Course Code: 20CS0903 **R20**

SIDDHARTH INSTITUTE OF ENGINEERING & TECHNOLOGY:PUTTUR
(AUTONOMOUS)
Siddharth Nagar, Narayanavanam Road – 517583

QUESTION BANK (DESCRIPTIVE)

Subject with Code: AT & CD(20CS0903) **Course & Branch**: B.Tech – CSM,CIC

Year &Sem: III-B.Tech & I-Sem **Regulation:** R20

UNIT –I
FINITE AUTOMATA AND REGULAR LANGUAGES

1	a	Consider the below finite automata and check whether the strings are accepted or not	L1][CO1]	[8M]

States (Q)	Input Alphabtes	
	0	1
→q0	q1	q3
q1	q0	q2
(q2)	q3	q1
q3	q2	q0

(i) 0001 (ii) 1010 (iii) 1001 (iv)0101

Sol:- (i) 0001

$\delta(q_0, 0001) \rightarrow \delta(q_1, 001)$
$\rightarrow \delta(q_0, 01)$
$\rightarrow \delta(q_1, 1)$
$\rightarrow q_2 \in F$

$\therefore q_2$ is the final state of the FA
So, the string '0001' is accepted by the given FA.

(ii) 1010

$\delta(q_0, 1010) \rightarrow \delta(q_3, 010)$

(3)

$\rightarrow \delta(q_2, 10)$
$\rightarrow \delta(q_1, 0)$
$\rightarrow q_0 \notin F$

$\therefore q_0$ is a non-final state, so the string 1010 is not accepted by the given FA.

(iii) 1001

$\delta(q_0, 1001) \rightarrow \delta(q_3, 001)$
$\rightarrow \delta(q_2, 01)$
$\rightarrow \delta(q_3, 1)$
$\rightarrow q_0 \notin F$

$\therefore q_0$ is a non-finate state, so the string 1001 i not accepted by the given FA.

(iv) 0101

$\delta(q_0, 0101) \rightarrow \delta(q_1, 101)$
$\rightarrow \delta(q_2, 01)$
$\rightarrow \delta(q_3, 1)$
$\rightarrow q_2 \in F.$

$\therefore q_2$ is a finate, so the string 0101 is accepted by the given FA.

	b	Define alphabets, strings, Languages? **Alphabet** • Definition − An alphabet is any finite set of symbols. • Example − ∑ = {a, b, c, d} is an alphabet set where ‘a’, ‘b’, ‘c’, and ‘d’ are symbols. **String** • Definition − A string is a finite sequence of symbols taken from ∑. • Example − ‘cabcad’ is a valid string on the alphabet set ∑ = {a, b, c, d} **Length of a String** • Definition − It is the number of symbols present in a string. (Denoted by |S|). • Examples − o If S = ‘cabcad’, |S|= 6 o If |S|= 0, it is called an empty string (Denoted by λ or ε) **Language** • Definition − A language is a subset of ∑* for some alphabet ∑. It can be finite or infinite. • Example − If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, aa, ba, bb }	[L3][C01]	[4M]

2	a	Compare DFA and NFA	[L2][CO1]	[4M]

DFA:-

DFA is a finite automata where, for all cases, when a single input is given to a single state, the machine goes to a single state, i.e., all the moves of the machine can be uniquely determined by the present state and input symbol.

A DFA can be represented as

$$M_{DFA} = \{Q, \Sigma, \delta, q_0, F\}$$

Q - set of states

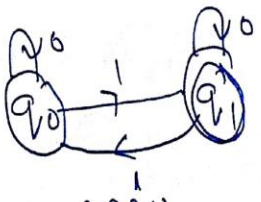
Σ - input Alphabets

δ - Transition function mapping from

$$Q \times \Sigma \rightarrow Q \quad \text{where } |Q| = 1.$$

q_0 - begining state

F - final state. eg:



(DFA)

NFA:-

NFA is a finite automata, where for some cases when a single input is given to a single state, the machine goes to more than one states, i.e, some of the moves of the machine can't be uniquely determined by the present state and the present input symbol.

An NFA can be represented as

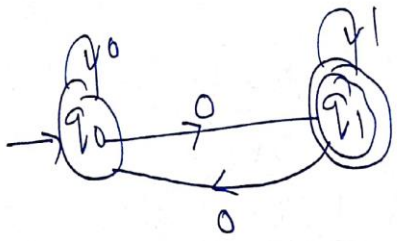
$M_{NFA} = \{Q, \Sigma, \delta, q_0, F\}$

where Q - set of states

Σ - i/p Alphabets

δ - Transition function mapping from $Q \times \Sigma \rightarrow 2^Q$, where 2^Q is power set of Q.

Eg:-



(NFA)

b Construct DFA for the given NFA [L6][CO2] [8M]

	Next state	
	0	1
→ q0	q0,q1	q0
q1	q2	q1
q2	q3	q3
(q3)	-	q2

Sol

* first construct the Transition table
* start from the begining state of the NFA, take states with in [].
* place the Next states for the begining state for given inputs in the Next state column, put them also in [].

Present states	Next state	
	0	1
$[q_0]$	$[q_0q_1]$	$[q_0]$
$[q_0q_1]$	$[q_0q_1q_2]$	$[q_0q_1]$
$[q_0q_1q_2]$	$[q_0q_1q_2q_3]$	$[q_0q_1q_3]$
$[q_0q_1q_2q_3]$	$[q_0q_1q_2q_3]$	$[q_0q_1q_2q_3]$
$[q_0q_1q_3]$	$[q_0q_1q_2]$	$[q_0q_1q_2]$

(5)

The begining state is $[q_0]$

The final states are $[q_0q_1q_2q_3]$, $[q_0q_1q_3]$

∴ The Transition Diagram for constructed DFA is

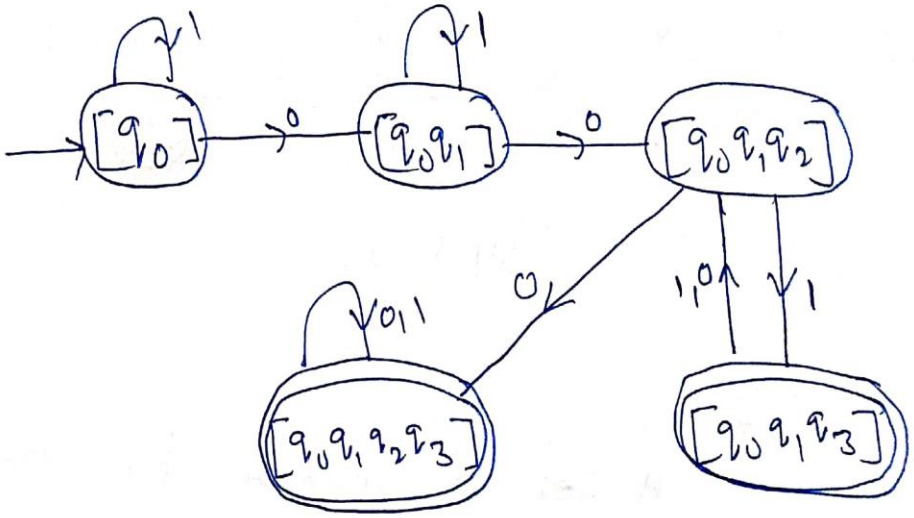


fig: The Transition Diagram for the DFA.

Course Code: 20CS0903 **R20**

3	a	Write the process of conversion from NFA with ε moves to DFA.	[L4][CO3]	[4M]

Let assume that in the NFA we want to remove the ε move which exists S_1 and S_2. This can be removed in the following way

1) Find all ε-closures of the states in the given NFA-ε

$$\varepsilon\text{-closure}(q_0) = \{P_1, P_2, \ldots P_n\}$$

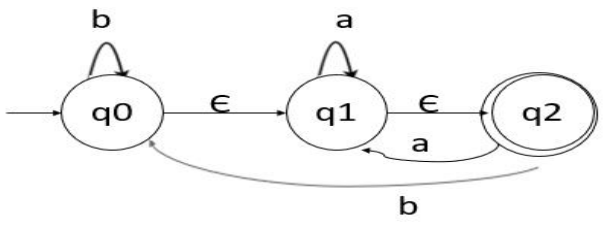
2) Finding δ' transitions on $[P_1, P_2 \ldots P_n]$ for each input $\delta(\{P_1, P_2 \ldots P_n\}, a)$ by using the following rule

$$\boxed{\delta'(q, a) = \varepsilon\text{-closure}(\delta(q, a))}$$

3) The states obtained $[P_1, P_2, P_3, \ldots P_n] \in Q_D$. The states contains final state in P_i is a final state in DFA.

4) The beginning state of the constructed DFA will be same as beginning of the given NFA-ε.

	b	Convert the following NFA with ε moves to DFA.	[L6][CO2]	[8M]



Sol:-

finding ε-closure for all the states

ε-closure$(q_0) = \{q_0, q_1, q_2\}$

ε-closure$(q_1) = \{q_1, q_2\}$

ε-closure$(q_2) = \{q_2\}$

δ- Transition

i) $\delta'(q_0, a) = \varepsilon$-closure$(\delta(q_0, a))$

$= \varepsilon$-closure(ϕ)

$\boxed{\delta'(q_0, a) = \phi}$

ii) $\delta'(q_0, b) = \varepsilon$-closure$(\delta(q_0, b))$

$= \varepsilon$-closure(q_0)

$\boxed{\delta'(q_0, b) = \{q_0, q_1, q_2\}}$ new state

iii) $\delta'(q_1, a) = \varepsilon$-closure$(\delta(q_1, a))$

$= \varepsilon$-closure(q_1)

$\boxed{\delta'(q_1, a) = \{q_1, q_2\}}$ new state

iv) $\delta'(q_1, b) = \varepsilon$-closure$(\delta(q_1, b))$

$= \varepsilon$-closure(ϕ)

$\boxed{\delta'(q_1, b) = \phi}$

v) $\delta'(q_2, a) = \varepsilon$-closure$(\delta(q_2, a))$

$= \varepsilon$-closure(q_1)

$\boxed{\delta'(q_2, a) = \{q_1, q_2\}}$ Repeated state

vi) $\delta'(q_2, b) = \varepsilon\text{-closure}(\delta(q_2, b))$

$= \varepsilon\text{-closure}(q_0)$

$= \varepsilon\text{-closure}($

$\boxed{\delta'(q_2, b) = \{q_0, q_1, q_2\}}$ Repeated

vii) $\delta'(\{q_0, q_1, q_2\}, a) = \varepsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$

$= \varepsilon\text{-closure}(\phi \cup q_1 \cup q_1)$

$= \varepsilon\text{-closure}(q_1)$

$\boxed{\delta'(\{q_0, q_1, q_2\}, a) = \{q_1, q_2\}}$ Repeated

viii) $\delta'(\{q_0, q_1, q_2\}, b) = \varepsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, b) \cup \delta(q_2, b))$

$= \varepsilon\text{-closure}(q_0 \cup \phi \cup q_0)$

$= \varepsilon\text{-closure}(q_0)$

$= \varepsilon\text{-closure}(q_0)$

$\boxed{\delta'(\{q_0, q_1, q_2\}, b) = \{q_0, q_1, q_2\}}$ Repeated

ix) $\delta'(\{q_1, q_2\}, a) = \varepsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$

$= \varepsilon\text{-closure}(q_1 \cup q_1)$

$= \varepsilon\text{-closure}(q_1)$

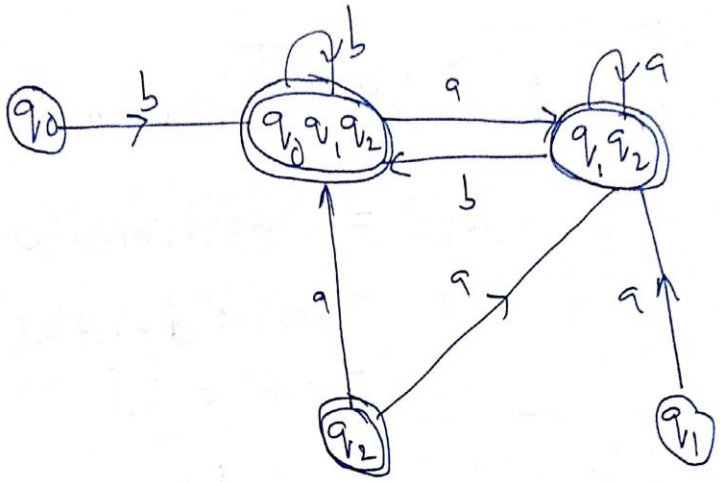
$\boxed{\delta(\{q_1, q_2\}, a) = \{q_1, q_2\}}$ Repeated

x) $\delta'(\{q_1, q_2\}, b) = \varepsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$

$= \varepsilon\text{-closure}(\phi \cup q_0)$

$= \varepsilon\text{-closure}(q_0)$

$\boxed{\delta'(\{q_1, q_2\}, b) = \{q_0, q_1, q_2\}}$ Repeated

		∴ The Transition Diagram for the DFA is  fig: DFA for the given NFA		
4	a	Write the process of equivalence two FA's?	[L4][CO3]	[4M]

Equivalance of Finite Automata's

Two FA's are said to be equivalent if they generate the same R.E. If the both FA are minimized, they reveal the equivalance by being minimized to the same number of states and same transitional functions.

The process of proving the Equivalance between two FA's

Let there are two FA, M and M' where the numb of inputs symbols are the same.

1) make a comparison table with n+1 columns, where n is the numb of input symbols

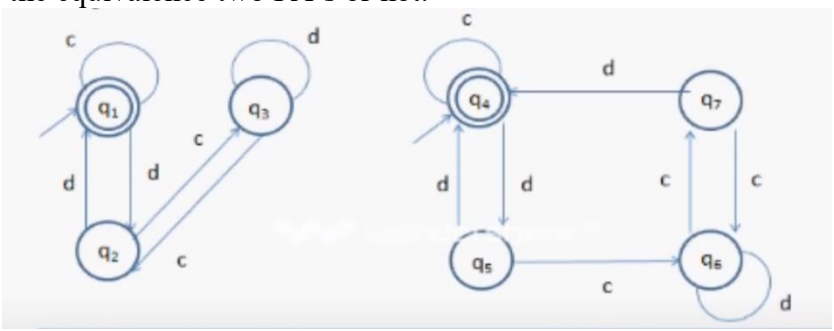
2) In the first column, there will be a pair of vertices (q, q') where $q \in M$ and $q' \in M'$. The first

pair of vertices will be initial state of two Machine M and M'. The second column consist of (q_{0}, q'_{0}) where q_a is reachable from the initial state of the machine M for the first input, q'_a is reachable from the initial state of the M' for first input. The other n-2 column consist of a pair of vertices from M and M' for n-1 inputs.

3) If any new pair of states appear in any of the n-1 next state columns, which were not yet taken in the first column, take that pair in the present state column and construct subsequent column elements like first row.

4) If a pair of states (q, q') appear in any of the n-columns for a pair of states in the present state column, where q is the final state of M and q' is the non-final state of M' or vice versa, terminate the construction and conclude that M and M' are not equivalent.

5) If a pair of states (q, q') where q, q' belongs to final state or q, q' belongs to non-final states and no new pair of states appear, which were not taken in the first column, stop the construction and declare that M and M' are equivalent.

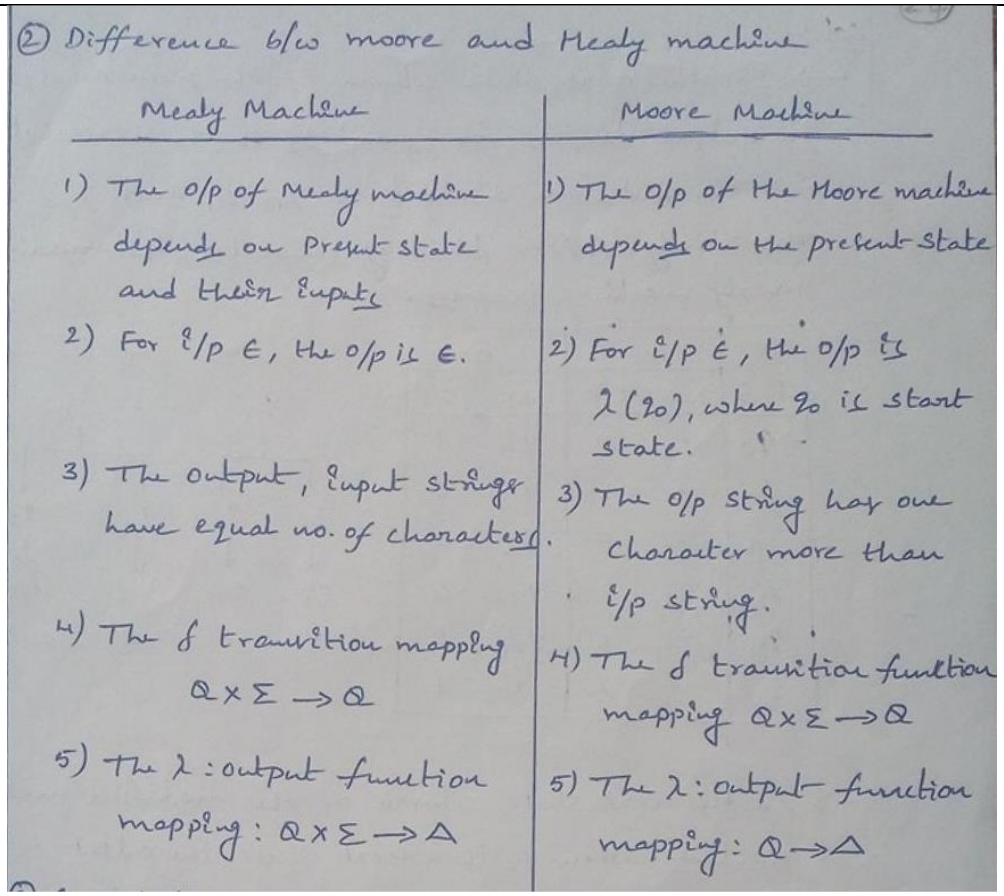
	b	Compare the equivalence two FA's or not.  Sol: • **Step 1** − First, construct the transition table for each input c and d. • **Step 2** − From the first machine M on receiving input c in state q1, we reach state q1 only which is the final state. • **Step 3** − From the second machine M1 for state q4 on receiving input c, we reach state q4 which is the final state. • **Step 4** − Thus for state (q1, q4) for input c, we get the next states as (q1, q4). Both are final states. • **Step 5** − Similarly, for input d in state (q1, q4), we get the next state as (q2, q5). Both are intermediate states. So, the first state in both machines is equal. • **Step 6** − Similarly, we perform the remaining states in two machines. • **Step 7** − In a pair of states, if both are final or if both are non-final, then we can say that two DFA's are equivalent and let's check for remaining. The transition table is as follows	[L4][CO3]	[8M]

States	c	d
{q1,q4}	{q1, q4}FS FS	{q2, q5}NS NS
{q2,q5}	{q3, q6}NS IS	{q1, q4}FS FS
{q3, q6}	{q2, q7}NS NS	{q3, q6}NS NS
{q2, q7}	{q3, q6}NS NS	{q1, q4}FS FS

Here, FS is the Final State and NS is the Non final State.

Therefore, by seeing the above table it is clear that we don't get one final and one intermediate in any pair. **Hence, we can declare that two DFA's are equivalent**.

5	a	Contrast Mealy machine and Moore machine.	[L4][CO1]	[6M]



② Difference b/w moore and Mealy machine

Mealy Machine	Moore Machine
1) The o/p of Mealy machine depends on present state and their inputs	1) The o/p of the Moore machine depends on the present state
2) For i/p ∈, the o/p is ∈.	2) For i/p ∈, the o/p is λ(q0), where q0 is start state.
3) The output, input strings have equal no. of characters.	3) The o/p string has one character more than i/p string.
4) The δ transition mapping Q × Σ → Q	4) The δ transition function mapping Q × Σ → Q
5) The λ: output function mapping: Q × Σ → Δ	5) The λ: output function mapping: Q → Δ

b

Convert the following Mealy machine into its equivalent Moore machine.

[L3][CO2] [6M]

Present State	I/P=0		I/P=1	
	Next State	O/P	Next State	O/P
→A	C	0	B	0
B	A	1	D	0
C	B	1	A	1
D	D	1	C	0

		Sol :- (12) Step1 : We can obtain all the next states along with the outputs for each current state of each input symbol A – 1 – A B – 0,1 – B_0, B_1 C – 0 – C D – 0,1 – D_0, D_1 Step2 : Now construct the transition table for Moore Machine The states are A, B_0, B_1, C, D_0, D_1 Next states Present state: I/p=0: C, A, A, B_1, D_1, D_1; I/p=1: B_0, D_0, D_0, A, ϵ, ϵ; output: 1, 0, 1, 0, 0, 1		
6	a	Define Melay machine and Moore machine.	[L3][CO1]	[6M]

The Melay Machine:-

→ The Melay Machine was proposed by George. H. Melay at the Bell labs in 1960.

→ The Melay Machine is one type of Finite automata with output, where the output depends on the present state and present input.

→ The Melay Machine consists of six tuples

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

where Q: finite non-empty set of states
Σ: set of input alphabets
Δ: set of output alphabets
δ: Transition function mapping $Q \times \Sigma \to Q$
λ: output function mapping $Q \times \Sigma \to \Delta$
q_0: initial state

Moore Machine:

→ The Moore machine was proposed by Edward. F. Moore in IBM around 1960.

→ The Moore machine is one type of Finite Automata where output depends on the present state only, but the o/p is independent of the present input.

→ The Moore machine consist of six tuples

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

where Q: Finite non-empty set of states
Σ: set of input alphabets
Δ: set of output alphabets
δ: Transition function mapping $Q \times \Sigma \to Q$
λ: output function mapping $Q \to \Delta$
q_0: Beginning state

c | Construct Mealy machine corresponding to Moore machine? | [L3][CO2] | [6M]

States (Q)	Next States		Output
	I/P=0	I/P=1	
→q1	q1	q2	0
q2	q1	q3	0
q3	q1	q3	1

Sol

* Draw the tabular format of a Mealy machine
* If λ is the o/p function in Moore machine and λ is the output function in Mealy machine, then λ can be obtained as

$$\lambda'(q_i, x_j) = \lambda(\delta(q_i, x_j))$$

$\lambda'(q_1,0) = \lambda(\delta(q_1,0)) = \lambda(q_1) = 0$

$\lambda'(q_1,1) = \lambda(\delta(q_1,1)) = \lambda(q_2) = 0$

$\lambda'(q_2,0) = \lambda(\delta(q_2,0)) = \lambda(q_1) = 0$

$\lambda'(q_2,1) = \lambda(\delta(q_2,1)) = \lambda(q_3) = 1$

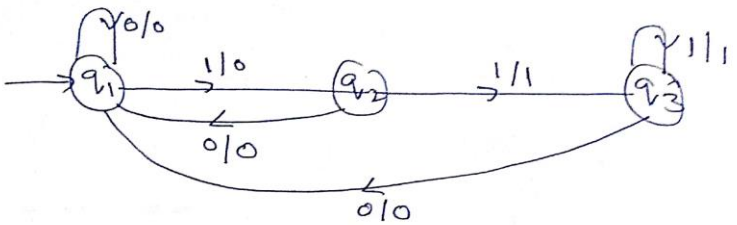
$\lambda'(q_3,0) = \lambda(\delta(q_3,0)) = \lambda(q_1) = 0$

$\lambda'(q_3,1) = \lambda(\delta(q_3,1)) = \lambda(q_3) = 1$

∴ Transition table for mealy machine is

States	I/p = 0		I/p = 1	
	Next state	output	Next state	output
q_1	q_1	0	q_2	0
q_2	q_1	0	q_3	1
q_3	q_1	0	q_3	1

∴ Transition diagram for mealy machine is



(The mealy machine)

7	a	List out the identities of Regular expression.	[L1][CO3]	[6M]

Identities of Regular Expression:

→ An identity is a relation which is tautologically true

→ In mathematics an equation which is true for every value of the variable is called an "identity equation"

→ In the RE also, there are some identities which are true for every RE.

1) $\phi + R = R + \phi = R$. Proof: LHS $= \phi + R$
$= \phi \cup R$
$= \{\} \cup \{\text{Elements of } R\}$
$= R = \text{R.H.S}$

2) $\phi R = R\phi = \phi$, Proof: LHS $= \phi R$
$= \phi \cap R$
$= \{\} \cap \{\text{Elements of } R\}$
$= \phi = \text{R.H.S}$

3) $\Lambda R = R\Lambda = R$, Proof: LHS $= \Lambda R$
= null string concatenated with any symbol of R
= same symbol $\in R = R =$ RHS

4) $\Lambda^* = \Lambda$ & $\phi^* = \Lambda$
Proof: LHS $= \Lambda^* = \{\Lambda, \Lambda\Lambda, \Lambda\Lambda\text{--}\}$
$= \{\Lambda, \Lambda, \Lambda\text{--}\}$ [according to identity (3)]
$= \Lambda =$ RHS

5) $R + R = R$
Proof: LHS $= R + R = R(\Lambda + \Lambda) = R\Lambda = R =$ RHS

6) $R^* R^* = R^*$
Proof: LHS $= R^* R^* = \{\Lambda, R, RR\text{--}\}\{\Lambda, R, RR\text{--}\}$
$= \{\Lambda\Lambda, \Lambda R, \Lambda RR\text{--}, R\Lambda, RR - RRR - \}$
$= \{\Lambda, R, RR, RRR\text{--}\}$ (using I_3)
$= R^* =$ RHS

7) $R^* R = RR^*$
proof: LHS: $R^* R = \{\Lambda, R, RR, RRR\text{--}\} R$
$= \{\Lambda R, RR, RRR, RRRR\text{--}\}$
$= R\{\Lambda, R, RR, RRR\text{--}\}$
$= RR^* =$ RHS

8) $(R^*)^* = R^*$
proof: LHS $= (R^*)^* = \{\Lambda, R^* R^*, R^* R^* R^*\text{---}\}$
$= \{\Lambda, R^*, R^*\text{--}\}$ (using I_6)
$= \{\Lambda, \{\Lambda, R, RR, RRR\text{--}\}, \{\Lambda, R, RR\text{--}\}$
$= R^* =$ RHS

9) $\Lambda + RR^* = \Lambda + R^*R = R^*$

Proof: LHS: $\Lambda + RR^* = \Lambda + R\{\Lambda, R, RR, RRR \cdots\}$

$= \Lambda + \{R\Lambda, RR, RRR, RRRR \cdots\}$

$= \Lambda + \{R, RR, RRR, RRRR \cdots\}$

$= \{\Lambda, R, RR, RRR, RRRR \cdots\}$

$= R^* = RHS$

10) $(PQ)^*P = P(QP)^*$

Proof: LHS: $(PQ)^*P = \{\Lambda, PQ, PQPQ, PQPQPQ, \cdots\}P$

$= \{P, PQP, PQPQP, PQPQPQP \cdots\}$

$= P\{\Lambda, QP, QPQP, QPQPQP \cdots\}$

$= P(QP)^* = RHS$

11) $(P+Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$ (D'Morgan's theorem)

12) $(P+Q)R = PR+QR$

Proof: Let $a \in (P+Q)R$

$= a \in PR$ or $QR = RHS$

---

b From the identities of RE, prove that [L3][CO3] [6M]

i) $10+(1010)^*[\Lambda+(1010)^*]=10+(1010)^*$

Sol LHS

$10 + (1010)^*[\Lambda + (1010)^*]$

$= 10 + \Lambda(1010)^* + (1010)^*(1010)^*$

$= 10 + (1010)^* + (1010)^*(1010)^*$ ($\Lambda R = R$)

$= 10 + (1010)^* + (1010)^*$ ($RR = R$)

$= 10 + (1010)^*$ ($R + R = R$)

$= RHS$

ii)$(1+100^*)+(1+100^*)(0+10^*)(0+10^*)^*=10^*(0+10^*)^*$

		Sol: LHS $(1+100^*)+(1+100^*)(0+10^*)(0+10^*)^*$ $=(1+100^*)(\wedge+(0+10^*)(0+10^*)^*)$ $=(1+100^*)(0+10^*)^{\wedge+RR^*}$ (according to $\wedge+RR^*=R^*$) $=1(\wedge+00^*)(0+10^*)^*$ $=10^{\wedge+RR^*}(0+10^*)^*$ = R.HS		
8	a	Prove R=Q+RP has unique solution, R=QP*	[L3][CO3]	[4M]

The Arden's Theorem:-

The Arden's theorem is used to construct the Regular Expression from a finite Automata.

Theorem:

statement: Let P, Q be two RE's over Σ. If P does not contain $\wedge$, then the equation $R=Q+RP$ has a solution (one and only) $R=QP^*$.

Proof: Now, point out the statements in the Arden's theorem in general form

→ P and Q are two Regular Expressions.
→ P does not contain the $\wedge$ symbol.
→ $R=Q+RP$ has a solution, i.e $R=QP^*$
→ This solution is the one and only solution of the Equation.

If $R=QP^*$ is a solution of the equation $R=Q+RP$ then by putting the value of R in the equation, we shall get the value '0'.

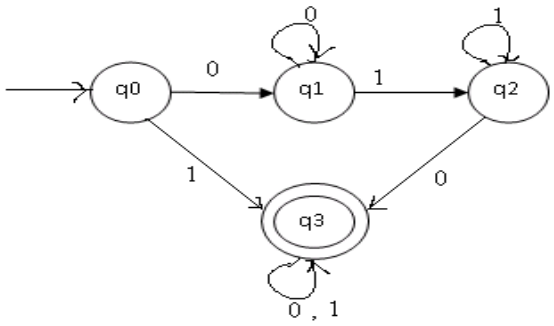
$R=Q+RP$
$R-Q-RP=0$

LHS $R-Q-RP$
(putting the value of $R=QP^*$ in LHS, we get)

$=QP^*-Q-QP^*P$
$=QP^*-Q(\wedge+P^*P)$
$=QP^*-QP^*$ $(\wedge+R^*R=R^*)$
$=0=$ R.HS

∴ So, from here it is proved that $R=QP^*$ is a solution of the Equation $R=Q+RP$.

	b	Construct RE from given FA by using Arden's Theorem.	[L6][CO3]	[8M]



Sol (16)

For the give finite Automata, the equations are

$q_0 = \Lambda$ ——— (1)

$q_1 = q_0 0 + q_1 0$ — (2)

$q_2 = q_1 1 + q_2 1$ — (3)

$q_3 = q_0 1 + q_2 0 + q_3 0 + q_3 1$ — (4)

Substitute Eq-(1) into Eq-(2)

$q_1 = q_0 0 + q_1 0$

$= \Lambda \cdot 0 + q_1 0$

$q_1 = 0 + q_1 0$ — (A1)

$q_1 = 0 + q_1 0$

$(R = Q + RP)$

The above Equation is in the form $R = Q + RP$

where $R = q_1$, $Q = 0$, $P = 0$

So, the solution of the Equation is $R = QP^*$

$\therefore \quad q_1 = 00^*$ — (5)

Now Substitute Eq-(5) into Eq-(3)

$q_2 = q_1 1 + q_2 1$

$q_2 = 00^* 1 + q_2 1$

$(R = Q + RP)$, where $R = q_2$, $Q = 00^* 1$, $P = 1$

$\therefore \quad R = QP^*$

$q_2 = 00^* 11^*$ — (6)

Now substitute Eq-(1) and Eq (6) into Eq-(4)

$$q_3 = q_0 1 + q_2 0 + q_3 0 + q_3 1$$

$$q_3 = \wedge \cdot 1 + 00^*11^*0 + q_3(0+1)$$

$$q_3 = 1 + 00^*11^*0 + q_3(0+1)$$

$$(R = \quad Q \quad + RP)$$

Where $R : q_3$, $Q : 1+00^*11^*0$, $P = (0+1)$

The Solution of Eq is

$$R = QP^*$$

$$\boxed{q_3 = (1+00^*11^*0)(0+1)^*}$$

As q_3 is the final state of the given F.A and Equation Consist of only the input symbol (Σ).

∴ The Regular Expression for the given F.A is

$$\boxed{(1+00^*11^*0)(0+1)^*}.$$

9	a	State Pumping lemma for regular languages	[L1][CO3]	[4M]

Pumping Lemma for Regular Languages

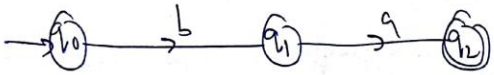
Pumping lemma is used to prove that certain sets are not regular. If any set fulfills all the conditions of pumping Lemma it can not be said that the set is regular.

Theorem: statement of the pumping lemma

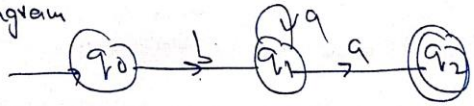
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA with n number of states. let L be a regular set accepted by M. Let w be a string that belongs to the set L and $|w| \geq m$.

If $m \geq n$, i.e, the length of the string is greater than or equal to the number of states, then there exists x, y, z such that $w = xyz$.

where $|xy| \leq n$, $|y| > 0$ and $xy^iz \in L$ for each $i \geq 0$.



The FA consist of three states q_0, q_1, and q_2. The string that is accepted by the FA is ba. The length of string is 2 which is less than the number of states of the FA. This can be described by the following diagram



The RE accepted by the automata is ba^*b. The expression can be divided into three parts x, y, z. where y is a looping portion, X is the portion befor looping and Z is the portion after the looping.

	b	Prove that $L = \{a^ib^i \mid i \geq 0\}$ is not regular	[L3][CO3]	[8M]

Sol

step 1 :- Let us assume the set L is regular. Let 'n' be the num of states of the FA accepting L.

step 2 :- Let, select a string $w \in L$, $|w| \geq n$,
By using pumping lemma, we can write $w = xyz$,
with $|xy| \leq n$ and $|y| > 0$.

Let $n = 2$, $w = aabb$
(x = a, y = a, z = bb)

$\therefore x = a, y = a, z = bb$

(i) $|xy| \leq n$
$|aa| \leq n$
$2 \leq 2$ it is true

(ii) $|y| > 0$
$|a| > 0$
$1 > 0$ it is true.

step 3 : Find a suitable integer such that $xy^iz \notin L$

Case (i) : $i = 0$
xy^iz
aa^ibb
aa^0bb
$abb \notin L$

$\therefore$ The given Language is not regular.

1	a	Give the Closure properties of Regular Sets	[L1][CO2]	[6M]

0		**Closure Properties of Regular Languages:** A set is closed under union if and only if the operations on two elements of the set produces another element of the set. If an element outside the set is produced then the operation is not closed. → closure is a property which describes when we can any two elements of the set, the result is also included in the set. → If we multiply two integer numbers, we will get another integer number. since this process is always true it is said that the integer numbers are closed under the operation of multiplication. There is simply no way to escape the set of integer numbers when multiplying. Let $S = \{1,2,3,4,5,6,7,8,9,10 \ldots\}$ is a set of integers numbers. $1 \times 2 = 2$ $2 \times 3 = 6$ $5 \times 2 = 10$ All are included in the set of integer numbers → we can conclude that integer numbers are closed under the operation of multiplication. → Two RE's L_1 and L_2 over Σ are closed under union operation. → if L_1 and L_2 are regular over Σ, their union $L_1 \cup L_2$ will be also regular.		
	b	What are the applications of Pumping Lemma?	[L1][CO3]	[6M]

Applications of the pumping Lemma

The pumping Lemma is used to prove that certain sets are not regular. If an Expression satisfys the conditions for pumping lemma to be good, then it cannot be said that the Expression is regular. But the opposite is true.

If any Expression breaks any condition of the pumping lemma, it can be declared that the Expression is not good.

This needs certain steps:

step I :- Assume that the set L is regular. Let n be the number of states of the FA accepting L.

step II :- choose a string w ($w \in L$) such that $|w| \geq n$.
By using pumping lemma, we can write

$w = xyz$, with $|xy| \leq n$ and $|y| > 0$

step III :-
Find a suitable integer i such that

$$xy^i z \notin L$$

This will be contradict our assumption.
From here 'L' will be declared as not regular.

Course Code: 20CS0903 **R20**

UNIT –II
CONTEXT FREE GRAMMAR AND TURING MACHINE

1	a	Analyze and explain with example Chomsky Hierarchy of Languages	[L4][CO1]	[6M]

The chomsky Hierarchy of Languages

The chomsky Hierachy is an important contribution in the field of formal Languages and automata theory. In the mid-1950, Noam chomsky, at the Harvard University, started working on formal languages and grammars.

He structured a hierarchy of formal languages based on the properties of the grammars required to generate the languages.

→ chomsky classified the grammar into four types depending on the production rules.

Grammar	Language	Machine format
Type 0	Unrestricted Language	Turing machine
Type 1	context sensitive Language	Linear bounded Automata
Type 2	context free Language	push Down Automata
Type 3	Regular expressions	Finite Automata

Type 0 :- Type 0 grammar is a phase structure grammar without any restriction, all grammars are type 0 grammar.

For type 0 grammar, the productions rules are in the format of

$\{(L_C)(NT)(R_C)\}$ → string of terminals (or) non-terminals or both

L_C: Left context NT: Non-Terminal

R_C: Right Context

Type 1 :- Type 1 grammar is called context-sensitive grammar.
for Type1 grammar if all rules in p are of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where $A \in NT$ (A is a single non-Terminal)
$\alpha, \beta \in (NT \cup \Sigma)^*$ (α and β are string of nonTerminals or Terminal)
$\gamma \in (NT \cup \Sigma)^+$ (γ is non-empty string of non-Terminal or Terminals)

Type 2 :- Type 2 grammar is called as context free grammar
In the LHS of the production, there will no left & right context.
for type 2 grammar, all the productions rules are in the form

$$NT \rightarrow \alpha$$

where $|NT| = 1$ and $\alpha \in (NT \cup T)^*$

Type 3 :- Type 3 grammar is called regular grammar
Here all the productions will be in the following form

$$A \rightarrow \alpha \ (or) \ A \rightarrow \alpha B$$

where $A, B \in NT$ and $\alpha \in$ Terminal

→ The chomsky classification is called the chomsky Hierarchy.
From this diagrammatical representation, we can say that all regular grammar is context free grammar, all context ~~sensitive~~ free grammar is context sensitive grammar and all context sensitive grammar is unrestricted grammar.

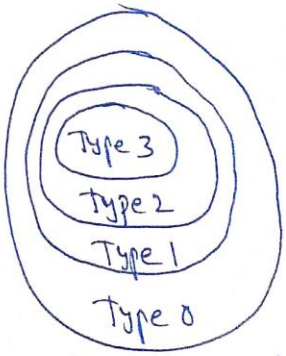


fig :- The chomsky Hierarchy.

b	Define the following terms: i) Useless symbol ii) Null production iii) Unit productions	[L1][CO4]	[6M]

Simplification of context free Grammars

CFGs may contain different types of useless symbols, unit productions and null productions. These type of symbols and productions increase the number of steps in generating a language from a CFG.

CFGs can be simplified in the following three phases

1) Removal of useless symbols
   a) Removal of non-generating symbols
   b) Removal of non-reachable symbols
2) Removal of unit productions
3) Removal of null productions

1) Removal of useless symbols

Useless symbols are of two types

Non-generating symbols are those symbols which do not produce any terminal string.

Non-reachable are those symbols which cannot be reached at any time starting from the start symbol.

Removal of UNIT productions :-

production in the form non Terminal → single non-Terminal is called a UNIT productions.

UNIT productions increases the no. of steps as well as the complexity of the time of generating language from the grammar.

Eg:- $A \rightarrow E$

$B \rightarrow D$ are called of UNIT productions.

Course Code: 20CS0903 **R20**

		Removal of NULL productions :- A production in the form $NT \rightarrow \varepsilon$ is called as null production. Procedure to Remove NULL productions step1 :- If $A \rightarrow \varepsilon$ is production to be eliminated, then we look on the all productions whose R.Hs contain A. step2 :- Replace each occurance of 'A' in each of the productions to obtain the non-ε productions. step3 :- The non-null productions must be added to the grammar to keep the language generating power the same.		
2	a	Describe what is meant by Simplifying the Grammar.	[L2][CO4]	[4M]

Simplification of context free Grammars

CFGs may contain different types of useless symbols, unit productions and null productions. These type of symbols and productions increase the number of steps in generating a language from a CFG.

CFGs can be simplified in the following three procedures

1) Removal of useless symbols
   a) Removal of non-generating symbols
   b) Removal of non-reachable symbols
2) Removal of unit productions
3) Removal of null productions

1) Removal of useless symbols

Useless symbols are of two types

Non-generating symbols are those symbols which do not produce any terminal string.

Non-reachable are those symbols which cannot be reached at any time starting from the start symbol.

Removal of UNIT productions :-

production in the form Non Terminal → single non-Terminal is called a UNIT productions.

UNIT productions increases the no. of steps as well as the complexity of the time of generating language from the grammar.

Eg:- A → E

B → D are called of UNIT productions.

Removal of NULL productions :-

A production in the form $NT \rightarrow \epsilon$ is called as null production.

Procedure to Remove NULL productions

step 1 :- If A → ε is production to be eliminated, then we look on the all productions whose R.H.S contain A.

step 2 :- Replace each occurance of 'A' in each of them productions to obtain the non-ε productions.

step 3 :- These non-null productions must be added to the grammar to keep the language generating power the same.

b	Evaluate simplification of the following context free grammar. S→ Aa /B B→ a/bC C→ a / ϵ		[L5][CO4]	[8M]

5d) first Removing useless symbols from the given grammar.

useless symbols are two types

* Non-generating symbols are those symbols which can't be produce terminal string.
* Non-reachable symbols are those symbols which can't be reached at any time starting from the start symbol.

→ In the given grammar 'A' is a non-generating symbol as it doesn't produce any terminal string. So we have to remove 'A', all the productions containing 'A' as symbol (L.H.S & R.H.S) must be removed. By Removing the productions, the minimized grammar will be

$$S \rightarrow B$$
$$B \rightarrow a \mid bC$$
$$C \rightarrow a \mid \varepsilon$$

Now finding Non-reachable symbols. In the above grammar NO-Non reachable symbols

Now Removing UNIT productions

In the Minimized grammar only one UNIT production is there

$S \rightarrow B$

By Removing ~~B/A/C~~ $S \to B$

Replace $B \to a$ in above productions

~~B/A/C~~ | $S \to a$

∴ After Removing UNIT productions the Minimized grammar is

$S \to a$

$B \to a \mid bC$

$C \to a \mid \varepsilon$

Now Removing NULL (ε) production in the above grammar only one null production

$C \to \varepsilon$

Removing $C \to \varepsilon$, look all productions R.H.S contain 'C'.

Those productions are

$B \to bC$

Replace $C \to \varepsilon$ in above production

$B \to b\varepsilon$

$B \to b$ is added in grammar

∴ After simplification of given CFG the modified grammar is.

$S \to a$

$B \to a \mid b$

$C \to a$

3		**Interpret simplification of the given grammar. Simplify the following CFG S→aSb S→A A→cAd A→cd**	[L5][CO 4]	[12 M]

Simplification of context free Grammars

CFGs may contain different types of useless symbols, unit productions and null productions. These type of symbols and productions increase the number of steps in generating a language from a CFG.

CFGs can be simplified in the following three process

1) Removal of useless symbols
   a) Removal of non-generating symbols
   b) Removal of non-reachable symbols
2) Removal of unit productions
3) Removal of null productions

1) Removal of useless symbols

Useless symbols are of two types

Non-generating symbols are those symbols which do not produce any terminal string.

Non-reachable are those symbols which cannot be reached at any time starting from the start symbol.

Removal of UNIT productions:-

production in the form Non Terminal → single non-Terminal is called a UNIT productions.

UNIT productions increases the no. of steps as well as the complexity of the time of generating language from the grammar.

Eg:- $A \rightarrow B$

$B \rightarrow D$ are called of UNIT productions.

Removal of NULL productions :-

A production in the form $NT \rightarrow \varepsilon$ is called as null production.

Procedure to Remove NULL productions

step1 :- If $A \rightarrow \varepsilon$ is production to be eliminated, then we look on the all productions whose R.Hs contain A.

step2 :- Replace each occurance of 'A' in each of the productions to obtain the non-ε productions.

step3 :- These non-null productions must be added to the grammar to keep the language generating power the same.

Sol

In the given grammar no useless symbols.

In the given grammar no null productions.

The given grammar contain one UNIT production

$S \rightarrow A$

by Removing $S \rightarrow A$, replacing A by 'cd'.

$S \rightarrow A$

$\boxed{S \rightarrow cd}$ $(A \rightarrow cd)$

After simplification the modified grammar is

$\boxed{\begin{array}{l} S \rightarrow aSb \\ A \rightarrow cAd \\ A \rightarrow cd \\ S \rightarrow cd \end{array}}$

4	a	Remove the unit production from the grammar S→AB A→E B→C C→D D→b E→a	[L3][CO4]	[6M]

Sol In the given grammar There are three UNIT productions

$A \to E$,
$B \to C$
$C \to D$

(i) Consider $A \to E$

Replace $E \to a$ in above production

$\boxed{A \to a}$

(ii) Consider $C \to D$

Replace $D \to b$ in above production

$\boxed{C \to b}$

iii) Consider $B \to C$

Replace $C \to b$ in above production

$\boxed{B \to b}$

∴ after Removing UNIT productions in the given grammar the modified grammar will be

$S \to AB$
$A \to a$
$B \to b$
$C \to b$
$D \to b$
$E \to a$

	b	Remove ϵ productons from the grammar S→ABaC A→BC B→b/ ϵ C→D/ϵ D→d	[L3][CO4]	[6M]

Sol In the given grammar Two null productions are there

$B \rightarrow \varepsilon$

$C \rightarrow \varepsilon$

* By Removing $B \rightarrow \varepsilon$, we look for all productions whose Right Hand side contains as 'B'.

Consider those productions

$S \rightarrow ABaC$

$A \rightarrow BC$

(i) Consider $S \rightarrow ABaC$

Replace $B \rightarrow \varepsilon$ in above production

$S \rightarrow A\varepsilon aC$

$\boxed{S \rightarrow AaC}$

(ii) Consider $A \rightarrow BC$

Replace $B \rightarrow \varepsilon$ in above production

$A \rightarrow \varepsilon C$

$\boxed{A \rightarrow C}$

* By Removing $C \rightarrow \varepsilon$, we look for all productions whose R.H.S contains as 'C'

Those productions are

$S \rightarrow ABaC$

$A \rightarrow BC$

		iii) Consider $S \to ABaC$ Replace $C \to \varepsilon$ in above production $S \to ABa\varepsilon$ $\boxed{S \to ABa}$ iv) Consider $A \to BC$ Replace $C \to \varepsilon$ in above production $A \to B\varepsilon$ $\boxed{A \to B}$ ∴ After Removing 'ε' productions in the given grammar the Modified grammar will be $S \to ABaC \mid AaC \mid ABa$ $A \to BC \mid C \mid B$ $B \to b$ $C \to D$ $D \to d$		
5	a	Write the process adapted to convert the grammar into CNF?	[L2][CO4]	[4M]

Chomsky Normal form

A CFG is said to be in CNF if all the productions of the grammar are in the following form.

Non-terminal → string of Exactly two non-Terminals

Non-terminal → Terminal

A CFG can be converted into CNF by the following process

step I :- 1. Eliminate all the useless symbols
2. Eliminate all the UNIT productions
3. Eliminate all the NULL productions

step II : 1. If all the productions are in the form
NT → string of Exactly TWO non-Terminals
NT → Terminal
Declare the CFG is CNF and stop.
2. else (Follow step III and/or IV and/or V)

step III :- Elimination of Terminals on the R.H.S of length Two or more

step IV : Restriction of the number of Variables on the R.H.S to TWO.

step V : Conversion of the string containing terminals and non-Terminals to the string of non-Terminal on the RHS.

b	Convert the following grammar into CNF. S→ bA/aB A→bAA/aS/a B→aBB/bS/a.		[L3][CO4]	[8M]

		Sol In the given grammar No useless symbols and no unit production and No Null production. The productions $S \to bA$ $S \to aB$ $A \to bAA$ $A \to aS$ $B \to aBB$ $B \to bS$ are not in CNF so, we have to convert. By Replacing Two New productions $C_a \to a$ $b \to b$ The Modified grammar will be $S \to DA$ $S \to CA$ $A \to DAA$ $A \to CS$ $B \to CBB$ $B \to DS$ In the above grammar Two productions are not in CNF $A \to DAA$ $B \to CBB$ By Replacing Two New production $E \to DA$ $F \to CB$ The New modified grammar will be $S \to DA \mid aB$ $A \to EA \mid CS \mid a$ $B \to FB \mid DS \mid a$ $C \to a$ $b \to b$ $E \to DA$ $F \to CB$ The above all productions are in CNF.		
6	a	State Pumping lemma for Context-free language	[L1][CO4]	[4M]

Pumping Lemma for CFL

we have become familiar with the term 'pumping lemma' in the regular expression chapter. The pumping lemma is also related to a CFL.

—>The pumping lemma for CFL is used to prove that certain sets are not context free.

-> Every CFL fulfills some general properties. But if a set or language fulfills all the properties of the pumping lemma for CFL, it cannot be said that the language is not context free.

Pumping Lemma (for CFL) is used to prove that a language is NOT Context Free.

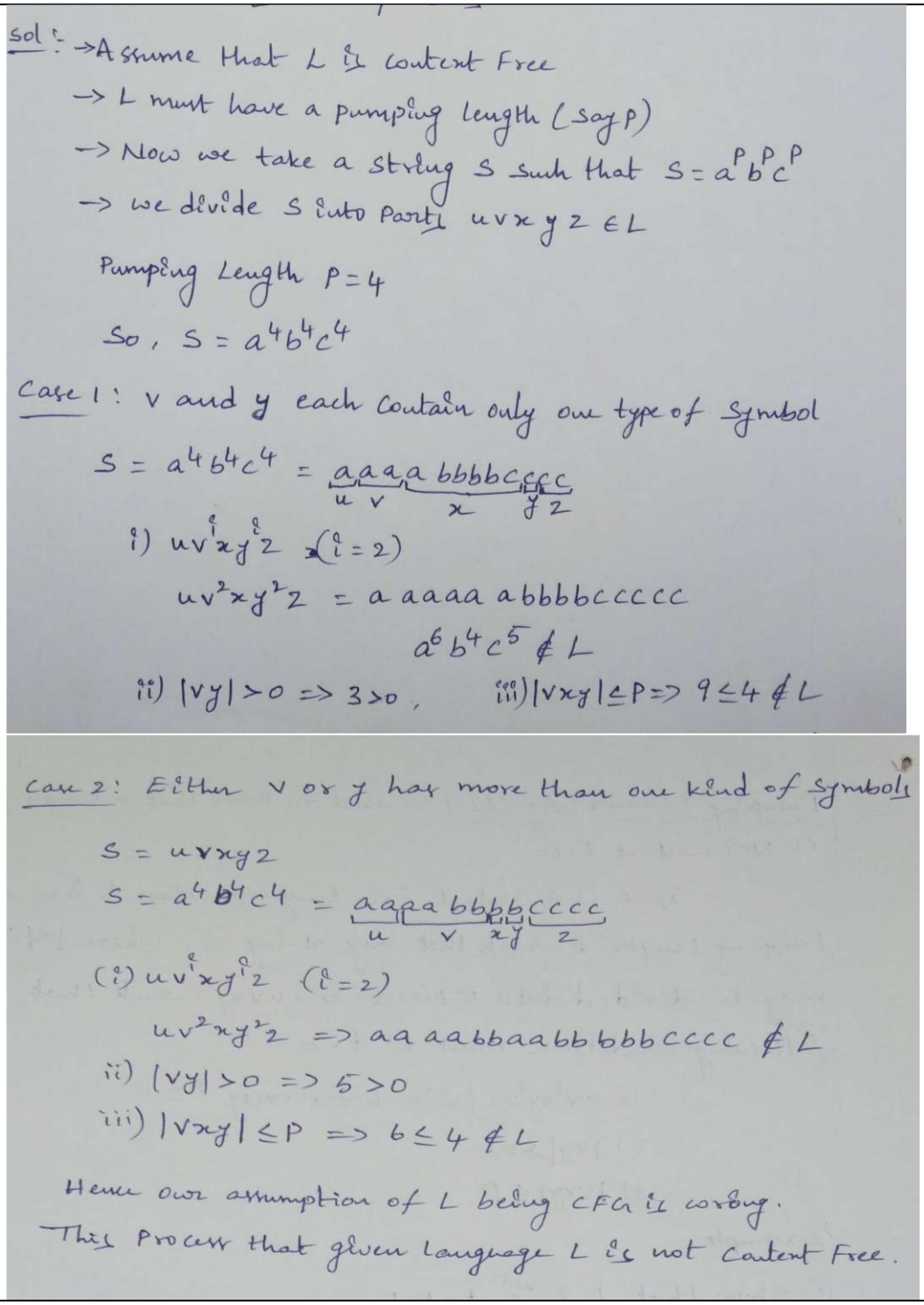
If L is a Context Free Language, then L has a Pumping Length 'P' such that any string 's', where $|S| \geq P$ may be divided into 5 pieces $S = uvxyz$ such that the following conditions must be true:

(1) uv^ixy^iz is in L for every $i \geq 0$

(2) $|vy| > 0$

(3) $|vxy| \leq P$

	b	Show that L = {$a^nb^nc^n$, where n>=1} is not context free.	[L3][CO4]	[8M]

				
7	a	State Turing machine.	[L1][CO 6]	[4M]

The Turing Machine, in short TM, is defind by 7

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

where

Q: Finite set of states

Σ: Finite set of input alphabets

Γ: Finite set of allowable tape symbol

δ: Transitional function

q_0: Initial state

B: A symbol of Γ called blank

F: Final state.

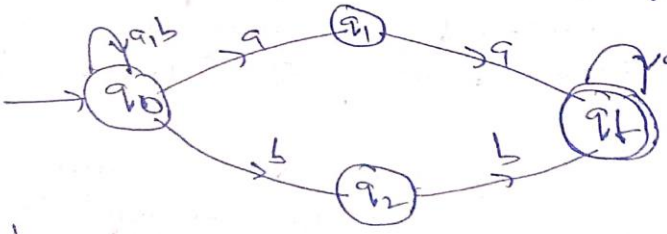
and δ is a mapping from $Q \times \Gamma \rightarrow (Q \times \Gamma \times \{L,$

b Construct a TM for regular Expression 01(00+11)(0+1)*1. [L2][CO6] 8M]

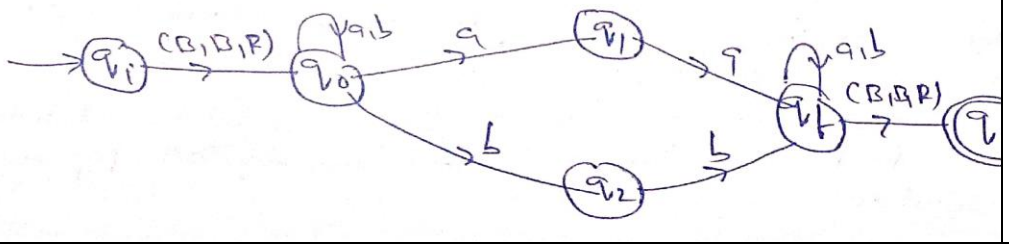
Procedure :-

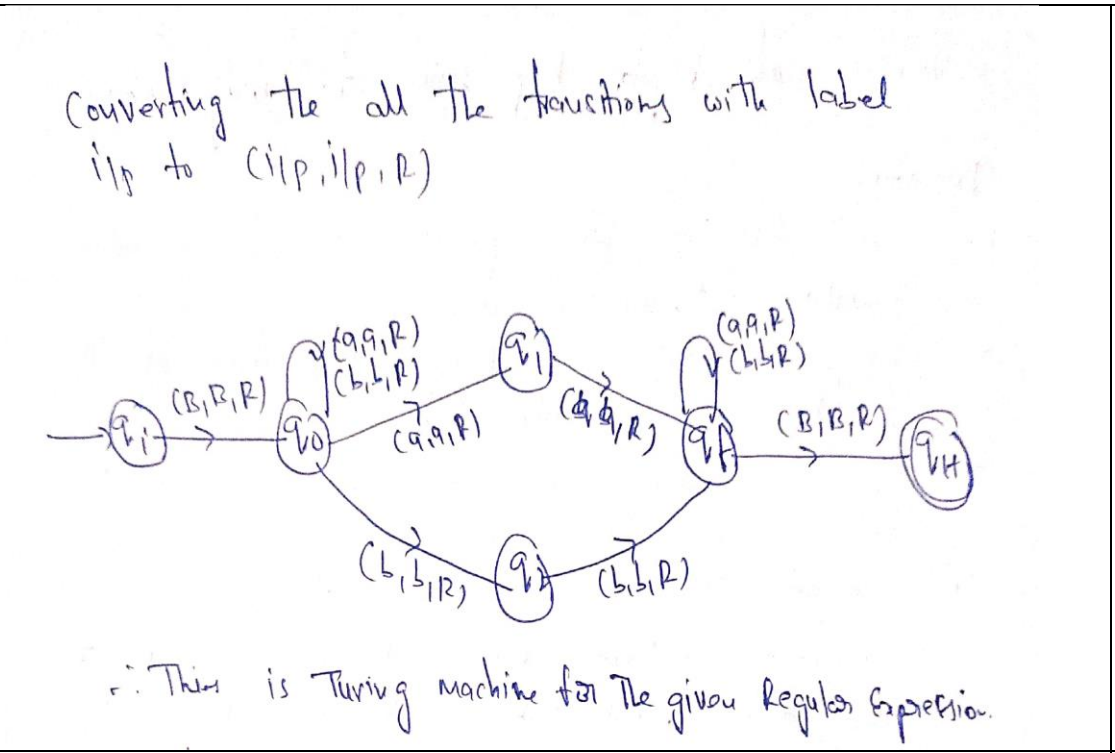
1. Convert the regular expression to an Equivale automata with out ϵ move
2. change the both initial and final states of automata to an intermediate state
3. Insert a new initial state with a transition (B to the automata initial state
4. convert the transitions with label i/p to (i/p
5. Insert a new final state with (B,B,R) automat's final state to the new final stat

Sol The finite Automata for the given Regula is



Inserting a new initial state q_i and final stat with label (B,B,R)





8		Explain the various types of Turing machine. **Multi-tape Turing machine** A **multi-tape Turing machine** is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. . Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads. This model intuitively seems much more powerful than the single-tape model, but any multi-tape machine, no matter how many tapes, can be simulated by a single-tape machine using only quadratically more computation time.Thus, multi-tape machines cannot calculate any more functions than single-tape machines, and none of the robust complexity classes (such as polynomial time) are affected by a change between single-tape and multi-tape machines. . **Multi-track Turing machines** **Multi-track Turing machines**, is a specific type of Multi-tape Turing machine, contain multiple tracks but just one tape head reads and writes on all tracks. Here, a single tape head reads n symbols from **n** tracks at one step. It accepts recursively enumerable languages like a normal single-track single-tape Turing Machine accepts. A Multi-track Turing machine can be formally described as a 6-tuple (Q, X, ∑, δ, q0,	[L2][CO 6]	[12 M]

F) where −

· **Q** is a finite set of states

· **X** is the tape alphabet

· ∑ is the input alphabet

· **δ** is a relation on states and symbols where

δ(Qi, [a1, a2, a3,….]) = (Qj, [b1, b2, b3,….], Left_shift or Right_shift)

· **q0** is the initial state
· **F** is the set of final states

a transition function of the form

$\delta: Q \times \Gamma 1 \times \Gamma 2 \times \Gamma 3 \rightarrow Q \times \Gamma 1 \times \Gamma 2 \times \Gamma 3 \times \{-1,+1\}$

.Non-Deterministic Turing Machine

In a **Non-Deterministic Turing Machine**, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 6-tuple

(Q, X, ∑, δ, q0, B, F) where −

· **Q** is a finite set of states

· **X** is the tape alphabet

· ∑ is the input alphabet

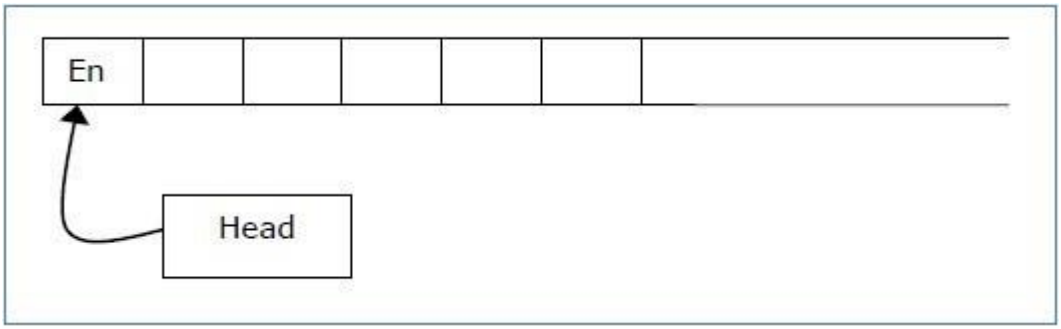
· **δ** is a transition function;

$\delta : Q \times X \rightarrow P(Q \times X \times \{\text{Left\_shift, Right\_shift}\})$.

· **q0** is the initial state

· **B** is the blank symbol

· **F** is the set of final states

		Infinite Turing Machine A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker. It is a two-track tape − · **Upper track** − It represents the cells to the right of the initial head position. · **Lower track** − It represents the cells to the left of the initial head position in reverse order.  .		
9	a	Differentiate PCP and MPCP.	[L4][CO 6]	[4M]

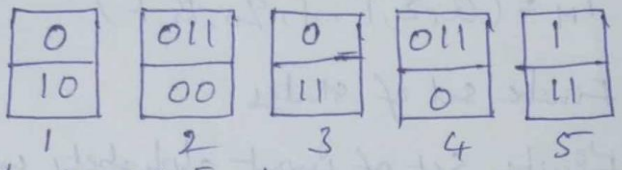
Post's Correspondence Problem (PCP).

The Post correspondence problem (PCP) was prop by an American mathematician Emil Leon Post in This is a well-known undecidable problem in the t of computer Science.

PCP is very useful for showing the undecidabil many other problems by means of reducibility.

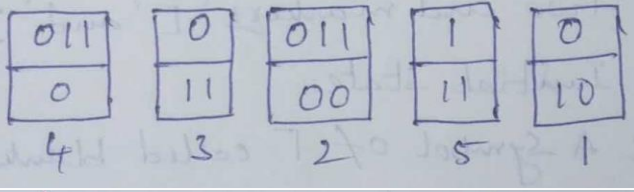
Before discussing the theory of PCP, let us play Let there be N cards where each of them are divided two parts. Each card contains a top string and a bottom

Example: Let N = 5 and the cards be the following.



Aim of the game: Is to arrange the cards in such a m that the top and the bottom strings become same.

Sol: Solution is possible and the sequence is 4 3 2



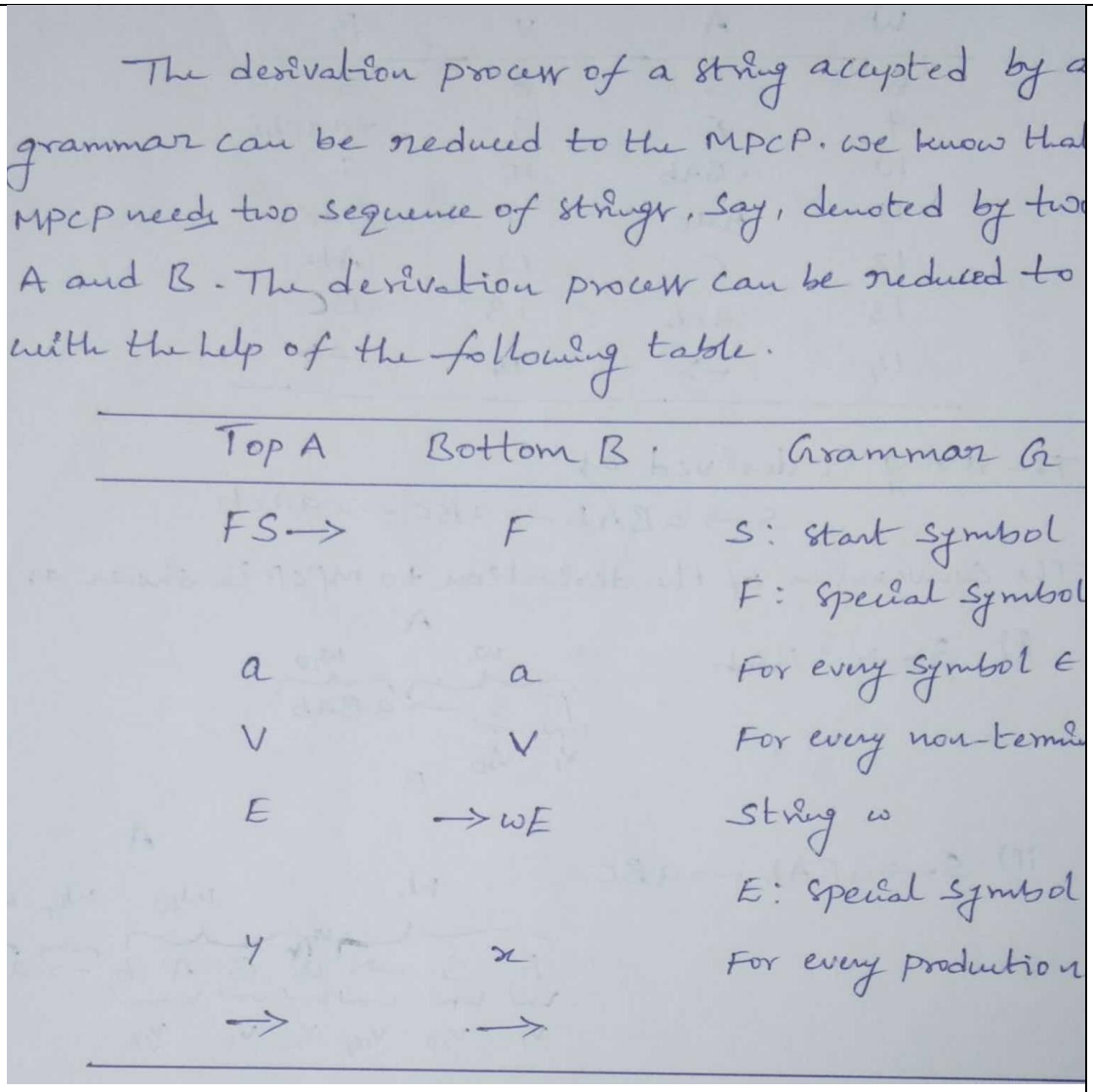
Modified PCP

If the first substring used in a PCP are w_i and x_i all instances, then the PCP is called modified PCP. T the MPCP is

Given two sequences of strings $w_1, w_2 \cdots w_n$ and $x_1, x_2, \cdots x_n$ over Σ

The solution of the problem is to find a non-empty sequence of integers $i_2, i_3, \cdots i_k$

Such that $w_{i_1} w_{i_2} w_{i_3} \cdots w_{i_k} = x_{i_1} x_{i_2} x_{i_3} \cdots$



The derivation process of a string accepted by a grammar can be reduced to the MPCP. We know that MPCP needs two sequence of strings, say, denoted by two A and B. The derivation process can be reduced to with the help of the following table.

Top A	Bottom B	Grammar G
FS →	F	S: start symbol F: special symbol
a	a	For every symbol ∈
V	V	For every non-termi
E	→ wE	String w E: special symbol
y	x	For every production
→	→	

b **Find the PCP solution for the following sets** [L5][CO 6] [8M]

A	B
10	101
01	100
0	10
100	0
1	010

To solve post Correspondence problem we try all the combinations of i_1, i_2, i_3 -- upto find the $w_i = x_i$, then we say that pcp has a solution.

The problem has a solution

$i_1=1,\ i_2=5,\ i_3=2,\ i_4=3,\ i_5=6,\ i_6=4,\ i_7=3$
$i_8=4.$

and the string is

10101001000100

procedure to solve:

step-1: we will start with tile in which numerator and denominator are starting with same number so we can start with tile 1. of I_1.

step-2: Lets go with tile 2 with I_5. string made by numerator 1, denominator 01.

step3: Repeating the same procedure, with the sequence 15234434. The string is 10101001000100 is formed in both numerator and denominator.

A = 10 1 01 0 100 100 0 100

B = 101 01 0 100 10 0 0 10 0

∴ The pcp has a solution.

The solution of the sequence is 15234434.

10	a	State the formal of PDA.	[L4][CO6]	[4M]

		pushdown Automata A PDA consist of 7-Tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ where Q – a finite set of states Σ – finite set of input symbols Γ – finite set of stack symbols δ – Transition function mapping f $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to (Q, \Gamma)$ q_0 – initial state of the PDA z_0 – is the stack bottom symb F – is the final state of the		
	b	Construct an equivalent PDA for the following CFG. S→ aAB | bBA A→bS|a B→aS | b.	[L5][CO6]	[8M]

Sol

step I :- In the given grammar all the productions are in GNF.

step II :- For a starting of the grammar, the Trans function will be

$$\delta(q_0, \epsilon, Z_0) \rightarrow (q_1, SZ_0)$$

step III :- Consider all the productions in the form $NT_1 \rightarrow$ (single T) (string of Non-Term

(i) Consider $S \rightarrow aAB$, Then Transition function

$$\delta(q_1, T, NT_1) \rightarrow (q_1, \text{string of Non-T})$$

$$\delta(q_1, a, S) \rightarrow (q_1, AB)$$

(ii) Consider $S \rightarrow bBA$, Then Transition func

$$\delta(q_1, b, S) \rightarrow (q_1, BA)$$

(iii) Consider $A \rightarrow bS$, Then Transition fu

$$\delta(q_1, b, A) \rightarrow (q_1, S)$$

(iv) Consider $B \rightarrow aS$, Then T.f will

$$\delta(q_1, a, B) \rightarrow (q_1, S)$$

step IV :- Consider all the productions in the $NT_1 \rightarrow$ single Terminal The T.f be $\delta(q_1, T, NT_1) \rightarrow (q_1, \epsilon)$

(i) Consider $A \to a$, then the transitional function

$$\delta(q_1, T, NT_1) \to (q_1, d)$$

$$\boxed{\delta(q_1, a, A) \to (q_1, d)}$$

(ii) Consider $B \to b$, then the T.f will be

$$\boxed{\delta(q_1, b, B) \to (q_1, d)}$$

∴ The PDA for the given CFG is

$M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

where $Q = \{q_0, q_1\}$

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

$\Gamma = \{z_0, S, A, B\}$

$q_0 = \{q_0\}$

$z_0 = \{z_0\}$

$F = \{\phi\}$

The Transitional functions are

$\delta(q_0, \epsilon, z_0) \to (q_1, Sz_0)$

$\delta(q_1, a, S) \to (q_1, AB)$

$\delta(q_1, b, S) \to (q_1, BA)$

$\delta(q_1, b, A) \to (q_1, S)$

$\delta(q_1, a, B) \to (q_1, S)$

$\delta(q_1, a, A) \to (q_1, d)$

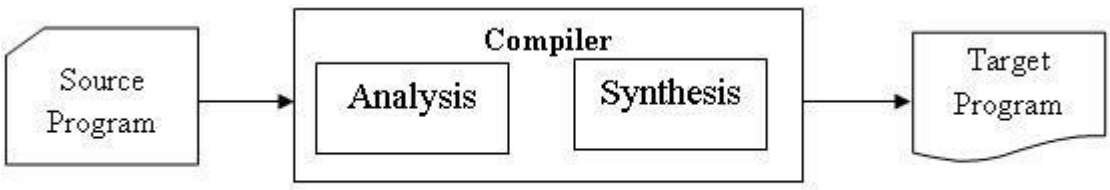
$\delta(q_1, b, B) \to (q_1, d)$

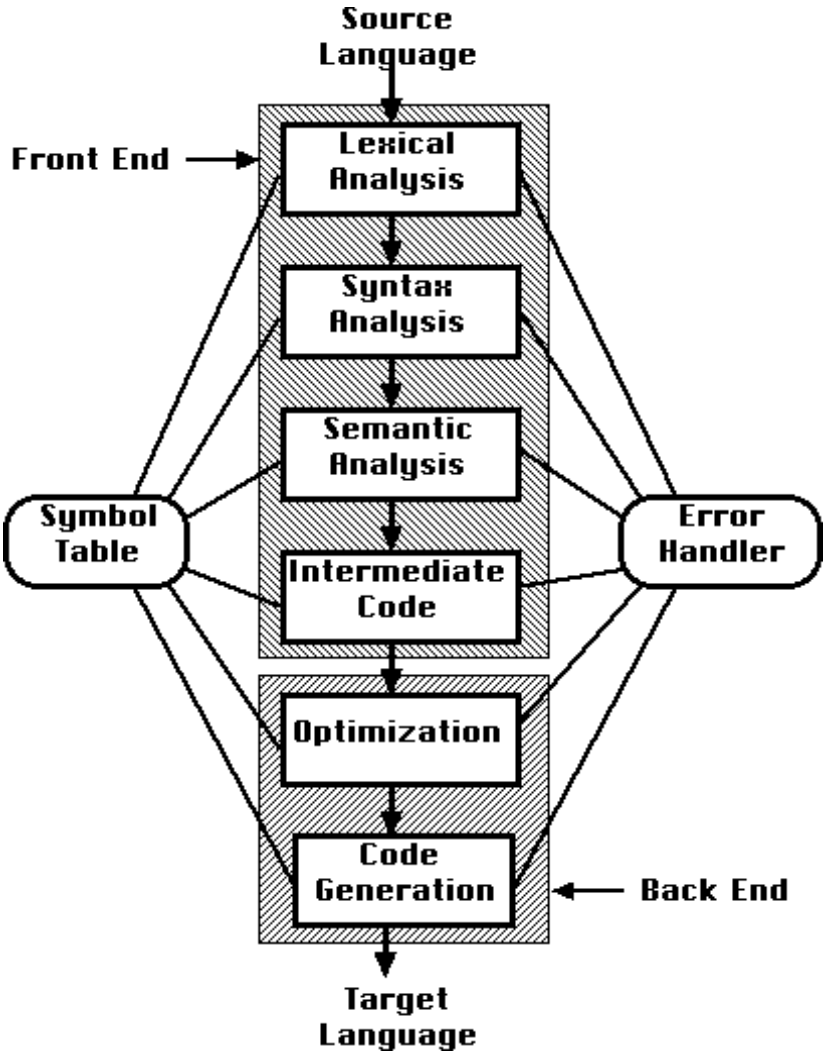
$\delta(q_1, d, z_0) \to (q_1, z_0)$

$\delta(q_1, d, z_0) \to (q_1, d)$

Course Code: 20CS0903 **R20**

UNIT –III
LEXICAL ANALYSIS AND TOP DOWN PARSING

1		Explain the phases of a compiler with neat diagram. The process of compilation can be done in two different ways A compiler as a single box that maps a source program into a semantically equivalenttargetprogram. Thereare twoparts inCompiler: ❖ Analysis ❖ Synthesis. The *analysis* part breaks up the source program into constituent pieces and imposes a grammaticalstructure on them. It then uses this structure to create an intermediate representation of the sourceprogram. If the analysis part detects that the source program is either syntactically ill formed orsemantically unsound, then it must provide informative messages, so the user can take correctiveaction. The analysis part also collects information about the source program and stores it in a datastructure called a *symbol table*, which is passed along with the intermediate representation to thesynthesispart. The *synthesis* part constructs the desired target program from the intermediate representation andthe information in the symbol table. The analysis part is often called the *front end* of thecompiler;thesynthesis partisthe*backend.*  Analysis and Synthesis model The Compiler has Six Phases: 1. LexicalAnalysis 2. SyntaxAnalysis 3. SemanticAnalysis 4. Intermediatecodegeneration 5. Code Optimization 6. Code generation	[L2][CO2]	[12 M]



LexicalAnalysis:

The first phase of a compiler is called *lexical analysis* or *scanning.* The lexical analyzer reads thestream of characters making up the source program and groups the characters into meaningfulsequences called *lexemes.* For each lexeme, the lexical analyzer produces as output a *token* of theform*(token-name,attribute-value)*thatitpassesontothesubsequentphase,syntax analysis.

For example, suppose a source program contains the assignment statement.

position=initial+rate*60

The characters in this assignment could be grouped into the following lexemes and mapped intothe followingtokens passedontothesyntaxanalyzer:

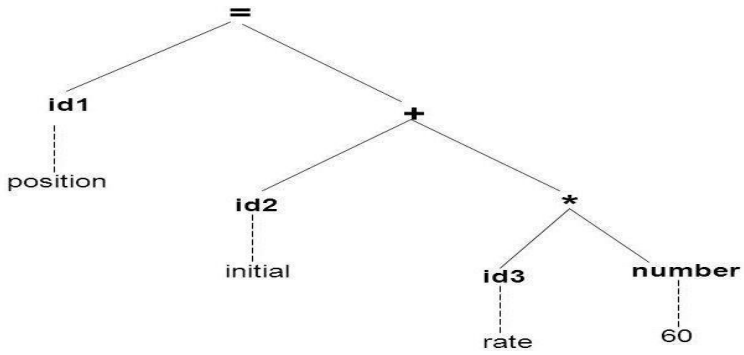
Blanks separating the lexemes would be discarded by the lexical

analyzer.Figure1.7shows the representation of the assignment statement(1.1)after lexical analysis as the sequence of tokens
(id,l)<=)(id,2)(+)(id,3)(*)(60) .....(1.2)

In this representation, the token names=,+,and* are abstract symbols for the assignment, addition, and multiplication operators, respectively.

SyntaxAnalysis:

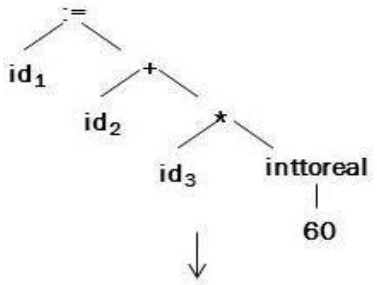
The second phase of the compiler is *syntax analysis* or *parsing.* The parser uses the firstcomponents of the tokens produced by the lexical analyzer to create a tree-likeintermediaterepresentation that depicts the grammatical structure of the token stream. A typical



representationis a *syntax tree* in which each interior node represents an operation and the children of the arguments of the operation.

SemanticAnalysis:

The *semantic analyzer* uses the syntax tree and the information in the symbol table tocheck the source program for semantic consistency with the language definition. An importantpart of semantic analysis is *type checking,* where the compiler checks that each operator hasmatching operands. For example, many programming language definitions require an array indexto be an integer; the compiler must report an error if a floating-point number is used to index anarray.



IntermediateCodeGeneration:

In the process of translating a source program into target code,a compiler may construct one or more intermediate representations, which can have a variety of forms.

This Intermediate representation should have two important

properties :it should be easy to produce and it should be easy to translate into the target machine. we consider an intermediate formcalled *three-address code*, which consists of a sequence of assembly-like instructions with threeoperands per instruction. Each operand can act like a register. The output of the intermediate codegenerator.

$temp_1 := inttoreal(60)$
$temp_2 := id_3 * temp_1$
$temp_3 := id_2 + temp_2$
$id_1 := temp3$

CodeOptimization:

The machine-independent code-optimization phase attempts to improve the intermediatecode so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the into float operation can be eliminated by replacing theinteger60bythefloating-pointnumber
60.0. Moreover, t3 is used only once to transmit its value to i d l so the optimizer can transform(1.3)into the shorter sequence.

t1 = id3 * 60.0

idl=id2+tl

Code Generation:

The code generator takes as inputan intermediaterepresentation of the source programand maps it into the target language. If the target language is machine code, registers Or memorylocations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

For example, using registers Rl andR2,the intermediate code in (1.4)might get translated into the machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id2
```

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers. The code loads the contents of address id3 intoregister R1, and then multiplies it with floating-point constant 60.0. The # signifies that 60.0 is tobe treated as an immediate constant. The third instruction movesid2into registerR2 and the fourth adds to it the value previously computed in register R1. Finally, the value in register R2 is stored into the address of id1.

Symbol-TableManagement:

The symbol tableisa data structure containinga record foreach variable name,withfields for the attributes of the name. The data structure should be designed to allow the compilerto find the recordforeachnamequicklyand to store retrieve data from that record quickly.

2	a	Explain in detail about the role of lexical analyzer in Compiler Design.	[L2][CO1]	[6M]

THE ROLE OF THE LEXICAL ANALYZER:

The first phase of a compiler, the main task of the lexical analyzer is to read the inputcharacters of the source program, group them into lexemes, and produce as output a sequence oftokens for each lexeme in the source program. The stream of tokens is sentto the parserforsyntax analysis. When the lexical analyzer discovers a lexeme constituting an identifier,it needstoenter that lexemeintothesymboltable.

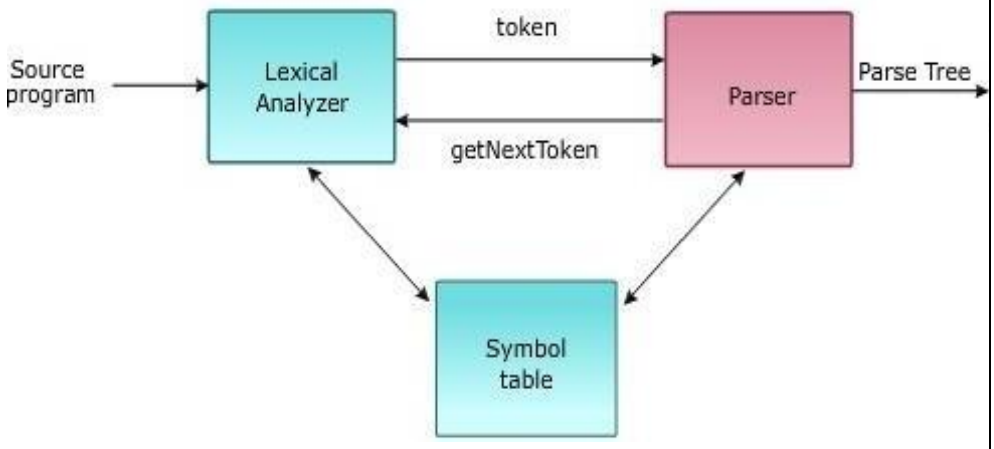


Fig: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it are used to separate tokens in the input).Another may perform certain other tasks besides identification of lexemes. One such taskis stripping outcomments and *whitespace* (blank, newline, tab, and perhaps other characters that task is correlating error messages generated by the compiler with the source program.

The lexical analyzers are divided into two processes: Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive white space characters into one.

Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

LexicalAnalysisVersus Parsing:(Reasons for separating Lexical analysis and Syntax analysis)

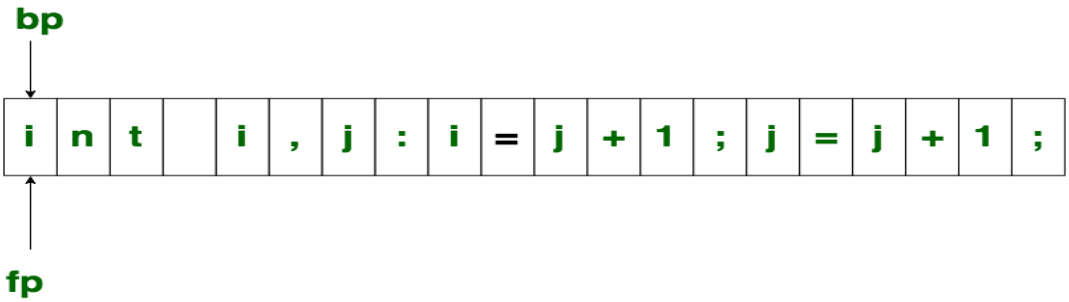
There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing(syntax analysis)phases.

1.Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.

2.Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task ,not the job of parsing.

3.Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

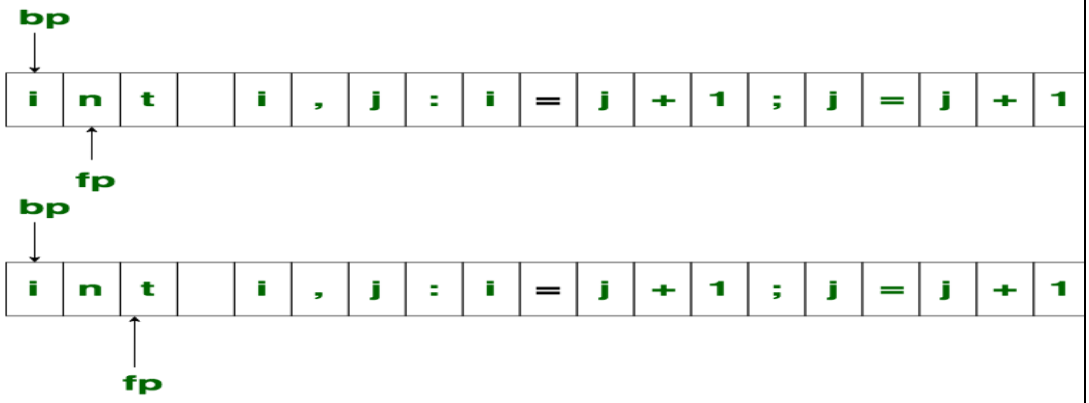
	b	Write about input buffering? **INPUT BUFFERING** The input buffering helps to find the correct lexeme; more than one character has to be seen beyond the next lexeme. A two-buffer scheme is initiated to handle large lookaheads safely. Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer-end have been adopted.	[L3][CO1]	[6M]



Initial Configuration

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**) to keep track of the pointer of the input scanned.

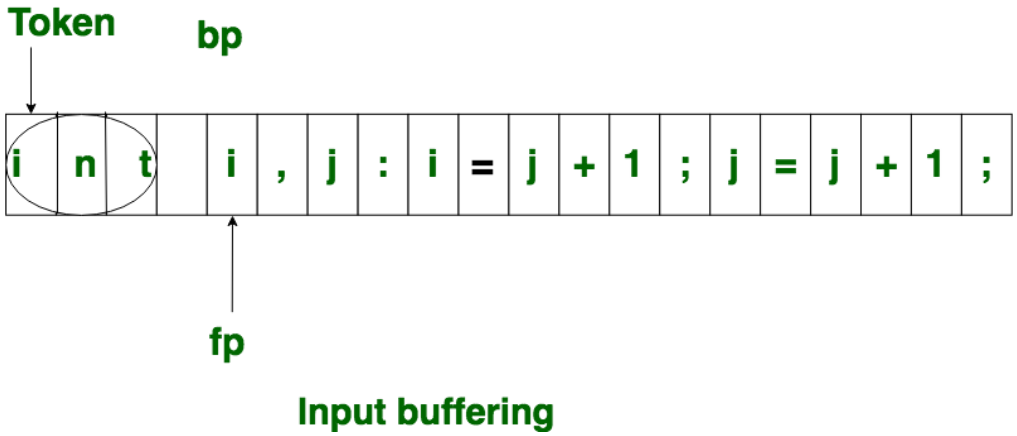
Initially both the pointers point to the first character of the input string as shown below



Input Buffering

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is

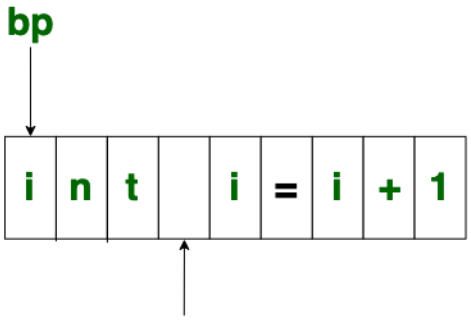
first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



Input buffering

Sentinels − Sentinels are used to making a check, each time when the forward pointer is converted, a check is completed to provide that one half of the buffer has not converted off. If it is completed, then the other half should be reloaded.

One Buffer Scheme: In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme

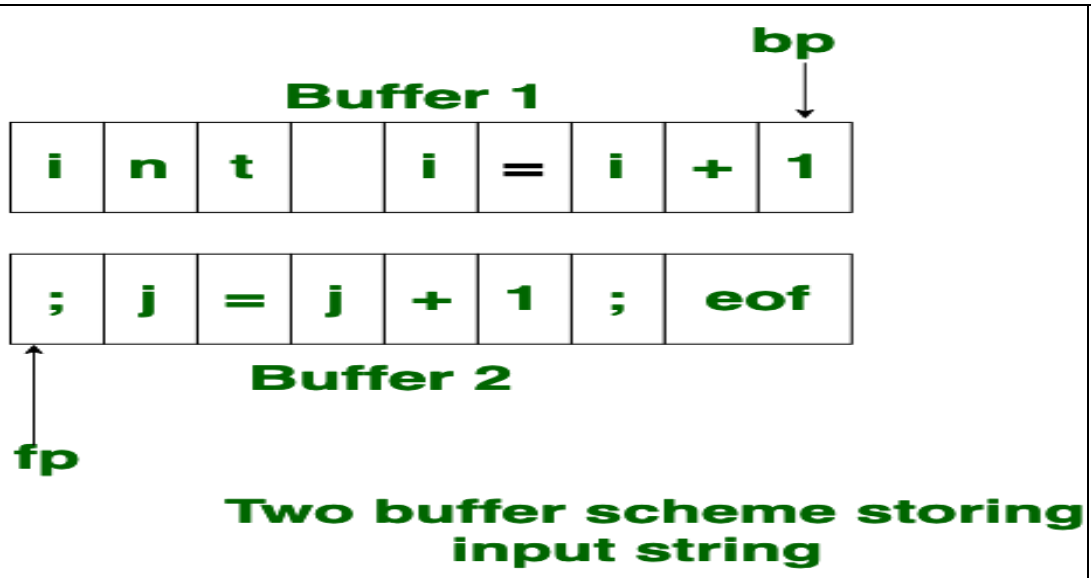


One buffer scheme storing input string

.

Two Buffer Scheme: To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely. Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token. to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer.

Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer. alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is calling **Sentinel** which is to identify the end of the buffer used.



Two buffer scheme storing input string

3	a	Explain LEX Tool with the structure of Lex Program?	[L2][CO3]	[8M]

An input file, which we call lex.l ,is written in the Lex language and describes the lexical analyzer to be generated.

The Lex compiler transforms lex.1 to a C program, in a file that is always named lex.yy.c.

The latter file is compiled by the C compiler into a file called a.out , as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

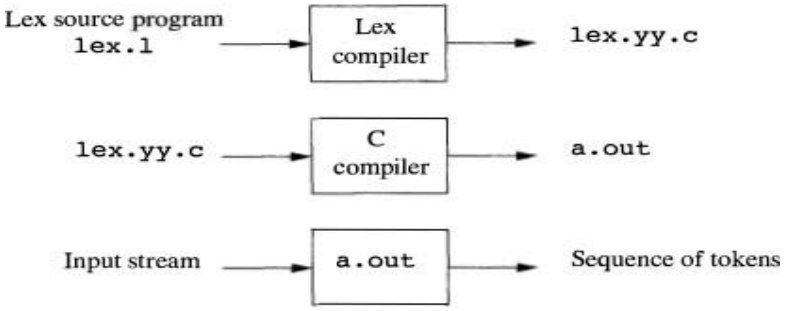


Fig. Architecture of LEX Tool

Structure of Lex Programs:

A Lex program has the following form:

declarations

%%

Translation rules

%%

Auxiliary functions

The declarations section includes declarations of variables, *manifest N constant s*(identifiers declared to stand for a constant ,e.g., the name of a token),and regular definitions.

The translation rules each have the form

		Pattern1{
		Action1}
		Pattern2{
		Action2}
		…….
		…….
		Pattern-n{Action-n}
		
		Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragements of the code, typically writtern in c, although many variants of Lex using other languages have been created. The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.		

	b	Illustrate Application of compiler technology Applicationsof CompilerTechnology: 1. ImplementationofHigh-LevelProgramming Languages. 2. OptimizationsforComputerArchitectures 3. DesignofNew ComputerArchitectures 4. ProgramTranslations 5. SoftwareProductivityTools **1. Implementation of High-Level Programming Languages** A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient ,that is ,the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. compilers include techniques to improve the performance of generated code, thus off setting the inefficiency introduced by high-level abstractions. **2. Optimizations for Computer Architectures** The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems takeadvantageofthesametwobasictechniques: *parallelism* and *memoryhierarchies*. Parallelismcanbefoundatseveral levels: at the *instructionlevel*, where multiple operations are executed simultaneously and atthe *processor level*, where different threads of the same application are run ondifferent	[L3][CO1]	[4M]

processors.

Memory hierarchies are a response to the basic limitation that we can build very fasts to rage or very large storage, but not storage e that is both fast and large

3. Design of New Computer Architectures

One of the best known examples of how compilers influenced the design ofcomputer architecture was the invention of the RISC (Reduced Instruction-SetComputer)architecture.Priortothisinvention,thetrendwastodevelopprogressively complex instruction sets intended to make assembly programmingeasier;thesearchitectureswereknownasCISC(ComplexInstruction-SetComputer).

For example, CISC instruction sets include complex memory-addressing modes tosupport data-structure accesses and procedure-invocation instructions that saveregistersandpass parametersonthestack

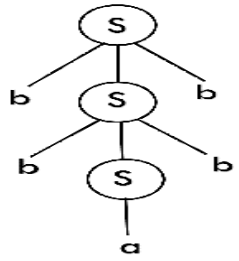
4. ProgramTranslations

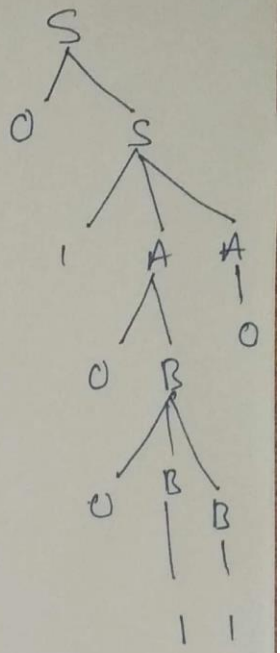
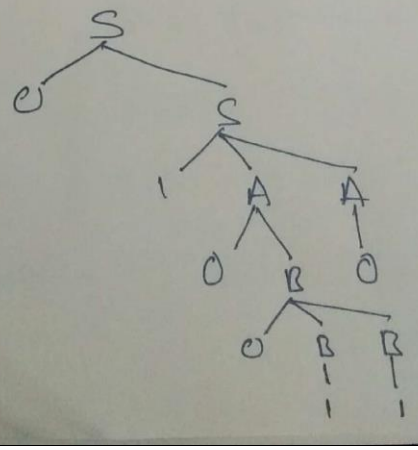
Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originallycompiled for another instruction set. Binary translation technology has beenused by various computer companies to increase the availability of software for their machines.

5. Software Productivity Tools

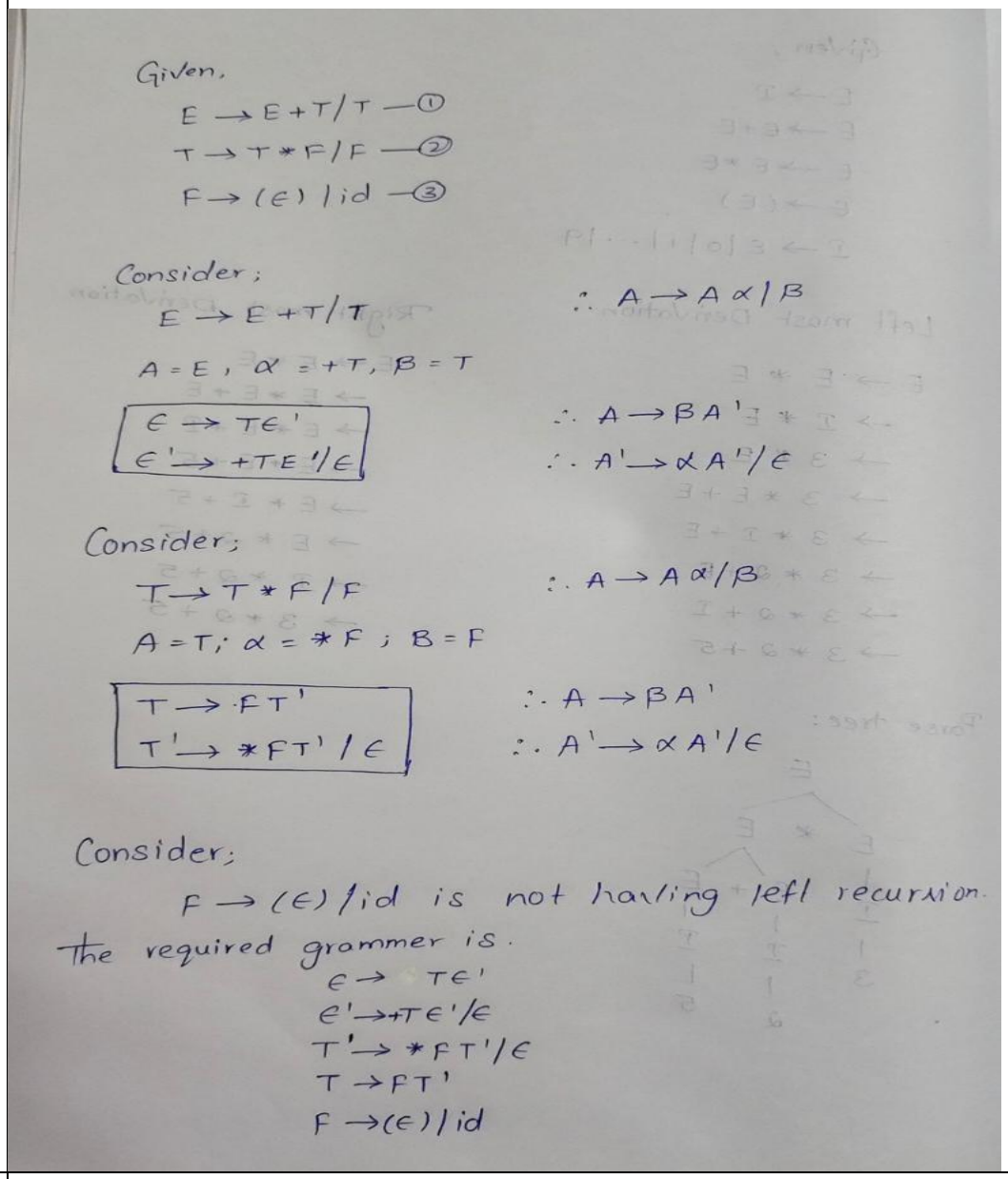
Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example,where an operation is applied to the wrong type of object, or if parameterspassed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analyzing the flow of datathrough a program. For example, if a pointer is assigned n u l l and then immediately dereferenced, the program is clearly in error.

4	a	State what is meant by derivation and parse tree with examples. PARSE TREE : Derivation tree is a graphical representation for the derivation of the given production rules of the context free grammar (CFG). It is a way to show how the derivation can be done to obtain some string from a given set of production rules. It is also called as the Parse tree. Draw a derivation tree for the string "bab" from the CFG given by	[L1][CO4]	[4M]

		S → bSb | a | b  input string: bbabb		
	b	Construct Leftmost and Rightmost derivation and derivation tree for the string 0100110 S→0S/1AA A→0/1A/0B B→1/0BB	[L6][CO4]	[8M]

		5d) Left Most Derivation :- S → 0S (S→1AA) → 01AA (A→0B) → 010BA (B→0BB) → 0100BBA (B→1) → 01001BA (B→1) → 010011A (A→0) → 0100110 Derivation Tree  Right Most Derivation :- S → 0S (S→1AA) → 01AA (A→0) → 01A0 (A→0B) → 010B0 (B→0BB) → 010BB0 (B→1) → 0100BB10 (B→1) → 0100110 Derivation Tree :- 		
5	a	Describe the procedure of eliminating Left recursion. ## Left Recursion A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same	[L1][CO1]	[6M]

		non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop. Removal of Left Recursion One way to remove left recursion is to use the following technique: **The production** **A => Aα | β** **is converted into following productions** **A => βA'** **A'=> αA' | ε** This does not impact the strings derived from the grammar, but it removes immediate left recursion. Example The production set S => Aα | β A => Sd after applying the above algorithm, should become S => Aα | β A => Aαd | βd and then, remove immediate left recursion using the first technique. A => βdA' A' => αdA' | ε Now none of the production has either direct or indirect left recursion.		
	b	Eliminate left recursion for the following grammar E→E+T/T T→T*F/F F→(E)/id	[L5][CO1]	[6M]



6	a	Explain Left recursion and Left factoring.	[L2][CO1]	[6M]

Left Recursion

A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

A => Aα | β

is converted into following productions

A => βA'

A'=> αA' | ε

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Example

The production set

S => Aα | β
A => Sd

after applying the above algorithm, should become

S => Aα | β
A => Aαd | βd

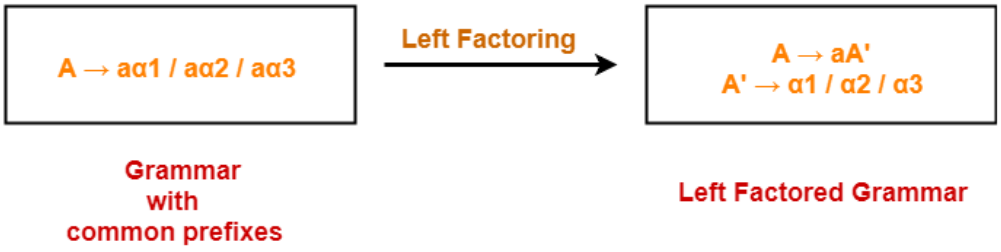
and then, remove immediate left recursion using the first technique.

A => βdA'
A' => αdA' | ε

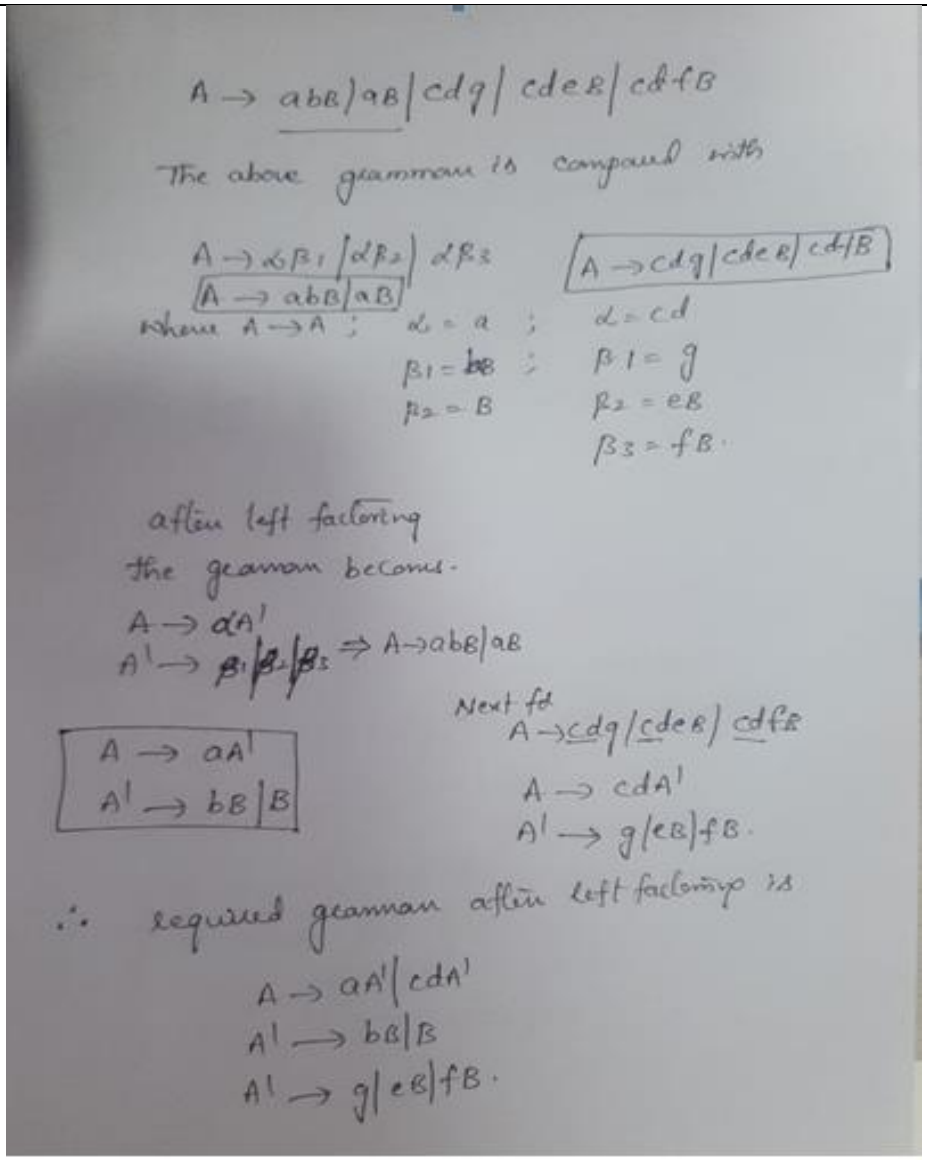
Now none of the production has either direct or indirect left recursion.

Left Factoring:

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

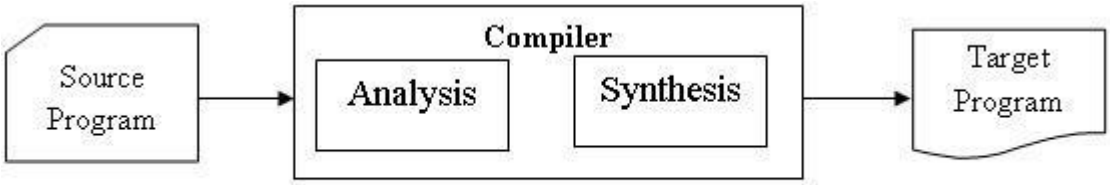


	b	Perform left factor for the grammar A→abB/aB/cdg/cdeB/cdfB	[L3][CO4]	[6M]



7	a	Describe the role of Compiler A compiler as a single box that maps a source program into a semantically equivalenttargetprogram. Thereare twoparts inCompiler: ❖ Analysis ❖ Synthesis. The *analysis* part breaks up the source program into constituent pieces and imposes a grammaticalstructure on them. It then uses this structure to create an intermediate representation of the sourceprogram. If the analysis part detects that the source program is either syntactically ill formed orsemantically unsound, then it must provide informative messages, so the user can take correctiveaction. The analysis part also collects information about the source program and stores it in a datastructure called a *symbol table*, which is passed along with the intermediate representation to thesynthesispart. The *synthesis* part constructs the desired target program from the intermediate representation andthe information in the symbol table. The analysis part is often called the *front end* of thecompiler;thesynthesis	[L1][CO1]	[4M]

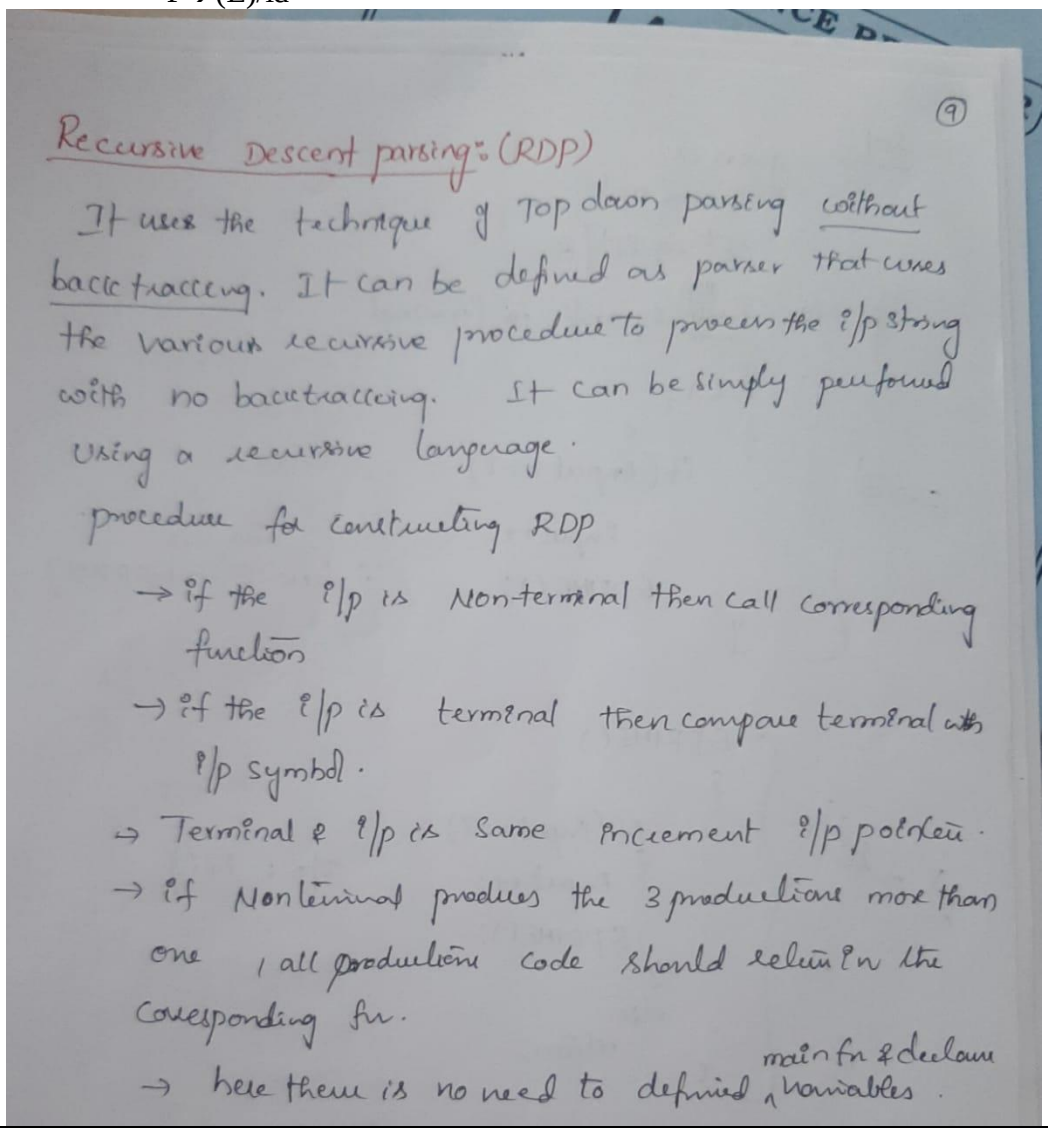
partisthe*backend.*



Analysis and Synthesis model

b | Design the recursive decent parser for the following grammar? | [L6][CO3] | [8M]

E→E+T/T
T→T*F/F
F→(E)/id

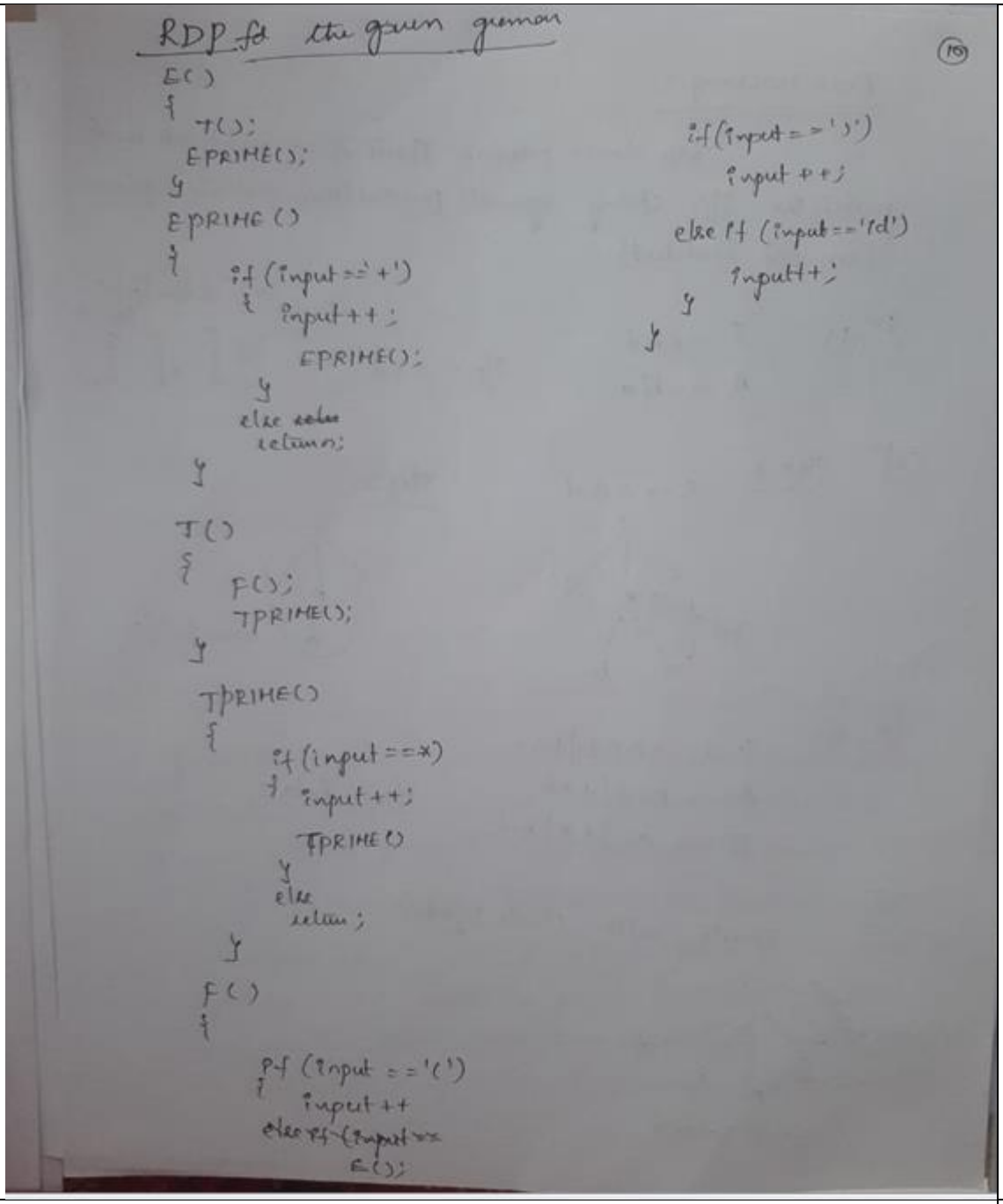


Recursive Descent parsing: (RDP)

It uses the technique of Top down parsing without backtracking. It can be defined as parser that uses the various recursive procedure to process the i/p string with no backtracking. It can be simply performed using a recursive language.

procedure for constructing RDP

→ if the i/p is Non-terminal then call corresponding function
→ if the i/p is terminal then compare terminal with i/p symbol.
→ Terminal & i/p is same increment i/p pointer.
→ if Non terminal produces the 3 productions more than one, all productions code should return in the corresponding fn.
→ here there is no need to defined main fn & declare variables.



8	a	Illustrate the rules to be followed in finding the FIRST and FOLLOW. **First Set** This set is created to know what terminal symbol is derived in the first position by a non-terminal. Rules For Calculating First Function- **Rule-01**: For a production rule $X \rightarrow \in$, $First(X) = \{ \in \}$ **Rule-02:** For any terminal symbol 'a', $First(a) = \{ a \}$ **Rule-03:**	[L3][CO1]	[4M]

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

- If $\in \notin$ First(Y_1), then First(X) = First(Y_1)
- If $\in \in$ First(Y_1), then First(X) = { First(Y_1) – $\in$ } $\cup$ First(Y_2Y_3)

Calculating First(Y_2Y_3)

- If $\in \notin$ First(Y_2), then First(Y_2Y_3) = First(Y_2)
- If $\in \in$ First(Y_2), then First(Y_2Y_3) = { First(Y_2) – $\in$ } $\cup$ First(Y_3)

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Follow Function-

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule $A \rightarrow \alpha B$,

$$Follow(B) = Follow(A)$$

Rule-03:

For any production rule $A \rightarrow \alpha B\beta$,

- If $\in \notin$ First(β), then Follow(B) = First(β)
- If $\in \in$ First(β), then Follow(B) = { First(β) – $\in$ } $\cup$ Follow(A)

b | Find FIRST and FOLLOW for the following grammar? E→ E+T/T T→T*F/F F→(E)/id | [L3][CO2] | [8M]

have-

- The given grammar is left recursive.
- So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' / \in$$

$$T \rightarrow FT'$$

$$T' \rightarrow * FT' / \in$$

$$F \rightarrow (E) / id$$

Now, the first and follow functions are as follows-

First Functions-

		• First(E) = First(T) = First(F) = { (, id } • First(E') = { + , ∈ } • First(T) = First(F) = { (, id } • First(T') = { * , ∈ } • First(F) = { (, id } **Follow Functions-** • Follow(E) = { $,) } • Follow(E') = Follow(E) = { $,) } • Follow(T) = { First(E') – ∈ } ∪ Follow(E) ∪ Follow(E') = { + , $,) } • Follow(T') = Follow(T) = { + , $,) } • Follow(F) = { First(T') – ∈ } ∪ Follow(T) ∪ Follow(T') = { * , + , $,) }		
9		Consider the grammar E→E+T/T, T→T*F/F, F→(E)|id Design predictive parsing table and check given grammar is LL(1) Grammar or not? **SOLUTION –** **Step1− Elimination of Left Recursion & perform Left Factoring** After eliminating LEFT RECURSION the modified grammar − E → TE′ E′ → +TE′|ε T′ → FT′ T′ → FT′|ε F → (E)|id **Step2− Computation of FIRST** FIRST(E) = FIRST(T) = FIRST(F) = {(, id} FIRST (E′) = {+, ε} FIRST (T′) = {*, ε} **Step3− Computation of FOLLOW** FOLLOW (E) = FOLLOW(E′) = {), $} FOLLOW (T) = FOLLOW(T′) = {+,), $} FOLLOW (F) = {+,*,),$} **Step4− Construction of Predictive Parsing Table** Create the table, i.e., write all non-terminals row-wise & all terminal column-wise.	[L6][CO3]	[12 M]

M[A, a]=

	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Now, fill the table by applying rules from the construction of the Predictive Parsing Table.

- **E → TE′**

Comparing E → TE′with A → α

E →	TE′
A →	A

∴ A = E, α = TE′

∴ FIRST(α) = FIRST(TE′) = FIRST(T) = {(, id}

Applying Rule (1) of Predictive Parsing Table

∴ ADD E → TE′ to M[E, ( ] and M[E, id]

∴ write E → TE′in front of Row (E)and Columns {(, id} (1)

- **E′ → +TE′|ε**

Comparing it with A → α

E →	+TE′
A →	α

∴ A = E′ α = +TE′

∴ FIRST(α) = FIRST(+TE′) = {+}

∴ ADD E → +TE′ to M[E′, +]

∴ write production E′ → +TE′ in front of Row (E′)and Column (+) (2)

- **E′ → ε**

Comparing it with A → α

E →	ε
A →	α

∴ α = ε

∴ FIRST(α) = {ε}

∴ Applying Rule (2)of the Predictive Parsing Table.

Find FOLLOW (E′) = {), $}

∴ ADD Production E′ → ε to M[E′,)] and M[E′, $]

∴ write E′ → ε in front of Row (E′)and Column {$,)} (3)

- **T → FT′**

Comparing it with A → α

T →	FT′
A →	α

∴ A = T, α = FT′

∴ FIRST(α) = FIRST (FT′) = FIRST (F) = {(, id}

∴ ADD Production T → FT′ to M[T, (] and M[T, id]

∴ write T → FT′ in front of Row (T)and Column {(, id} (4)

- **T′ →*FT′**

Comparing it with A → α

T →	*FT′
A →	α

∴ FIRST(α) = FIRST (* FT′) = {*}

∴ ADD Production T → +FT′ to M[T′,*]

∴ write T′ →∗ FT′ in front of Row (T′)and Column {*} (5)

- **T′ → ε**

Comparing it with A → α

T′ →	ε
A →	α

∴ A = T′

α = ε

∴ FIRST(α) = FIRST {ε} = {ε}

∴ Applying Rule (2)of the Predictive Parsing Table.

Find FOLLOW (A) = FOLLOW (T′) = {+,), $}

∴ ADD T′ → ε to M[T′, +], M[T′,)] and M[T′, $]

∴ write T′ → ε in front of Row (T′)and Column {+,), $} (6)

- **F →(E)**

Comparing it with A → α

F →	(E)
A →	A

∴ A = F, α = E

∴ FIRST(α) = FIRST ((E)) = {(}

∴ ADD F → (E) to M[F, (]

∴ write F → (E) in front of Row (F)and Column (( ) (7)

- **F → id**

Comparing it with A → α

F →	Id
A →	A

∴ FIRST(α) = FIRST (id) = {id}

∴ ADD F → id to M[F, id]

∴ write F → id in front of Row (F)and Column (id) (8)

Combining all statements (1) to (8) will generate the following Predictive or LL (1) Parsing Table −

	Id	+	*	(	)	$
E	E → TE′			E → TE′		
E′		E′ → +TE′			E′ → ε	T′ → ε
T	T → FT′			T → FT′		
T′		T′ → ε	T′ →* FT′		T′ → ε	T′ → ε
F	F → id			F → (E)		

The given grammar is LL(1) grammar.

10 Consider the grammar

$E \rightarrow TE^1$

$E^1 \rightarrow +TE^1 \mid -TE^1 \mid \varepsilon$

$T \rightarrow FT^1$

$T^1 \rightarrow *FT^1 \mid / FT^1 \mid \varepsilon$

$F \rightarrow GG^1$

$G^1 \rightarrow \wedge F / \varepsilon$

$G \rightarrow (E) / id$

Calculate FIRST and FOLLOW for the above grammar

[L4][CO2] [12 M]

Given;

$E \rightarrow TE'$

$E' \rightarrow +TE'/-TE'/\varepsilon$

$T \rightarrow FT'$

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

$F \rightarrow GG'$

$G' \rightarrow \wedge F/\varepsilon$

$G \rightarrow (E)/id$

FIRST ( ) :

FIRST (G) = { c, id }

FIRST (G') = { ∧, ε }

FIRST (F) = { FIRST (G) } = { c, id }

FIRST (T') = { * , /, ε }

FIRST (T) = { FIRST (F) } = { c, id }

FIRST (E') = { +, −, ε }

FIRST (E) = { FIRST (T) } = { c, id }.

FOLLOW ( ) :

FOLLOW (E) = { $,) }

FOLLOW (T) = FIRST (E') = { +, −, (, $,) }

FOLLOW (E') = { $,) }

FOLLOW (F) = FIRST (T') = { *, /, ε }

= { *, /, +, −, $,) }

FOLLOW (G) = FIRST (G') = { ∧, ε }

= { ∧, +, /, +, −, $,) }

FOLLOW (G') = FOLLOW (F) = { *, /, +, −, $,) }

FOLLOW (T') = FOLLOW (T) = { +, −, $,) }

Course Code: 20CS0903 **R20**

UNIT –IV
BOTTOM-UP PARSING AND SEMANTIC ANANLYSIS

1	a	Explain about handle pruning	[L2][CO1]	[6M]

HANDLE PRUNING:

Removing the children of the left-hand side non-terminal from the parse tree is called Handle Pruning.
A rightmost derivation in reverse can be obtained by handle pruning.

Sentential form: S => a here, 'a' is called sentential form, 'a' can be a mix of terminals and nonterminals.
Consider Grammar : S -> aSa | bSb | ε

Derivation: S => aSa => abSba => abbSbba => abbbba

Left Sentential and Right Sentential Form:

- A left-sentential form is a sentential form that occurs in the *leftmost derivation of some sentence.*
- A right-sentential form is a sentential form that occurs in the *rightmost derivation of some sentence.*

Example 1:

Right Sequential Form	Handle	Reducing Production
id + id * id	id	E ⇒ id
E + id * id	id	E ⇒ id
E + E * id	id	E ⇒ id
E + E * E	E + E	E ⇒ E + E
E * E	E * E	E ⇒ E * E
E (Root)		

Example 2:

Right Sequential Form	Handle	Production
id + id + id	id	E ⇒ id
E + id + id	id	E ⇒ id
E + **E** + id	id	E ⇒ id
E + **E** + **E**	**E** + **E**	E ⇒ E + E
E + **E**	**E** + **E**	E ⇒ E + E
E (Root)		

b **Summarize about SLR parsing** [L2][CO1] [6M]

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)
  -

For each type of LR parsing the following steps have to be followed:

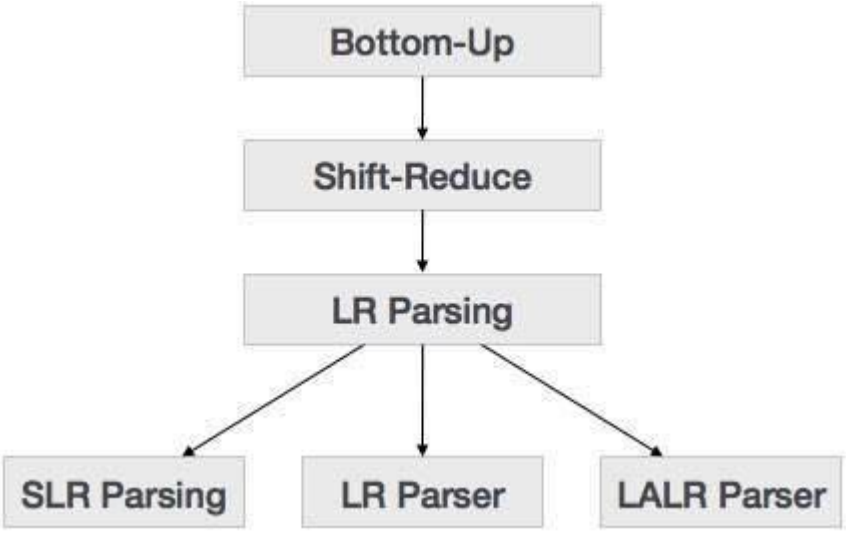
- Introduce the augmented grammar
- Construction of LR(0) items for SLR and LR(1) Items for CLR AND LALR parser
- Construction of DFA for the given parser
- Construction of Parsing Table for the given parser
- Parse the input string for the given parser

2 a **Describe bottom up parsing** [L1][CO2] [4M]

BOTTOM UP PARSING

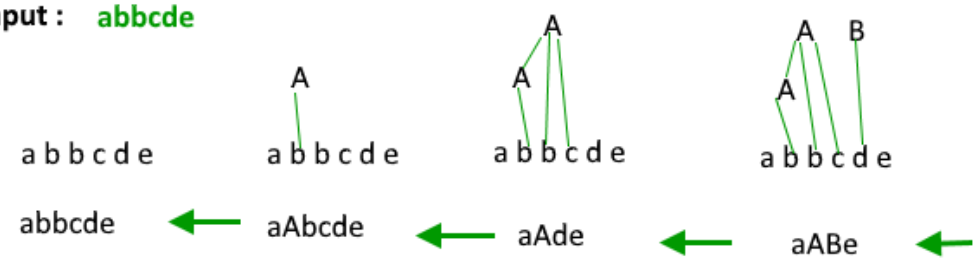
Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a

sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



S → aABe
A → Abc/b
B → d

Input : abbcde



b **Differences between SLR,CLR, LALR parsers** [L4][CO2] [8M]

Comparison between SLR, CLR, and LALR Parser:

S. no.	SLR Parsers	Canonical LR Parsers	LALR Parsers
1	SLR parser are easiest to implement.	CLR parsers are difficult to implement.	LALR parsers are difficult to implement than SLR parser but less than CLR parsers.
2.	SLR parsers make use of canonical collection of LR(0) items for constructing the parsing tables.	CLR parsers are uses LR(1) collection of items for constructing the parsing tables part.	LALR parsers LR(1) collection , items with items having same core merged into a single itemset.

		3	SLR parsers don't do any lookahead i.e., they lookahead zero	CLR parsers lookahead one symbol.	LALR parsers lookahead one symbol.		
		4.	SLR parsers are cost effective to construct in terms of time and space.	CLR parsers are expensive to construct in terms – of time and space.	The cost of constructing LALR parsers is i intermediate between SLR and CLR parser.		
		5.	SLR parsers have hundreds of states.	CLR parses have thousands of states.	LALR parsers have hundreds of state and is same as number states in SLR parsers.		
		6.	SLR parsers uses FOLLOW information to guide reductions.	CLR parsers uses lookahead symbol to guide reductions.	LALR parsers uses lookahead symbol to guide reductions.		
		7.	SLR parsers may fail to l produce a table for certain class of grammars on which ether succeed.	CLR parser works on very large class of grammar.	LALR parser works on very large class grammars.		
		8.	Every SLR(1) grammar is LR(1) grammar and LALR(1).	Every LR(1) grammar may not be SLR(1) grammar.	Every LALR(1) grammar may not be SLR(1) but every LALR(1) grammar is LR(1) grammar.		
		9.	A shift-reduce or reduce-reduce conflict may arise in SLR parsing table.	A shift-reduce or reduce-reduce conflicted may arise but chances are less than that in SLR parsing tables.	A shift-reduce conflict can not arise but a reduce-reduce conflict may arise.		
		10.	SLR parser is least powerful.	A CLR parsers is most powerful among the family canonical of bottom-up parsers.	A LALR parser is intermediate in power between SLR and LR parser.		
3		Prepare Shift Reduce Parsing for the input string using the grammar S→(L)|a L→L,S|S a)(a,(a,a)) b)(a,a) a) (a,(a,a))				[L6][CO3]	[12M]

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a

$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S)	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

b) (a,a)

Stack	Input Buffer	Parsing Action
$	(a,a) $	Shift
$ (	a , a) $	Shift

		$ (a	, a) $	Reduce S → a	
		$ (S	, a) $	Reduce L → S	
		$ (L	,a) $	Shift	
		$ (L ,	) $	Shift	
		$ (L , a	) $	Reduce S → a	
		$ (L ,S	) $	Reduce L → L,S	
		$ (L	$	Shift	
		$ (L)	$	Reduce S → (L)	
		$ S	$	Accept	
4	a	Define augmented grammar. **Augmented Grammar –** If grammar G has start symbol S, then augmented Grammar is new Grammar G′ with new start symbol S′. Also, it will contain the production S′ → S.		[L1][CO2]	[2M]
	b	Construct the LR(0) items for the following Grammar S→L=R S→R L→*R L→id R→L		[L6][CO3]	[10M]

Sol

Initially add Augmented grammar based on starting symbol of the grammar.

I_0 :
$S' \rightarrow .S$
$S \rightarrow .L=R$
$S \rightarrow .R$
$L \rightarrow .*R$
$L \rightarrow .id$
$R \rightarrow .L$

Now finding goto and closure on I_0.

(i) goto $[I_0, S]$

I_1: $S' \rightarrow S.$

(ii) goto $[I_0, L]$

I_2: $S \rightarrow L.=R$
$R \rightarrow L.$

(iii) goto $[I_0, R]$

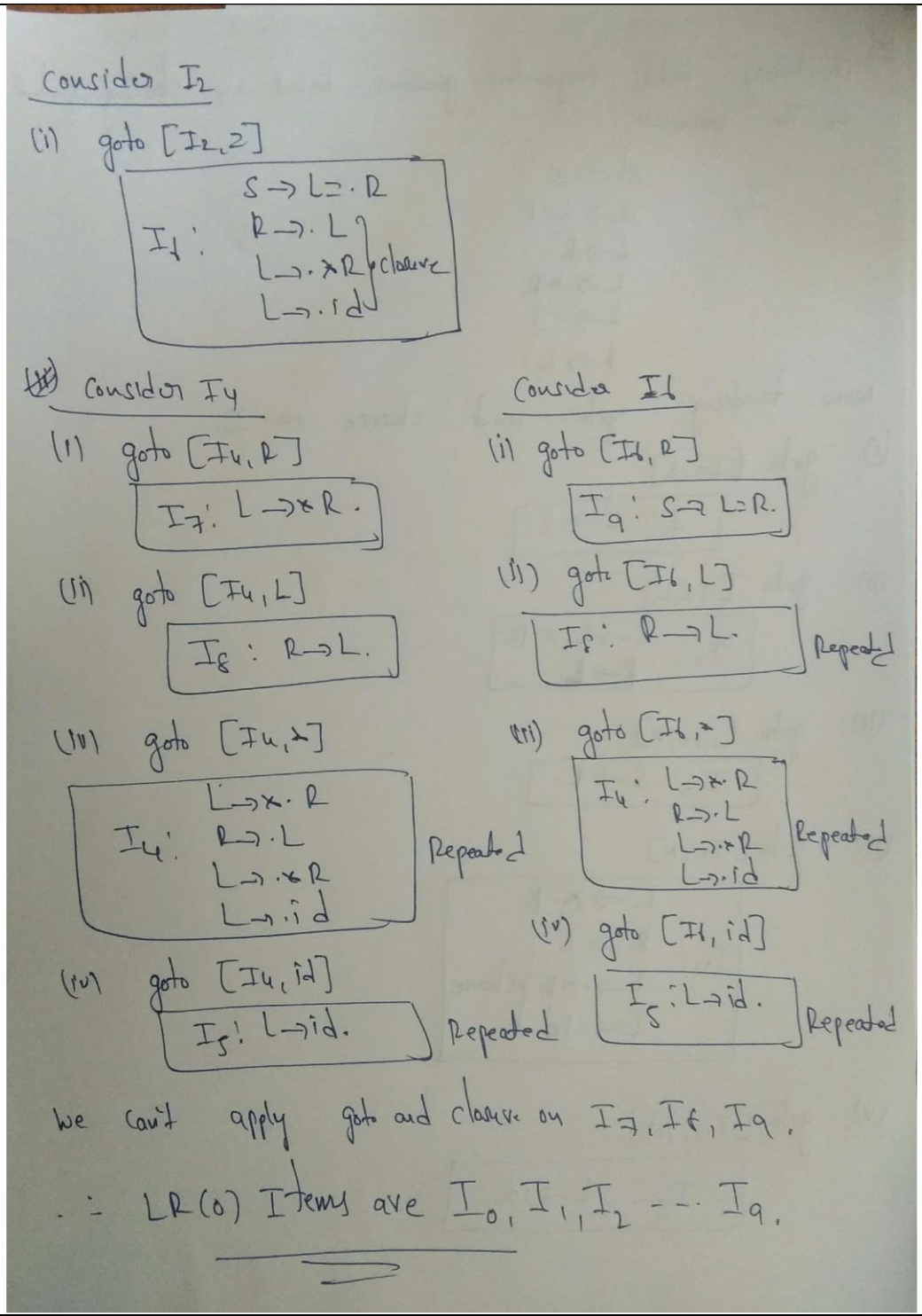
I_3: $S \rightarrow R.$

(iv) goto $[I_0, *]$

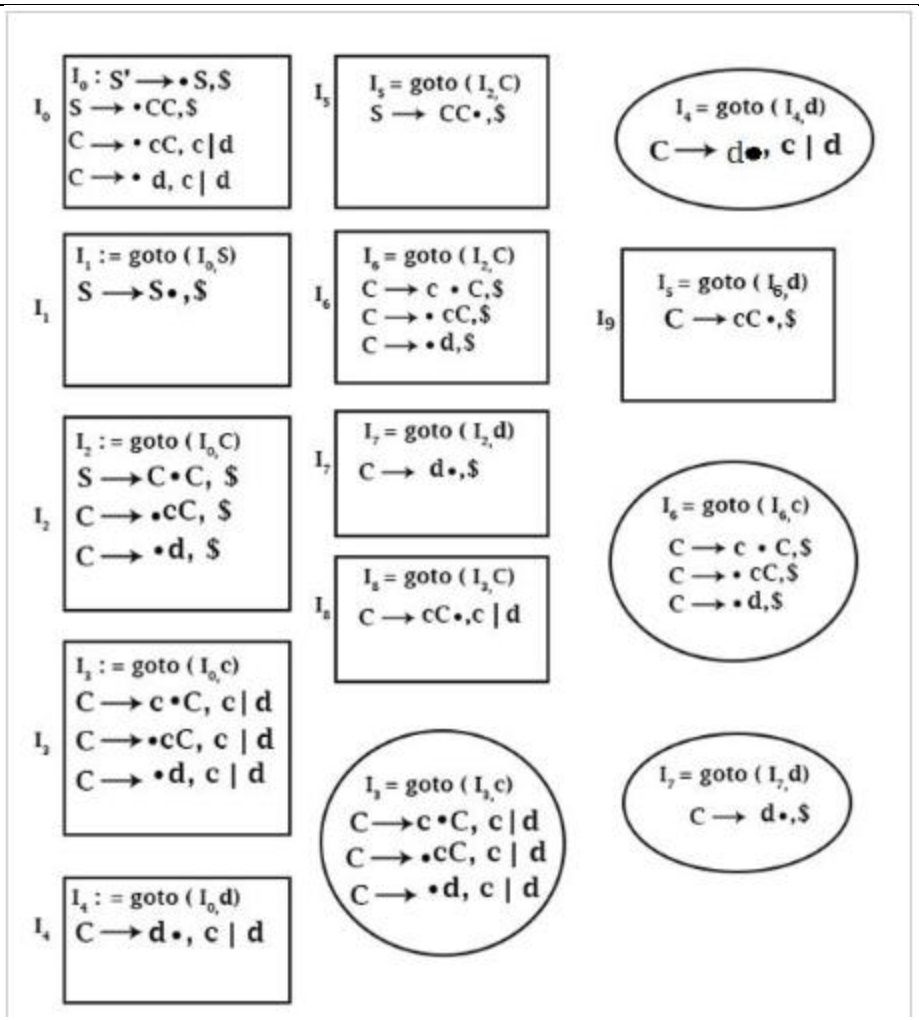
I_4:
$L \rightarrow *.R$
$R \rightarrow .L$ } closure
$L \rightarrow .*R$ } closure
$L \rightarrow .id$ } closure

(v) goto $[I_0, id]$

I_5: $L \rightarrow id.$



5		Construct CLR Parsing table for the given grammar S→CC C→aC/d **Solution** **Step1** − Construct LR (1) Set of items. First of all, all the LR (1) set of items should be generated.	[L6][CO3]	[12M]



CLR Parsing table:

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

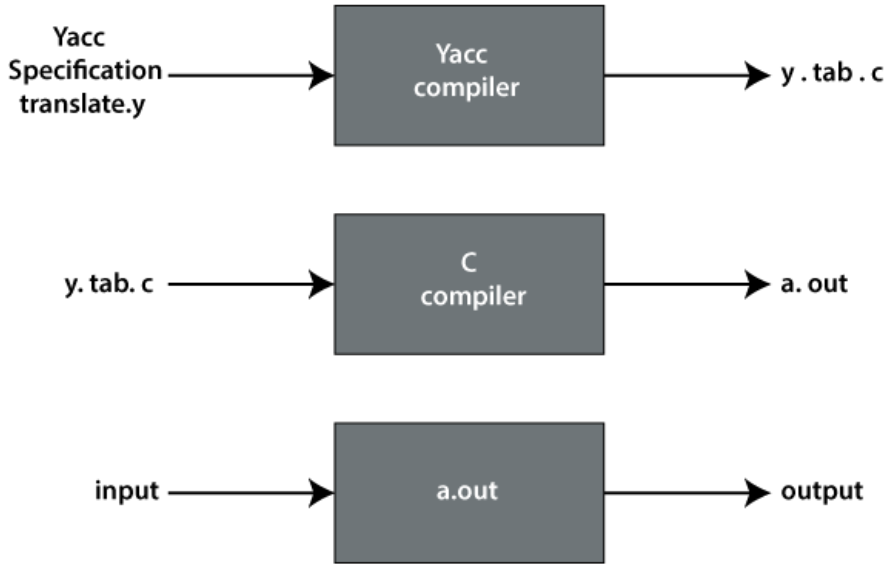
6		Design the LALR parser for the following Grammar S → AA A → aA A →b **LALR (1) Grammar** S → AA A → aA A → b Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.	[L6][CO3]	[12M]

S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I0 State:
Add Augment production to the I0 State and Compute the ClosureL
I0 = Closure (S` → •S)
Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes
I0 = S` → •S, $
S → •AA, $
Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.
I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)
Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes
I2= S → A•A, $
A → •aA, $
A → •b, $
I3= Go to (I0, a) = Closure (A → a•A, a/b)
Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes
I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b
Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)
I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)
Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes
I6 = A → a•A, $
A → •aA, $
A → •b, $
Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)
I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $
If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.
I3 = { A → a•A, a/b

		A → •aA, a/b A → •b, a/b } I6= { A → a•A, $ A → •aA, $ A → •b, $ } Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36. I36 = { A → a•A, a/b/$ A → •aA, a/b/$ A → •b, a/b/$ } The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47. I47 = {A → b•, a/b/$} The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89. I89 = {A → aA•, a/b/$} **LALR (1) Parsing table:**		

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

7	a	Define YACC parser in Syntax Analysis. ○ YACC stands for Yet Another Compiler Compiler. ○ YACC provides a tool to produce a parser for a given grammar. ○ YACC is a program designed to compile a LALR (1) grammar. ○ It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar and the output is a C program	[L1][CO3]	[2M]
	b	Explain in detail about YACC Parser generator tool. YACC Tool	[L2][CO3]	[10M]



A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided into three parts.

/* definitions */

....

%%

/* rules */

....

%%

/* auxiliary routines */

....

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:

%token NUMBER

%token ID

-

- Yacc automatically assigns numbers for tokens, but it can be overridden by

%token NUMBER 621

		• Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes. • The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column. • It can also include the specification of the starting symbol in the grammar: • %start nonterminal • Input File: Rule Part: • The rules part contains grammar definition in a modified BNF form. • Actions is C code in { } and can be embedded inside (Translation schemes). Input File: Auxiliary Routines Part: • The auxiliary routines part is only C code. • It includes function definitions for every function needed in rules part. • It can also contain the main() function definition if the parser is going to be run as a program. • The main() function must call the function yyparse().		
8	a	Explain syntax directed definition with simple examples **Syntax Directed Definition** Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production X –> a is associated with it a set of production rules of the form s = f(b_1, b_2, ……b_k) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }). Example 1: E --> E_1 + T { E.val = E_1.val + T.val} Example 2: Consider the following grammar S --> E	[L2][CO2]	[6M]

E --> E_1 + T
E --> T
T --> T_1 * F
T --> F
F --> digit
The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

b Describe in detail the Translation scheme of SDD. [L2][CO2] [6M]

Syntax Directed Translation is a set of productions that have semantic rules embedded inside it. The syntax-directed translation helps in the semantic analysis phase in the compiler. SDT has semantic actions along with the production in the grammar.

This article about postfix SDT and postfix translation schemes with parser stack implementation of it. Postfix SDTs are the SDTs that have semantic actions at the right end of the production. This article also includes SDT with actions inside the production, eliminating left recursion from SDT and SDTs for L-attributed definitions.

Postfix Translation Schemes:

- The syntax-directed translation which has its semantic actions at the end of the production is called the **postfix translation scheme.**
- This type of translation of SDT has its corresponding semantics at the last in the RHS of the production.
- SDTs which contain the semantic actions at the right ends of the production are called **postfix SDTs.**

Example of Postfix SDT

```
S ⇢ A*B{S.val = A.val * B.val}
A ⇢B1{A.val = B.val + 1}
B ⇢num{B.val = num.lexval}
```

Parser-Stack Implementation of Postfix SDTs:

Postfix SDTs are implemented when the semantic actions are at the right end of the production and with the bottom-up parser(LR parser or shift-reduce parser) with the non-terminals having synthesized attributes.

- The parser stack contains the record for the non-terminals in the grammar and their corresponding attributes.
- The non-terminal symbols of the production are pushed onto the parser stack.
- If the attributes are synthesized and semantic actions are at the right ends then attributes of the non-terminals are evaluated for the symbol in the top of the stack.
- When the reduction occurs at the top of the stack, the attributes are available in the stack, and after the action occurs these attributes are replaced by the corresponding LHS non-terminal and its attribute.
- Now, the LHS non-terminal and its attributes are at the top of the stack.

Production

```
A ⇢ BC{A.str = B.str . C.str}
B ⇢a {B.str = a}
C ⇢b{C.str = b}
```

Initially, the parser stack:

```
B       C        Non-terminals
B.str   C.str    Synthesized attributes
          ↑
        Top of Stack
```

After the reduction occurs A ⇢BC then after B, C and their attributes are replaced by A and in the attribute. Now, the stack:

```
A            Non-terminals
A.str        Synthesized attributes
↑
```

Top of stack

SDT with action inside the production:

When the semantic actions are present anywhere on the right side of the production then it is **SDT with action inside the production.**
It is evaluated and actions are performed immediately after the left non-terminal is processed.

This type of SDT includes both S-attributed and L-attributed SDTs.
If the SDT is parsed in a bottom-up parser then, actions are performed immediately after the occurrence of a non-terminal at the top of the parser stack.

If the SDT is parsed in a top-down parser then, actions are before the expansion of the non-terminal or if the terminal checks for input.

Example of SDT with action inside the production

```
S ⇢ A +{print '+'} B
A ⇢ {print 'num'}B
B ⇢ num{print 'num'}
```

9 a Define a syntax-directed translation and explain with example. [L2][CO2] [6M]

SYNTAX DIRECTED TRANSLATION

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer

b Give the evaluation order of SDT with an example. [L5][CO2] [6M]

Evaluation order of SDD

Evaluation order for SDD includes how the SDD(Syntax Directed Definition) is evaluated with the help of attributes, dependency

graphs, semantic rules, and S and L attributed definitions. SDD helps in the semantic analysis in the compiler so it's important to know about how SDDs are evaluated and their evaluation order. This article provides detailed information about the SDD evaluation. It requires some basic knowledge of grammar, production, parses tree, annotated parse tree, synthesized and inherited attributes.

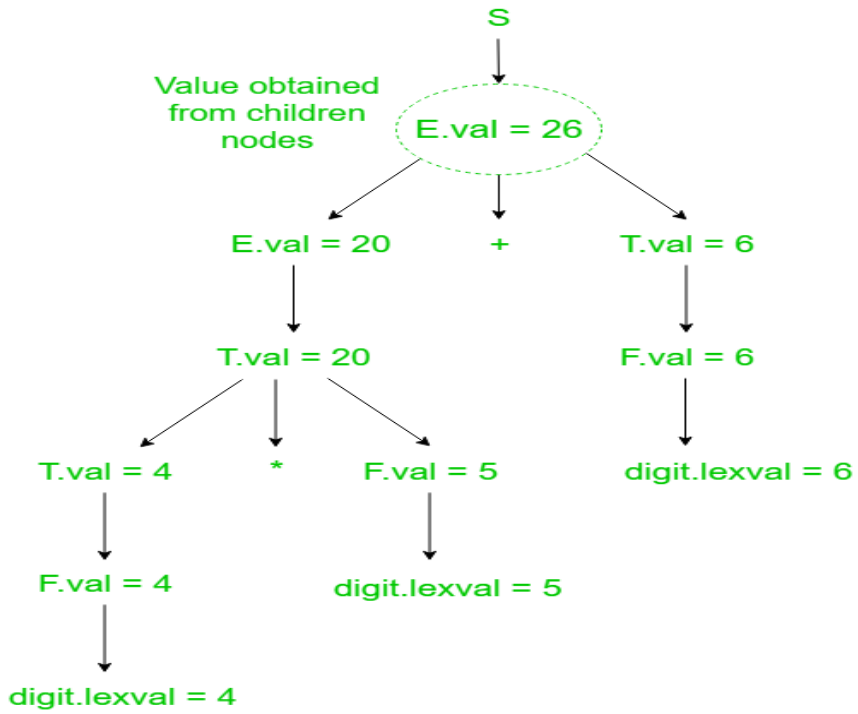
Terminologies:

- Parse Tree: A parse tree is a tree that represents the syntax of the production hierarchically.
- Annotated Parse Tree: Annotated Parse tree contains the values and attributes at each node.
- Synthesized Attributes: When the evaluation of any node's attribute is based on children.
- Inherited Attributes: When the evaluation of any node's attribute is based on children or parents.

Ordering the Evaluation of Attributes:

The dependency graph provides the evaluation order of attributes of the nodes of the parse tree. An edge(i.e. first node to the second node) in the dependency graph represents that the attribute of the second node is dependent on the attribute of the first node for further evaluation. This order of evaluation gives a linear order called topological order.
There is no way to evaluate SDD on a parse tree when there is a cycle present in the graph and due to the cycle, no topological order exists.

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval



Annotated Parse Tree

10 Discuss Type Checking with suitable examples. [L2][CO4] [12M]

TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called static checking, detects and reports programming errors.

Some examples of static checks:

1. Type checks- A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. Flow-of-control checks- Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An enclosing statement, such as break, does not exist in switch statement.

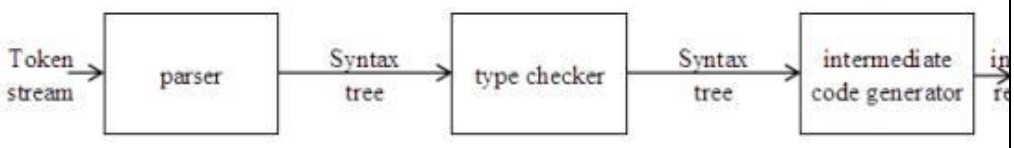


Fig. 2.6 Position of type checker

Fig. 2.6 Position of type checker

A typechecker verifies that the type of a construct matches that expected by its context. For example : arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that

the operands of mod have type integer. Type information gathered by a type checker may be needed when code is generated.

Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.
For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

The type of a language construct will be denoted by a "type expression." A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.
   A special basic type, type_error , will signal an error during type checking; void denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.

   Constructors include:

   Arrays : If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

   Products : If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

   Records : The difference between a record and a product is that the names. The record type constructor will be applied to a tuple formed from field names and field types.

   For example:

```
type row = record
address: integer;
        lexeme: array[1..15] of char
```

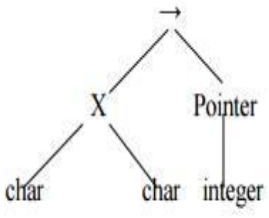
```
        end;
    var table: array[1...101] of row;
```

declares the type name row representing the type expression record((address X integer) X (lexeme X array(1..15,char))) and the variable table to be an array of records of this type.

Pointers : If T is a type expression, then pointer(T) is a type expression denoting the type "pointer to an object of type T". For example, var p: ↑ row declares variable p to have type pointer(row).

Functions : A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.



Fg. 5.7 Tree representation for char x char → pointer (integer)

Fg. 5.7 Tree representation for char x char → pointer (integer)

Type systems

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. It is specified in a syntax-directed manner. Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

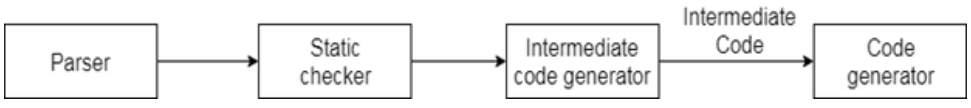
Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A sound type system eliminates the need for dynamic checking fo allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound

		type system assigns a type other than type_error to a program part, then type errors cannot occur when the target code for the program part is run. **Strongly typed language** A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.		

UNIT –V
CODE OPTIMIZATION AND CODE GENERATION

1		Analyse different types of Intermediate Code with an example. Intermediate code Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.  Fig: Position of intermediate code generator o If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine. o The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine. o Intermediate code generator receives input from its	[L4][CO5]	[12M]

predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.

- Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

The following are commonly used intermediate code representations:

Postfix Notation: Also known as reverse Polish notation or suffix notation. The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation, the operator follows the operand.

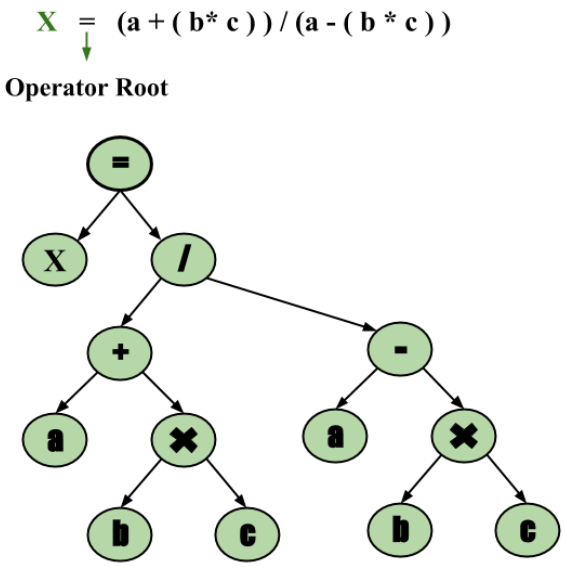
Example 1: The postfix representation of the expression (a + b) * c is : ab + c *

Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+

1. Three-Address Code: A statement involving no more than three references(two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as a three address code. Three address statement is of form x = y op z, where x, y, and z will have address (memory location). Sometimes a statement might contain less than three references but it is still called a three address statement.
   Example:

2. **Syntax Tree**: A syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by the single link in the syntax tree the internal nodes are operators and child nodes are operands. To form a syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example: x = (a + b * c) / (a – b * c)



Three Address Code:

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.
General representation –
a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

The three address code for the expression a + b * c + d :
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d where T 1 , T 2 , T 3 are temporary variables.
Example-1: Convert the expression a * – (b + c) into three address code.

$t_1 = b + c$
$t_2 = uminus\ t_1$
$t_3 = a * t_2$

		Example-2: Write three address code for following code
		for(i = 1; i<=10; i++)
		{
		a[i] = x * 5;
		}
		i = 1
		L : t_1 = x * 5
		t_2 = &a
		t_3 = sizeof(int)
		t_4 = t_3 * i
		t_5 = t_2 + t_4
		*t_5 = t_1
		i = i + 1
		if i<=10 goto L
		There are 3 ways to implement a Three-Address Code in compilerdesign
		i)Quadruple
		ii)Triples
		iii) Indirect Triple		

2		**Explain Representation of Three Address Codes with suitable Examples** Implementation of Three Address Code – There are 3 representations of three address code namely 1. Quadruple 2. Triples 3. Indirect Triples 1. Quadruple – It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression. Advantage – • Easy to rearrange code for global optimization. • One can quickly access value of temporary variables using symbol table. Disadvantage – • Contain lot of temporaries. • Temporary variable creation increases time and space complexity. Example – Consider expression a = b * – c + b * – c. The three address code is: t1 = uminus c	[L2][CO5]	[12M]

t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples –
This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.
Disadvantage –

- Temporaries are implicit and difficult to rearrange code It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –
This representation makes use of pointer to the listing of all references to computations which is made separately and

stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.
Example – Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

3 Produce quadruple, triples and indirect triples for following expression: (x + y) * (y + z) + (x + y + z) [L6][CO5] [12M]

Explanation – The three address code is:
t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

4 a Discuss function preserving transformations. [L2][CO6] [6M]

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation,
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

t1: = 4*i

t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

x=Pi;
A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated.

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

b Describe about loop optimization technique . [L2][CO5] [6M]

Loop optimization

- Loop optimization
  - Consumes 90% of the execution time

  ⇒ a larger payoff to optimize the code within a loop

- Techniques
  - Loop invariant detection and code motion
  - Induction variable elimination
  - Strength reduction in loops
  - Loop unrolling
  - Loop peeling

- Loop invariant detection and code motion
  - If the result of a statement or expression does not change within a loop, and it has no external side-effect
  - Computation can be moved to outside of the loop
  - Example

    for (i=0; i<n; i++) a[i] := a[i] + x/y;

    - Three address code

```
for (i=0; i<n; i++)
  {
     c := x/y;
     a[i] := a[i] + c;
  }
```

⇒ c := x/y;
for (i=0; i<n; i++) a[i] := a[i] + c;

Code Motion:

- Move the code from inner side to outer side

Example:

```
if (a<b) then                    temp = x * 2
z = x * 2          ⇒             if (a<b) then
else                             z = temp
 y = 10                          else
                                        y = 10
```

Induction variable Elimination:

```
t := b * c
FOR i := 1 TO10000 DO
        BEGIN
        a := t
d := i * 3...
        END
```

...

- *Where d* is an induction variable.

Strength Reduction:

- Replacement of an operator with a less costly one.

  Example:

```
                                  temp = 5;
```

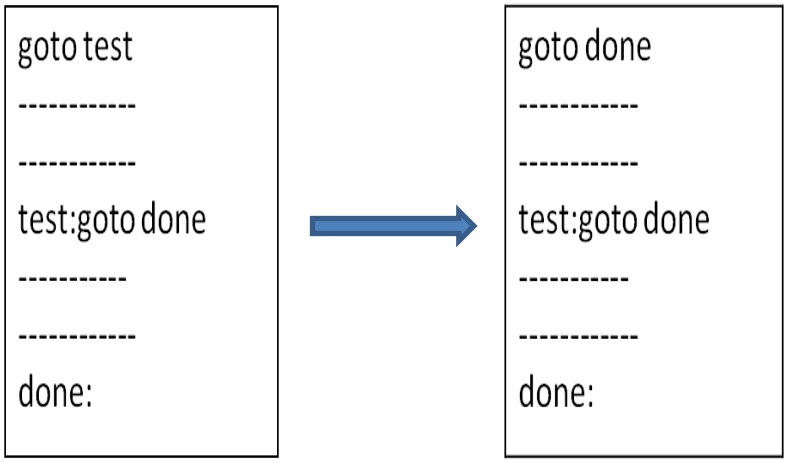
		for i=1 to 10 do … x = i * 5 … end  for i=1 to 10 do … x = temp … temp = temp + 5 end Typical cases of strength reduction occurs in address calculation of array references. **Loop unrolling:** • Execute loop body multiple times at each iteration • Get free of the conditional branches, if possible • Allow optimization to cross multiple iterations of the loop • Especially for parallel instruction execution • Space time tradeoff • Increase in code size, reduce some instructions **Loop peeling:** • Similar to unrolling • But unroll the first and/or last few iterations **Algebraic identities:** • Worth recognizing single instructions with a constant operand: A * 1 = A A * 0 = 0 A / 1 = A A * 2 = A + A More delicate with floating-point • Strength reduction: **A ^ 2 = A * A**		

5		Explain the peephole optimization Technique with examples.
		Peephole Optimization
		• Generally code generation algorithms produce code statement by statement.
		• This may contain redundant instructions.
		• This efficiency of such code can be improved by applying peephole optimization.
		• It is a simple but effective technique for locally improved target code.
		• It is a small window moving on target code and transformations can be made.
		
		Characteristics of Peephole Optimization
		• The peephole optimization can be applied on the target code using following characteristics.
		❖ Redundant Instruction elimination.
		❖ Flow of control optimization.
		❖ Algebraic Simplifications.
		❖ Reduction in strength.
		Redundant Instruction elimination
		• Especially the redundant loads and stores can be eliminated in this type of transformations.
		Example:
		MOV R0,X
		MOV X,R0
		In the above code second instruction can be eliminated because x is already in R0**.**
			[L2][CO5]	[12M]

Flow of control optimization.

- By using Peephole optimization , unnecessary jumps on jumps can be eliminated.

Example:



Algebraic Simplifications

- Using peephole optimization we simplyfy algebraic expressions.
- Example :

  X=x+0 or x=x*1

can be eliminated by using peephole optimization.

Reduction in strength

- Reduction in strength means substitute lowest operators in place of highest operators.

  X=2*2 → X=2+2

  X*X ---> X^2

6	a	Define and Show Dead-code elimination with example. **Dead Code Elimination:** • A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. • A dead (or useless) code — statements that compute values that never get used. – Ineffective statements • x := y + 1 (immediately redefined, eliminate!) • y := 5 ⇒ y := 5 • x := 2 * z x := 2 * z – A variable is dead if it is never used after last definition **Eliminate assignments to dead variables**	[L1][CO4]	[6M]
	b	List and explain the Issues in the design of a code generator **<u>Issues in the Design of a Code Generator</u>** • The most important criterion for a code generator is that it should produce correct code. • Correctness takes on special significance because of the number of special cases that a code generator might face. • Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal. There are 6 issues in the generation code as follows: • Input to the Code Generator • Memory manager	[L2][CO6]	[6M]

- The Target Program
- Instruction Selection
- Register Allocation
- Evaluation Order

1. Input to the Code Generator:

- The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table.
- The many choices for the IR include
  - **Three-address** representations such as quadruples, triples, indirect triples;
  - **Linear** representations such as postfix notation; and
  - **Graphical** representations such as syntax trees and DAG's.

2. The Target Program:

- The most common target-machine architectures are **RISC** (reduced instruction set computer), **CISC** (complex instruction set computer), and stack based.
  - A **RISC machine** typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
  - A **CISC machine** typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- In a **stack-based machine**, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.
- **Stack-based architectures** were re-energized with the introduction of the **Java Virtual Machine** (JVM). The **JVM** is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers.

3. Instruction Selection:

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of performing this mapping is determined by a factors such as
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code.
- For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.
  - For example, every three-address statement of the form **x = y + z**, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD  R0, y       // R0 = y        (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z   (add z to R0)
ST  x, R0       // x = R0        (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

a = b + c
d = a + e

would be translated into

```
LD  R0, b        // R0 = b
ADD R0, R0, c    // R0 = R0 + c
ST  a, R0        // a = R0
LD  R0, a        // R0 = a
ADD R0, R0, e    // R0 = R0 + e
ST  d, R0        // d = R0
```

4. Register Allocation:

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine,
- The use of registers is often subdivided into two subproblems:
  1. **Register allocation**, during which we select the set of variables that will reside in registers at each point in the program.
  2. **Register assignment**, during which we pick the specific register that a variable will reside in.

t = a + b
t = t * c
t = t / d
Optimal Three address code for the above code is

MOV R0,R0,a
ADD R0,R0,b
MUL R0,R0,c
DIV R0, R0,d
ST t,R0

5. Evaluation Order:

- The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others.

Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

7	a	Analyse the different forms in target program.	[L4][CO6]	[6M]

The Target Program:

- The most common target-machine architectures are **RISC** (reduced instruction set computer), **CISC** (complex instruction set computer), and stack based.
  - A **RISC machine** typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
  - A **CISC machine** typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- In a **stack-based machine**, operations are done by pushing

operands onto a stack and then performing the operations on the operands at the top of the stack.

- **Stack-based architectures** were re-energized with the introduction of the **Java Virtual Machine** (JVM). The **JVM** is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers.

Instruction Selection:

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of performing this mapping is determined by a factors such as
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code.
- For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.
  - For example, every three-address statement of the form **x = y + z**, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD  R0, y       // R0 = y        (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z   (add z to R0)
ST  x, R0       // x = R0        (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD  R0, b       // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0       // a = R0
LD  R0, a       // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0       // d = R0
```

b Explain the target machine in code generator. [L2][CO6] [6M]

The Target Language

we shall use as a target language assembly code for a simple computer that is representative of many register machines.
There are Two Methods:

- A Simple Target Machine Model
- Program and Instruction Costs

1. A Simple Target Machine Model:

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte addressable machine with n general-purpose registers, RO, R1, . . . , Rn - 1.

- Our target computer models a three-address machine done following Operations
  - Load operation.

- ❖ Store operation.
- ❖ Computation operations.
- ❖ Unconditional jumps
- ❖ Conditional jumps

Load operation:

- ➢ The instruction **LD dst, addr**
  loads the value in location addr into location dst This instruction denotes the assignment dst = addr.
- ➢ The most common form of this instruction is **LD r, x** which loads the value in location x into register r.
- ➢ An instruction of the form **LD r, r2** is a register-to-register copy in which the contents of register r2 are copied into register r.

Store operation:

- ➢ The instruction ST x, r
  Which Stores the value in register r into the location x.
- ➢ This instruction denotes the assignment x = r.

Computation operations:

- ➢ Computation operations of the form OP dst, src1,src2,
  Where OP is a operator like ADD or SUB, and dst, src1, and src2 are locations, not necessarily distinct.
- ➢ The effect of this machine instruction is to apply the operation represented by OP to the values in locations src1 and src2, and place the result of this operation in location dst.
- ➢ For example, SUB n,r2,r3 computes
  n = r2 - r3. Any value formerly stored in n is lost, but if r1 is r2 or r 3 , the old value is read first. Unary operators that take only one operand do not have a src2.

Unconditional jumps:

Unconditional jumps: The instruction BR L causes control to branch to the machine instruction with label L. (BR stands for branch.)

Conditional jumps

Conditional jumps of the form Bcond r, L,

where r is a register, L is a label, and cond stands for any of the common tests on values in the register r.

For example, BLTZ r, L causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

- We may execute the three-address instruction b = **a [i]** by the machine instructions:

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a + contents(R1))
ST  b, R2          // b = R2
```

2. Program and Instruction Costs

- Some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

- Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given

source program is an undecidable problem in general.

- The Addressing Modes are used as follows

Addressing Mode	Form	Address	Cost
Absolute	M	M	1
Register	R	R	0
Indexed	C(R)	C+Contents(R)	1
Indirect Register	*R	Contents(R)	0
Indirect Indexed	*c(R)	Contents(c+contents(R))	1
literal	#c	c	1

8	a	Define flow Graph **Flow Graphs:** • Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. • There is an edge from block B to block *C* if and only if it is possible for the first instruction in block *C* to immediately follow the last instruction in block B. • There are two ways that such an edge could be justified: – There is a conditional or unconditional jump from the end of B to the beginning of C. – C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.	[L1][CO4]	[2M]
	b	Interpret optimization techniques on Basic Blocks with simple examples? **Optimization of Basic Blocks** Obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. **1. The DAG Representation of Basic Blocks:** • Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph). • We construct a DAG for a basic block as follows: 1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block. 2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s. 3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block. 4. Certain nodes are designated ***output nodes***. These are the nodes whose variables are ***live on exit*** from the block; that is, their values may be used later, in another block of the	[L3][CO5]	[10M]

flow graph.

- The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.
    1. We can **eliminate *local common subexpressions***, that is, instructions that compute a value that has already been computed.
    2. We can **eliminate *dead code***, that is, instructions that compute a value that is never used.
    3. We can **reorder statements** that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
    4. We can **apply algebraic laws to reorder operands** of three-address instructions, and sometimes t hereby simplify the computation.

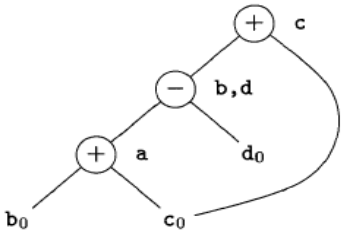
2. Finding Local Common Subexpressions:

- Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator.

Example 8.10: A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement c = b + c, we know that the use of b in b + c refers to the node of Fig. 8.12 labeled -, because that is the most recent definition of b. Thus, we do not confuse the values computed at statements one and three.



ıre 8.12: DAG for basic block in Example 8.10

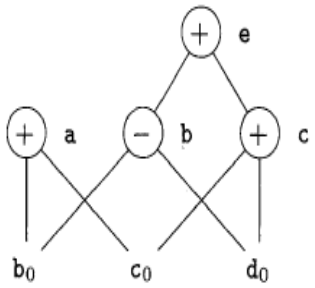
However, the node corresponding to the fourth statement d = a - d has the operator - and the nodes with attached variables a and do as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add d to the list of definitions for the node labeled -.

3. Dead Code Elimination:

- The operation on DAG's that corresponds to dead-code elimination can be implemented as follows.

We delete from a DAG any root (**node with no ancestors**) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

- **Example 8.12:** If, in Fig. 8.13, a and b are live but c and e are not, we can immediately remove the root labeled ***e.*** Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached.

		a = b + c; b = b - d c = c + d e = b + c  ure 8.13: DAG for basic block in Example 8.11 **4. The Use of Algebraic Identities:** • Algebraic identities represent another important class of optimizations on basic blocks. • For example, we may apply arithmetic identities, such as to eliminate computations from a basic block. $x+0=0+x=x$ $\quad x-0=x$ $x\times 1=1\times x=x$ $\quad x/1=x$ • Another class of algebraic optimizations includes local reduction in strength, that is, replacing a more expensive operator by a cheaper one as in: EXPENSIVE = CHEAPER $x^2 = x\times x$ $2\times x = x+x$ $x/2 = x\times 0.5$ • A third class of related optimizations is **constant folding**. **Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression *2 * 3.14* would be replaced by 6.28.**		
9	a	Analyze Simple code generator **A Simple Code Generator** • Consider an algorithm that generates code for a single basic block. • It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can **avoid generating unnecessary loads** and **stores**. • One of the primary **issues** during code generation is deciding **how to use registers** to best advantage. • There are four principal uses of registers: 1. In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation. 2. Registers make good temporaries - places to hold the result of a subexpression while a larger expression is being evaluated, 3. Registers are used to hold (global) values that are computed in one basic block and used in other blocks.	[L4][CO6]	[6M]

4. Registers are often used to help with run-time storage management.

1. Register and Address Descriptors:

- Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the **needed operands into registers.**
- In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers.
- The desired data structure has the following descriptors:
    1. a **register descriptor** keeps track of the variable names whose current value is in that register.
    2. an **address descriptor** keeps track of the location or locations where the current value of that variable can be found.

2. The Code-Generation Algorithm:

- An essential part of the algorithm is a function **getReg(I)**, which selects registers for each memory location associated with the three-address instruction I.
- A possible improvement to the algorithm is to generate code for both x = y + z and x = z + y whenever + is a commutative operator, and pick the better code sequence.

Machine Instructions for Operations:

- For a three-address instruction such as x = y + z, do the following:
    1. Use getReg (x = y + z) to select registers for x, y, and z. Call these R_x, R_y and R_z.
    2. If y is not in R_y (according to the register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
    3. Similarly, if z is not in R_z, issue and instruction LD R_z, z', where z' is a location for *z* .
    4. Issue the instruction ADD R_x, R_y, R_z.

Machine Instructions for Copy Statements:

- There is an important special case: a three-address copy statement of the form x = y.
- We assume that getReg will always choose the same register for both x and y.

Ending the Basic Block:

As we have described the algorithm, variables used by the block may wind up with their only location being a register. If the variable is a temporary used only within the block, that is fine; when the block ends, we can forget about the value of the temporary and assume its register is empty.

Managing Register and Address Descriptors:

As the code-generation algorithm issues load, store, and other machine instructions, it needs to update the register and address descriptors. The rules are as follows:

1. For the instruction LD R, x.
   (a). Change the register descriptor for register R so it holds only x.
   (b). Change the address descriptor for x by adding register R as an additional location.
2. For the instruction ST x, R, change the address descriptor for x to include its own memory location.

3. For an operation such as ADD R_x, R_y , R_z, implementing a three-address instruction x = y + z.
   (a). Change the register descriptor for R_x so that it holds only x.
   (b). Change the address descriptor for x so that its only location is R_x.
   (c). Remove R_x from the address descriptor of any variable other than x.
4. When we process a copy statement x = y, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):
   (a) Add x to the register descriptor for R_y.
   (b) Change the address descriptor for x so that its only location is R_y.

Example 8.16 : Let us translate the basic block consisting of the three-address statements

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

A summary of all the machine-code instructions generated is in Fig. 8.16. The figure also shows the register and address descriptors before and after the translation of each three-address instruction.

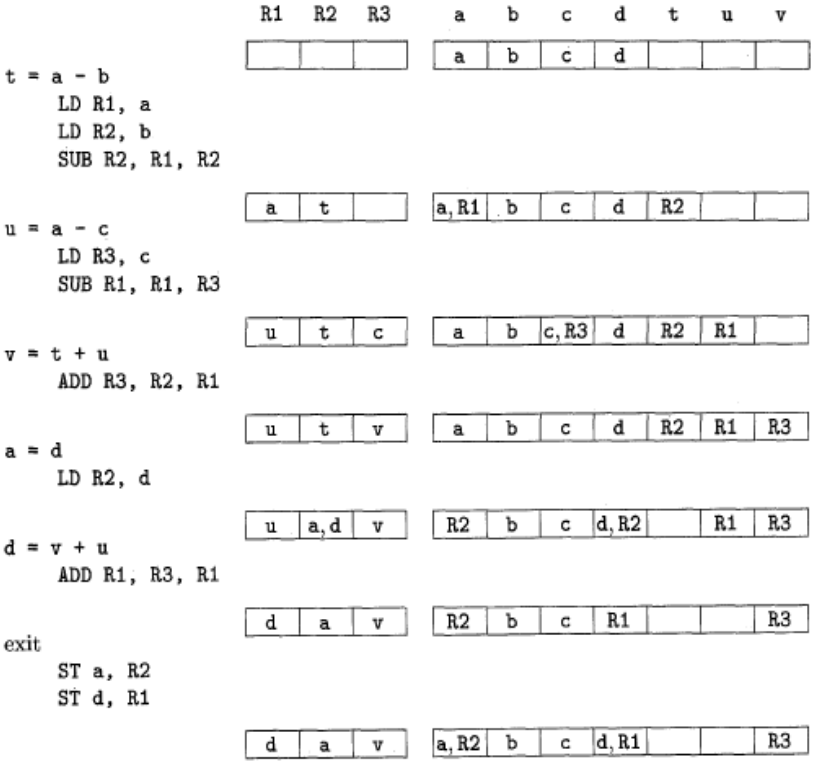


Figure 8.16: Instructions generated and the changes in the register and address descriptors

b Evaluate Register allocation and register assignment techniques [L5][CO6] [6M]

Register Allocation and Assignment

- Instructions involving only register operands are faster than those involving memory operands.
- One approach to register allocation and assignment is to assign specific values in the target program to certain registers.

1. Global Register Allocation:

- Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop.
- One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop.

2. Usage Counts:

- Assume that the savings to be realized by keeping a variable x in a register for the duration of a loop L is one unit of cost for each reference to x if x is already in a register.
- On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units.
- Thus, an approximate formula for the benefit to be realized from allocating a register x within loop L is

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B) \qquad (8.1)$$

Example 8.17 : Consider the the basic blocks in the inner loop depicted in Fig. 8.17, where jump and conditional jump statements have been omitted. Assume registers RO, R1, and R2 are allocated to hold values throughout the loop.

For example, notice that both e and f are live at the end of B1, but of these, only e is live on entry to B2 and only f on entry to B3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.

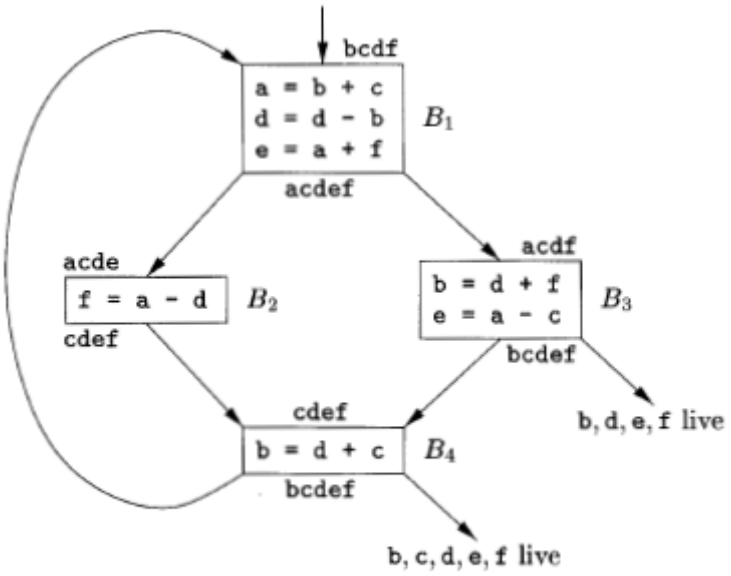


Figure 8.17: Flow graph of an inner loop

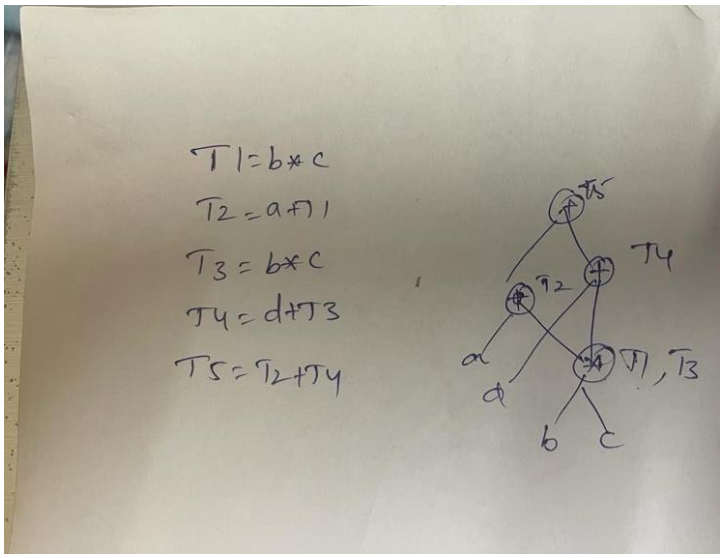
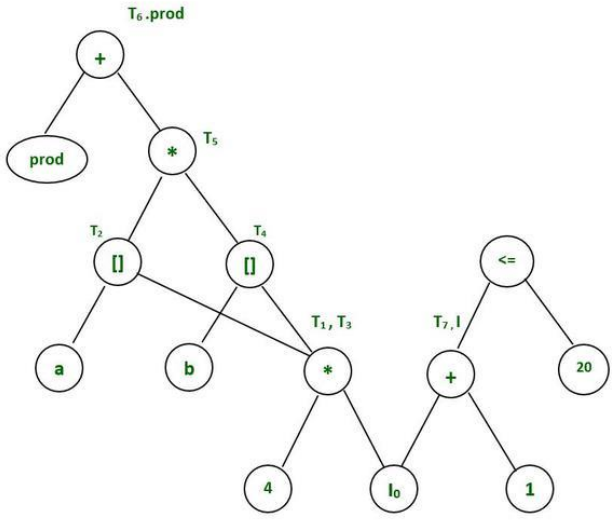
3. Register Assignment for Outer Loops:

- Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops.
- If an outer loop L1 contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1 - L2.

4. Register Allocation by Graph Coloring:

- When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored *(spilled)* into a memory location in order to free up a register.

Graph coloring is a simple, systematic technique for allocating

		registers and managing **register spills.**.		
10	a	Create the DAG for following statement. a+b*c+d+b*c **First convert given expression into 3-address code then construct the DAG for the converted 3-address code**  	[L6][CO6]	[4M]
	b	Construct the DAG for the following basic blocks 1. t1:=4*i 2. t2:=a[t1] 3. t3:=4*i 4. t4:=b[t3] 5. t5:=t2*t4 6. t6:=prod+t5 7. prod:=t6 8. t7:=i+1 9. i:=t7 if i<=20 goto 1  	[L6][CO6]	[8M]

Prepared by:

B Pavan kumar
Assoc. Prof./CSE