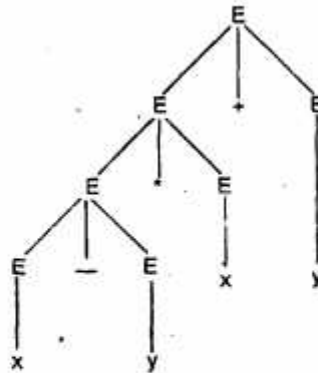
$$+* - x y xy$$

$$\text{Infix} \Rightarrow x - y * x + y$$

$$E \Rightarrow \overset{\downarrow}{E} + E \Rightarrow \overset{\downarrow}{E} * E + E \Rightarrow \overset{\downarrow}{E} - E * E + E$$

$$\Rightarrow x - E * E + E \Rightarrow x - y * E + E$$

$$\Rightarrow x - y * x + E \Rightarrow x - y * x + y$$

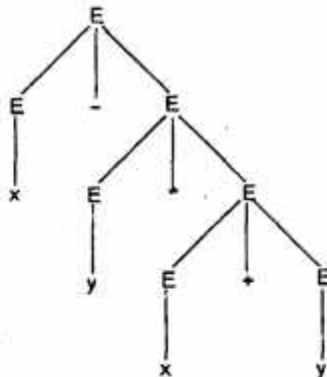


For Rightmost Derivation

$$E \Rightarrow E - E \Rightarrow E - E * E \Rightarrow E - E * E + E$$

$$\Rightarrow E - E * E + y \Rightarrow E - E * x + y \Rightarrow E - y * x + y$$

$$\Rightarrow x - y * x + y$$



(b) **This Grammar is Unambiguous** : Because, context-free Grammar G is said to be ambiguous if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

And this grammar generates only one left derivation tree and right derivation tree.

Q. 7. Let L be a set accepted by NFA. Then prove that there exists a deterministic finite automaton that accepts L. Construct a DFA for the following NFA :

$$M = (\{q_0, q_1\}, \{0, 1\}, s, q_0, \{q_1\})$$

$$\text{Where } S = (q_0, 0) = \{q_0, q_1\}, S(q_0, 1) = \{q_1\}, S(q_1, 0) = \emptyset, S(q_1, 1) = \{q_0, q_1\}$$

Ans. Theorem : For every NFA, there exists a DFA which simulates the behaviour of NFA. Alternatively, if L is the set accepted by NFA, then there exists a DFA which also accepts L.

Proof : Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA accepting L. We construct a DFA M' as follows :

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

Where

(i) $Q' = Q^*$ (any state in Q' is denoted by $[q_1, q_2, \dots, q_j]$ where $q_1, q_2, \dots, q_j \in Q$):

(ii) $q_0' = [q_0]$

(iii) F' is the set of all subsets of Q containing an element of F .

(iv) $\delta'([q_1, q_2, \dots, q_j], a) = \{q_1, a\} \cup \delta(q_2, a) \cup \dots \cup \delta(q_j, a)$

Equivalently, $\delta'([q_1, q_2, \dots, q_j], a) = [p_1, \dots, p_j]$ if and only if

$$\delta(\{q_1, \dots, q_j\}, a) = \{p_1, p_2, \dots, p_j\}$$

Before proving $L = T(M')$, we prove an auxiliary result :

$$\delta'(q_0', x) = [q_1, \dots, q_j] \quad \dots(1)$$

If and only if $\delta(q_0, x) = \{q_1, \dots, q_j\}$ for all x in Σ^* we prove by induction on $|x|$, the "if" part, i.e

$$\delta'(q_0', x) = [q_1, q_2, \dots, q_j], \text{ if } \delta(q_0, x) = \{q_1, \dots, q_j\} \quad \dots(2)$$

When $|x| = 0$, $\delta(q_0, \Lambda) = \{q_0\}$ and by definition of δ' ,

$$\delta'(q_0', \Lambda) = q_0' = [q_0]$$

So equation (2) is true for x with $|x| = 0$. Thus there is basis for induction.

Assume equation (2) is true for all strings y with $|y| \leq m$. Let x be a string of length $m+1$. We can write x as ya . Where $|y| = m$ and $a \in \Sigma$. Let $\delta(q_0, y) = \{p_1, \dots, p_j\}$ and $\delta(q_0, ya) = \{r_1, r_2, \dots, r_k\}$. As $|y| \leq m$ by induction hypothesis, we have

$$\delta'(q_0', y) = [p_1, \dots, p_j] \quad \dots(3)$$

$$\text{Also } \{r_1, r_2, \dots, r_k\} = \delta(q_0, ya) = \delta(\delta(q_0, y), a) = \delta(\{p_1, \dots, p_j\}, a)$$

By definition of δ' .

$$\delta'([p_1, \dots, p_j], a) = [r_1, \dots, r_k] \quad \dots(4)$$

$$\begin{aligned} \text{Hence } \delta'(q_0', ya) &= \delta'(\delta'(q_0', y), a) = \delta'([p_1, \dots, p_j], a) \text{ by equation (3)} \\ &= [r_1, \dots, r_k] \text{ by equation (4)} \end{aligned}$$

Thus, we have proved equation (2) for $x = ya$

By induction equation (2) is true for all strings x . The other part (i.e. only if" part) can be proved similarly, and so equation (1) is established.

Now, $x \in T(M)$ if and only if $\delta(q_0, x)$ contains a state of F . By equation (1), $\delta(q_0, x)$ contains a state of F if and only if $\delta'(q_0', x)$ is in F' . Hence, $x \in T(M)$ if and only if $x \in T(M')$. This proves that DFA M' accepts L .

Example : Construct a DFA for the following NFA.

$$M = (\{q_0, q_1\}, \{0, 1\}, S, q_0, \{q_1\})$$

$$\text{Where } S(q_0, 0) = \{q_0, q_1\} \quad S(q_0, 1) = \{q_1\}$$

$$S(q_1, 0) = \phi$$

$$S(q_1, 1) = \{q_0, q_1\}$$

NFA			→	DFA			I/P
State/I/P	0	1		States	0	1	
→ q ₀	{q ₀ , q ₁ }	{q ₁ }	→	[q ₀]	[q ₀ , q ₁]	[q ₁]	
q ₁	ϕ	{q ₀ , q ₁ }		[q ₁]	ϕ	[q ₀ , q ₁]	Transition Function
				[q ₀ , q ₁]	[q ₀ , q ₁]	[q ₀ , q ₁]	

- (1) Set of states are [q₀], [q₁], [q₀, q₁]
- (2) Initial state is [q₀]
- (3) Transition function (δ) are given above.
- (4) Final states are [q₁], [q₀, q₁]
- (5) Input alphabets are {0, 1}.

Q. 8. Write notes on the following :

- (a) Universal Turing Machine
- (b) Moore Machine
- (c) Context Sensitive Language

Ans. (a) Universal Turing Machine : "A Turing machine is a special purpose computer. Once δ is defined, the machine is restricted to carrying out one particular type of computation. Digital computer, on the other hand, are general purpose machines that can be programmed to do different jobs at different times. Consequently, Turing machines cannot be considered equivalent to general purpose digital computers".

This objection can be overcome by designing a reprogrammable Turing machine, called a universal Turing machine.

A Universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w , can simulate the computation of M on w . To construct such an M_u , we first choose a standard way of describing Turing machines. We may, without loss of generality assume that

$$Q = \{q_1, q_2, \dots, q_n\}$$

With q_1 the initial state, q_2 the single final state, and $z = \{a_1, a_2, \dots, a_m\}$, where a represents the blank. We then, select an encoding in which q_1 is represented by 1, q_2 is represented by ", and so on. Similarly a_1 is encoded as 1, a_2 as 11 etc.

The symbol 0 will be used as a separator between the 1's with the initial and final state and the blank defined by this convention, any Turing machine can be described completely with δ only. The

transition function is encoded according to this scheme, with the arguments and result in some prescribed sequence. For example, $\delta(q_1, q_2) = (q_2, q_3, L)$ might appear as

.....10110110111010.....

(b) **Moore Machine** : The finite automata which have binary output, i.e. they accept the string or 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 other alphabet. The value of 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 Moore machine. It is more convenient to use Moore machine in automata theory.

Definitions : The Moore machine is a six tuple

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

- (1) Q is a finite set of states;
- (2) Σ is the input alphabet;
- (3) Δ is the output alphabet;
- (4) δ is the transition function $\Sigma \times Q$ into Q ;
- (5) λ is the output function mapping Q into Δ ; and
- (6) q_0 is the initial state.

Example :

Present State	Next State δ		Output
	$a = 0$	$a = 1$	λ
$\rightarrow q_0$	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

For the input string 0111, the transition of states is given by $q_0 \rightarrow q_3 \rightarrow q_0 \rightarrow q_1 \rightarrow q_2$. The output string is 00010.

For a Moore machine if the input string is of length n , the output string is of length $n + 1$.

(c) **Context-Sensitive Language** : Between the restricted, context-free grammars and the general, unrestricted grammars a great variety of "somewhat restricted" grammars can be defined. Not all cases yield interesting results; among the ones that do, the context-sensitive grammars have received considerable attention. These grammars generate languages associated with a restricted class of Turing machines, linear bounded automata.

\Rightarrow A grammar $G = (V, T, S, P)$ is said to be Context-Sensitive if all productions are of the form

$$x \rightarrow y$$

Where

$$x, y \in (V \cup T)^+$$

$$\& \quad |x| \leq |y|$$

This definition shows that, it is non-contracting, in the sense that the length of successive sentential forms can never decrease.

A language L is said to be context-sensitive if there exists a context-sensitive grammar G , such that $L = L(G)$.

Example : The language $L = \{a^n b^n c^n : n \geq 1\}$ is a context-sensitive language. We show this by exhibiting a context-sensitive grammar for the language. One such grammar is

$$\begin{aligned} S &\rightarrow abc \mid aAbc, Ab \rightarrow bA, Ac \rightarrow Bbcc, \\ bB &\rightarrow Bb, aB \rightarrow aa \mid aaA \end{aligned}$$

We can see how this works by looking at a derivation of $a^3 b^3 c^3$

$$\begin{aligned} a &\Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \Rightarrow aBbbcc \Rightarrow \\ &\Rightarrow aaAbbcc \Rightarrow aabAbcc \Rightarrow aabbAcc \Rightarrow aabbBbcc \\ &\Rightarrow aabBbbcc \Rightarrow aaBbbbcc \Rightarrow aaabbcc \end{aligned}$$