Unit-IV

VIJAYSHRI
PAGE NO.:
DATE:

SDT (Syntax directed translation)

"It's Third Phase of compiler phases"

Parser uses a CFG (context free Grammar) to validate the input string & produce output for next phase of the compiler. output could be either a parse tree or abstract tree.

Now interleave (मिलाना / विद्या देना) Semantic analysis with Syntax analysis phase of the compiler, we use Syntax directed translation

* Syntax directed translation (SDT) ⇒ To handle semantic features of the programming languages the following two techniques are used:-
  - Formal method.
  - Adhoc method

In the formal method, we can use CFG to handle semantic features of the programming language but it is very difficult to implement CFG. Hence programming languages semantic features can be handle by Adhoc technique is called as Syntax directed translation (SDT)

SDT ⇒ Definition ⇒ SDT is nothing but it's are augmented rules to the grammar that facilitate semantic analysis.

SDT involves passing information bottom-up and/or top-down: the parse tree in form of attributes attached to the nodes. SDT rules use:-

(i) lexical values of nodes (ii) constants (iii) attributes associated to the non-terminals in their definitions.

DATE:

General Approach of SDT is to construct a Parse tree / Syntax tree & compute the val of attributes at the nodes.
Whenever parser applyies parsing the semant rules associated with this CFG production will executed & they verifies the meaning of the statement.

SDT = Grammar + semantic rules

SDT

Grammar	semantic rules
$E \to E+T$	{ E.value = E.value + T.value }
$E \to T$	{ E.value = T.value }
$T \to T*F$	{ T.value = T.value * F.value }
$T \to F$	{ T.value = F.value }
$F \to num$	{ F.value = num.lexvalue }

* Semantic rules notations ⇒ semantic rules notations ~~are~~ may be two types
  - → Syntax directed definition (SDD)
  - → syntax directed translation (SDT) scheme

⇒ Syntax directed definition (SDD) ⇒ SDD is a kin of abstract specification. It is generalizatio of CFG in which each grammar production S→ is associated with it a set of production rules. (or)
More augumented CFG that specifies the value attributes by associating semantic rules with gra

Productions Example $E \to E+T$ { E.code = E.code // T.code
Concanate

Types of SDD attributes

(or)

SDD mainly two types :-

- Synthesized attribute
- Inherited attribute

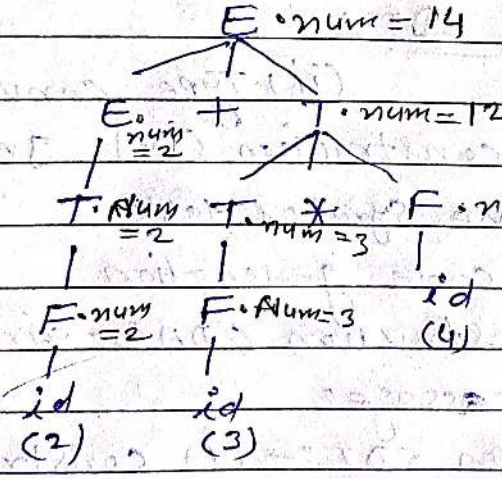
① **Synthesized attributes** ⇒ Bottom-up approach

These are those attribut which derive their values from their children nodes i.e. value of synthesized attributes at node is computed from the value of attributes at children nodes in Parse tree.

Example :- $E \rightarrow T * F$ { E.value = T.value * F.value }

Here E.value derive it's value from T.value & F.value

* **Annotated Parse tree** :- The Parse tree containing the value of attributes at each node for given input string is called annotated or decorated Parse tree



Features of annotated tree :-

(i) High level specification

(ii) Hides implementation details

(iii) Explicit order of evaluation is not specified.

(2) Inherited Attributes ⇒ These are the attributes wh derive their values from their parent or siblings node i.e. (that is) value of inherited attributes are computed by parent or sibling nod

Example :- $A \rightarrow BCD$ { C.Value = A.value, B.value, D.Va

⟹ Syntax directed translation (SDT) scheme ⇒

→ It is a CFG. SDT is used to evaluate the order of semantic rules

→ In SDT, the semantic rules are embedded within the right side of the productions

→ The position at which an action is to be execut is shown by enclosed b/w braces. It is writt within the right side of the production

* Advantages of SDT ⇒

(i) Type checking (ii) Type conversion

(iii) Symbol table construction (iv) Inserting / retrievin values from the symbol table.

(v) Intermediate code generation

(vi) Syntax tree construction (DAG - directed acylic grap

(vii) Issue error messages

* Steps to design SDT :- (i) construct the parse tr

(ii) construct the annotated parse (or) decorated tr

(iii) computing the attribute value at each node by the parse tree.

(iv) Generalized rules of attribute computation attaching to CFG (semantic actions)

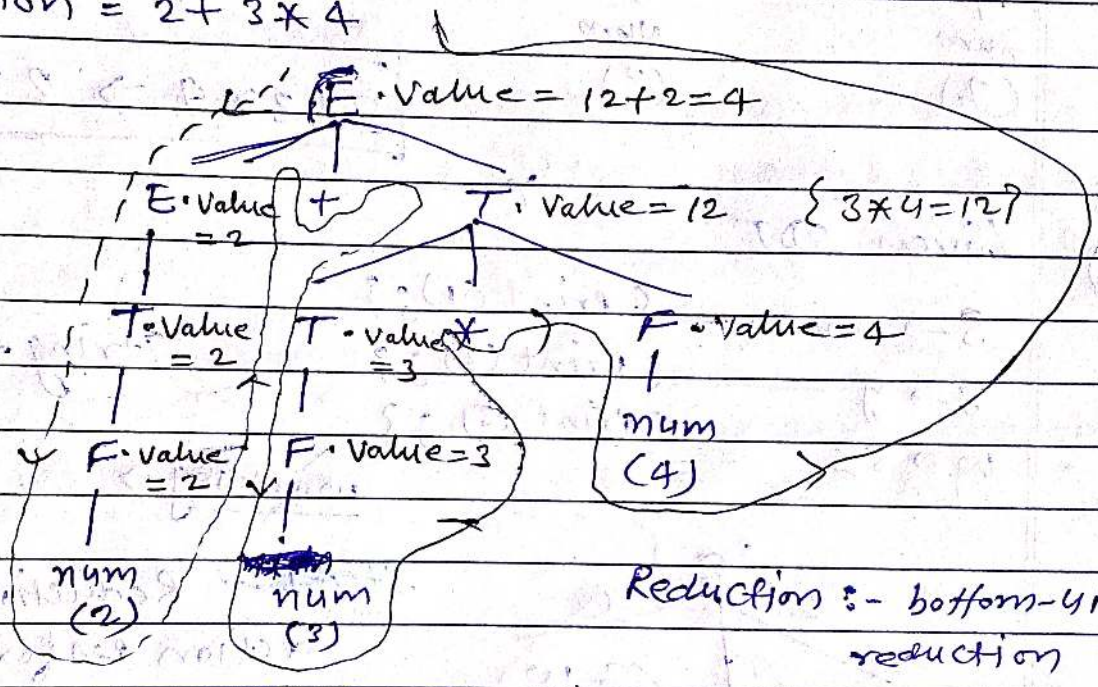
Construct SDT for evaluation of expression

(OR)

Q.1) Construct SDT for ✎ Expression

Grammar	Action (rules)
$E \rightarrow E+T$	{ E.value = E.value + T.value }
$E \rightarrow T$	{ E.value = T.value }
$T \rightarrow T*F$	{ T.value = T.value * F.value
$T \rightarrow F$	{ T.value = F.value }
$F \rightarrow num$	{ F.value = num.lex value }

Expression = 2+3*4

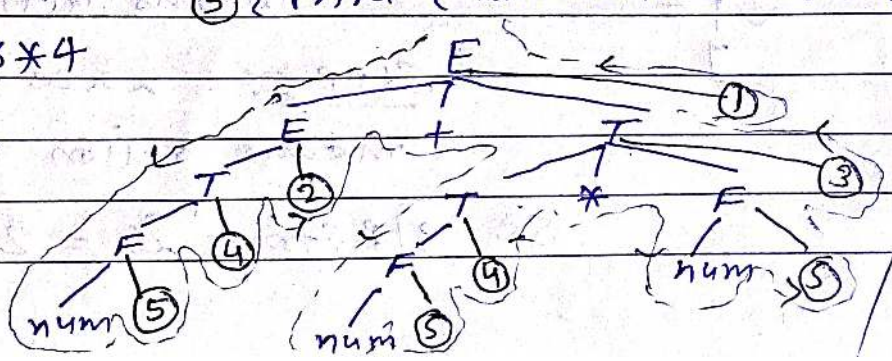


Reduction :- bottom-up reduction

Q.2) Given SDT

	semantic action no.
$E \rightarrow E+T$	① { Print("+"); }
$E \rightarrow T$	② { } ← Nothing (No action)
$T \rightarrow T*F$	③ { print("*"); }
$T \rightarrow F$	④ { }
$F \rightarrow num$	⑤ { Print(num.lex value); }

Expression = 2+3*4



Convert infix to postfix

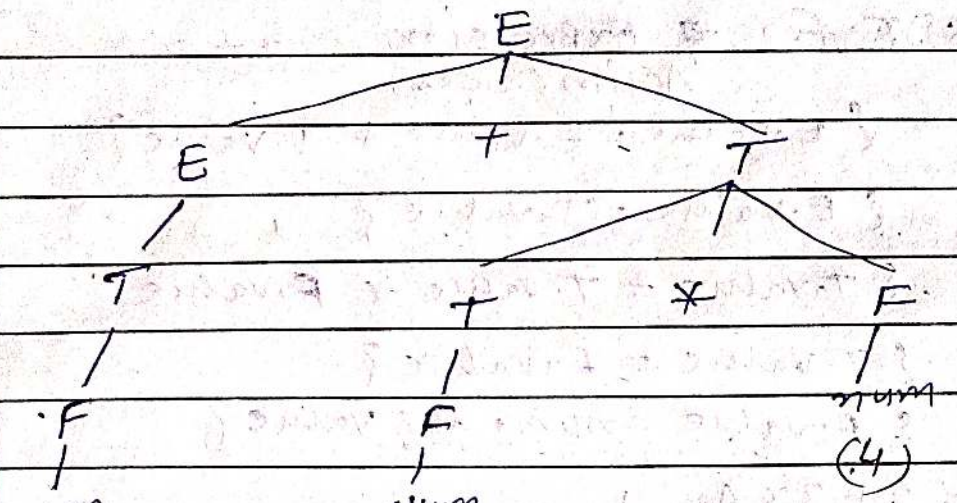
234*+

(From TOP-down Parser)

move करते हैं TOP to bottom & left to right जो action no. आता है उसके according action Perform करते जाते हैं जीसके base पर result 234*+ आता है

PAGE NO.:
DATE:

From Bottom-up Parser ⇒ 2+3*4



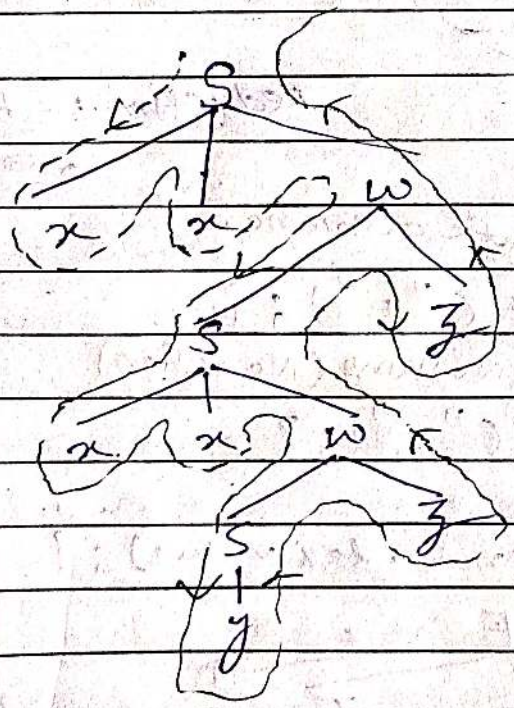
E → E+T { print "+" }
E → T { }
T → T*F { print "*" }
T → F { }
F → num { print (num.lex value) }

2+3*4 ⇒ 234*+

(Q.3) Given SDT

S → xxw { print (1); }
S → y { print (2); }
W → Sz { print (3); }

String = xxxyzz



working ⇒

Step-1 :- Reduction y → S means action perform print 2

Step-2 :- Sz reduction to W means action perform 3

Step-3 :- xxw reduce to S means action perform 1

Step-4 :- Sz reduce to W means action 3 perform

Step-5 :- xxw reduce to S means action 1 perform

So o/p string is = 23131

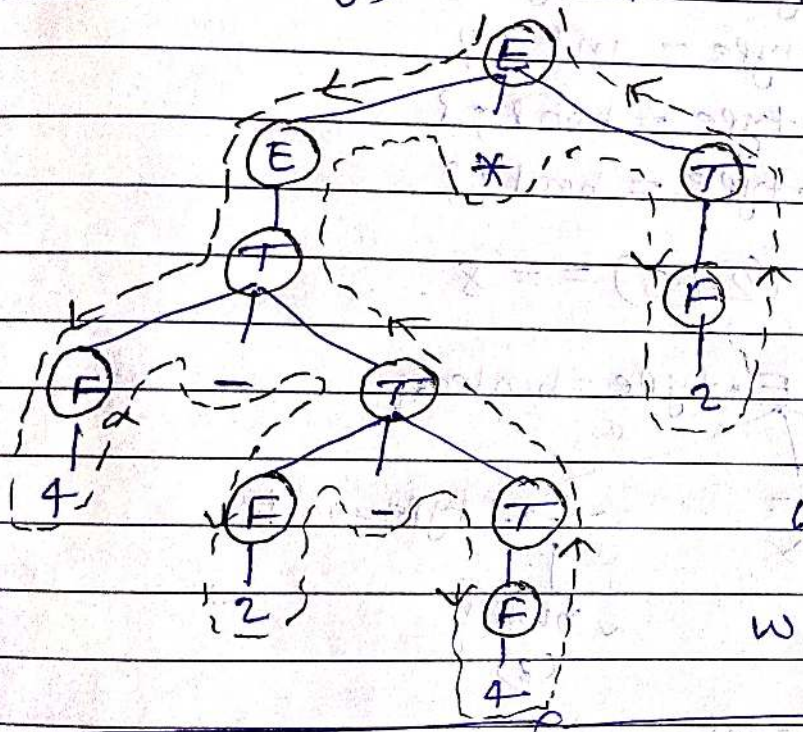
VIJAYSHRI
PAGE NO.:
DATE:

gmf

Q.3) Given SDT — Action (rules)

$E \to E * T$	{ E.value = E.value * T.value; }
$E \to T$	{ E.value = T.value; }
$T \to F - T$	{ T.value = F.value - T.value ; }
$T \to F$	{ T.value = F.value; }
$F \to 2$	{ F.value = 2; }
$F \to 4$	{ F.value = 4; }

word (String) $w = 4 - 2 - 4 * 2$



All circles show the number of reductions = 10

According to operator precedance

$w = ((4 - (2 - 4)) * 2)$

$w \equiv (4 + 2) * 2 = 6 * 2 = 12$ Ans

gmf

Q.4) Given SDT

$E \to E \# T$	{ E.value = E.value * T.value ; }
$E \to T$	{ E.value = T.value ; }
$T \to T \& F$	{ T.value = T.value + F.value ; }
$T \to F$	{ T.value = F.value ; }
$F \to num$	{ F.value = num.lex value ; }

Expression ⇒ $2 \# 3 \& 5 \# 6 \& 4$

Soln

$2 * (3 + 5) * (6 + 4)$

$(2 * 8) * 10$

$= 16 * 10$

$= 160$

precedance rules
recursion → Associative
level first depth level is higher precedance as compaire to low level depth After that recursion value first

VIJAYSHRI
PAGE NO.:
DATE:

Type check

Q.5] $E \rightarrow E_1 + E_2$ { if (E_1.type == E_2.type) && (E_1.type = int then E.type = int else Error ; }

$E \rightarrow E_1 == E_2$ { if (E_1.type == E_2.type) && (E_1.type = int/boolean)) then E.type = boolean else Error ; }

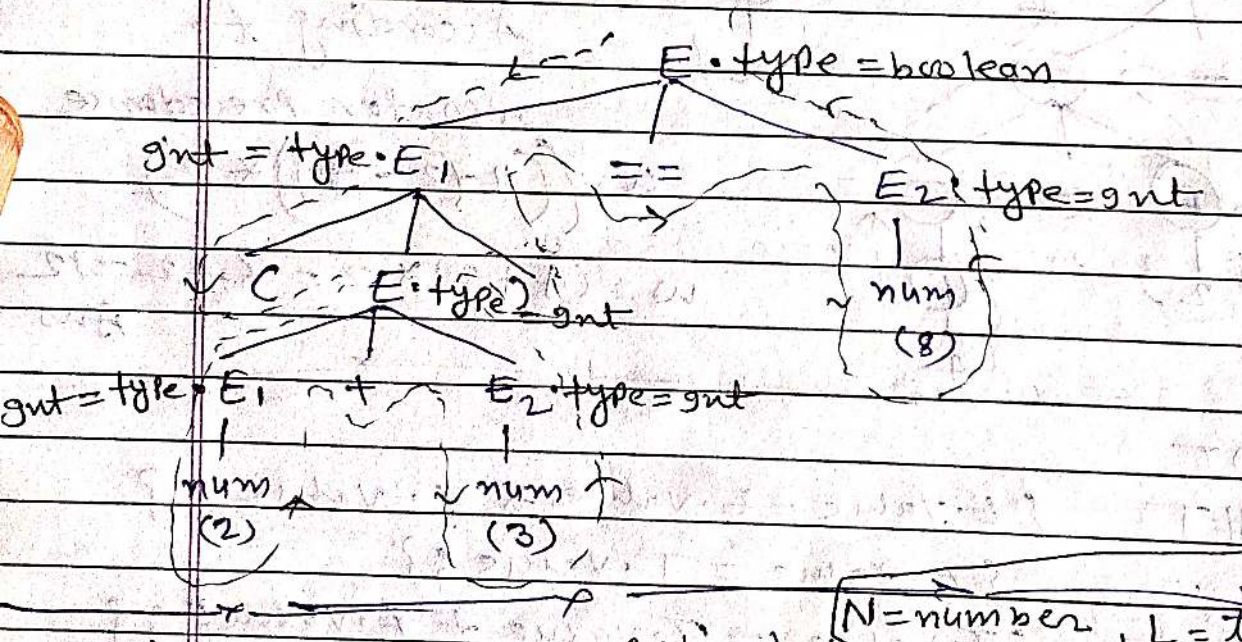
$E \rightarrow (E)$ { E.type = E.type ; }

$E \rightarrow$ num { E.type = int ; }

$E \rightarrow$ True { E.type = bool ; }

$E \rightarrow$ false { E.type = bool ; }

Expression :- (2+3) == 8



(Action) N = number, L = list of bit & B = bit

Q.6] make a semantic rules for given grammar

		count 1's	count 0's	count bit
$N \rightarrow L$		{ N.count = L.count }		
$L \rightarrow LB$		{ L.count = L.count + B.count }	} same	} same
$L \rightarrow B$		{ L.count = B.count }		
$B \rightarrow 0$		{ B.count = 0 }	{ B.count = 1 }	{ B.count = 1 }
$B \rightarrow 1$		{ B.count = 1 }	{ B.count = 0 }	{ B.count = 1 }

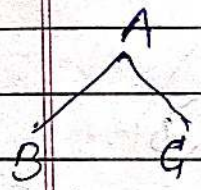
* SDT Attributes / Types of SDT ⇒

- → S-Attributed SDT
- → L-Attributed SDT

synthesis attribute :-

$A \rightarrow BC$

{A.Value = B.Value, C.Value}



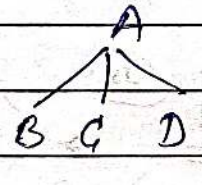
inherited attributes

$A \rightarrow BCD$

{C.value = A.value, B.value, D.value}

{C.value = B.value, D.value}

{C.value = B.value / D.value}



⇒ S-Attributed SDT ⇒

1. use only synthesized attribute.

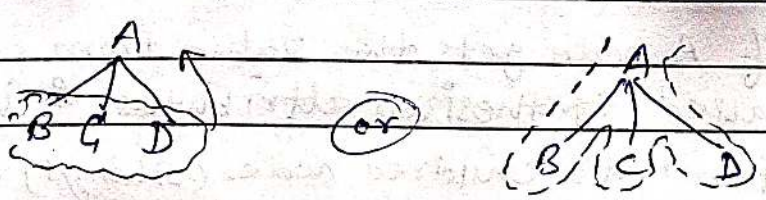
Example :- $A \rightarrow BC$ {A.Value = B.Value, C.Value}

2. Semantic action placed at right end of production

Ex :- $E \rightarrow E+T$ {E.Value = E.Value + T.Value}

3. Attribute are evaluated Bottom-up manner

$A \rightarrow BCD$



④ it's More restricted

L-Attributed SDT ⇒

① used both synthesized & inherited attribute. Each inherited is restricted to inherite either from Parent node or left siblings nodes only

$A \to BCD$

{C.value = A.Value} (or) {C.value = B.value}

{A.Value = B.value, C.value, D.value}

But {C.value = D.value} ✗ is not allow because D is right sibling node of C node.

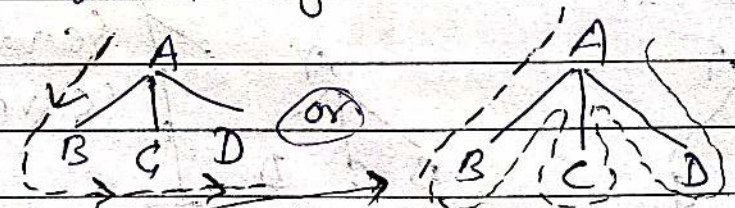
② Semantic action are Placed any where on RHS side

$A \to BCD$ {---} $A = \{---\} BCD$

$A \to B\{---\}CD$

③ Attributes are evaluated by traversing parse tree depth First & left to right

$A \to BCD$



④ it's free {B की value A से & C की value A, B & D की value A, B, C से हम calculate कर सकते है}

Q.1) In S-Attributed :- $A \to BCD$ then value of C

C.value = A.value, C.value = B.value, C.value = D.value

But in L-Attribute

C.value = A.value, C.value = B.value But C.value = D.value ✗

If A node gets the value from it's children node is called synthesized attributes & if A node gets the value from it's childred node (siblings) & Parent node is called ~~Inherent~~ inherite attributes

* Intermediate code generation ⇒ It's 4th Phase of compiler design Phases. It's generate Three address code (TAC) form

Intermediate code generation may be following types

ICG (Intermediate code generation)

- LINEAR FORM
  - Postfix notation
  - Three Address code (TAC) notation
- TREE FORM
  - Syntax tree
  - Directed acyclic graph (DAG)

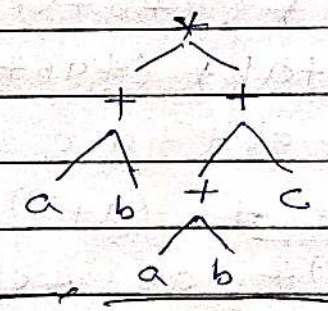
Example :- Expression = (a+b) * (a+b+c)

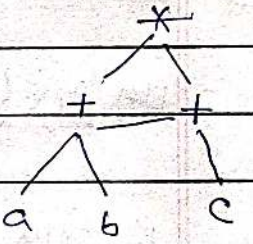
ab+ ab+c+ *

$t_1 = a+b$

$t_2 = t_1+c$

$t_3 = t_1 * t_2$





* Postfix notation ⇒ Notations manly three types

- Infix notation
- Prefix notation
- Postfix notation

Infix	Prefix	Postfix
a+b	+ab	ab+
a+b*c	+a*bc	a+bc* ⇒ abc*+
(a+b)*c	+ab*c	ab+*c = ab+c*
	*+abc	

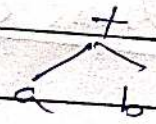
P.T.O

DATE:

* Syntax tree ⇒ leaf node always operand (variable) & intermedia node & root node as a operator

Examples:-

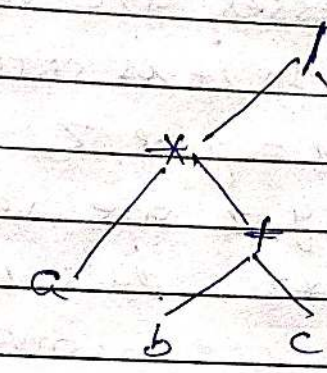
a+b



a*(b+c)/d ⇒

a* bc+/d ⇒ abc+*/d

= abc+*d/



Infix notation ⇒ Left - root - Right node. [L-R'-R]

Prefix notation ⇒ root - left - Right [R'-L-R]

Postfix notation ⇒ left - Right - root [L-R-R']

From syntax tree

Q.1)

Infix ⇒ L-R'-R = a*(b+c)/d

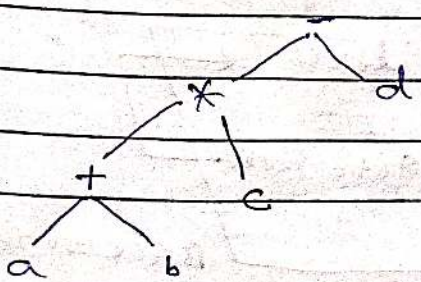
Prefix notation ⇒ R'-L-R ⇒

/ * a + b c d

Postfix notation ⇒ L-R-R'

a b c + * d /

Q.2) ((a+b) * c) -d.



Prefix ⇒ R'-L-R

- * + a b c d

Infix ⇒ L-R'-R

a+b * c - d

Post fix ⇒ L-R-R'

a b + c * d -

* Three Address code (TAC) ⇒ In three address code (TAC), at most three addresses are used to represent any statement. So it is called TAC.

Example:- $x = y$ op z

Here x, y, z are variables (operand) & op = operator

$a = x + y * z$ It's 4 address statement so we can convert into three address code

$t_1 = y * z$, $t_2 = x + t_1$, $a = t_2$

Types of three address statement ⇒

(i) Logical & Binary operator ⇒ $x = y$ op z

(ii) Unary operator ⇒ $x =$ op y

(iii) Copy statement ⇒ $x = y$

(iv) Procedure call ⇒ ~~return y~~

Parament x, Parameter y

(or)

Parameter x call;
~~return y~~
P(x) {
≡
return y;
}

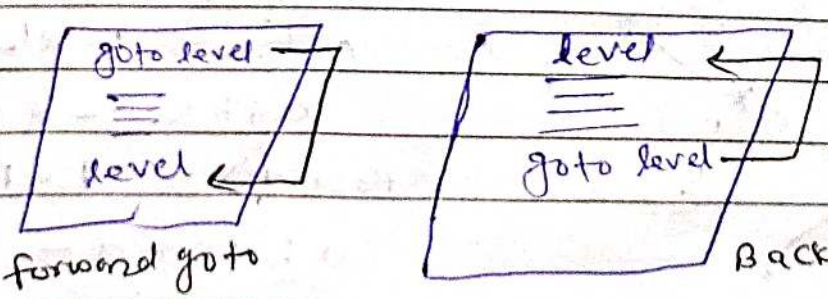
(v) Index Assignment ⇒ $x = y[i]$

$x[i] = y$

(vi) Conditional jump ⇒ if x relational operator y goto Label
(<=, >=)

(or)

If $x <= y$ goto level



forward goto

Backward goto

(viii) Address & Pointer Assignment ⇒ $x = \&y$

$x = *y$

Q.1) Generate/Generation of three address code (3-TAC)

$a = b + c + d$

Sol^n) Here + operator has ~~higher~~ higher Precedence over "=" operator & "+" operator is left Associative.

$t_1 = b + c$, $t_2 = t_1 + d$, $a = t_2$

Q.2) Generation of 3-AC (Three Address code) for

$-(a*b) + (c+d) - (a+b+c+d)$

Sol^n)

$t_1 = a * b$

$t_2 =$ minus t_1

$t_3 = c + d$

$t_4 = t_1 + t_3$

$t_5 = a + b$

$t_6 = t_2 + t_5$

$t_7 = t_4 - t_6$

$-[T_1] + [T_2] - [T_3]$

$\Downarrow$

$[T_4] + [T_2] - [T_3]$

$\Downarrow$

$[T_5] - [T_3]$

$\Downarrow$

$[T_6]$

Q.3) Generate TAC for

if $A < B$

then 1

else 0

Sol^n) (i) If $(A < B)$ goto 4

(ii) $t_1 = 0$

(iii) goto 5

(iv) $t_1 = 1$

Q.4] Generate TAC for
"if a<b and c<d then t=1 else t=0"

Soln (i) if (a<b) goto 3
(ii) goto 4
(iii) if (c<d) goto 6
(iv) t=0
(v) goto 7
(vi) t=1
(vii)

* Implementation of TAC (three address code) ⇒

Three address code (TAC) can be implemented as a record with the Address fields. These are three representations used for three address code

TAC representation → Quadruples
→ Triples
→ Indirect triples

⇒ **Quadruples ⇒** In quadruples representation, each instruction is divided into four fields operand, arg1, arg2 & result.

where

* The operation field is used to represent the internal code for operation.
* The arg1 & arg2 (argument) fields represent the two operation used.
* The result field is used to store the result of an expression

Q.1) Translate the following expression to quadruple, Triple & indirect triples.

$$a + b * c / e \uparrow f + b * a$$

Soln Priority $\uparrow \geq * \geq / \geq +$ & Associativity.

construct an TAC for given expression

$t_1 = e \uparrow f$, $t_2 = b * c$, $t_3 = t_2 / t_1$

$t_4 = b * a$ $t_5 = a + t_3$, $t_6 = t_5 + t_4$

Quadruple ⇒

Location	operation	Argument-1	Argument-2	Result
(0)	$\uparrow$	e	f	t_1
(1)	*	b	c	t_2
(2)	/	t_2	t_1	t_3
(3)	*	b	a	t_4
(4)	+	a	t_3	t_5
(5)	+	t_5	t_4	t_6

Exception in quadruple representation ⇒

* The statement $x = op\ y$ where op is a unary operator is represented by placing op in the operation field. y in the argument 1 field & x in the result field. The argument 2 field is not used.
* A statement like parameter t_1 is represented by placing parameter in the operator field & t_1 in the argument 1 field.
* Un conditional & conditional jump statements are represented by placing the target labels (levels) in the result field.

⇒ Triples ⇒ In trille representation, the use of temporary variables is avoided & instead references to instructions are made.

operator, arg1, arg2

Advantage:- result is not stored so we save time & memory

Location	Operation	argument-1	Argument-2
(0)	↑	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	4	(3)

Disadvantages:- It is very difficult for optimization compiler.

⇒ Indirect triples ⇒ This representation is an enhancement over triples representation. It uses an additional instruction array to list the pointer to the triples in the desired order. Thus it uses pointers instead of position to store results which enables the optimizers to freely reposition the sub expression to produce an optimized code.

Execution Statement order table

	Statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

triple table

location	operator	argument 1	argument 2
~~Unimes~~ (0)	↑	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Q.2 Expression = $-(a+b) * (c+d) + (a+b+c)$

① $t_1 = a+b$ ② $t_2 = -t_1$ ③ $t_3 = c+d$

④ $t_4 = t_2 * t_3$ ⑤ $t_5 = a+b$ ⑥ $t_6 = t_5 + c$

⑦ $t_7 = t_4 + t_6$

Quadruples ⇒

S.No.	operation	operand-1	operand-2	Result
1	+	a	b	t_1
2	-	t_1	—	t_2
3	+	c	d	t_3
4	*	t_2	t_3	t_4
5	+	a	b	t_5
6	+	t_5	c	t_6
7	+	t_4	t_6	t_7

Advantage :- Statements can be moved arround

Disadvantage :- Too much of space is wasted

Triple

Advantage :- space is not wasted

Disadvantage :- statements can not be moved.

S.No.	operator	operand 1	operand - 2
1	+	a	b
2	−	(1)	
3	+	c	d
4	*	(2)	(3)
5	+	a	b
6	+	(5)	c
7	+	(4)	(6)

Indirect triples ⇒

	Pointer
(i)	(1)
(ii)	(2)
(iii)	(3)
(iv)	(4)
(v)	(5)
(vi)	(6)
vii	(7)

Advantage :- Statements can be moved.

Disadvantage :- Too memory access (ज्यादा)

Q. Translate the following expression to quadruple triple & indirect triple.

$a = b * -c + b * -c$

Soln Construct TAC

① $t_1 = -c$ ② $t_2 = b * t_1$ ③ $t_3 = -c$

④ $t_4 = b * t_3$ ⑤ $t_5 = t_2 + t_4$ ⑥ $a = t_5$

PAGE NO.:
DATE:

Quadruple ⇒

location	operator	argument-1	Argument-2	result
(1)	Unary minus	c	–	t_1
(2)	*	b	t_1	t_2
(3)	Unary minus	c	–	t_3
(4)	*	b	t_3	t_4
(5)	+	t_2	t_4	t_5
(6)	=	t_5	–	a

Triple ⇒

location	operator	argument-1	Argument-2
(1)	Unary minus	c	–
(2)	*	b	(1)
(3)	Unary minus	c	–
(4)	*	b	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Indirect triple ⇒

	Statement	location	operator	argument 1	argument 2
35	(1)	(1)	Unary minus	c	–
36	(2)	(2)	*	b	(1)
37	(3)	(3)	Unary minus	c	–
38	(4)	(4)	*	b	(3)
39	(5)	(5)	+	(2)	(4)
40	(6)	(6)	=	a	(5)

↑
Statement Execution order table.

PT. ⇒

Q.1 Write the quadruples triples for the following

$(x+y) * (y+z) + (x+y+z)$

Soln Convert into TAC

① $t_1 = x+y$, ② $t_2 = y+z$ ③ $t_3 = t_1 * t_2$

④ $t_4 = t_1 + z$ ⑤ $t_5 = t_3 + t_4$

Quadruples

location	operator	arg-1	arg-2	Result
1	+	x	y	t_1
2	+	y	z	t_2
3	*	t_1	t_2	t_3
4	+	t_1	z	t_4
5	+	t_3	t_4	t_5

Triple

Location	Operator	Argument-1	Argument-2
1	+	x	y
2	+	y	z
3	*	(1)	(2)
4	+	(1)	z
5	+	(3)	(4)

(TAC)

More About translation & an intermediate code generation

- Array reference in arithmetic
- Procedure call
- Case statement

⇒ Array reference in Arithmetic ⇒

- 1-D Array
- 2-D Array

1-D Array

Example:- int a[5] = {15, 10, 11, 44, 39}

	Base address			width(w) 4 byte		
Actual address in memory	1100	1104	1108	1112	1116	←
Element	15	10	11	44	39	
Address with Respect to Array	0	1	2	3	4	← i
	↑ Lower bound				↑ Upper bound	

Address A[i] = Base Address + width * (i - Lower bound)

(or)

Address of A[i] = B + w * (i - LB)

Base Address B = 1100 width (w) = w

Lower bound (LB) = 0 i = 4

A[4] = 1100 + 4 * (4 - 0) = 1100 + 16 = 1116

Q.1 Convert into three address code.

```
int i;
int a[10];
i = 1;
while (i < 10)
{
    a[i] = 0;
    i = i + 1;
}
```

```
(1)  i = 1
(2)  if (i < 10) goto (4)
(3)  goto (8)
(4)  t1 = 4 * i
(5)  a[t1] = 0
(6)  i = i + 1
(7)  goto (2)
(8)  stop
```

2-D Array ⇒

→ Row major implementation

→ column major implementation

Lr →	0	1	2	3
row index 0	8 (1000)	6 (1004)	3 (1008)	2 (1012)
1	4	5	9 (1024)	1
2	6	3	2	4

A[1][2]

* Row major implementation ⇒

Row-1 | Row-2 | Row-3

A[0][0] = A[M][N]	8	6	3	2	4	5	9	1	6	3	2	4
	1000	1004	1008	1012	1016	1020	1024	1028	1032	1036	1040	1044

$A[i][j] = \text{base address} + w * [N * (i - L_r) + (j - L_c)]$

w: size of element; L_r: lower bound of row; L_c: lower bound of column

Example $A[1][2] = 1000 + 4 * [4(1-0) + (2-0)] \Rightarrow 1000 + 4 * (4+2) = 1000 + 24$

$= 1024 \Rightarrow A[1][2] = 9$

* Column Major implementation ⇒

col-1 | col-2 | col-3 | col-4

	8	4	6	6	5	3	3	9	2	2	1	4
	1000	1004	1008	1012	1016	1020	1024	1028	1032	1036	1040	1044

$A[i][j] = \text{base address} + w * [(i - L_r) + m * (j - L_c)]$

$A[1][2] = 1000 + 4 * [(1-0) + 3 * (2-0)]$ (M = 3, N = 2)

$= 1000 + 4[1 + 6] = 1000 + 28 = 1028 = 9$

Q.1 Convert into TAC

```
int i
int a[10][10]
i = 0
while (i<10)
{
    a[i][j] = 1;
    i++;
}
```

(1) i = 0
(2) if (i<10) goto (4)
(3) goto (8)
(4) $t_1 = 44 * i$
(5) a[t_1] = 1
(6) i = i+1
(7) goto (2)
(8) Stop

a[10][0] = B + w[10(i - L_r) + (j - L_c)]

a[i][i] ⇒ = B + 4[10(i-0) + (i-0)]

P.T.O.

Convert into TAC

* **Case statement :-**

```
switch(ch)
{
case 1: c=a+b;
        break;
case 2: c=a-b;
        break;
}
```

(1) if ch=1 goto (3)
(2) if ch=2 then goto (5)
(3) c=a+b (or) t_1=a+b, c=t_1
(4) goto (7)
(5) c=a-b (or) t_1=a-b, c=t_1
(6) goto (7)
(7) Stop

* **Procedure call ⇒ if given function**

P(A_1, A_2, A_3, ... A_n)

Parameter A_1
Parameter A_2
Parameter A_3
{
Parameter A_n
call P, n

(procedure)

where P = function name
n = no. of actual parameter

Example

```
void main()
{
int x, y;
swap(&x, &y);
}
void swap(int *a, int *b)
{ int i;
  i = *b;
  *b = *a;
  *a = i
}
```

TAC

(1) call main
(2) Parameter &x
(3) Parameter &y
(4) call swap, 2 — no. of parameter
(5) i = *b;
(6) *b = *a;
(7) *a = i
(8) stop

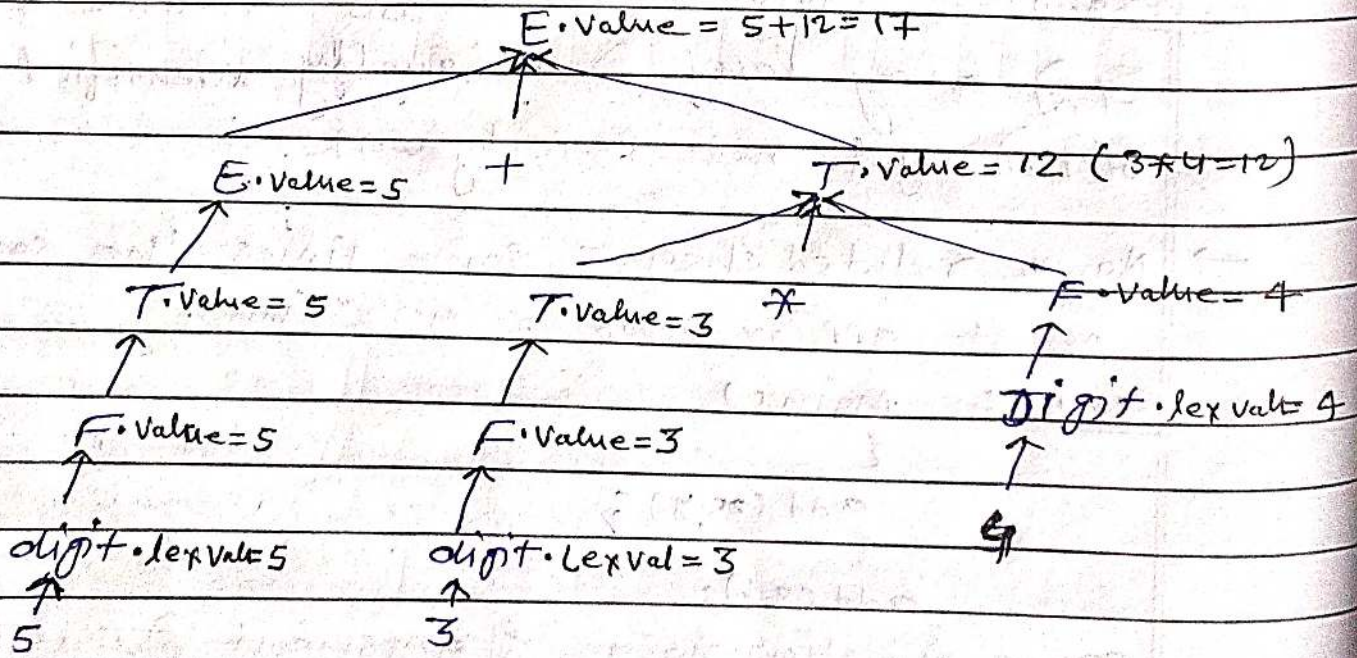
Dependency Graph ⇒ The directed graph that represents the inter-dependencies between synthesized & inherited attributes at nodes in the parse tree is called "Dependency graph". Dependency graph determines the evaluation order of the semantic ~~order~~ rules.

Example:-

$E \rightarrow E+T$	{ E.value = E.value + T.value }
$E \rightarrow T$	{ E.value = T.value }
$T \rightarrow T*F$	{ T.value = T.value * F.value }
$T \rightarrow F$	{ T.value = F.value }
$F \rightarrow (E)$	{ F.value = E.value }
$F \rightarrow digit$	{ F.value = digit.lexvalue }

digits are :- 1|2|3|4|5|6|7|- - - - .

input string = "5 + 3 * 4" Draw a dependancy graph



It's dependency graph ⇒ Semantic rules + evaluation order = dependency graph.

Q.1) Generate TAC for the following

```
c=0
do
{
  if (a<b) then
     x++;
  else
     x--;
  c++;
}
while (c<5)
```

① c=0
② if (a<b) goto 4
③ goto 7
④ $T_1 = x+1$
⑤ $x = T_1$
⑥ goto 9
⑦ $T_2 = x-1$
⑧ $x = T_2$
⑨ $T_3 = c+1$
⑩ $c = T_3$
⑪ if (c<5) goto 2
⑫ ---

Q.2) Generate TAC (Three address code) for

```
while (A<C and B>D) do
  if A=1 then C=C+1
else
  while A<=D
    do A=A+B
```

Sol^{n}

① if (A<C) goto 3
② goto 15
③ if (B>D) goto 5
④ goto 15
⑤ if (A=1) goto 7
⑥ goto 10
⑦ $T_1 = C+1$
⑧ $C = T_1$
⑨ ~~$T_1 = C+1$~~ goto 1
⑩ if (A<=D) goto 12
⑪ goto 1
⑫ $T_2 = A+B$
⑬ $A = T_2$
⑭ goto 10
⑮ ---

[3-AC]

* TAC [Three Address code] ⇒

- → for boolean expression
- → for "if" expression
- → for "if ---then" expression
- → for "while" loop
- → for "for" loop
- → For do ---while loop
- → code repeate
- → switch case.

① For boolean expression

Step-I

(a) if a<b

Sol^{n}

(1) if a<b goto 4
(2) $t_1 = 0$
(3) goto next (5)
(4) $t_1 = 1$
(5) - - - -

Note:- goto next means out of domain (program)

Step-II

if (a<b) or (c<d)
{
x=x+1;
}

Sol^{n}

(1) if (a<b) goto 4
(2) if (c<d) goto 4
(3) goto Next
(4) x=x+1
(5) - - - -

② For "if" expression

if (a<b) and (c<d)
{
x=x+1
}

Sol^{n}

(1) if a<b goto 3
(2) goto Next
(3) if c<d goto 5
(4) goto Next
(5) x=x+1
(6) - - - -

③ for if ---than expression

Step-I

if a<b then
x=y+3

Sol^{n}

(1) if (a<b) goto 3
(2) goto next
(3) $t_1 = y+2$
(4) x=t_1
(5) - - - -

DATE:

Step-II ⇒ if (a<b) then
x = x+1
else
y = y*z

Sol

① if (a<b) goto 5
② $t_1 = y * z$
③ $y = t_1$
④ goto Next
⑤ $t_2 = x + 1$
⑥ $x = t_2$

OR

① if (a<b) goto 3
② goto 5
③ $t_1 = x + 1$
④ $x = t_1$
⑤ goto Next
⑥ $t_2 = y * z$
⑦ $y = t_2$

---

④ For while loop

```
i = 1
while (i<=n)
{
  x = x*z
  i = i+1
}
```

Sol

① $i = 1$
② if $i <= n$ goto 4
③ goto Next
④ $t_1 = x * z$
⑤ $x = t_1$
⑥ $t_2 = i + 1$
⑦ $i = t_2$
⑧ goto 2

OR

① $i = 1$
② if $i <= n$ goto 4
③ goto Next
④ $t_1 = x * z$
⑤ $x = t_1$
⑥ $t_2 = i + 1$
⑦ $i = t_2$
⑧ goto 2

---

⑤ For "for" loop ⇒

```
for (i=1 to n)
{
  x = x*z;
}
```

Sol

① $i = 1$
② if $i <= n$ goto 4
③ goto Next
④ $t_1 = x * z$
⑤ $x = t_1$
⑥ $t_2 = i + 1$
⑦ $i = t_2$
⑧ goto 2

DATE:

Example → for (i=1; i<=20; i++)
{
x = y+z;
}

Sol^n
① i=1
② if i<=20 goto 7
③ goto NEXT
④ $t_1 = i+1$
⑤ $i = t_1$
⑥ goto 2
⑦ $t_2 = y+z$
⑧ $x = t_2$
⑨ goto 4

(6) for
DO - - while loop →

do
x = y+z
while (a < b)

Sol^n
① $t_1 = y+z$
② $x = t_1$
③ if a<b goto 1
④ goto NEXT

i=0
do
i = i+1
x = x+1
y = y*z
while (i<n)

Sol^n
① i=0
② i = i+1
③ x = x+1
④ y = y*z
⑤ if i<n
⑥ goto Next

or → $t_1 = i+1$, $i = t_1$

(7) code repeate

i=0
Repeate
x = x+1
y = y*z
i = i+1
Until (i>=n)

Sol^n
① i=0
② x = x+1
③ y = y*z
④ i = i+1
⑤ if (i>=n) goto Next
⑥ goto 2

or → $t_1 = x+1$, $x = t_1$
or → $t_2 = y*z$, $y_2 = t_2$

(8) For "switch case"

```
switch a
begin
case 0: x=x+1;
        break;
case 2: y=y*z;
        break;
case 4: z=z/2;
        break;
Default: z=x+y;
         break;
end.
```

Soln

```
(1)  a
(2)  if a=0 goto 4
(3)  goto 5
(4)  x=x+1
(5)  goto Next
(6)  if a=2 goto 8
(7)  goto 9
(8)  y=y*z
(9)  goto Next
(10) if a=4 goto 12
(11) goto 13
(12) z=z/2
(13) goto Next
(14) if a=default goto 16
(15) goto 17
(16) z=x+y
(17) goto Next
```

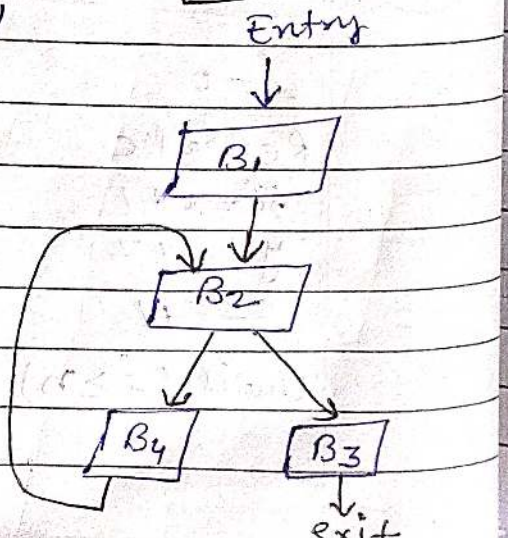
Q.1) Construct 3-AC for following

```
int A[i=20];
    x=0;
for (i=1 to n)
{
A[i]=x;
x=x+2;
}
```

Soln

```
(1)  x=0 - Leader - B1
(2)  i=1
(3)  if i<=n goto 5 - leader - B2
(4)  goto Next - leader - B3
(5)  t1 = const A - leader - B4
(6)  t2 = i*w   or  20
(7)  t3 = x
(8)  t1[t2] = t3
(9)  t4 = x+2
(10) x = t4
(11) t5 = i+1
(12) i = t5
(13) goto 3
```

B1 ⇒ 1, 2
B2 = 3
B3 = 4
B4 = 5 to 13



Q.2] $A[i] = B[K+j] + C[l]$

Sol^n.

(1) $t_1 = const\_A - L - B_1$
(2) $t_2 = const\_B$
(3) $t_3 = const\_C$
(4) $t_4 = K + j$
(5) $t_5 = t_4 * w_B$
(6) $t_6 = t_2[t_5]$
(7) $t_7 = l * w_c$
(8) $t_8 = t_3[t_7]$
(9) $t_9 = t_6 + t_8$
(10) $t_{10} = i * w_A$
(11) $t_{11} = t_1[t_{10}]$
(12) $t_{11} = t_9$

Q.3]

```
For i=1 to n
{
For j=1 to n
{
C[i,j] = A[i,j] + B[i,j]
}
}
```

Sol^n

(1) $t_1 = const\_A$
(2) $t_2 = const\_B$
(3) $t_3 = const\_C$
(4) $i = 1$
(5) $j = 1$
(6) if $i \le n$ goto 8
(7) goto NEXT
(8) if $j \le n$ goto 12
(9) goto 6
(10) $t_4 = i + 1$
(11) $i = t_4$
(12) goto 6
(13) $t_5 = i * n$
(14) $t_6 = t_5 + j$
(15) $t_7 = t_6 * w$
(16) $t_8 = t_1[t_7]$
(17) $t_9 = i * n$
(18) $t_{10} = t_9 + j$
(19) $t_{11} = t_{10} * w_B$
(20) $t_{12} = t_2[t_{11}]$
(21) $t_{13} = i * n$
(22) $t_{14} = t_{13} + j$
(23) $t_{15} = t_{14} * w_c$
(24) $t_{16} = t_3[t_{15}]$
(25) $t_{17} = t_8 + t_{12}$
(26) $t_{16} = t_{17}$
(27) $t_{18} = j + 1$
(28) $j = t_{18}$
(29) goto 8