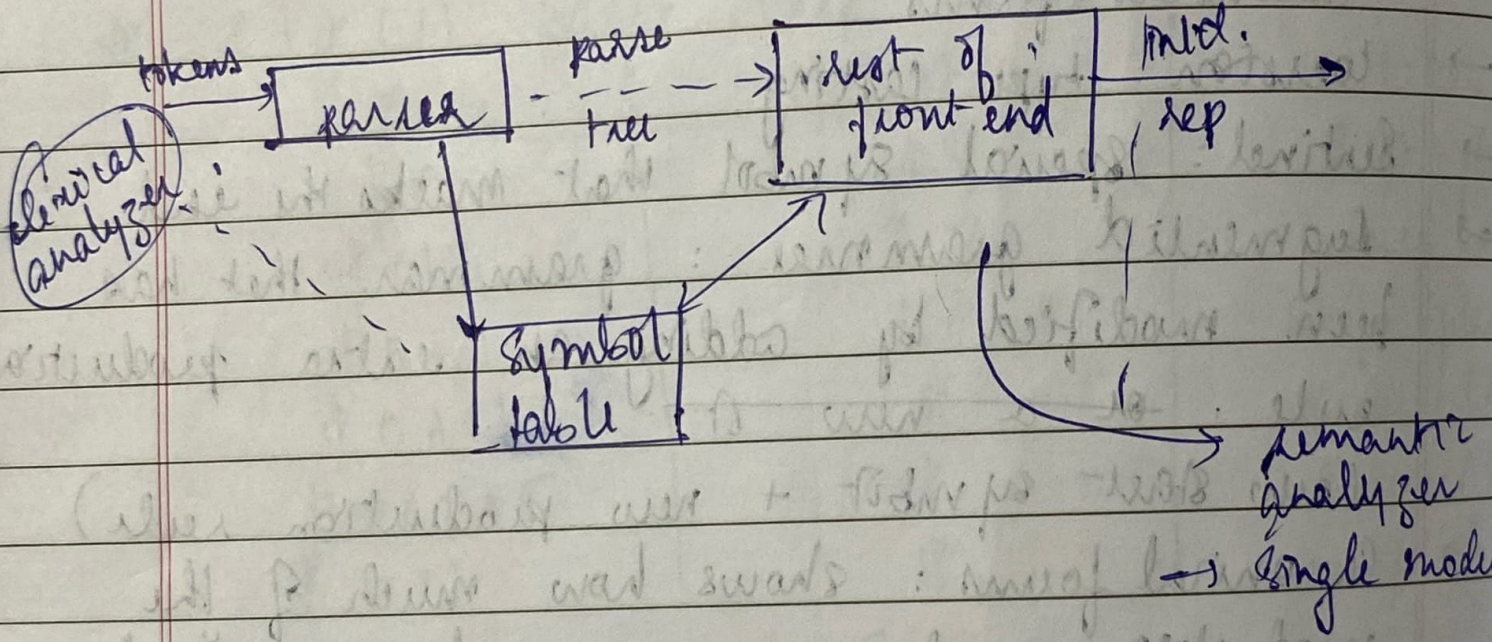
Unit - 3



→ Syntax directed translation
" guided by CFGs.
" Attach attributes to grammar symbol
" SDD - Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

⇒ SDD : is a context free grammar together with attributes & rules
↳ attributes are associated with grammar symbols.

→ rules are associated with prod.
eg of attributes: numbers, types, table refs or strings (may even be code in the intermediate language).

⇒ Synthesized Attribute:
* SA for a non-terminal A at a parse tree node N is defined by semantic rule associated with the production at N
* Production must have A as its head.
* SA at node N is defined only in terms of attribute values at the children of N and at N itself.

eg:
production $E \rightarrow E_1 + T$
semantic rule $E.code = E_1.code \,||\, T.code \,||\, "+"$

⇒ Inherited attribute for a non terminal B at
* for a non terminal B at node N is defined by a semantic rule associated with the production at the parent of N.
where production must have B as a symbol in its body.
* defined only in terms of attribute values at N's parent, N itself and N's siblings

eg: $T \rightarrow FT'$ | $T'.inh = F.val$
$T' \rightarrow *FT_1$ | $T_1'.inh = T'.inh \times F.val$

* Synthesized attributes at N to be defined in terms of inherited attributes values at N itself.
$T' \rightarrow \epsilon$ | $T'.syn = T'.inh$
* do not allow an inherited attribute at N to be defined in terms of attribute values at the children of node N.

* Terminals can have synthesized attributes, but not inherited attributes.
* attributes for terminals have lexical values that are supplied by the lexical analyzer.

F → digit | F.val = digit.lexval

Example

	Production	Semantic rules
(1)	L → En	L.val = E.val
(2)	E → E1 + T	E.val = E1.val + T.val
(3)	E → T	E.val = T.val
(4)	T → T1 * F	T.val = T1.val x F.val
(5)	T → F	T.val = F.val
(6)	F → (E)	F.val = E.val
(7)	F → digit	F.val = digit.lexval

12.11.25

Unit - 3

Syntax Directed Definition.

CFG along with semantic values

E.val = E.val + T.val

E.val = T.val

(1) Synthesized attribute

A → BCD

A.S = B.S

A.S = C.S

A.S = D.S

(2) Inherited Attribute

A → BCD

C.x = A.x

B.x = B.x

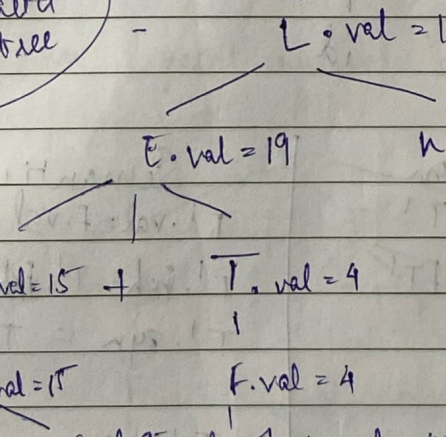
C.x = D.x

* S- attributed : will only use synthesized attribute, bottom up parsing
* L-attributed : uses both synthesized and inherited. L.n = D.n is invalid since. C comes after D on the grammar (uses only the value left of itself).

Production	Semantic rules
L → En	L.val = E.val
E → E+T	E.val = E.val + T.val
E → T	E.val = T.val
T → T*F	T.val = T.val x F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval.

eg. Input string : 3*5+4n.

annotated parse tree -



* Start from the top and go to left most node.

basically 3*5+4n 3 has to come in the leftmost leaf node.

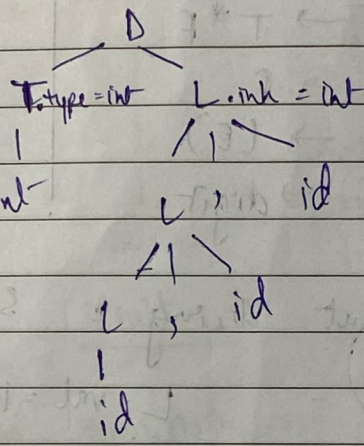
P.T.O ⇒

inherited attrib.

Production	Semantic rules.
D → TL	L.inh = T.type
T → int	T.type = int
T → float	T.type = float
L → L, id	L.inh = L.inh addtype (identity, L.inh)
L → id	addtype (identity, L.inh)

Input String : int a, b, c

annotated parse tree:

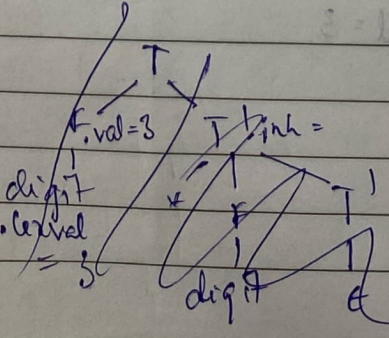


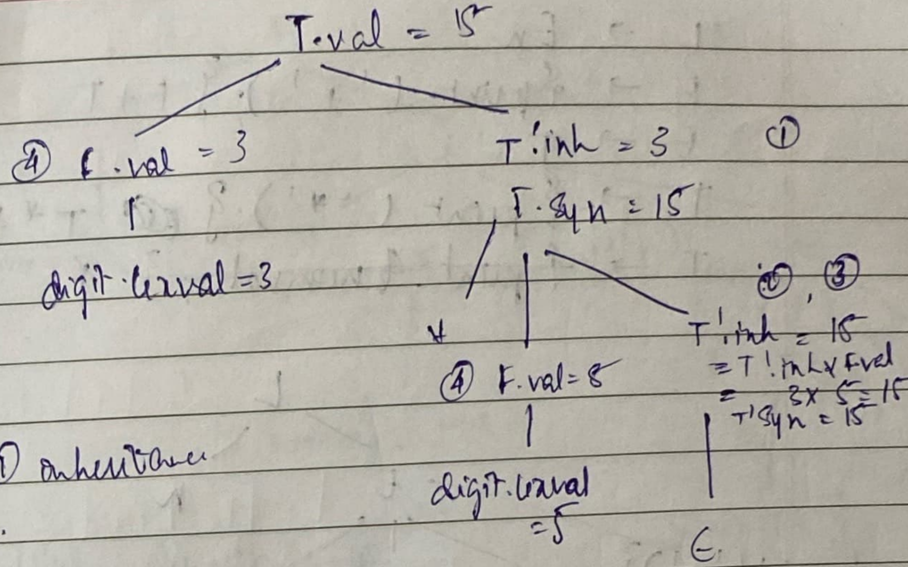
L.inh nodes will take either parent / sibling value.

**	Production	Semantic rules.
①	T → FT'	T'.val = F.val ; T.val = T'.syn
②	T' → *FT'	T'.inh = T'.inh * F.val T'.syn = T'.syn
	T → ϵ	
③	T' → ϵ	T'.syn = T'.inh
④	F → digit	F.val = digit.lexval.

Input String = 3 * 5

Annotated parse tree





from ① inheritance rules.

Syntax Directed Translation.

eg. infix to postfix : 2 + 3 * 4

E → E+T { print ('+'); }

E → T

T → T*F { print ('*'); }

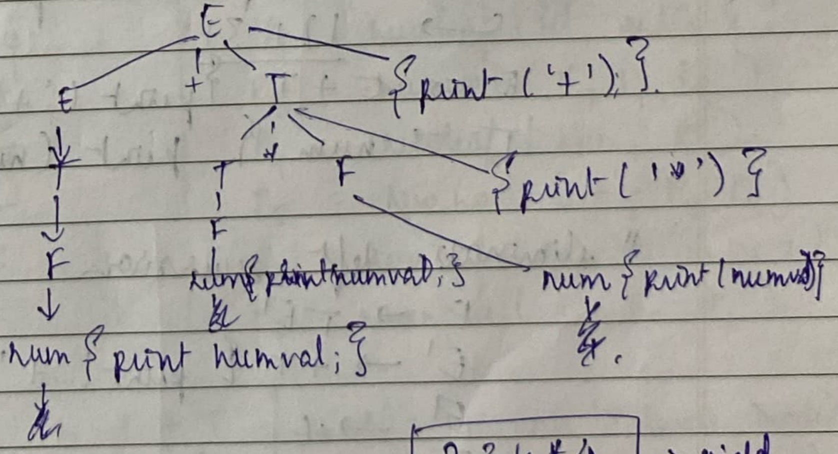
T → F

F → num { print numval; }

2+3 4 *

2 3 4 * +

only 2 question infix to postfix (only +) prefix



234*4 → yield

eg. Infix to prefix : 3*5+4

* 354

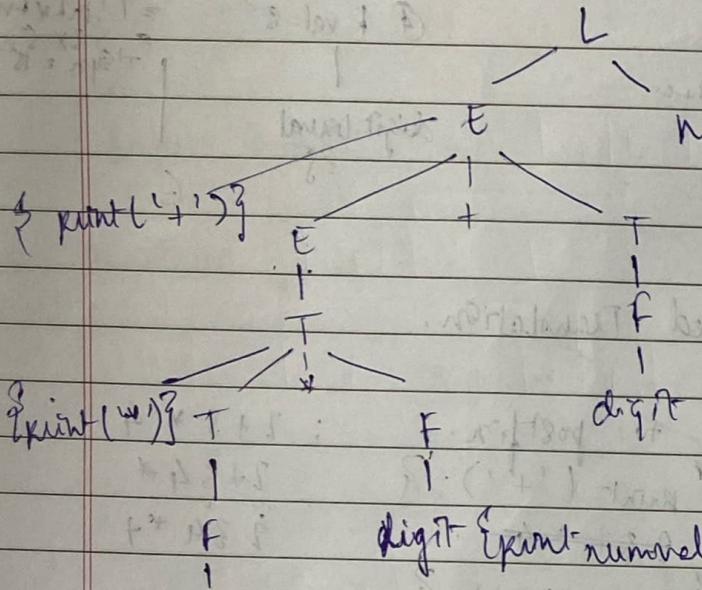
+*354

write print statement before the grammar

↳ yield (+*354)

L → En
E → {print('+');} E+T
E → T
T → {print('*');} T*F
T → ~~{print(&numval);}~~ F

F → (E)
F → digit {&numval;}



eg: 1+2+3 infix to postfix

↪ 12+ 3+

E → E+T {print('+');} | T

T → num {print(num value);}

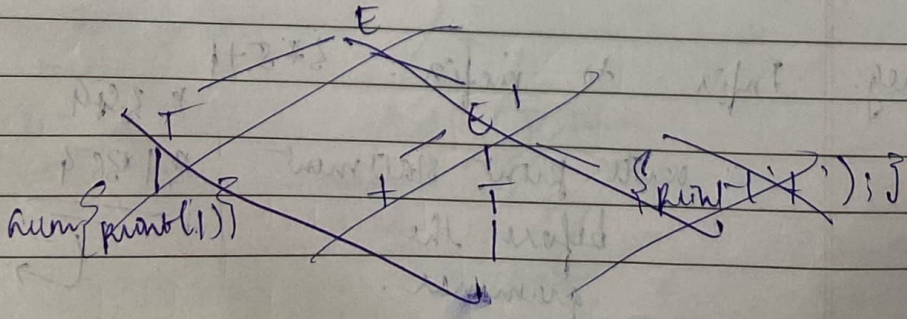
eliminate left recursion.

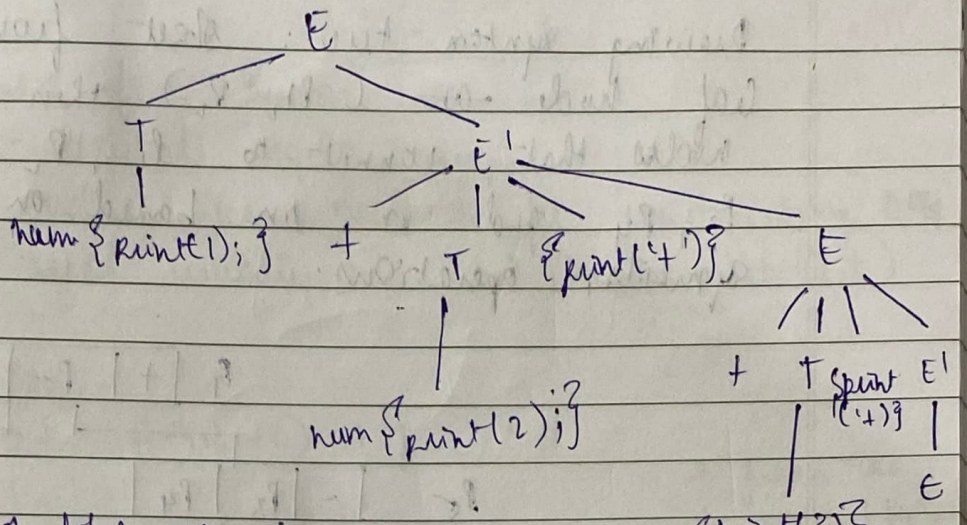
E → TE'

E' → +T {print('+');} E'

E' → ε

T → num {print(numvalue);}.





* most likely denoting infix to postfix.

Construction of Syntax Tree

① new node (op, left, right)
② new leaf (id, entry - ST) ⇢ symbol table entry to when encountering identifier.
③ new leaf (num, value)

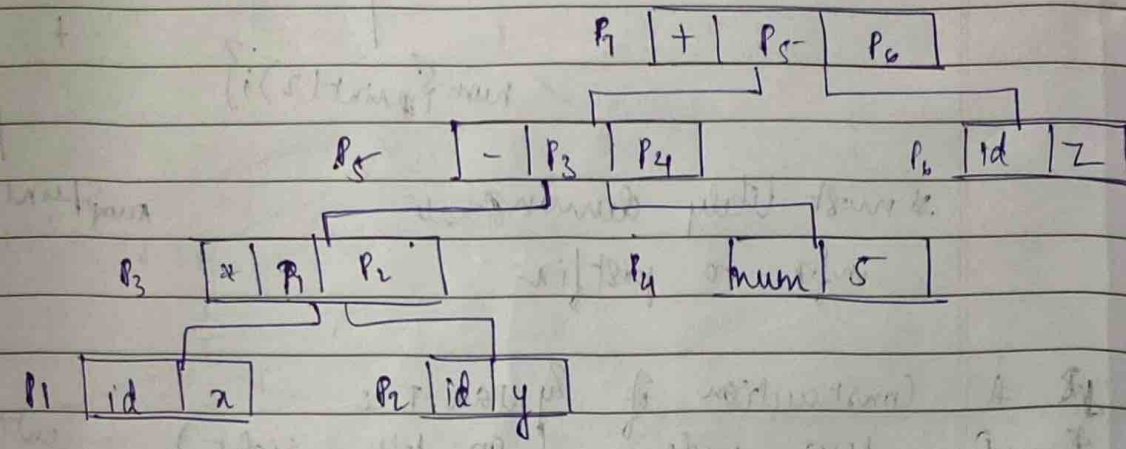
eg: a * y - 5 + z

Symbol	operation.
x	P_1 = new leaf (id, entry x)
y	P_2 = new leaf (id, entry y)
*	P_3 = new node (*, P_1, P_2)
5	P_4 = new leaf (num, 5)
-	P_5 = new node (-, P_3, P_4)
z	P_6 = new leaf (id, entry z)
+	P_7 = new node (+, P_5, P_6)

SDD

E → E + T	Enode = new node (+, Enode, Tnode)
E → E - T	Enode = new node (-, n n)
E → E * T	Enode = n (*, n n)
E → T	Enode = Tnode.
T → id	Tnode = new leaf (id, entry-ST)
T → num	T.node = n (num, value)

Drawing Syntax tree: Start from the leaf node ~~do~~ (P_1, P_2) then create the nodes that connect to P_1, P_2 then for P_3, P_4 and so on based on the symbols, operations.



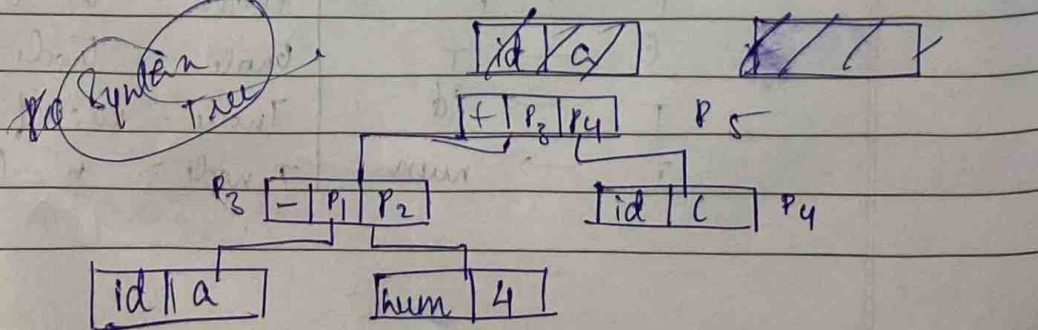
Q. Solve for $a - 4 + c$

Symbol	operation
a	P_1 = new leaf (id, entry a)
4	P_2 = new leaf (num, 4)
−	P_3 = new node (op, P_1, P_2)
c	P_4 = new leaf (id, entry c)
+	P_5 = new node (op P_3, P_4)

SDD

$E \rightarrow E - T$	Enode = new node (-, Enode, Tnode)
$E \rightarrow E + T$	Enode = " (+ " ")
$E \rightarrow T$	Enode = Tnode
$T \rightarrow id$	Tnode = new leaf (id, entry-ST
$T \rightarrow num.$	Tnode = new leaf (num, valu

Syntax Tree



Q. Write the procedure to create a syntax tree (with final tree).

Q. Draw DAG for given expr.

19.11.25

Unit - 4

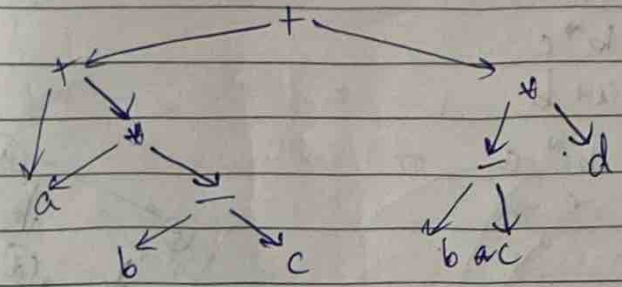
Variants of a Syntax Tree

① DAG - directed acyclic graph.

② The value no. method for constructing DAG (not asked usually). ↳ (mean what is it)

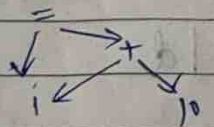
→ DAG

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



* graph can have multiple parent nodes.

eg: $i = i + 10$



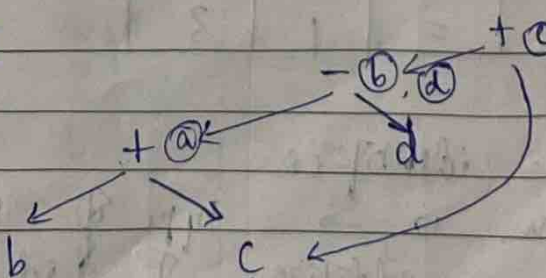
→ value no. method (array of records)
entire expression stored as array of records.

P:T-O ⇒

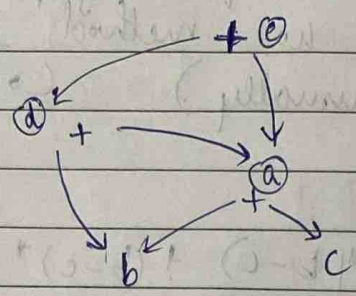
Q. $a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

* the c and c are not the same (both don't have the same value) but b, b are same.

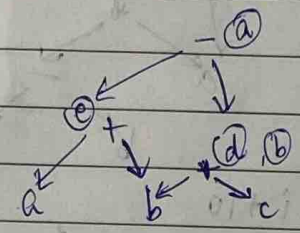
DAG



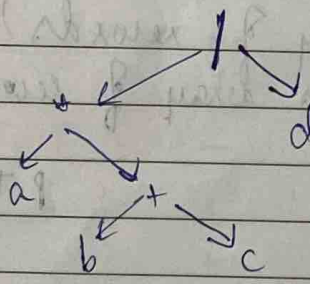
2. $a = b + c$
$d = b + a$
$e = d + a$

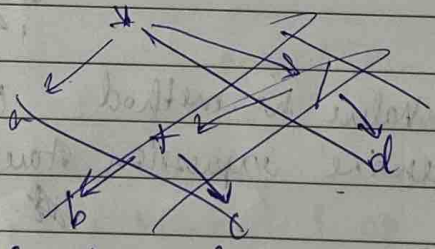


3. $d = b * c$
$e = a + b$
$b = b * c$
$a = e - d$



4. $a * (b + c) / d$





(left - right ; *, / and +, - have same level.)

⇒ Value no. method for $i = i + 10$

1	id		
2	num	10	
3	+	1	2
4	=	1	3

id : identifies
num, 10 → type of 10
③ → addition of id and num (1+2)
④ → assignment of value 3 to id.

Q. Write the DAG and rep. the eq. in ①, ②, ③ forms.

12 mark

Three Address code representation :-

A representation is known as three address code if there is a distination with almost ~~atleast~~ 2 operands and almost 1 operation.

⇒ [dest = operand 1 op. operand 2]

⇒ Three forms

① Quadruple ② Triple ③ Indirect triple

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

converting to three address code:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$a = t_1 + t_2$

$\Rightarrow t_5 = t_2 + t_4$

	op	arg 1	arg 2	result
[0]	−	c		t_1
[1]	*	b	t_1	t_2
[2]	−	c		t_3
[3]	*	b	t_3	t_4
[4]	+	t_2	t_4	t_5

⇓

QUADRUPLE

→ TRIPLE

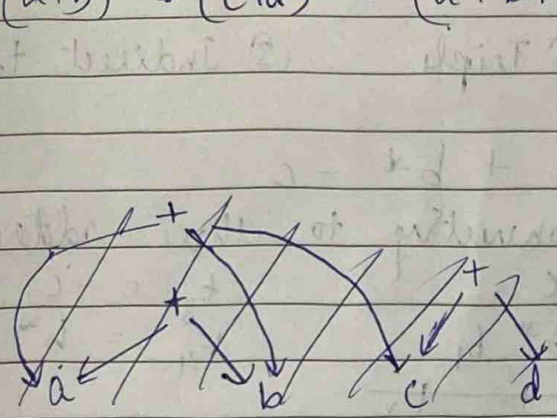
	op	arg 1	arg 2
[0]	−	c	
[1]	*	b	[0]
[2]	−	c	
[3]	*	b	[2]
[4]	+	[1]	[3]

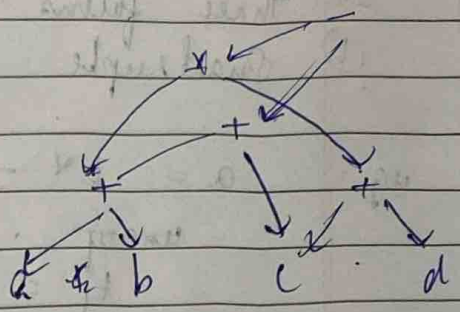
→ INDIRECT TRIPLE (triple table + address table)

1000	[0]
1001	[1]
1002	[2]
1003	[3]
1004	[4]

→ address table
each row for faster access.

Q. $(a+b) \uparrow (c+d) - (a+b+c)$





$t_1 = a + b$
$t_2 = c + d$
$t_3 = t_1 + c$
$t_4 = t_1 * t_2$
$t_5 = t_4 - t_3$

→ indirect triple

1000	[0]
1001	[1]
1002	[2]
1003	[3]
1004	[4]

→ Quadruple

	op	arg1	arg2	result
[0]	+	a	b	t_1
[1]	+	c	d	t_2
[2]	+	t_1	c	t_3
[3]	*	t_1	t_2	t_4
[4]	-	t_4	t_3	t_5

→ triple

	op	arg1	arg2
[0]	+	a	b
[1]	+	c	d
[2]	+	[0]	c
[3]	*	[0]	[1]
[4]	-	[3]	[2]

Types and Declarations (read theory from T.B)

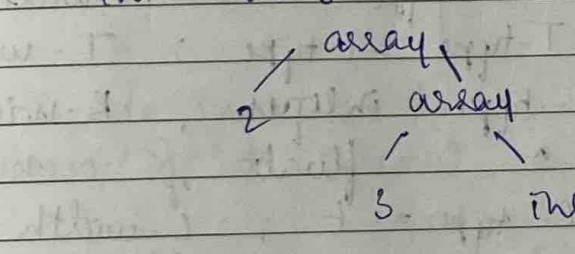
→ Type expression
→ Type equivalence
→ Declarations : problem only from this.
→ Storage layout for local names → eq. of declaration

Type expression is either a basic type or is formed by applying an operator called type constructor to a type expression

eg: types → int, float
constructor → list, tuple, array

eg: int [2] [3]

* how to represent array type.



generic type:
array (num, type)
here: array (2, ~~integer~~ array)
array (3, integer)
↳ inside array.
array (2, (3, integer))

Type equivalence indicates the comparison equivalency (list of integers compared to list of int's.;
array of diff. name → alias. same as array of int).
↳ check T.B.

Declarations.

eg: D → T id ; D | ϵ
T → B C | record '{' D '}'
B → int | float
C → ϵ | '[' num ']' C

declaration → type identifier
Type → int | float
identifier → ~~id, id~~ identifier , id | id.

→ D → T L
T → int | float
L → L , id | id.

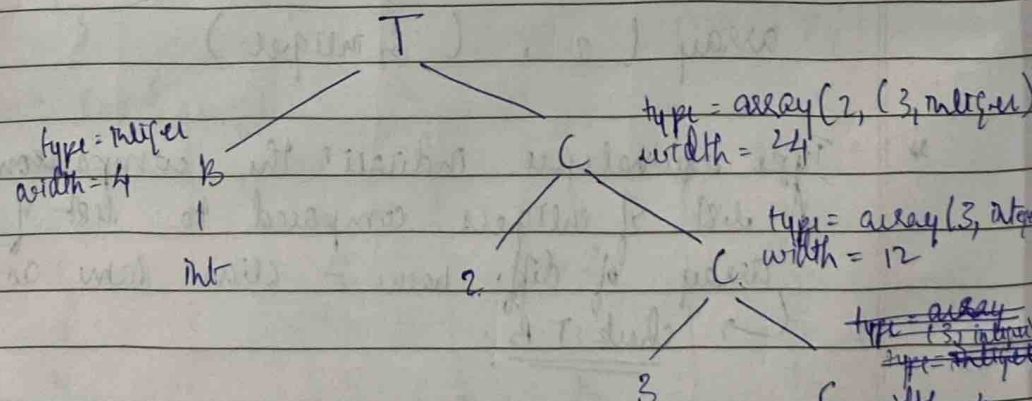
Simple grammar example to declare integer type variables.

eg: int a, b, c

→ Storage layout for local names

T → B { t = B.type ; w = B.width }
C { T.type = C.type ; T.width = C.width }
B → int { B.type = integer , B.width = 4 }
B → float { n = float , n = 8 }
C → ϵ { C.type = t ; C.width = w }
C → [num] C1 { C.type = array (num.value, C1.type);
C.width = num.value × C1.width; }

* int [2] [3]



* Sequence of declaration.

eg: P → D
D → D ; D
D → id : Type { enter (id.name, T.type, offset);
offset = offset + T.width }
T → integer { T.type = integer , T.width = 4 }
T → real { n = real , n = 8 }

→ first declaration $offset = 0$

→ seq. of declarations defines where or which address we place values in.

☆ Read from T.B. same questions.

☆☆ # Translation for boolean expressions.

① OR

$B \rightarrow B_1 \;||\; B_2$

translating into 3 address code:

$B_1.true = B.true$ (not direct assignment → if B_1 is true then B is true)

(if B_1 is true go to address of $B.true$).

$B_1.false = newlabel()$ → indicates need to check B_2.

$B_2.true = B.true$

↳ path where entire expression is true

$B_2.false = B.false$.

$B.code = B_1.code \;||\; label(B_1.false) \;||\; B_2.code$

② AND

$B \rightarrow B_1 \;\&\&\; B_2$

$B_1.true = B$ ke $newlabel()$

$B_2.true = B.true$

$B_2.false = B.false$

$B_1.false = B.false$

$B.code = B_1.code \;||\; label(B_1.true) \;||\; B_2.code$

Sorry not sorry :)

(3) NOT

$B \rightarrow !B_1$

$B_1.true = B.false$

$B_1.false = B.true$

$B.code = B_1.code$

B.code always computed from B_1.

Production	Semantic rules
$B \rightarrow True$	B.code = gen ('goto', B.true)
$B \rightarrow False$	B.code = gen (" , B.false)
$B \rightarrow E_1 \, rel \, E_2$	$B.code = E_1.code \,\|\|\, E_2.code \,\|\|$ gen('if' E_1 addr relop E_2 addr goto B.true) || gen(goto B.false)

rel: relational operator

eg: E_1 can be (2+5) or any expression or computation that has true / false val.

||| or &&

Complete semantic rules:

$B \rightarrow E_1 \, rel \, E_2$

$B.code = E_1.code \,||\, E_2.code \,||$ gen ('if' E_1 addr relop E_2 addr goto B.true) || gen (goto B.false)

Translation for Assign operator

① $S \rightarrow id = E$

$E \rightarrow E_1 + E_2$ (2) | $-E_1$ (3) | (E_1) (4) | id (5)

eg: $S = a$ ($S = id$)

$S = 10$ ($S = E_1$)

$S = 2 + 3$ ($S = E_1 + E_2$)

$S = 2 \, 3$;

Semantic rules of the grammar :

① S.code = E.code | gen (top.get (id.lexeme) = E.addr)

E.code : resolution of id = E
→ computation of expression.
then get the address of the identifier from the symbol table and save the value.

② E.addr = new Temp()
E.code = E_1.code || E_2.code ||
gen (E.addr = E_1.addr + E_2.addr)

③ E.addr = new Temp()
E.code = E_1.code || gen (E.addr = minus E_1.addr)

④ E.addr = E_1.addr
E.code = E_1.code

⑤ E.addr = top.get(id.lexeme)
E.code = ''

6.11.25 ⑥ Type checking

Q. Rules for type checking

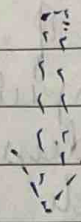
① $E \rightarrow E_1 + E_2$ { if ((E_1.type == E_2.type) && (E_1.type = int))
then E.type = int
else error ; }

② $E \rightarrow E_1 == E_2$ { if ((E_1.type == E_2.type)
&& (E_1.type = int | bool))
then E.type = boolean
else error ; }

③ $E_1 \rightarrow (E_1)$ { E.type = E_1.type }
$E \rightarrow$ num { E.type = int }

E → true
E → false
} { E.type = bool }.

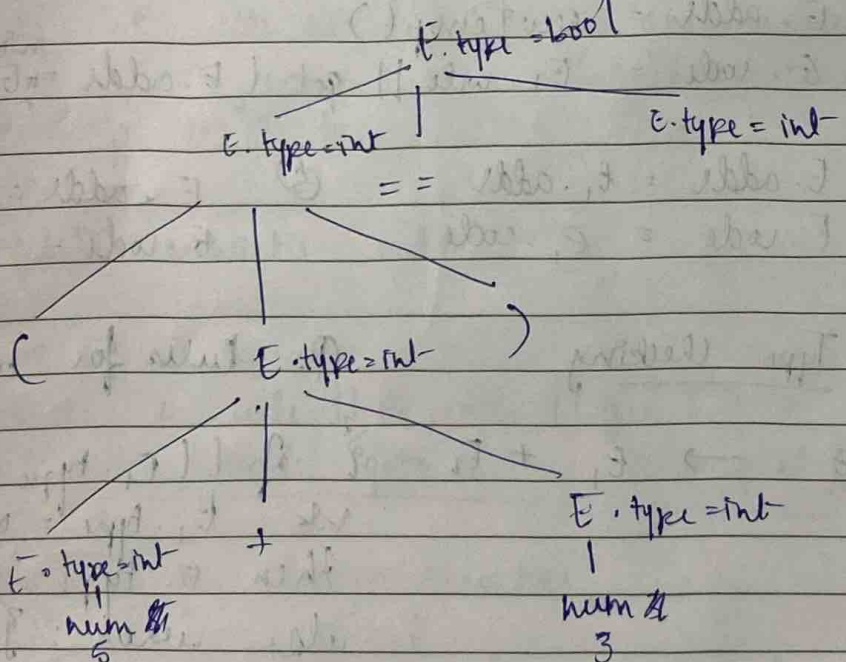
using the type casting rules we draw the annotated parse tree



& write the semantic rules for type checking.

Q) Draw the annotated tree and determine if it's a meaningful expression or not.
(5+3) == 8.

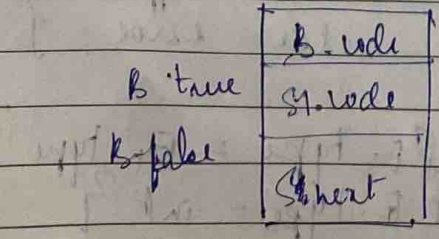
Steps: ① Parse tree ② compute value at each node



Intermediate code for flow control statements

S → if (B) then S1 (if)

(6 SM all conditions)



S → if ( ) S1

S1.next

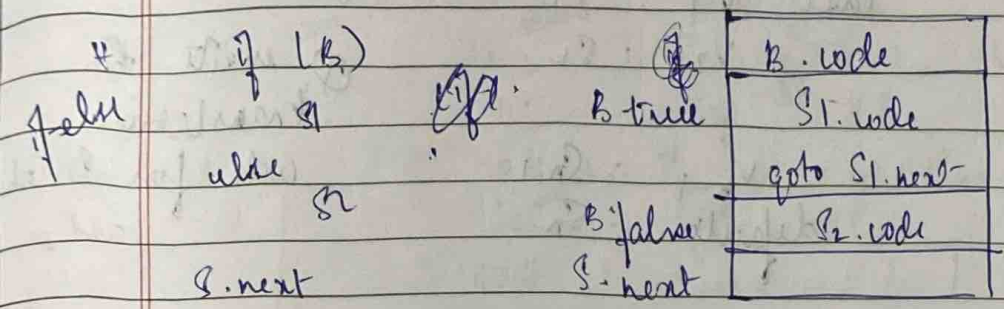
code that is next (follows)

Q. write the Translation code for flow control stmt.

→ Semantic Rule

B.true = ~~number of~~ newlabel()
B.false = S_1.next = S.next
S.code = B.code || label(B.true) || S_1.code.

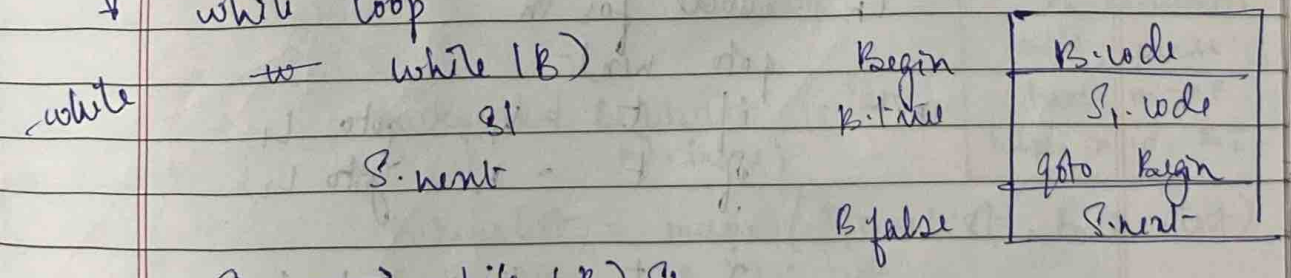
if (B) S1 else S2



B.true = newlabel()
B.false = newlabel()
S_1.next = S_2.next = S.next
S.code = B.code || label(B.true) ||
S_1.code || gen(goto S.next) ||
label(B.false) || S_2.code.

while loop

while (B)
S1
S.next



S → while (B) S_1

Begin = newlabel()
B.true = newlabel() B.false = S.next
S_1.next = Begin
S.code = label(Begin) ||
B.code || label(B.true) ||
S_1.code || gen(goto Begin)

→ always asked from last 15 years in SEE and CIE
⇒ Same problem will be asked.

~~while statement~~ Switch case

```
Switch (E)
{
    case V1 : S1
         V2 : S2
         :
         Vn-1 : Sn-1
    default : Sn
}
```

(not a flow control stmt.)

Q. write the translation code for switch case.

```
code to evaluate E into E
→ goto Test
   L1 : code for S1
        goto next
   L2 : code for S2
        goto next
   :
   Ln : code for Sn
        goto next
test : if t = V1 goto L1
       if t = V2 goto L2
       :
       if t = Vn-1 goto Ln-1
       goto Ln
next.
```

Q. if (x < 100 || y > 200 && x != y)   x = 0

note: if x < 100 is false the compiler never resolves rest of the expers. similarly if x > 200 is false the expr. is ~~rendt~~ resolved to false

⇒ short hand circuit

(not L → R)

Q. Resolve using backpatching

Q. Use an eg. to explain backpatching.

Backpatching: compiler decides which addr. to go to

→ Decoding: Pass 1:

address		
true → 100 :	if $x < 100$ then	106
101 :	goto	102
false 102 :	if $y > 200$ then	104
103 :	goto	107
104 :	if $x != y$ then	106
105 :	goto	107
106 :	true	
107 :	false	

fill the blanks in pass 2 ; pass 1 the compiler does not know where to go, pass 2 it determines where to go.

→ Translation rules for backpatching

$B \rightarrow B_1 \,||\, m\, B_2$

{

Backpatch (B1.falselist, m.instr)

B.truelist = merge (B_1.truelist, B_2.truelist)

B.falselist = B_2.falselist

}

(marker) m indicates the next addr in semantic rule; parse tree $m \rightarrow \epsilon$. helps in id. B_2

→ compiler goes to B_2 directly when B1 is false

$B \rightarrow B_1 \,\&\&\, m B_2$

{

Backpatch (B_1.truelist, m.instr)

B.truelist = B_2.truelist

B.falselist = merge (B1.falselist, B_2.falselist)

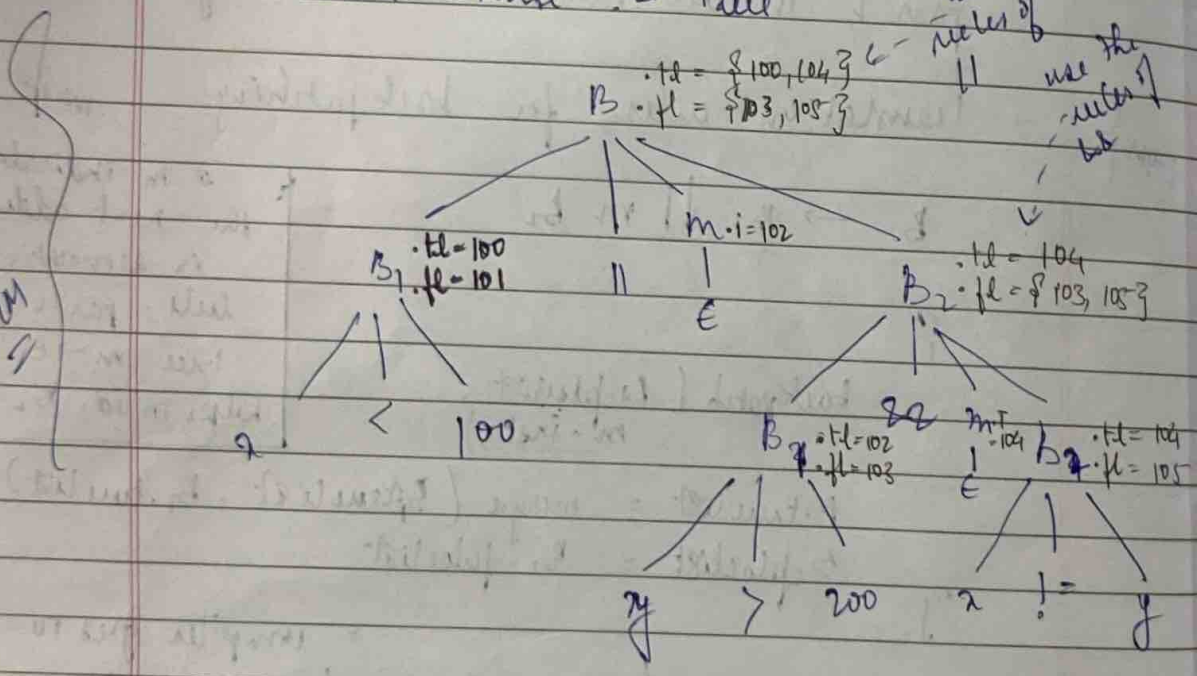
}

B → ! B_1
{
B.truelist = B_1.falselist
B.falselist = B_1.truelist
}

B → (B_1)
{
B.truelist := B_1.truelist
B.falselist = B_1.falselist
}

M → ϵ
M.instr = next instr

→ Annotated Parse Tree



Q. Base questions: unification algorithm
activation record.