USN | | | | | | | | | |

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 3 – March 2024

Sub:	Automata Theory and Compiler Design					Sub Code:	21CS51	Branch :	ISE
Date:	/3/2024	Duration:	90 min's	Max Marks:	50	Sem / Sec:	V A, B, C		OBE

Answer any FIVE FULL Questions — MARKS | CO | RBT

1 (a) Obtain a PDA to accept the language. [2+2=4] CO3 L2

i. L= {w∈(a+b)* | w such that w has an equal number of a's and b's}

(1.) $\delta(q_0, a, z_0) = (q_0, a z_0)$ — push a
(2.) $\delta(q_0, b, z_0) = (q_0, b z_0)$ — push b
(3.) $\delta(q_0, a, a) = (q_0, aa)$ — push a
(4.) $\delta(q_0, b, b) = (q_0, bb)$ — push b
(5.) $\delta(q_0, a, b) = (q_0, \epsilon)$ — pop b
(6.) $\delta(q_0, b, a) = (q_0, \epsilon)$ — pop a
(7.) $\delta(q_0, \epsilon, z_0) = (q_f, \epsilon)$ — accepted.

accept by empty store $F = \{$

ii. $L=\{a^n b^n \mid n>=1\}$ by a final state.

Construct PDA

P_1 : $L = \{a^n b^n \mid n >= 1\}$ $Q = \{q_0, q_1, q_f\}$ no required if $F=\{\phi\}$

$Q = \{q_0, q_1, q_f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, z_0\}$,

q_0 - start state

$F = \{q_f\}$ $F=\{\phi\}$

z_0 - initial push down symbol

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

$\delta(q_0, a, z_0) = (q_0, a z_0)$

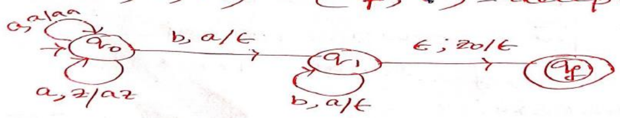
$\delta(q_0, a, a) = (q_0, aa)$

$\delta(q_0, b, a) = (q_1, \epsilon)$

$\delta(q_1, b, a) = (q_1, \epsilon)$

$\delta(q_1, \epsilon, z_0) = (q_f, z_0)$ - accepted / (q_1, ϵ)

[In this case $F=\{\phi\}$]



(b) Discuss recursive descent parsing with a suitable example. [6] CO2 L3

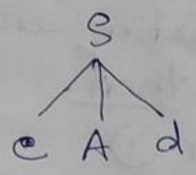
Recursuve descent parsing:

- General recursive descent parsing may require backtracking i.e., it may require repeated scans over the input.
- Consider grammar

$$S \to cAd.$$
$$A \to ab \mid a.$$

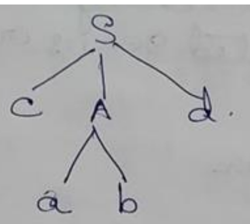
To construct a parse tree (top-down) for the input string $w = cad$, begin the tree with a single node S.
The input pointer is pointing to 'c', the first symbol of 'w'.

- S has only one production, so expand S and we get a tree (a).



(a)

- The leftmost leaf, labeled 'c' matches the first symbol of input 'w', so we advance the input pointer to 'a' of $w = cad$, the second symbol of w.
- Consider the next leaf labelled 'A'. we expand A using first alternative $A \to ab$ to obtain the tree (b).

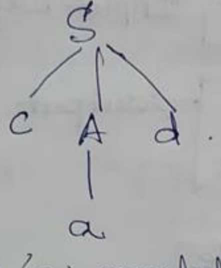


(b)

- We have a match for the second input 'a', so we advance the input pointer to 'd', the third input symbol and compare 'd' against next leaf labelled 'b'.

Since 'b', does not match with 'd', we report failure, and go back to 'A' to check whether there is any other alternative for 'A', that has not been tried but that might produce a 'match'.

- We reset the pointer to '2' position in A.
- The second alternative for A $(A \to a)$ produces the tree (c)



- The leaf 'a' matches the second symbol of 'w' and the leaf 'd' matches the third symbol of 'w'. Hence we halt and announce successful completion of parsing.

, Left recursion grammar for example:-

$E \rightarrow E+T/T$ ~~can~~ with backtracking can get into infinite loop. ~~To~~ the avoid ~~and~~ eliminate recursion,

$T \rightarrow T*F/F$

$F \rightarrow (E)/id.$

by transforming grammar to:-

$E \rightarrow TE'$

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

$T \rightarrow FT'.$

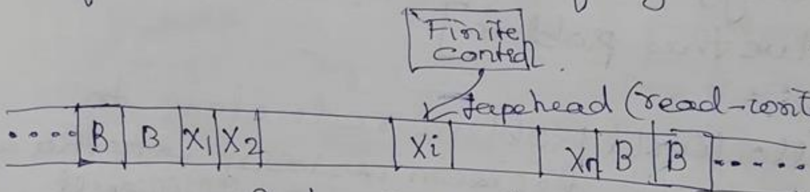
$T' \rightarrow *FT'/\varepsilon.$

$F \rightarrow (E)/id.$

2 (a) Design a Turing Machine and show its moves. [2] CO3 L2

* Turing m/c consists of a finite control, which can be in any of a ~~finite~~ finite set of states.

* There is a tape divided into squares or cells ; each cell can hold any one of a finite number of symbols.



A turing m/c.

* Initially, the input, which is a finite-length string of symbols chosen from input alphabet, is placed on the tape.
* All other tape cells, extending infinitely to the left and the right, initially hold a special symbol called the <u>blank</u>.
* The blank is a tape symbol, but not an input symbol. There may be other tape symbols besides the ~~other~~ input and the blank symbols.
* There is a tape head always positioned at one of the tape cells. The TM is said to be scanning that cell.
* The string to be scanned will be stored from the leftmost position ~~on~~ the tape. The string to be scanned should end with blanks.
* The read/write head can move in both left and right direction.

* The various actions performed by the m/c are :-

δ : The transition function - The arguments of $\delta(q, X)$ are a state 'q' and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where :

1. p is the next state, in Q.
2. Y is the symbol in Γ, written in the

* The turing m/c can be represented using
   1) Transition tables.
   2) Instantaneous descriptions (ID)
   3) Transition diagram.

The formal notation for turing machine (TM) can be defined by :-

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

Q : finite set of states of the finite control.

Σ : finite set of input symbols.

Γ : Complete set of tape symbols; Σ is always a subset of Γ.

δ: The transition function - The arguments of $\delta(q, X)$ are a state 'q' and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

① p is the next state, in Q.

② Y is the symbol in Γ, written in the cell being scanned, replacing whatever symbol was there.

③ D is the direction, either L or R standing for "left" or "right" respectively and telling us the direction in which the head moves.

q_0: The start state, a member of Q in which the finite control is found initially.

B: The blank symbol. This symbol is in Γ but not in Σ (not input symbol). The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of final or accepting states, a subset of Q.

* Moves of TM is represented by $\vdash^*$ (0, 1 or more moves).

* Suppose $\delta(q, X_i) = (p, Y, L)$ [Move left]

$$\underbrace{X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n}\ \vdash_M X_1 X_2 \cdots X_{i-2}\, p\, X_{i-1}\, Y X_{i+1} \cdots X_n$$

[X_i is replaced by Y]

This move reflects change to state 'p' and the tape head is positioned at cell [i-1]. There are two important exceptions.

① If $i=1$, then M (TM) moves to the blank to the left of X_i. In that case

$$\underbrace{q X_1 X_2 \cdots X_n} \vdash_M p B Y X_2 \cdots X_n$$

② If $i=n$ and $Y=B$, then the symbol B written over X_n joins the infinite sequence of ~~leading~~ trailing blanks and does not appear in the next ID. Thus.

$$X_1 X_2 \cdots X_{n-1} q X_n \vdash_M X_1 X_2 \cdots X_{n-2}\, p\, X_{n-1}.$$

* Now suppose $\delta(q, X_i) = (P, Y, R)$ [Move right]

Then
$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash_M$$
$$X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n.$$

Here move reflects that the head has moved to cell [i+1].

* There are 2 important exceptions :-

① If $i=n$, then $i+1$st cell holds a blank, and that cell was not part of the previous ID. Thus,

$$X_1 X_2 \cdots X_{n-1}\, q\, X_n \vdash_M X_1 X_2 \cdots X_{n-1}\, Y\, p B.$$

② If $i=1$, ~~then~~ and $Y=B$, then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID. Thus,

$$q X_1 X_2 \cdots X_n \vdash_M p X_2 \cdots X_n.$$

Acceptance of a language by TM.

TM can do one of the following:

(1) Halt and accept by entering into final state.

(2) Halt and reject. This is possible if $\delta(q, X)$ is not defined. ~~where~~

(3) TM will never halt and enters into an infinite loop.

* The language accepted by TM is called recursively enumerable language (RE language).

(b)	Design a predictive parser for the following grammar. Show the stack implementation for the input id+id*id. Construct a parse table by using FIRST and FOLLOW algorithm. E->E+T/T T->T*F/F	[3+3+2=8]	CO2	L3

F->(E)/ id

Construction of the parsing table will be done by using FIRST and FOLLOW.

FIRST(E) = { (, id } {Rule 4.2}

FIRST(E') = { ε, + } {Rule 2, Rule 3}

FIRST(T) = { (, id } {Rule 4.2}.

FIRST (T') = { *, ε }. {Rule 3, Rule 4.1}

FIRST(F) = { (, id }. {Rule 4.1}.

FOLLOW(E) = {), $ } {) follows E, Rule 1 for $ }

FOLLOW (E') = {), $ }. {Rule 3}.

FOLLOW (T) = { +,), $ } { Rule 2.2 for +, Rule 3 for), $ }

FOLLOW (T') = { +,), $ } { Rule 3 using T→FT' }.

FOLLOW (F) = { *, +,), $ }. { * from FIRST(T') using Rule 2.2. +,), by using Rule 3 }

$ E'T'F	id $	F→id.
$ E'T' id	id $	
$ E'T'	$	T'→ε.
$ E'	$	E'→ε.
$\boxed{\$}$	$\boxed{\$}$	

END OF MARKER

Moves made by predictive parser on i/p: id+id*id

Table Construction: Input Symbols

Non-Terminals	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Blank Entry denotes error.

3 (a)	Construct a Turing Machine which accepts the language L= $\{0^n1^n \mid n>=1\}$. Show the result of the moves using a transition table and the Turing Machine model at the end.	[2+2=4]	CO3	L2

(18)

[Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by an infinity of blanks. Alternatively, the TM will change 0 to an X and then a 1 to a Y, until all 0's and 1's have been matched.]

Consider situation.

$$X X 0 0 Y Y 1 1$$
$$\uparrow$$
$$q_0.$$

Here the first two 0's are replaced by Xs and first two 1's are replaced by Ys. In this situation, the read-write head points to the left most 0 and the m/c is in state q_0. This is the configuration and let us design the Turing m/c.

Step 1: In state q_0, replace 0 by X, change the state to q_1 and move the pointer towards right. The transition can be –

$$\delta(q_0, 0) = (q_1, X, R).$$

The resulting configuration is:

$$X X X 0 Y Y 1 1.$$
$$\uparrow$$
$$q_1$$

Step 2: In state q_1, we have to obtain the left-most 1 and replace it by Y. So let us move the pointer to point to leftmost 1. When the pointer is moved towards 1, the symbols encountered may be 0 and Y. Irrespective of what symbol is encountered, replace 0 by 0, and Y by Y, and remain in the state q_1 and move the pointer towards right. The transition can be of the form:-

$$\delta(q_1, 0) = (q_1, 0, R)$$
$$\delta(q_1, Y) = (q_1, Y, R)$$

Step 2: When these transitions are repeatedly applied, the following configuration is obtained.

$$\begin{array}{c} XXX0YY11 \\ \uparrow \\ q_1 \end{array}$$

Step 3: In state q_1, if the input symbol to be scanned is a 1, then replace 1 by Y, change the state to q_2 and move the pointer towards left. The transition can be of the form.

$$\delta(q_1, 1) = (q_2, Y, L)$$

The resulting configuration is:-

$$\begin{array}{c} XXX0YYY1 \\ \uparrow \\ q_2 \end{array}$$

Step 2: When these transitions are repeatedly applied, the following configuration is obtained.

XXX0YY11
$\uparrow$
q_1

Step 3: In state q_1, if the input symbol to be scanned is a 1, then replace 1 by Y, change the state to q_2 and move the pointer towards left. The transition can be of the form,

$$\delta(q_1, 1) = (q_2, Y, L)$$

The resulting configuration is:-

XXX0YYY1
$\uparrow$
q_2

Note that the pointer should be moved towards left. ~~This is because~~ Now we have to scan for the leftmost 0 and so the pointer move towards left.

Step 4: To obtain leftmost 0, we need to obtain rightmost X first. During this scanning process, we may encounter Y's and 0's. So the transitions may be

$\delta(q_2, Y) = (q_2, Y, L)$
$\delta(q_2, 0) = (q_2, 0, L)$.

The following configuration is obtained -

$$X X X 0 Y Y Y 1$$
$$\uparrow$$
$$q_2$$

Step 5 Now we have the right most X. To get the leftmost 0, replace X by X, change the state to q_0 and move the pointer towards right. The transition can be of the form:

$$\delta(q_2, X) = (q_0, X, R)$$ and the

configuration is -

$$X X X 0 Y Y Y 1$$
$$\uparrow$$
$$q_0$$

Then $\delta(q_0, 0) = (q_1, X, R)$
$\delta(q_1, 0) = (q_1, 0, R)$
$\delta(q_1, Y) = (q_1, Y, R)$
$\delta(q_1, 1) = (q_2, Y, L)$
$\delta(q_2, Y) = (q_2, Y, L)$
$\delta(q_2, 0) = (q_2, 0, L)$
$\delta(q_2, X) = (q_0, X, R)$

Now repeat the steps 1 through 5 to get the configuration as shown:-

$$X X X X Y Y Y Y$$
$$\uparrow$$
$$q_0$$

$\delta(q_0, Y) = (q_3, Y, R)$

In state q_3, we see only Y's and no more 1's. So we replace Y by Y and remain in q_3 only.
Repeatedly applying this transition, we obtain

$$XXXXYYYY\underset{\substack{\uparrow \\ q_3}}{B}$$

The string ends with infinite number of blanks and so in state q_3 if we encounter symbol B, this means the end of string is reached and there exists n nos of 0's followed by n nos of 1's.

So in state q_3, on i/p B, change the state to q_4, replace B by B and move pointer towards right and the string is accepted.
The transition for this is of the form:

$$\delta(q_3, B) = (q_4, B, R)$$

where q_4 is the final state and the configuration obtained is:

X X X X Y Y Y Y B B

$\uparrow$

q_4.

So the TM to accept the language,

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

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

where $Q = \{q_0, q_1, q_2, q_3, q_4\}$.

$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, X, Y, B\}$

$q_0 \in F$.

$B \in \Gamma$.

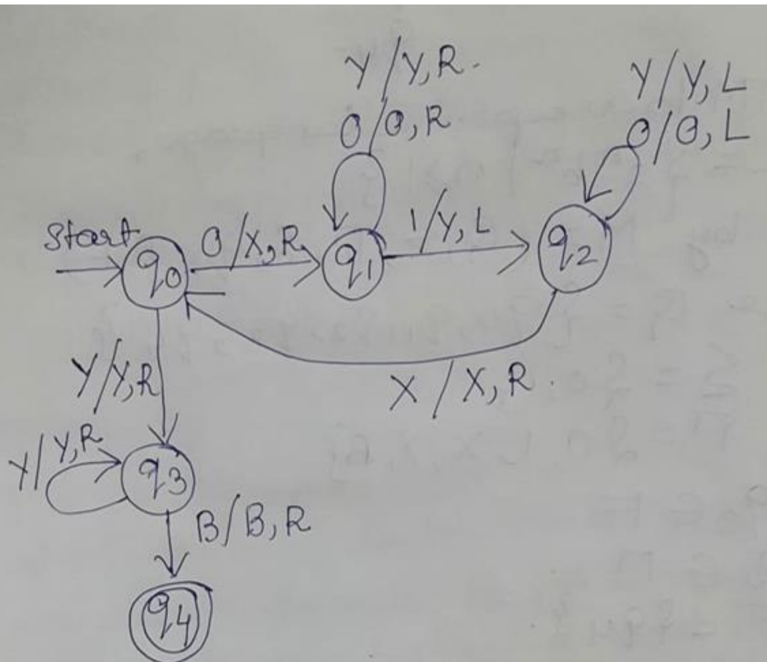
$F = \{q_4\}$.

δ is as shown above.

/* Write all the δ we obtained :- */

The transition table is :-

δ states	Tape Symbols (Γ) 0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—



<u>To accept the string</u>: The sequence of moves or computations (IDs) for the string 0011 is shown:

(Initial ID) $q_0 0011 \vdash X q_1 011 \vdash X0 q_1 11$
$\vdash X q_2 0Y1 \vdash q_2 X0Y1 \vdash X q_0 0Y1 \vdash XX q_1 Y1$
$\vdash XXY q_1 1 \vdash XX q_2 YY \vdash X q_2 XYY$
$\vdash XX q_0 YY \vdash XXY q_3 Y \vdash XXYY q_3$
$\vdash XXYYB q_4$ (Final ID)

Since final state q_4 is reached, the string 0011 is accepted.

(b)	What is a handle and Handle pruning? Design a shift-reduce parser for the input string id+id*id following the production rules: E->E+E, E->E*E, E->(E), E->id	[2+4=6]	CO2	L3

Solution: Handle Definition: Bottom-up parsing during a left-to-right scan of the input constructs a right- most derivation in reverse. Informally, a **“handle**" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
The table below shows the hnadles during parsing of id1*id2.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha \beta w$, then production $A \to \beta$. in the position following α is a handle of $\alpha \beta w$.

Handle pruning Definition: A rightmost derivation in reverse can be obtained by \handle pruning." That is, we start with a string of terminals w to be parsed. If w is a sentence

To reconstruct this derivation in reverse order, we locate the handle βn in

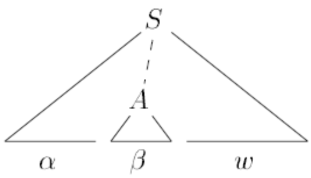


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

of the grammar at hand, then let $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

γn and replace β_n by the head of the relevant production $A_n \to \beta_n$ to obtain the previous right-sentential form γ_{n-1}. If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

Shift-reduce parser:

(1) We start with a string of terminals 'w' to be parsed.

$$
\left.\begin{aligned}
S &\rightarrow aABe \\
&\rightarrow aAde && B\rightarrow d \\
&\rightarrow aAbcde && A\rightarrow Abc \\
&\rightarrow aabcde && A\rightarrow a
\end{aligned}\right\} \text{Rightmost derivation.}
$$

Soln Stack Implementation (Shift-reduce parser).

Stack	input	Action.
\$	id+id*id\$	Shift.
\$ id	+id*id\$	E→id (reduce).
\$ E	+id*id\$	Shift
\$ E+	id*id\$	Shift
\$ E+id	*id\$	Reduce (E→id)
\$ E+E	*id\$	Reduce (E→E+E)
\$ E	*id\$	Shift
\$ E*	id\$	Shift
\$ E*id	\$	Reduce (E→id)
\$ E*E	\$	Reduce (E→E*E)
\$ $\boxed{E}$	$\boxed{\$}$	Accept

Four possible Actions :-

A shift-reduce parser can make 4 possible actions.

(1) In a shift action, the next input symbol is shifted onto the top of the stack.

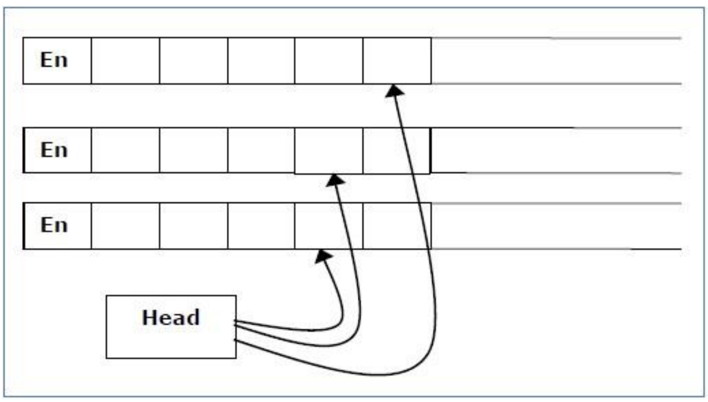
(2) In a reduce action, the parser knows the right end of the handle and it must locate the left end of the handle (non-terminal) ~~with~~ of the production and reduce it.

(3) In an accept action, the parser announces successful completion of parsing.

(4) In an error action, the parser discovers syntax errors and calls error-recovery routine.

4 (a) Explain with neat diagrams the variants of the Turing machine. [6] CO3 L1

1. Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. 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.



A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q_0, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where
  $\delta: Q \times X^k \rightarrow Q \times (X \times \{Left\_shift, Right\_shift, No\_shift\})^k$
  where there is **k** number of tapes
- $\mathbf{q_0}$ is the initial state
- **F** is the set of final states

Note − Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

2. Multi-track Turing machines, 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, q_0, 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
  $\delta(Q_i, [a_1, a_2, a_3,....]) = (Q_j, [b_1, b_2, b_3,....], Left\_shift \text{ or } Right\_shift)$
- $\mathbf{q_0}$ is the initial state
- **F** is the set of final states

Note − For every single-track Turing Machine **S**, there is an equivalent multi-track Turing Machine **M** such that **L(S) = L(M)**.

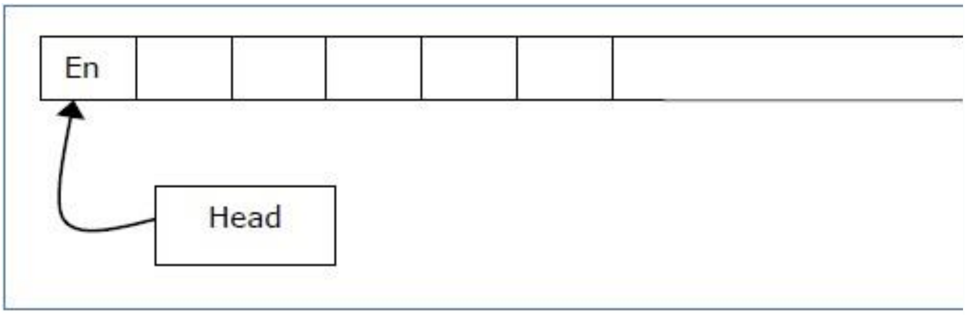
3. In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

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, q_0, B, F) where −

Q is a finite set of states
X is the tape alphabet
∑ is the input alphabet
δ is a transition function;
$\delta : Q \times X \rightarrow P(Q \times X \times \{Left\_shift, Right\_shift\})$.
q_0 is the initial state
B is the blank symbol
F is the set of final states

4. Semi-infinite tape: 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.

The infinite length input string is initially written on the tape in contiguous tape cells.

The machine starts from the initial state **q_0** and the head scans from the left end marker ‘End’. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called **accept state** and **reject state**. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note − Turing machines with semi-infinite tape are equivalent to standard Turing machines.

5. A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

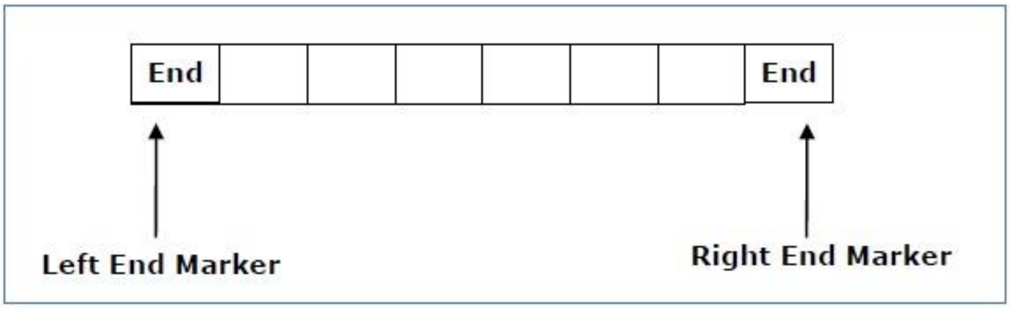
Here,

Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

A linear bounded automaton can be defined as an 8-tuple (Q, X, ∑, q_0, ML, MR, δ, F) where −

Q is a finite set of states
X is the tape alphabet
∑ is the input alphabet
$\mathbf{q_0}$ is the initial state
$\mathbf{M_L}$ is the left end marker
$\mathbf{M_R}$ is the right end marker where $M_R \neq M_L$
δ is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
F is the set of final states



A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable.**.

4(b)	Design a LALR parser.	[4]	CO2	L3

Solution:

(1) $S \rightarrow CC$

$C \rightarrow cC/d$.

Solution: Construct augmented grammar.

I_0 r_0 $S' \rightarrow \cdot S, \$$ // Initialize $S' \rightarrow \cdot S$, its lookahead will be $\$$,

r_1 $S \rightarrow \cdot CC, \$$

r_2 $C \rightarrow \cdot cC, c/d$. // As $\cdot S$, ∴ $S \rightarrow \cdot CC$, FIRST($\$$) = $\$$.

r_3 $C \rightarrow \cdot d, c/d$. // FIRST{c, d} = {c, d}.

I_1 goto (I_0, S)

$S' \rightarrow S \cdot, \$$ // $\$$ because only dot shift i.e lookahead is same if dot is shifted.

I_2 goto (I_0, C) // capital C.

$S \rightarrow C \cdot C, \$$

$C \rightarrow \cdot cC, \$$ FIRST($\$$) = $\$$.

$C \rightarrow \cdot d, \$$

I_3 goto (I_0, c) // small c

$C \rightarrow c \cdot C, c/d$.

$C \rightarrow \cdot cC, c/d$ FIRST {c, d} = {c, d}.

$C \rightarrow \cdot d, c/d$. or if its terminal, place directly in lookahead.

I_4 goto(I_0, d)

$C \rightarrow d \cdot, c/d$

I_5 goto(I_2, C)

$S \rightarrow CC \cdot, \$$

goto(I_2,

I_7 goto(I_2, d)

$C \rightarrow d \cdot, \$$

I_3 goto(I_3, c)

$C \rightarrow c \cdot C, c/d.$

$C \rightarrow \cdot cC, c/d.$

$C \rightarrow \cdot d, c/d.$

I_6 goto(I_2, c)

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

I_8 goto(I_3, C)

$C \rightarrow cC \cdot, c/d.$

I_4 goto(I_3, d)

$C \rightarrow d \cdot, c/d.$

I_9 goto(I_6, C)

$C \rightarrow cC \cdot, \$$

I_7 goto(I_6, d)

$C \rightarrow d \cdot, \$$

I_6 goto(I_6, c)

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$.$

From Cannonical LR(1) it is seen that 3 and 6 is repeating. Similarly, 4, 7 and 8, 9 are repeating.

Hence, combine them. I.e – Reduce the State 3 and 6 to 36

4 and 7 to 47

8 and 9 to 89.

	Action			Goto	
State	c	d	$	S	C
0	S36	S47		1	2
1					
2			accept		
36	S36	S47			5
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

5 (a) Write a note on the Church Turing Hypothesis and Problems that computers cannot solve. [4] CO4 L1

The Church Turing Thesis, also known as the Church's Thesis or Turing's Thesis, is a hypothesis in computer science that states any real-world computations can be translated into an equivalent computation involving a Turing machine. This conjecture represents the underpinning principle of modern computers.

A quick glance at some practical applications of the Church Turing Thesis can help illuminate its importance:

Design of Digital Computers: Digital computers function based on the principles laid down by the Church Turing Thesis. If there exists an algorithm to solve a problem, a computer can be programmed to implement that algorithm.

Creation of Programming Languages: The design principles of almost all high-level programming languages are also rooted in this thesis. They all allow for the expression of a general-purpose set of instructions — algorithms in other words — that a computer can execute.

Fundamentals of Artificial Intelligence: When exploring artificial intelligence and machine learning, the Church Turing Thesis is often invoked. For instance, if a human intelligence process can be encapsulated as an algorithm, this thesis suggests a machine can be programmed to replicate that process.

It's remarkably eye-opening to realise that from the commonplace laptop in your possession to the complex AI models, they echo the principles of this

impactful thesis, thereby, shedding light on its ubiquitous relevance and application.
Church Turing Thesis Examples: Understanding Through Practice
The interplay between theory and practice lies at the heart of the Church Turing Thesis. To grasp this abstract concept, concrete examples provide the perfect bridge. Each elucidates how real-world computations get abstracted into the realm of Turing Machines, guiding you on the path of mastery. Let's consider a simple but effective example. Imagine the process of baking a cake from a recipe. This is a step-by-step process that, in essence, is a real-world algorithm. Following the Church Turing Thesis, one can structure this process into a form that a Turing machine (or a computer) can comprehend and execute.
Demystifying Church Turing Thesis with Effective Examples
Consider the aforementioned example in more detail:
Algorithm for Baking a Cake:
1. Gather all ingredients
2. Preheat the oven
3. Mix ingredients
4. Pour mixture into a pan
5. Bake in the preheated oven

Given this algorithm, let's construct a pseudocode mapping:

```
BEGIN
  IF ingredients present THEN
    Preheat oven
    Mix ingredients
    Pour mixture into pan
    Bake in oven
  ELSE
    Display 'Gather all ingredients first!'
  END IF
END
```

This constructed pseudocode now translates the original algorithm into a format that a Turing Machine — or a modern computer — could execute (albeit metaphorically, since computers can't physically bake cakes). Through this example, you can start to understand the real power and practical application of the Church Turing Thesis. It's not merely an abstract concept, but a principle that provides the backbone for virtually all modern computation. So, whether you're considering a career in computer science, a related field, or simply looking for a deeper understanding of the digital world, the Church Turing Thesis provides fundamental insights into the mechanisms that drive modern computation.

Problems that computers cannot solve.

Programs that print "Hello, World"

- A C program that prints "Hello, World" is:

```
main()
{
print("hello, world\n");
}
```

- Define a "*hello, world problem*" to be:

Determine whether a given C program, with a given input, prints *hello, world* as the first 12 characters in what it prints.

- ♦ Describe the problem *alternatively* using symbols:

Is there a program *H* that could examine any program *P* and any input *I* for *P*, and tell whether *P*, run with *I* as its input, would print *hello*, *world*?
(A program *H* means an algorithm in concept here.)

- The answer is: *undecidable*!
- That is, there exists no such program *H*.
- We can prove this by contradiction next.

8.1.2 Hypothetical "Hello, World" Tester

■ We want to prove that no program *H*, called *hypothetical* "Hello, World" *tester*, as mentioned above exists by contradiction using the following steps.

- ♦ Step 1 --- assume *H* exists in a form as shown in Fig. 8.1 (Fig 8.3 in the textbook).

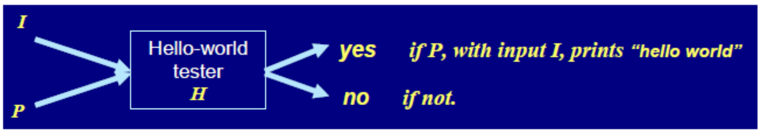


Fig. 8.1 A hypothetical "Hello, World" tester.

- ♦ Step 2 --- transform *H* into another form H_2 in a simple way which can be done by C programs.
- ♦ Step 3 --- prove that H_2 does not exist and so that *H* does not exist, either.

■ Implementation of Step 2 above ---

(1) Transform *H* to H_1 in a way as illustrated by Fig. 8.2 (Fig. 8.4 in the textbook).

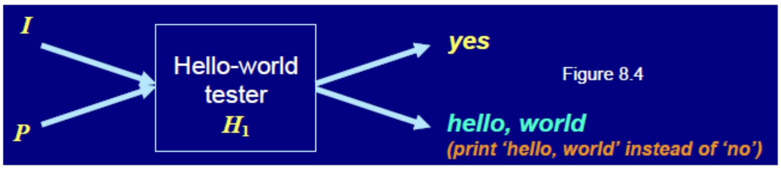


Fig. 8.2 A transformed "hello-world tester" H_1.

(2) Transform H_1 to H_2 in a way as illustrated by Fig. 8.3 (Fig. 8.5 in the textbook).

Fig. 8.2 A second transformed "hello-world tester" H_2.

- The function of H_2 constructed in Step 2 is ---

  given any program P as input,

  if P prints **hello, world** *as first output, then H_2 makes output yes;*
  if P does not prints **hello, world** *as first output, then H_2 prints* **hello, world**.

- Implementation of Step 3 above (proving H_2 does not exist) ---
  - Let P for H_2 in Fig. 8.2 (last figure) be H_2 itself, as illustrated in Fig. 8.3 (Fig. 8.6 in the textbook).

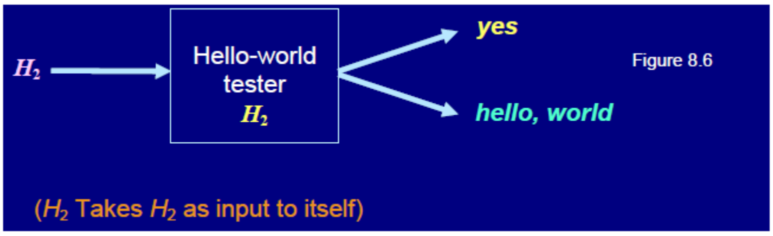


Fig. 8.3 A second transformed "hello-world tester" H_2 taking itself as input.

  - Now, we have the following reasoning (assuming the term "box" means "Hello-world tester" ---

    (1) If

    the box H_2, given itself as input, makes output yes,

    then according to the above-described function of H_2, this means that

    the box H_2, given itself as input, prints **hello, world** *as the first output.*

    But this is contradictory because we just suppose that

    the box H_2, given itself as input, makes output yes.

    (2) The above contradiction means the other alternative must be true since there are only two choices, that is ---

    the box H_2, given itself as input, prints **hello, world** *as the first output.*

    But according to the above-described function of H_2, this means that

    such H_2, when taken as input to the box H_2 (itself), will make the box H_2 to make output yes.

This is a contradiction again because we just say that

the box H_2, given itself as input, prints **hello, world** *as the first output.*

- ♦ Since both cases lead to contradiction, we conclude that the assumption that H_2 exists is wrong by the principle of contradiction for proof.

- ♦ H_2 does not exist $\Rightarrow$ H_1 does not exist (otherwise, H_2 must exist)
  $\Rightarrow$ H does not exist (otherwise, H_1 must exist), done!
  ("$\Rightarrow$" means "imply" here)

■ The above *self-contradiction* technique, similar to the *diagonalization* technique (to be introduced later), was used by Alan Turing for proving undecidable problems.

Reducing One Problem to Another

Now we have an undecidable problem, which can be used to prove other undecidable problems by a technique of *problem reduction*.

♦ That is, if we know P_1 is undecidable, then we may *reduce P_1 to a new problem P_2*, so that we can prove P_2 undecidable by contradiction in the following way

- *If P_2 is decidable, then P_1 is decidable.*
- *But P_1 is known undecidable. So, contradiction!*
- *Consequently, P_2 is undecidable.*

■ An illustration of the above idea is illustrated in Fig. 8.4.

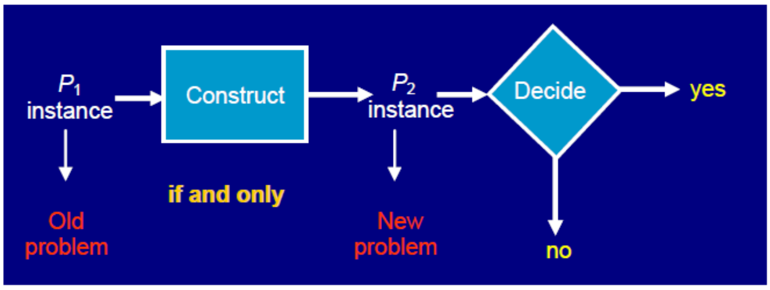


Fig. 8.4 An illustration of reducing one problem to another.

■ **Example 8.1** ---

We want to prove a new problem P_2 (called ***calls-foo*** *problem*):

*"does program Q, given input y, ever call function **foo**?"*

to be undecidable.

Solution:

- Reduce P_1: *the* **hello-world** *problem* to P_2: *the* ***calls-foo*** *problem* in the following way:
  - If Q has a function called ***foo***, *rename it and all calls* to that function $\Rightarrow$ a new program Q_1 doing the same as Q. ("$\Rightarrow$" means "leading to" here)
  - Add to Q_1 a function ***foo*** doing nothing & *not* being called $\Rightarrow$ a new program Q_2.
  - Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array $A \Rightarrow$ a new program Q_3.
  - Modify Q_3 in such a way that whenever it executes any output statement, it checks A to see if it has written 12 characters or more, and if so, whether *hello, world* are the first characters. In that case (i.e., if so), call the new function ***foo*** $\Rightarrow$ a new program R with input y.

- Now,
  - if Q with input y prints *hello, world* as its first output, then R will call ***foo***;
  - if Q with input y does not print *hello, world*, then R will never call ***foo***.
- That is, program R, with input y, calls ***foo*** if and only if program Q, with input y, prints *hello, world*.

- So, if we can decide whether R, with input y, calls ***foo***, then we can decide whether Q, with input y, prints *hello, world*.
- But *the latter is impossible* as has been proved before, so the former is impossible.

(b) Discuss three methods of evaluation order of a syntax-directed definition (SDD). [6] CO2 L3

Solution:

① Dependency graphs :- Useful tool for determining an evaluation order for the attribute in a parse tree. Dependency graph shows how those values are computed.

Example: Consider:

production	Semantic rule.
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$.

Let each node be N. The dotted line represent parse tree. The children corresponds to the body of the production. The synthesized attribute 'val' at node N is computed ~~by~~ using the values of 'val' at the two children (labelled E and T). Thus a portion of the dependency graph looks like

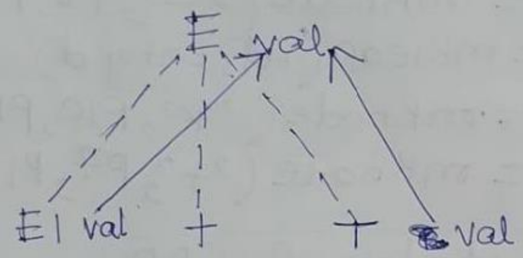


Fig: E.val is synthesized from E_1.val and T.val

(2) Bottom-up evaluation of S-attributed definition :- (from LR grammar)

$L \rightarrow E_n$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow digit$

i/p : $3*5+4n$.

Solution.

input	Stack State	Value	Production Used
(1) 3 * 5 + 4 n	—	—	—
(2) * 5 + 4 n	3	3	F → digit
(3) * 5 + 4 n	F	3	T → F
(4) * 5 + 4 n	T	3	
(5) 5 + 4 n	T *	3 –	
(6) + 4 n	T * 5	3 – 5	F → digit.
(7) + 4 n	T	3 – 5	T → T * F
(8) + 4 n	E	15	E → T
(9) + 4 n	E	15	
(10) 4 n	E +	15 –	
(11) n	E + 4	15 – 4	
(12) n	E + F	15 – 4	F → digit
(13) n	E + T	15 – 4	T → F
(14) n	E	19	E → E + T
(15)	E n	19.	L → E n

The parsing program compares input and stack and produces output.

4 Conditions :
1) Shift
2) Reduce
3) accept
4) Error.

(Q3) Bottom-Up evaluation of inherited attribute :-

Prod	Semantic rules
1) $D \rightarrow TL$	$L.inh := T.Type$
2) $T \rightarrow int$	$T.type := integer$
3) $T \rightarrow real$	$T.type := real / float$
4) $L \rightarrow L , id$	$L1.inh := L.inh$
	addType(id.entry, L.inh)
5) $L \rightarrow id$	addType(id.entry, L.inh)

Syntax directed defn

The table shows syntax-directed ~~trans-lation of~~ definition of simple type declarations.

1) D → (Non-terminal) represents declaration ~~from~~ which, from production 1, consists of a type T followed by a list ~~of~~ L of Identifiers.

* T has one attribute, T.type which is the type ~~of~~ in the declaration D.

* Non-terminal L also has one attribute, which we call inh to emphasize that it is an inherited attribute.

* The purpose of L.inh is to pass the declaration type down the list of identifiers, so that it can be added to the appropriate symbol ~~entry~~ table entries.

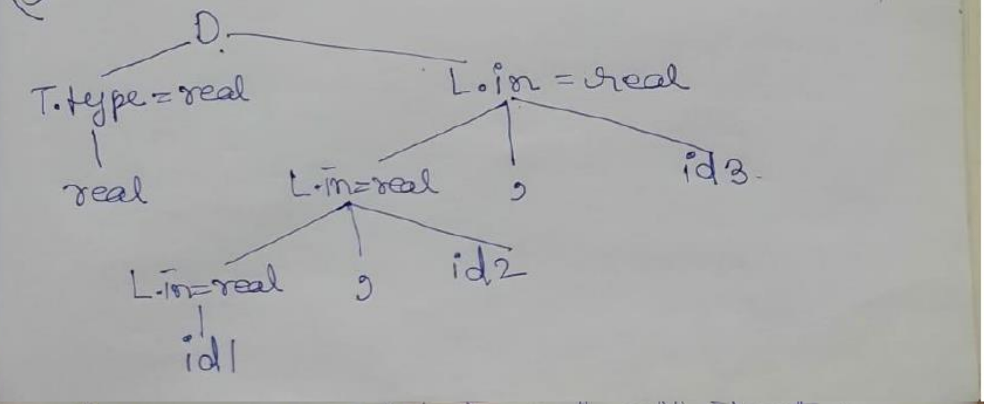
2) and 3) evaluates the synthesized attribute T.type giving it appropriate value, integer or float.
This type is passed to the attribute L.inh in the rule for prod 1. prod 4 passes L.inh down the parse tree. That is the value L1.inh is computed at a parse-tree node by copying the value of L.inh from the parent of that node; the parent corresponds to the head of the production.

* Prod 4 and 5 have a rule in which a function addType is called with 2 arguments:

i) id.entry, a lexical value that points to a symbol-table object.

ii) L.inh, the type being assigned to every identifier on the list.

(Q) Let the input be – real p, q, r
id1 id2 id3



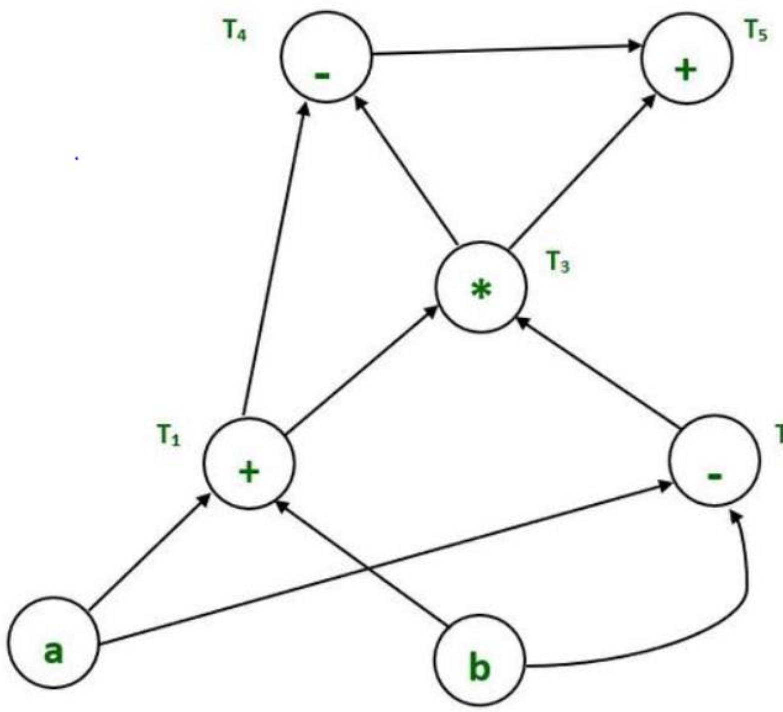
Bottom-Up Eval of Inherited attribute 5

input	state	production Used
1) real p, q, r	—	
2) p, q, r	real	T → real.
3) p, q, r	T	L → id
4) , q, r	Tp	
5) q, r	TL,	
6) , r	TL, q	L → L, id
7) , r	TL	
8) r	TL,	
9) —	TL, r	L → L, id
10) —	TL	D → TL
11) —	D	

6 (a) Construct a Directed Acyclic Graph for the following expression: [5] CO2 L3

T1 = a + b
T2 = a – b
T3 = T1 * T2
T4 = T1 – T3
T5 = T4 + T3

Solution:



(b) Explain three address code. Design a quadruple, triple, and indirect triple for the expression d= b*-c - b*-c [5] CO2 L3

Solution:

Quadruple:

	op	arg1	arg2	result
0	Uminus	c	—	t1
1	*	b	t1	t2
2	Uminus	c	—	t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

(2) Triples : Consists of three fields – Op, arg1, arg2.

	Op	arg1	arg2
0	Uminus	c	
1	*	b	0
2	Uminus	c	—
3	*	b	2
4	+	1	3
5	=	a	4

(3) Indirect triple : Listing of pointers rather than listing of triples themselves.

	statement/instruction		Op	arg1	arg2
0 →	(14)	14	Uminus	c	
1	(15)	15	*	b	(14)
2	(16)	16	Uminus	c	
3	(17)	17	*	b	(16)
4	(18)	18	+	15	17
5	(19)	19	Assign	a	18