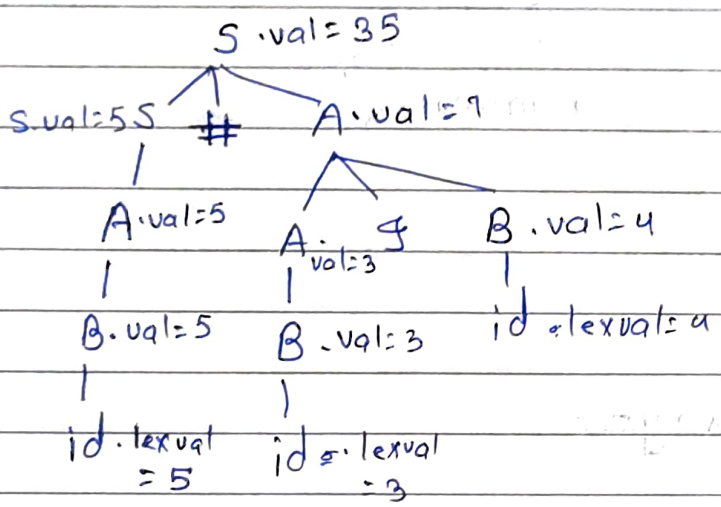
Syntax directed translation.

1]

grammar	~~rules~~. semantic rules
$S \to S \# A \mid A$	$\{ S.val = S.val * A.val \}$
$A \to A \$ B \mid B$	$\{ A.val = A.val + B.val \}$
$B \to id$	$\{ B.val = id.lexval \}$

given string 5 # 3 $ 4
Give its output.

→



annotated parse tree.

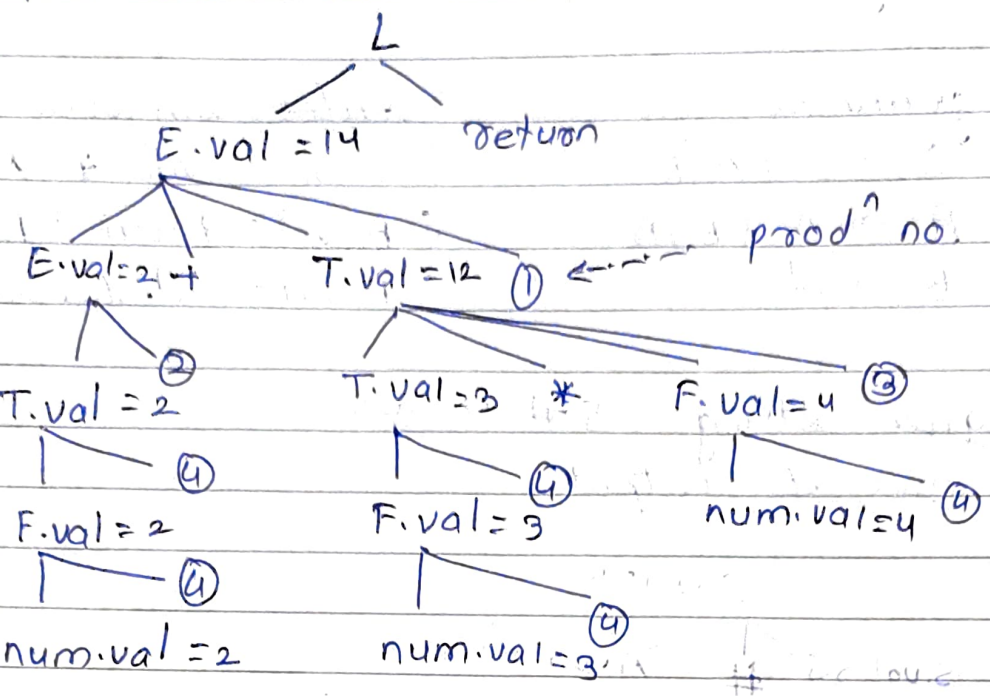
2] Postfix 234 * +.

→

$L \to En$	
$E \to E+T$	$\{ print('+') \}$ ①
$E \to T$	$\{ \}$ ②
$T \to T*F$	$\{ print('*') \}$ ③
$T \to F$	$\{ \}$ ④
$F \to num$	$\{ print('num.val') \}$ ⑤

SDT. – semantic rules
–[variable + extra attribute
+
each prod^n] → semantic rule.

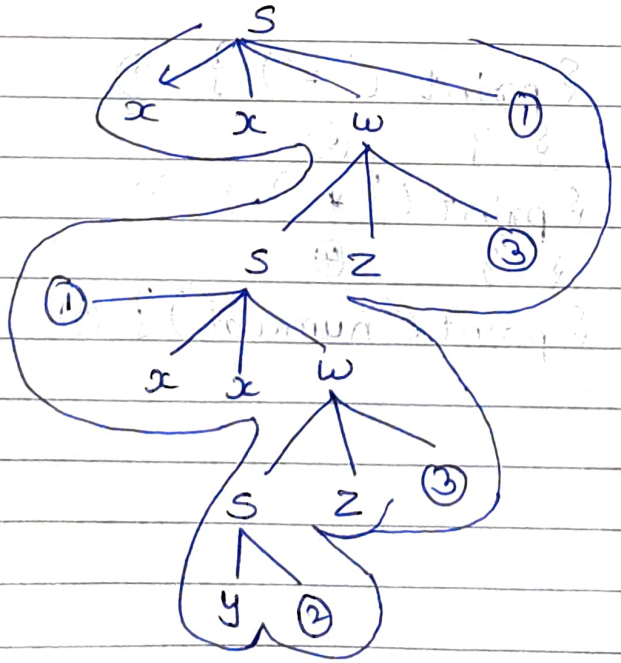
Ex - E → E1 + T E.val



③ S → x x w
S → y
w → sz
string = xxxxyzz

→

S → x x w { print (1); }
S → y { print (2); }
w → s z { print (3); }



23131

grammar + semantic rule = SDT

4]

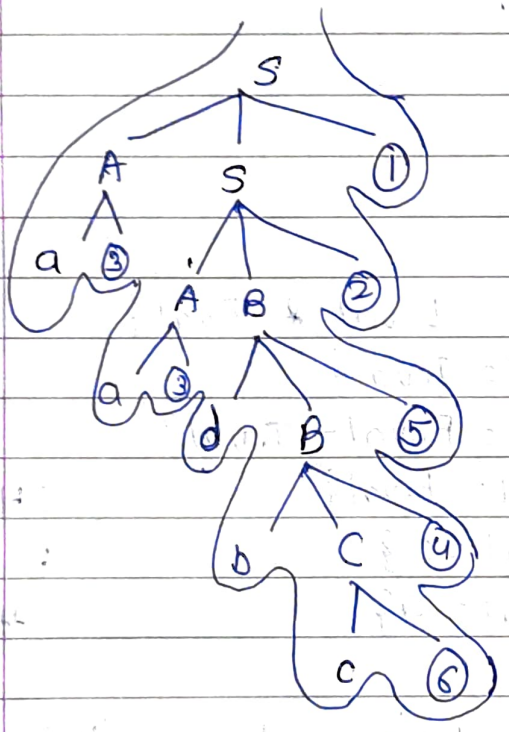
grammar	semantic rules	
$S \to AS$	{printf (1)}	①
$S \to AB$	{pf (2)}	②
$A \to a$	{pf (3)}	③
$B \to bC$	{pf (4)}	④
$B \to dB$	{pf (5)}	⑤
$C \to c$	{pf (6)}	⑥

given SDT & input is aadbc.
What is output?

→ ① Top-down parsing :-

(Write parse tree corresponding to given input)

(You can use top-down / bottom-up)



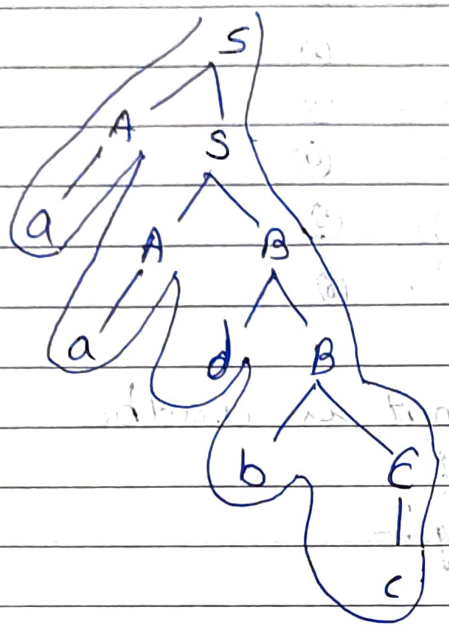
In top-down parsing, we write corresponding semantic action.

output = 3364521

SDD – semantic action
- each prod^n – set of actions need to be performed.
- enclosed {}
- Ex - $E \to E_1 + T$ {print '+'}

② bottom-up parsing.

In bottom-up parsing, we don't write corresponding semantic action.



op - 3364521

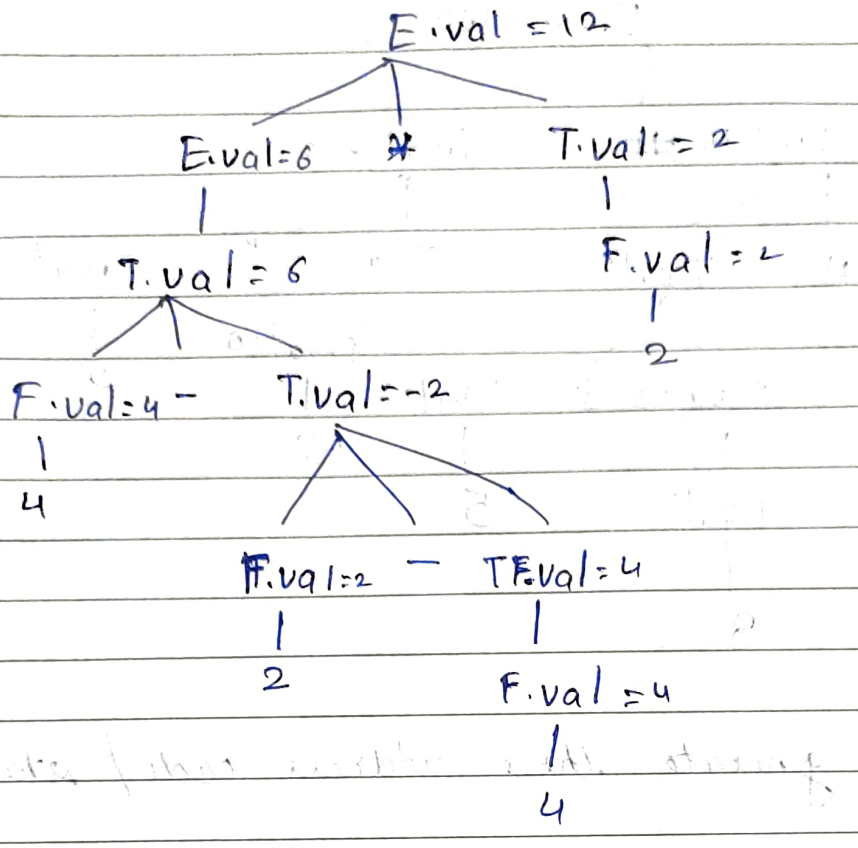
5]

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

(Here val is an attribute)

Construct parse tree & derive string = 4-2-4*2.

↓ annotated



6] $E \rightarrow E \# T$ $\{E.val = E.val * T.val\}$

$E \rightarrow T$ $\{E.val = T.val\}$

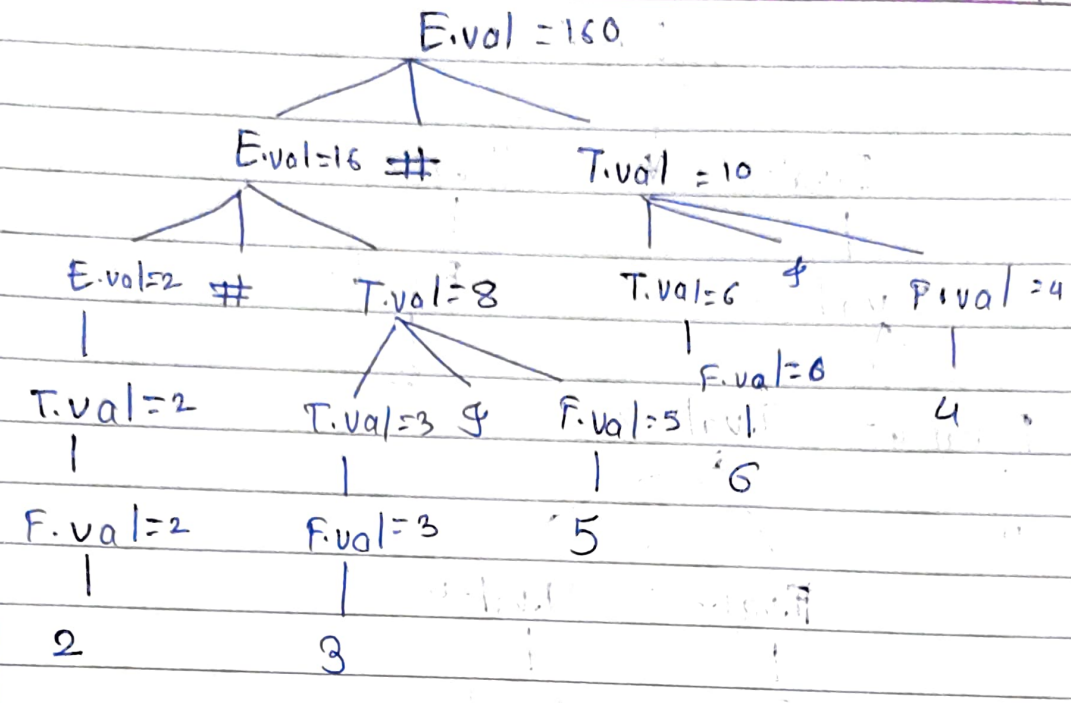
$T \rightarrow T \$ F$ $\{T.val = T.val + F.val\}$

$T \rightarrow F$ $\{T.val = F.val\}$

$F \rightarrow num$ $\{F.val = num.lval\}$

* Construct annotated parse tree & derive string
  2 # 3 $ 5 # 6 $ 4

→



SDD to generate three address code / state -

$x := a + b * c$

$t1 = a$

$t2 = b$

$t3 = t1 + t2$

$t4 = c$

$t5 = t3 * t4$

three address code =

$t1 = b * c$

$t2 = a + t1$

$x = t2$

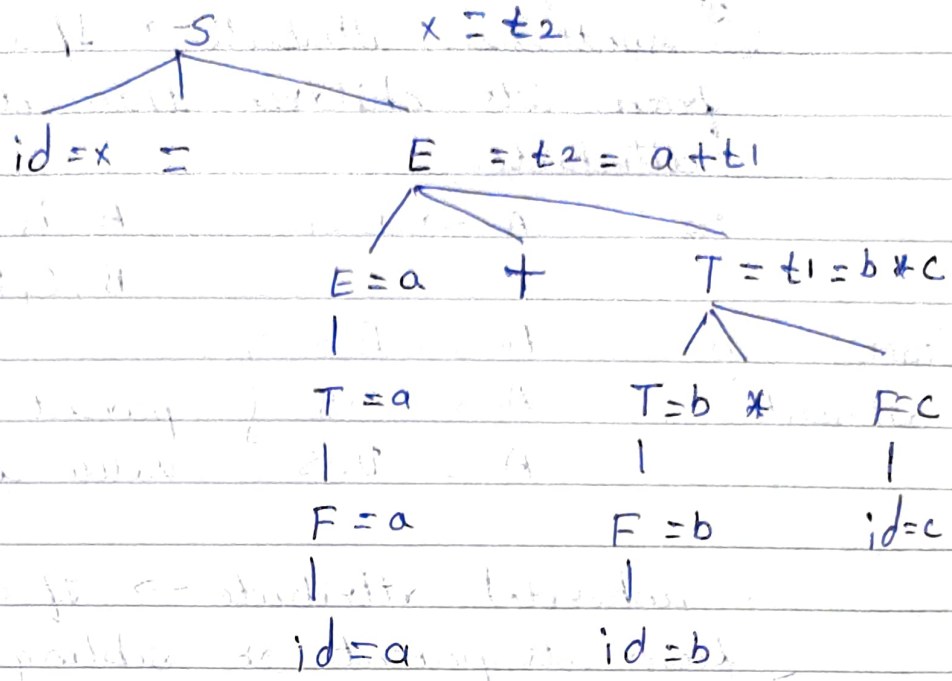
rules.

$S \rightarrow id = E$	{ gen (id.name = E.place)}
$E \rightarrow E + T$	{ E.place = newtemp (); gen (E.place = E.place + T.place)}
$E \rightarrow T$	{ E.place = T.place}
$T \rightarrow T * F$	{ T.place = newtemp(); gen(T.place = T.place * F.place)}
$T \rightarrow F$	{ T.place = F.place}
$F \rightarrow id$	{ F.place = id.name}

(Here place is an attribute).

Page No.
Date

Parse tree =



* S - attributed

① It uses only synthesised attribute.

② Semantic actions placed at right hand side.
Ex - $A \rightarrow BC \{\}$

③ Attributes are evaluated by bottom up parser.

L - attributed

It uses both inherited & synthesised attribute.
Each inherited attribute is restricted to inherit either from parent & left sibling only.
Ex - $A \rightarrow BCD$
① $C.i = A.i$ ✓
② $C.i = B.i$ ✓
③ $C.i = D.i$ ✗

Semantic actions placed anywhere at R.H.S.
Ex - $A \rightarrow BC\{\}$
$A \rightarrow \{\} BC$
$A \rightarrow B\{\}C$.

Attributes are evaluated by depth first traversal in bottom up parser.

* Types of attributes -

① synthesised attribute → If a node takes value from its children then it is synthesised attribute.

Ex - $A \rightarrow BCD$ — A is parent node. B, C, D are children node.

(Here S is attribute)

$A.S = B.S$
$A.S = C.S$
$A.S = D.S$

} parent node is taking value from its children nod

② inherited attribute → If a node takes value from its parent or sibling.

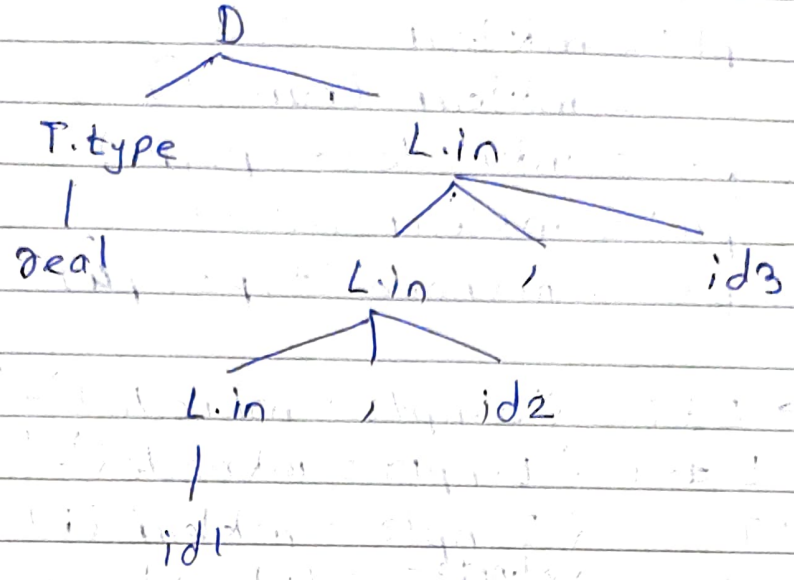
Ex - $A \rightarrow BCD$

$C.i = A.i$ (Here i is attribute)
$C.i = B.i$
$C.i = D.i$

* Given below SDT. By using inherited attribute draw parse tree for string (real id1, id2, id3).

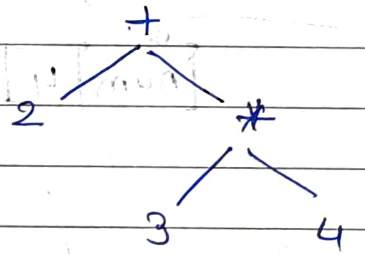
prod	semantic rules.
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$ addtype (id.entry, L.in)
$L \rightarrow id$	addtype (id.entry, L.in)

→ parse tree -

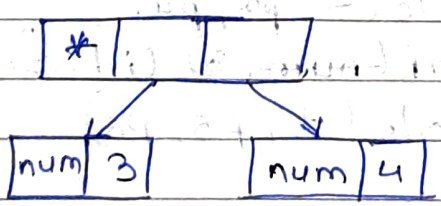


* ~~***~~ Construction of syntax tree.

- syntax tree is abstract form of parse tree.
- ① 2 + 3 * 4



- There are 3 major functions -
① mknode (operator node, left child, right child)



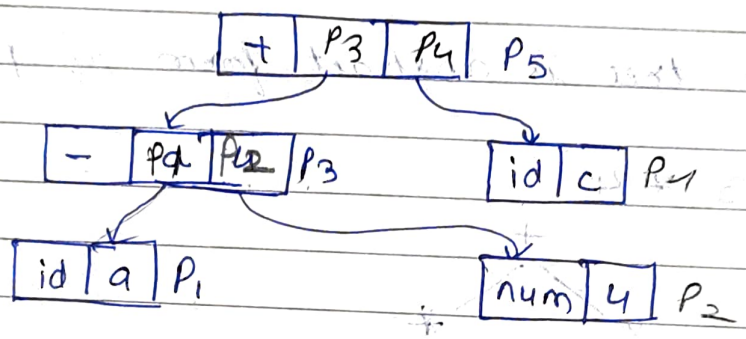
② mkleaf (id, entry) - creates identifier node

③ mkleaf (num, val) - create leaf node.

Ex ① - a - 4 + c

→ p1 := mkleaf (id, a)
p2 := mkleaf (num, 4)
p3 := mknode (-, p1, p2)
p4 := mkleaf (id, c)
p5 := mknode (+, p3, p4)

E → E+T {E.nptr = mknode ('+', E.nptr, T.nptr)}
E → E-T {E.nptr = mknode ('-', E.nptr, T.nptr)}
E → T {E.nptr = mkleaf (id, T.entry)}
T → id {T.nptr = mkleaf (id, id.entry)}
T → num {T.nptr = mkleaf (num, num.entry)}



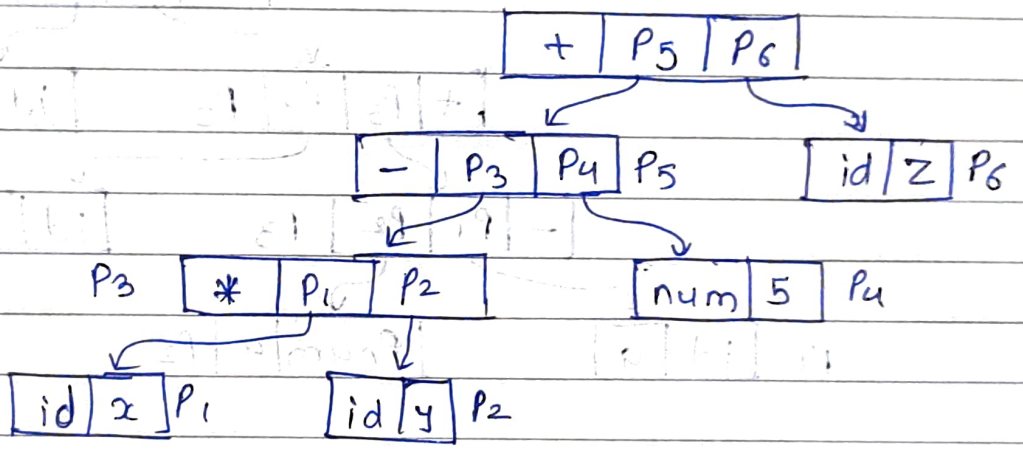
Ex ② - x * y - 5 + z

→ p1 := mkleaf (id, x)
p2 := mkleaf (id, y)
p3 := mknode (*, p1, p2)
p4 := mkleaf (num, 5)
p5 := mknode (-, p3, p4)
p6 := mkleaf (id, z)
p7 := mknode (+, p5, p6)

SDD for x * y # - 5 + z.

prodⁿ	semantic rules.
E → E1 + T	{E.nptr = mknode('+', E1.nptr, T.nptr)}
E → E1 - T	{E.nptr = mknode('-', E1.nptr, T.nptr)}
E → E1 * T	{E.nptr = mknode('*', E1.nptr, T.nptr)}
E → T	{E.nptr = T.nptr.mkleaf(id, T.entry)}
T → id	{T.nptr = mkleaf(id, id.entry)}
T → num	{T.nptr = mkleaf(num, num.entry)}

syntax tree -



Ex ① Ex ③ a - 4 + c * b

→

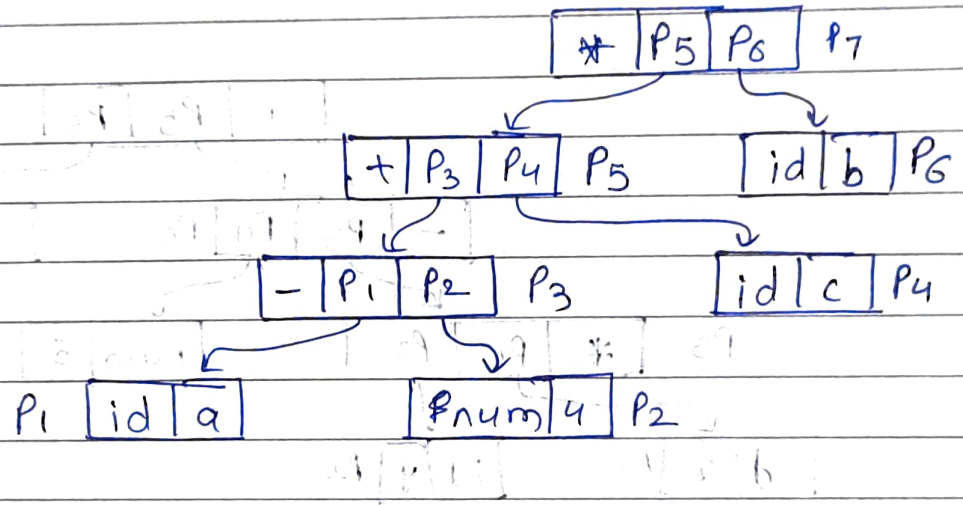
steps :-

p1 := mkleaf(id, a)
p2 := mkleaf(num, 4)
p3 := mknode(-, p1, p2)
p4 := mkleaf(id, c)
p5 := mknode(+, p3, p4)
p6 := mkleaf(id, b)
p7 := mknode(*, p5, p6).

SDD

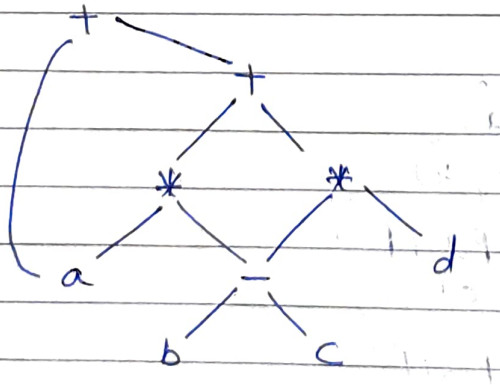
$E \rightarrow E_1 + T$	{E.nptr = mknode('+', E_1.nptr, T.nptr)}
$E \rightarrow E_1 - T$	{E.nptr = mknode('-', E_1.nptr, T.nptr)}
$E \rightarrow E_1 * T$	{E.nptr = mknode('*', E_1.nptr, T.nptr)}
$E \rightarrow T$	{E.nptr = mkleaf(id, T.entry)}
$T \rightarrow id$	{E.T.nptr = mkleaf(id, id.entry)}
$T \rightarrow num$	{T.nptr = mkleaf(num, num.entry)}

syntax tree



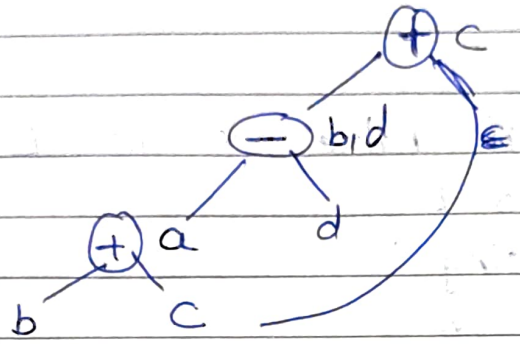
* Construction of DAG.

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



② $a = b + c$ $\quad$ $c = b + c$

$b = a - d$ $\quad$ $d = a - d$

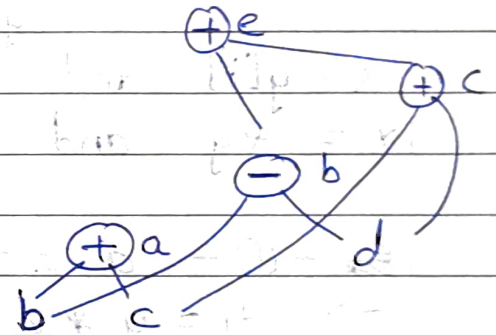


③ $a = b + c$

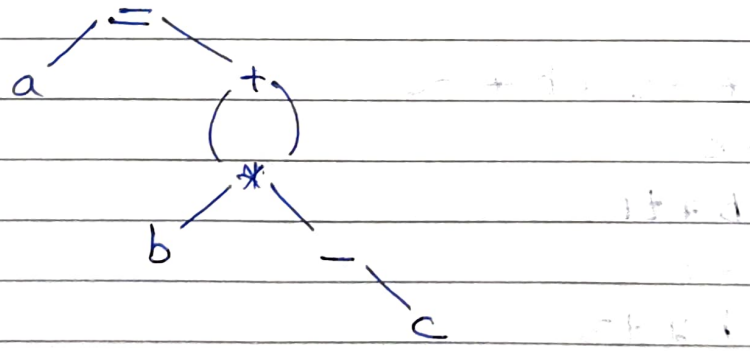
$b = b - d$

$c = c + d$

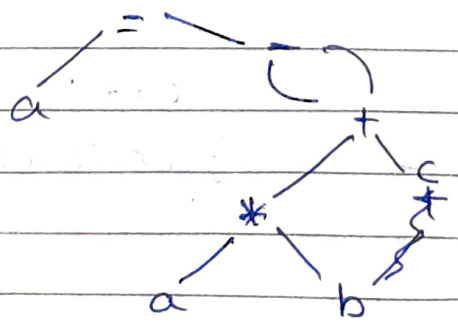
$e = b + c$



④ $a = b * -c + b * -c$



⑤ $a = (a * b + c) - (a * b + c)$



* Forms of three address instr^{n} –

x = y op z (binary)
x = op y (unary)
x = y
goto L
if x goto L and if false x goto L
if x relop y goto L.
procedure calls using :
- param x
call p, n
y = call p, n.
x = y[i] and x[i] = y
x = &y and x = *y and *x = y.

Ex – ① – a := x + y * z // three address code
→ t1 = y * z
t2 = x + t1
a = t2

② x := b * -c + b * -c
→ t1 = -c
t2 = b * t1
t3 = -c
t4 = b * t3
t5 = t2 + t4
x = t5

③ 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 ( )
4) T1 = x + 1
5) x = T1

(4) while (A < C and B > d) do
    if A = 1 then c = c + 1
    else
        while A <= D
            do A = A + B;

→

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

→

if ch = 1 goto L1
if ch = 2 goto L2
L1 : T1 = a + b
     a = T1
     goto Last
L2 : T2 = a - b
     a = T2
     goto Last.

* Data structure for TAC.

1) Quadraples -
It has 4 fields = op, arg1, arg2, result.
Ex - x := b * -c + b * -c

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
x  := t5
```

	operator	arg1	arg2	result
1)	uniary minus	c	-	t1
2)	*	b	t1	t2
3)	unary minus	c	-	t3
4)	*	b	t3	t4
5)	+	t2	t4	t5
6)	=	t5	-	x

2) triples -
It has 3 fields = op, arg1, arg2.
Ex - x = a + a * (b-c) + (b-c) * d.

```
t1 := b - c
t2 := a * t1
t3 := b - c
t4 := t3 * d
t5 := a + t2
t6 := t5 + t4
x  := t6
```

	op	arg1	arg2
0	minus	b	c
1	*	a	(0)
2	minus	b	c
3	*	(2)	d
4	+	a	(1)
5	+	(4)	(3)
6	=	x ~~(5)~~	(5)

3) indirect triples -
triples + list of pointers to triples

Page No. _____
Date _____

Ex - a = b * - c + b * - c

		op
t1 := - c	35	(0)
t2 := b * t1	36	(1)
t3 := - c	37	(2)
t4 := b * t3	38	(3)
t5 := t2 + t4	39	(4)
a := t5	40	(5)

	op	arg1	arg2
(0)	unary minus	c	-
1	*	b	(0)
2	unary minus	c	-
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

3. given string x := a + b * c . Write TAC & SDD for TAC.

→ x := a + b * c

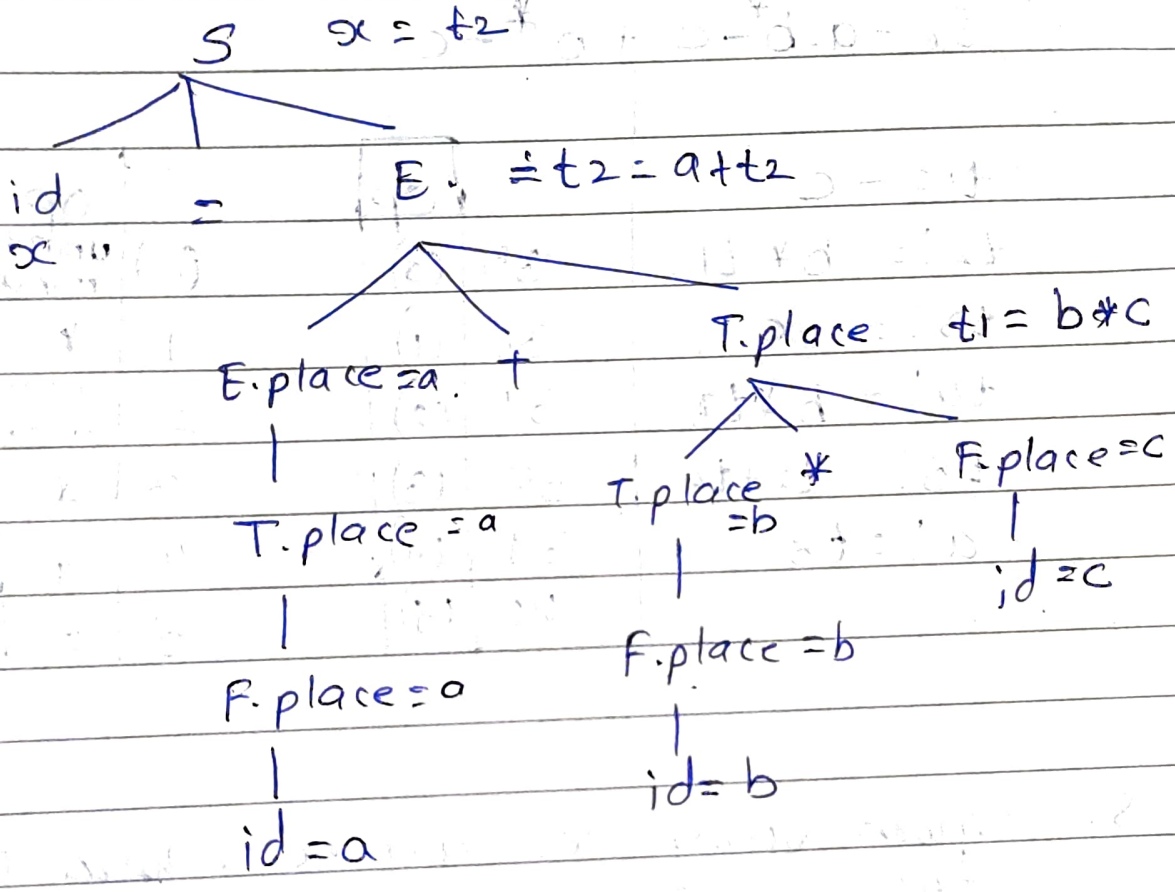
t1 = b * c

t2 = a + t1

x := t2

SDD

S → id = E	{ gen (id.name = E.place) }
E → E + T	{E.place = newtemp(); gen(E.place + T.place)}
E → T	{ E.place = T.place}
T → T * F	{T.place = newtemp(); gen (T.place * F.place)}
T → F	{T.place = F.place}
F → id	{F.place = id.name }


S
x = t2
id
x
=
E.
t2 = a + t2
E.place = a
t
T.place
t1 = b * c
T.place = a
T.place
=b
*
F.place = c
id = c
F.place = a
F.place = b
id = a
id = b