Chapter - 5

26.6.2024
Zinia ma'am

Syntax Directed Translation

Syntax ⟶ construction (Ex : Sub (S) + verb (V) + obj (O). ⟶ syntax)

Semantic ⟶ meaning

I eat rice.

↓ translation ⟶ to understand the meaning

आणि जाऊ शकते

Syntax Analyzer ⟶ Semantic analyzer

(Parse tree) (Parse tree with additional info that has a meaning)

SDD (Syntax Directed Definition)

⟶ CFG with attributes & rules

$E \rightarrow E + T$

E.type

↓

Example of attribute

X ⟶ symbol

a ⟶ one of X's attribute

⟶ X.a

* attributes may be number, type, table reference, values for instance

Ex : S_n ⟶ SVO ⟶ O.eng, V.eng, S.eng

S_n ⟶ S_n.eng

English sentence

I eat rice

↓ ↓ ↓

$S_n.eng = S.eng \,||\, V.eng \,||\, O.eng$

by concatenating

$S_n \longrightarrow S\ V\ O$

S_n ↓ bangla

S ↓ S.bang

V ↓ V.bang

O ↓ O.bang

S_n · bang

আমি ভাত খাই

$S_n \cdot bang = S.bang \,||\, O.bang \,||\, V.bang$

↳ This rule is called semantic rules.

attribute for non-term :

→ synthesized
→ inherited

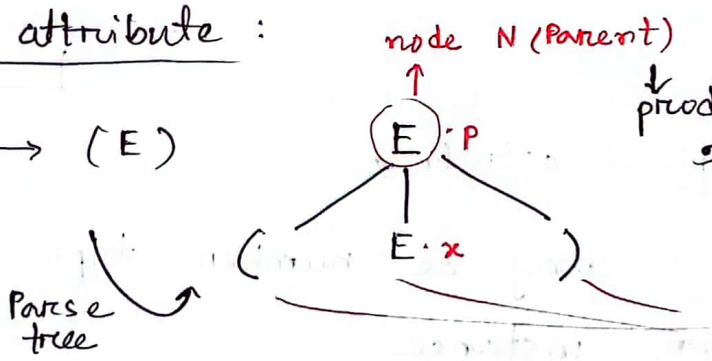
terminal

E → (E)

↳ non-terminal

Synthesized attribute :

E → (E)

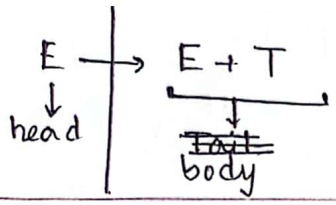


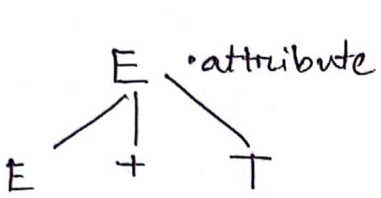
production এর head এ থাকবে

child থেকে নেয়া node N নিজের কোন attribute value (here val) পায়, then that's synthesized attribute

$E \cdot p = E \cdot x + E \cdot p$

↓ parent এর ↓ child এর ↓ নিজের

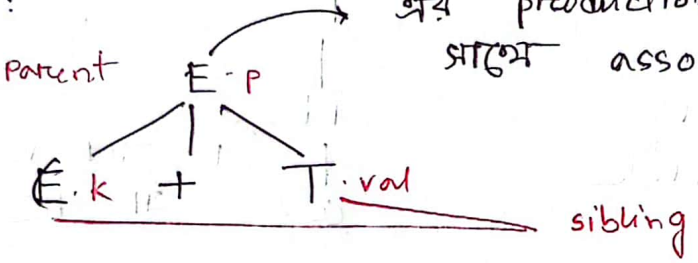




will be synthesized if it's from itself or it's children.

Inherited :

Ex :

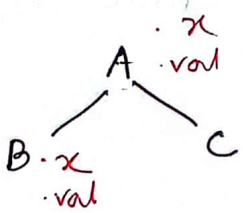


এর production rule যখন তার parent এর সাথে associated.

$$E.k = E.p + T.val$$

Ex :

$A \longrightarrow BC$



semantic rule :

$B.val = B.x$

↓

B, production এর body তে আছে ;
and নিজের থেকে attribute নিচ্ছে ,

↳ inherited attr

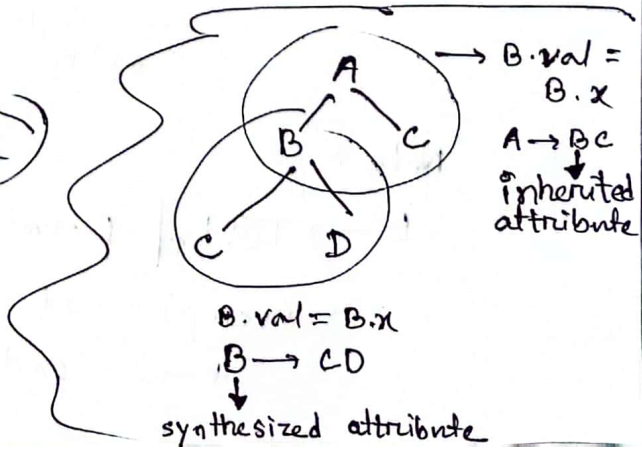
semantic rule :

$A.val = A.x$

↓

A, head এ আছে .
নিজের থেকে att. নিচ্ছে ,

↳ synthesized attribute



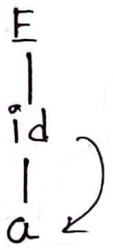
attribute for terminal :

$E \longrightarrow id$

- terminal এর attribute is always synthesized which comes from lexical analyzer. (lexical value)

$E \longrightarrow id$

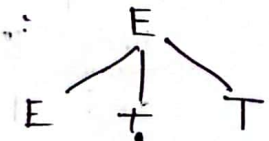
id . lex val
attribute



lex থেকে value না পাই

- There are no semantic rules in SDD itself for computing the value of an attribute for terminal.

Ex :



(terminal)

$E \cdot val = E \cdot val + T \cdot val$

Ex : $E \longrightarrow E@T$ | $E \cdot val = E \cdot val + T \cdot val$

Production rule

Semantic rule

SDD

Assume, This is addition sign

meaning, আমরা semantic rule এ লিখি

Note :

$E \longrightarrow T@E_1 \mid E \cdot val = E_1 \cdot val + T \cdot val$

- Subscript add করার to understand which E is in head and which E is in body.

Ex 5.1 :

Terminal → 6 ∄

Non - " → 4 ∄ (L, E, T, F)

L → E n (n → newline) → L.val = E.val

F → (E) → F.val = E.val

F → digit → F.val = digit.lexval

attribute → val, lexval

L.val → synthesized

E.val →

T.val →

F.val →

all are synthesized attributes → S-attributed SDD

↓

all are in the head of the production rule → synthesized.

Attribute Grammar :

Just attribute value নিয়েই কাজ করছি, calculation করে রেখে দিচ্ছি ⟶ attribute grammar

Side effects ⟶ symbol table এ value update, printing something

⟶ Not a attribute grammar

- Ex 5.1 ⟶ attribute grammar.

Evaluating an SDD at the Nodes of a Parse tree

⟶ প্রতিটা node এর attribute value as additional info in the parse tree

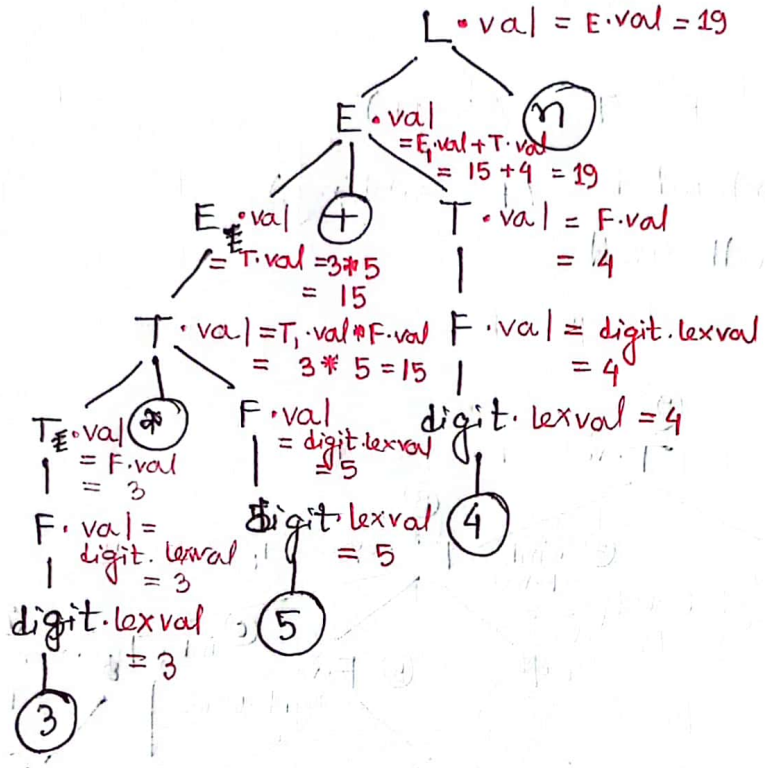
অনুবাদ ⟶ ANNOTATED PARSE TREE

how to evaluate (in which order)?

Ex : 5.1 ⟶ bottom to up evaluate করবো (Post order traversal)

Ex : 5.2

Annotated Parse tree for 3*5+4n (From Ex : 5.1)



Circular Dependency :

SDD with inherited & synthesized attribute.

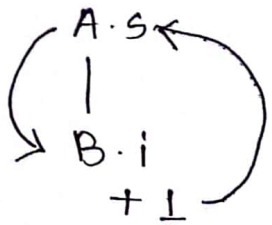
$A \rightarrow B$

$A.s = B.i$ (B is child) — synthesized

$B.i = A.s + 1$ — inherited (A is parent)

In this case,

* we need to redesign the SDD.
* we need to know evaluation order here.



Ex : 5.3

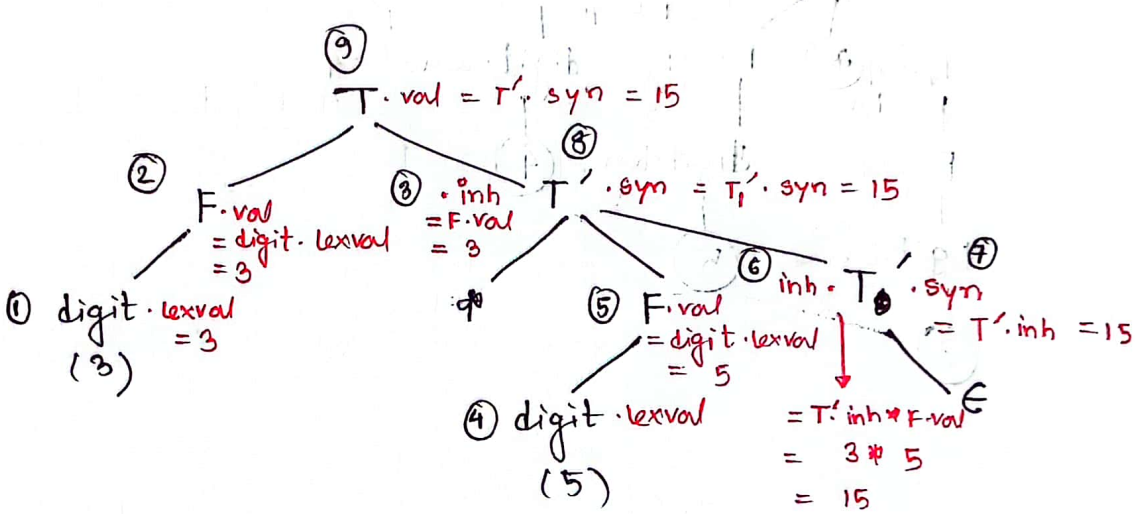
Annotated Parse tree for 3 * 5

Non term → T, F, T′ (3 ℸ)

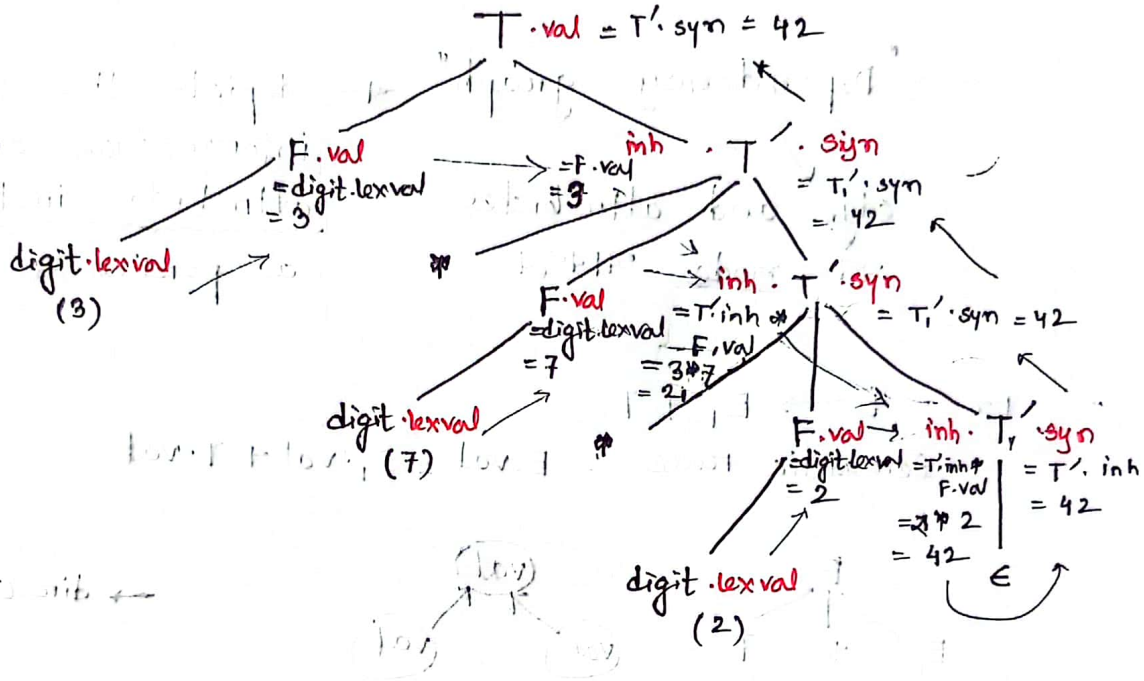
T′.inh ⟶ inherited (F sibling)

T.val → synthesized

Two types of attributes exist. so dependency check is a must.



3 * 7 * 2 :


T .val = T'.syn = 42
F.val
=digit.lexval
=3
digit.lexval
(3)
=F.val
=3
inh
T'
.syn
= T1'.syn
= 42
F.val
=digit.lexval
=7
digit.lexval
(7)
inh.T'
=T'.inh *
F.val
=3*7
=21
= T1'.syn = 42
F.val
=digit.lexval
= 2
digit.lexval
(2)
inh.T1'
=T'.inh *
F.val
=21 * 2
= 42
.syn
= T'.inh
= 42
ε

27.06.2024

Evaluation Order of SDD's :

↳ কোন value আগে calculate করব ← to understand this.

→ "Dependency graph" ← depicts the flow of information among the attribute instances in a particular parse tree

↙ edge and attributes এর node থাকবে

Ex : $E \rightarrow E_1 + T$

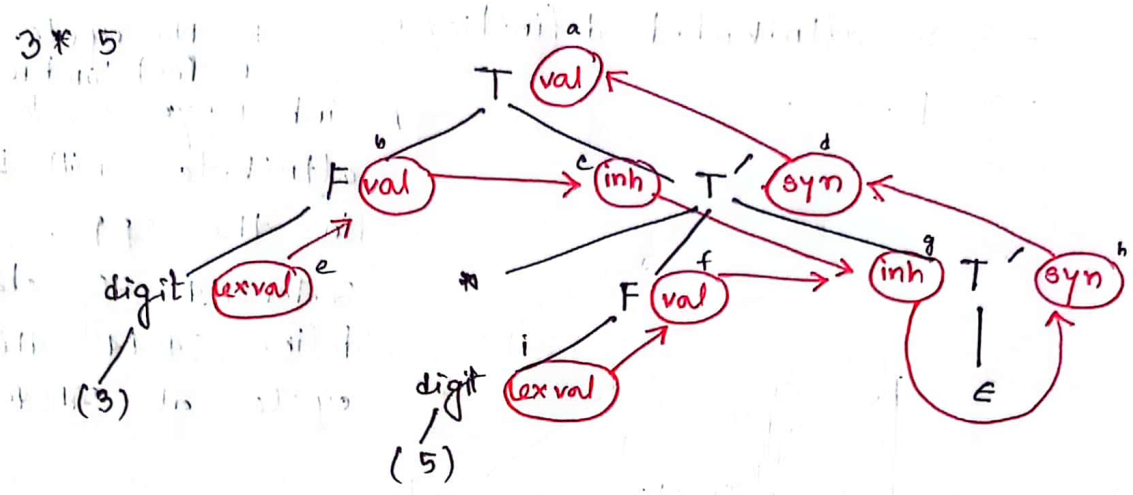
Semantic rule : $E.val = E_1.val + T.val$

→ directed graph

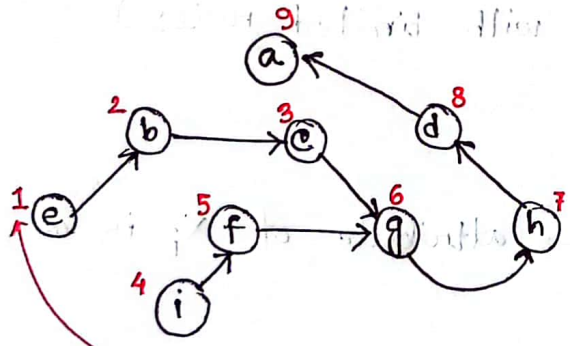
$X \rightarrow AB$, semantic rule : $B.c = 2 \times (X.a)$

Note : If there's any cycle in the graph, then we need to change the semantic rules.

Ex : 5.5 :



To sort the nodes → Topological sort ← we need to identify which node isn't dependent on anyone.



Note :
↳ e b i f c g h d a
This is also correct order. So, we can draw more than one dependency graph using topological sort.

In order যার 0 থাকে নিবো

	1	2	3	4	5	6	7	8	9
Topological sort :	e	b	c	i	f	g	h	d	a

↳ after sorting, from the dependency graph. we can easily evaluate the annotated parse tree.

Classes of SDD :

→ S - attributed definitions ⟶ No cycle
⟶ Post ordering

→ L - " " ⟶ inh + syn → both attribute will be there in the SDD.

এমনভাবে এই class define করব যাতে কোনো cycle না থাকে ,

L - attributed definition :

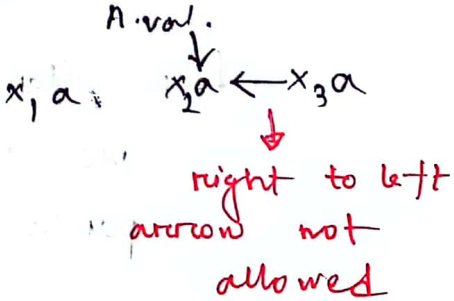
* dependency graph edges can go from left to right, but not from right to left.

* Syn + (inh attributes with limited rules)

Rules :

$A \longrightarrow X_1 X_2 X_3$ ⟶ attribute of X_i is a.

- $X_1 \cdot a = A \cdot val$ ✓
- $X_3 \cdot a = A \cdot val + X_1 \cdot a + X_2 \cdot a$ ✓
- $X_2 \cdot a = A \cdot val + X_3 \cdot a$ ✗ →



- $X_3 \cdot a = A \cdot val + X_3 \cdot val$ ✓

inh → from sibling, head or itself
left এ থাকতে হবে to be L-attributed

Example → S-attributed SDD
↳ all grammar symbols have only synthesized attribute.
(child থেকে calculate হচ্ছে values)

L-attributed SDD
↳ all S-attributed SDD are L-attributed SDD because they both support synthesized attributes

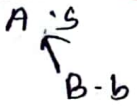
Example 5.8 → Not an S-attributed SDD
(inh + syn)

L-attributed SDD

Ex: ② $T' \rightarrow *FT_1'$ $\quad T_1'.inh = T'.inh \times F.val$
F, T_1 এর left এ
T: head
F: sibling (left এ আছে)

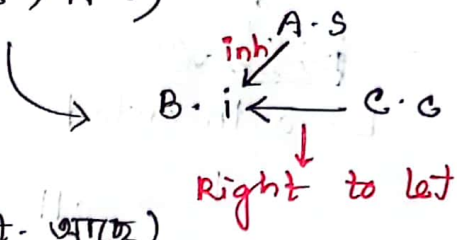
① $T \rightarrow FT'$ $\quad T'.inh = F.val$
left এ
left এ sibling

Example 5.9 →



$A \rightarrow BC \qquad A.s = B.b$ (Syn)

$B.i = f(C.c, A.s)$



So, not an S-attributed (inh att. আছে)

not " L- " SDD (right sibling থেকে value নিচ্ছে)

3.07.2024

Semantic rule with controlled side effects :

SDD of desk calc → print a result
↳ side effect

" " semantic analyzer → enter type of ID into symbol table

Attribute grammar → No side effect
Translation scheme → left-right evaluation

incidental side effect → root node এ side effect যার জন্য কোনো problem হয় না।

L → En print (E.val)

production → side effect → dummy node → ঐ production এর head এর সাথে associate করবো।

5.10 SDD for simple type declarations :

int a
int a, b, c
T L

float a
float a, b, c
T L

D → T L
declaration type list

T → int
T → float

L → L1, id
L → id

⑧ SDD given → semantic rule generate করবো

$D \rightarrow TL$

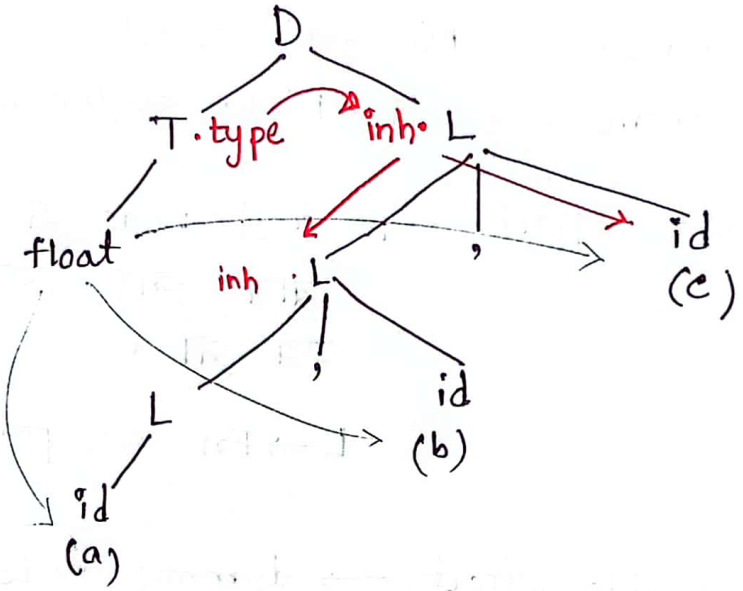
→ int L

→ int L, id

→ int L, id, id

→ int id, id, id → int a, b, c

float a, b, c :



* a, b, c এর type another branch এ (float)

↳ inherited att. ব্যবহার L এ

L.inh = T.type

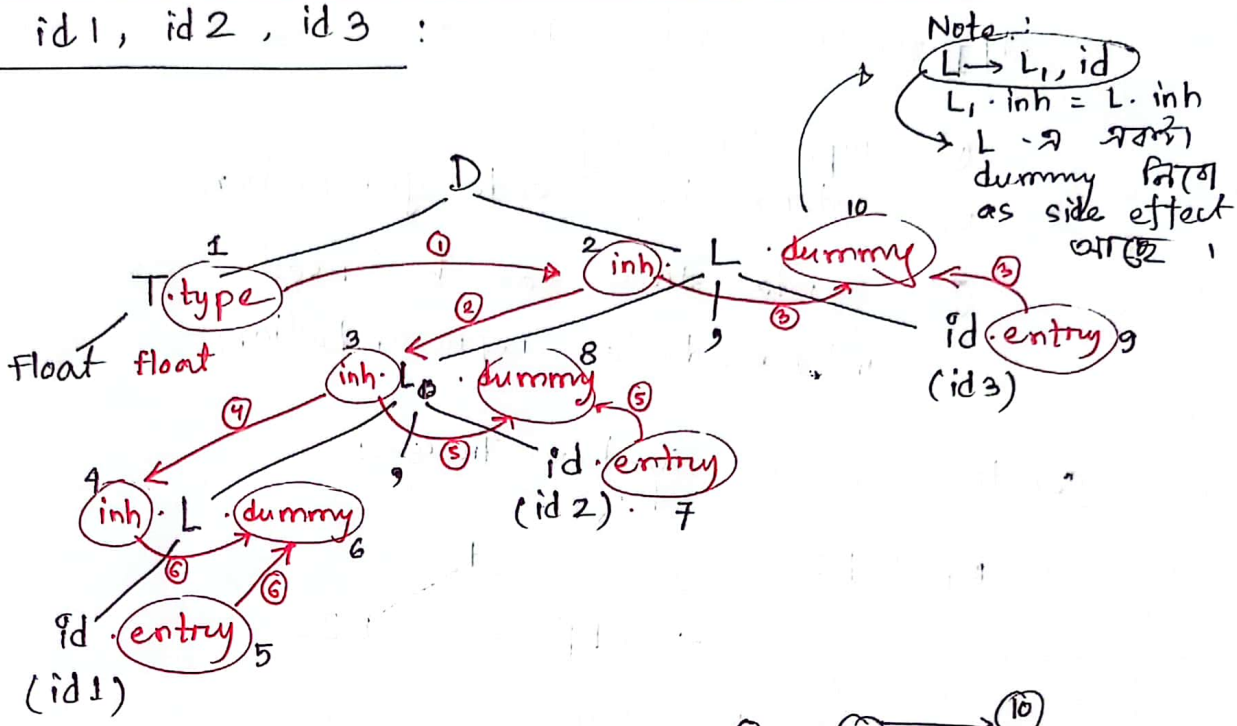
* $L_1.inh = L.inh$ (L head)

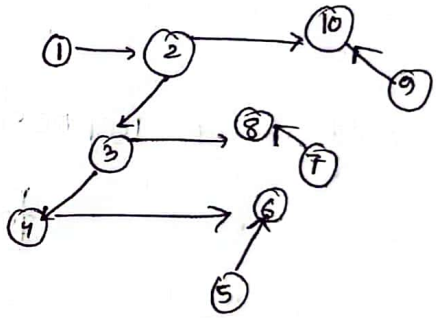
* (id)c এর type symbol T. entry করতে,

addType (id.entry, L.inh)

inherited attr. সাবধান ⟶ dependency graph

float id1, id2, id3 :





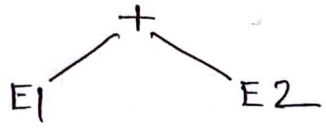
we can make annotated parse tree from this.

Application of SDT :

1. Type checking
2. intermediate code generation
3. Syntax tree can be used as intermediate representation

construction of Syntax Trees :

$E_1 + E_2$



node ⟶ represented as record

op. field	other field

leaf ⟶ value lex. analyzer fn(a

op. field	val. field

Interior node with two children

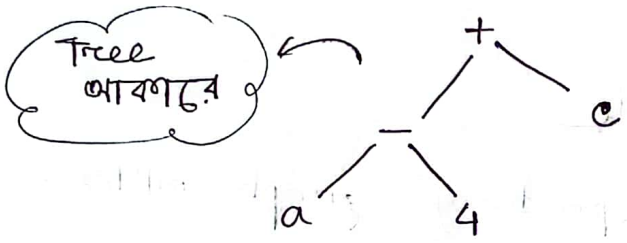
op. field	left child	right child

Ex : 5.11

- node creation ⟶ 'new' ⟶ side effect
- S-attributed definition
- Ex : a + b + 6 − 7 + c
- dependency graph লাগবে না (inh attr. নাই) ↳ S-attributed

Syntax tree for a − 4 + c :

Note: left এসোসিয়েটিভ operation আগে হবে



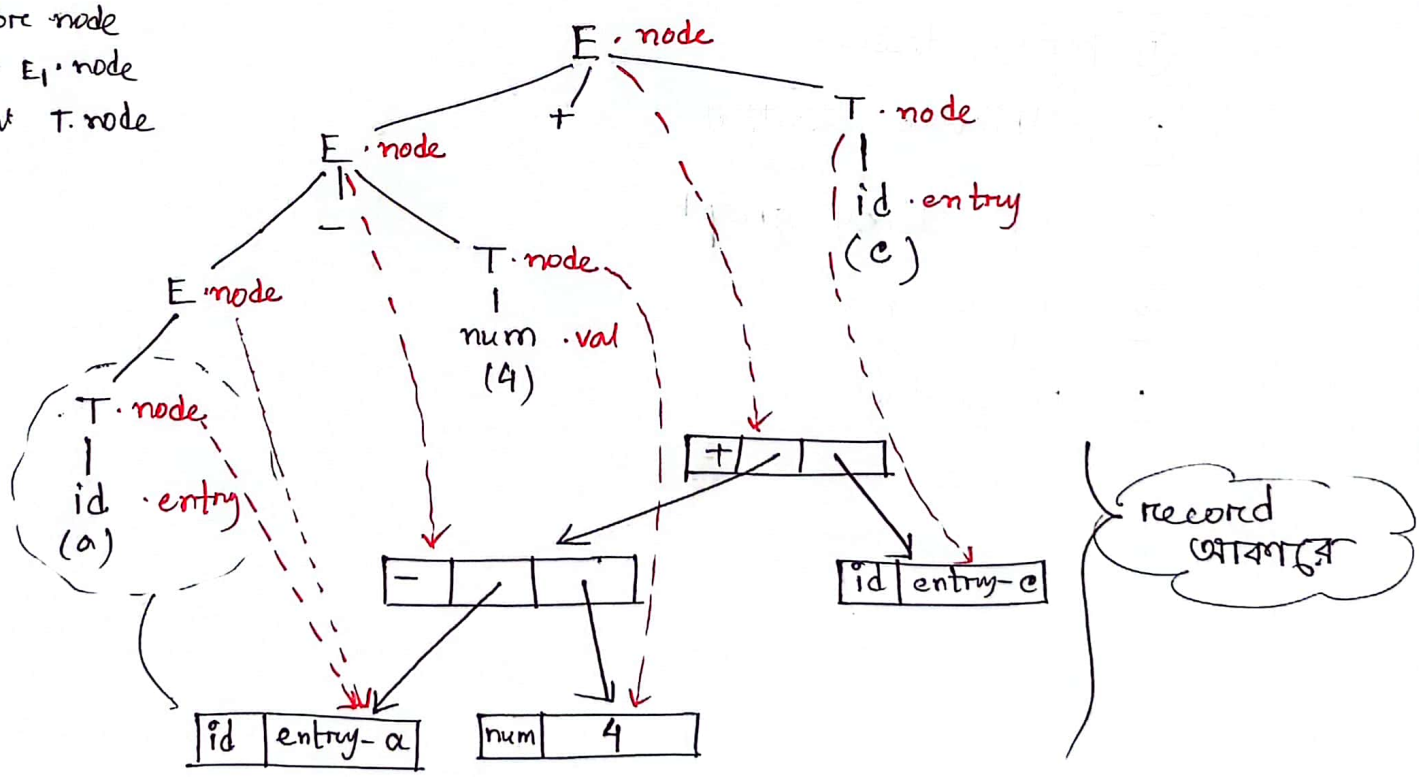
Note:

⟶ parse tree

- - - ⟶ for node reference / node pointer

⟶ ⟶ syntax tree

E ⟶ interior node
↳ left E_1 · node
⟶ right T · node



parallelly this steps to be have written

```
P1 = new Leaf ( id , entry-a)
P2 = new Leaf ( num , 4 )
P3 = new Node ( '-' , P1 , P2 )
P4 = n Leaf (id , entry-c)
P5 = new Node ('+' , P3 , P4)
```

5.12 –

(syn + inh) attribute

additional thing ⟶ dependency graph লাগবে.

Steps :

① Parse tree

② attribute বসাবো

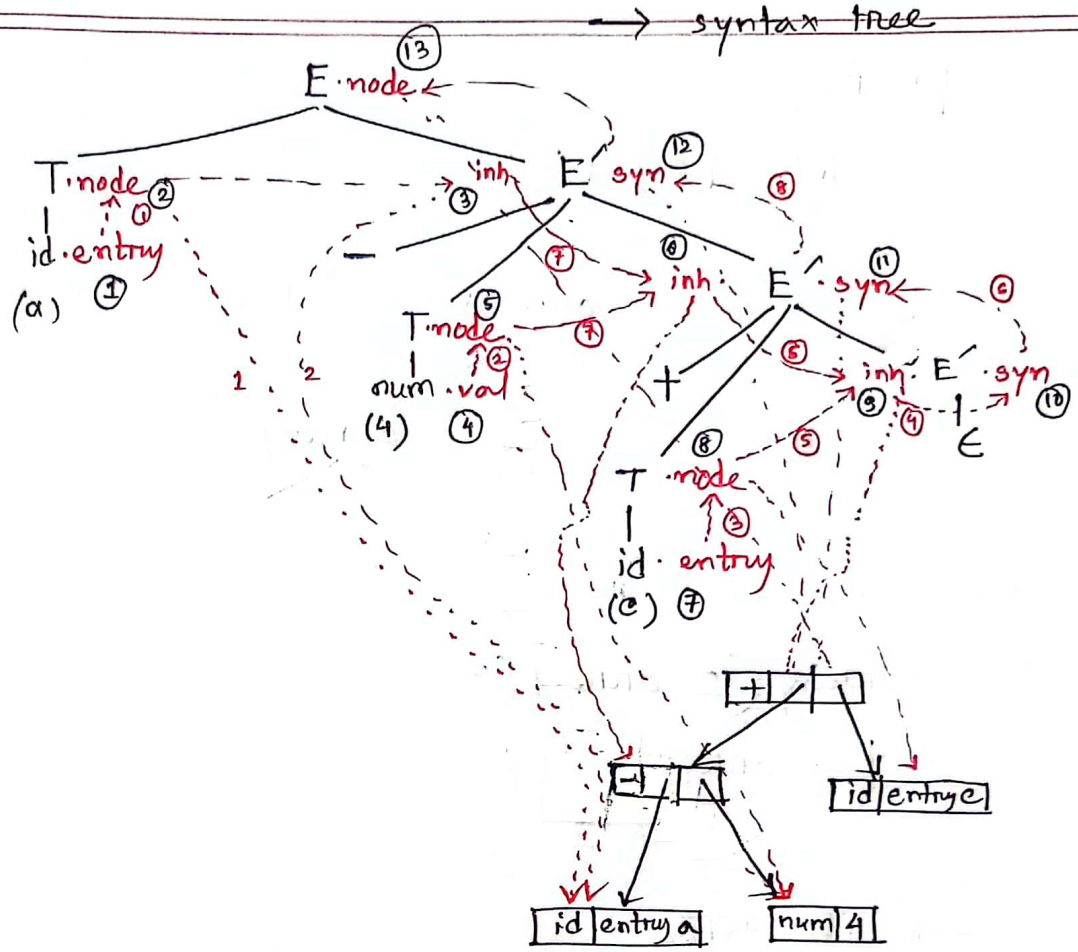
③ dependency graph

④ Topological

⑤ Evaluation

a - 4 + c :

—— parse tree
---> dependency graph
•••> node ref
——> syntax tree

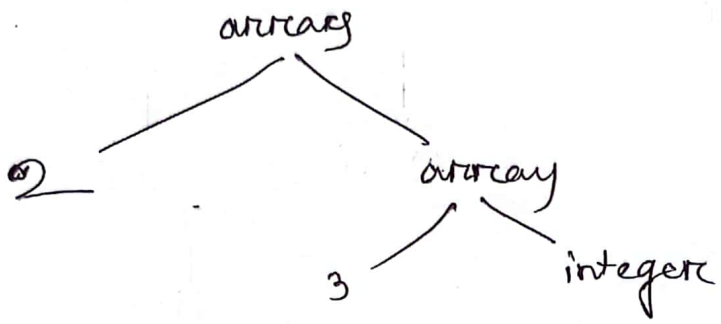


Steps of node creation :

1. P1 = leaf(id , entry - a)
2. P2 = leaf (num , 4)
3. P3 = Node (- , P1 , P2)
4. leaf (id , entry - c)
5. Node (+ , P3 , P4)

5.13

int [2] [3]



array (2, array (3, integer))

↳ array of 2 arrays of 3 integers

→ inh आएगा

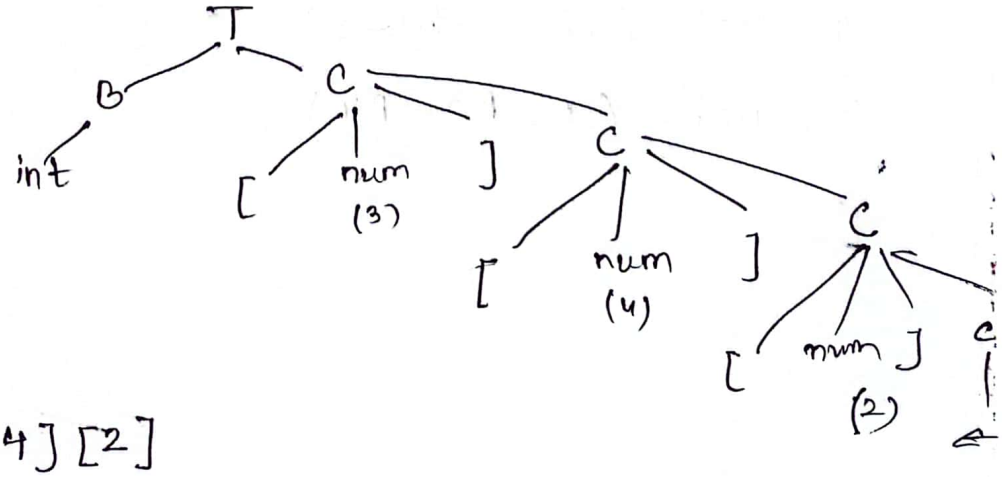
→ side effect आएगा

T → Type

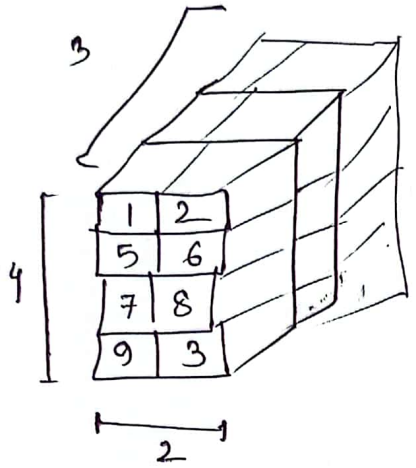
B → basic type

C → component

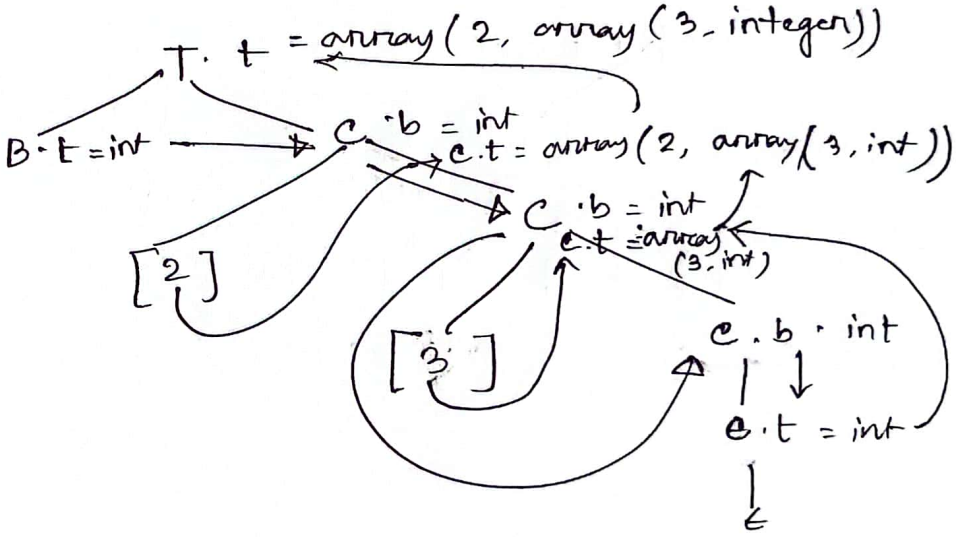
[num] → size



int [3] [4] [2]


3
1
2
5
6
7
8
9
3
4
2

for int [2][3] :


T. t = array(2, array(3, integer))
B.t = int
C.b = int
C.t = array(2, array(3, int))
C.b = int
C.t = array
(3, int)
[2]
[3]
C.b. int
C.t = int

4.7.2024

Syntax Directed Translation Scheme :

$E \rightarrow E_1 + T$ — Production (4+5)

$E.code = E_1.code \,||\, T.code \,||\, '+'$ — semantic rule (45+)

Specification → SDD

Implementation → SDT

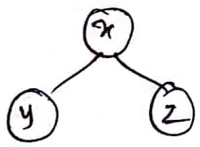
$E \rightarrow E_1 + T \; \{ print\ '+' \}$

semantic action within production body.

SDT →

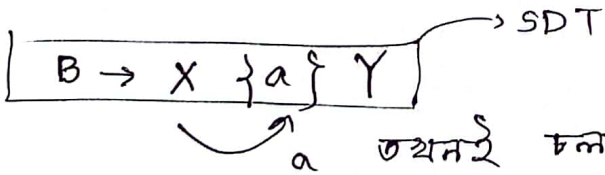
① parse tree

② preorder on left to right depth order



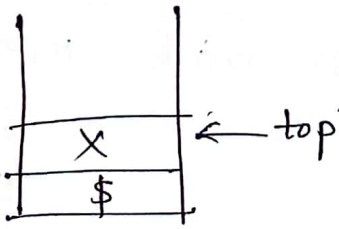
Preorder : x y z

Post " : y z x



a তখনই চলবে যখন X - terminal পেয়ে যাব

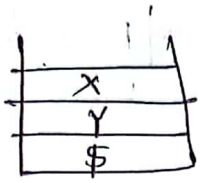
more precisely,



X top এ থাকলে action a execute হবে।

Post order = leftmost derivation

$B \rightarrow XY$



> reverse order এ stack এ যাবে।

Postfix scheme ⟶ synthesized attr. all

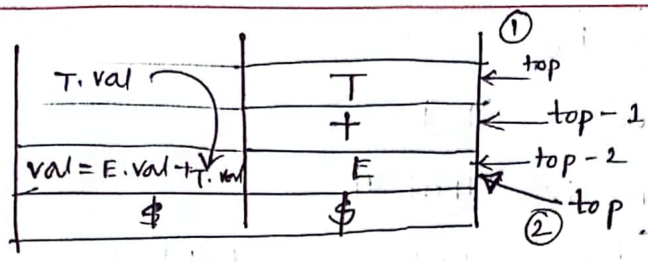
Parser stack implementation of Postfix SDT's :

⟶ LR parsing
- ⟶ rightmost derivation
- ⟶ left to right scan

$L \rightarrow En$
$\rightarrow E+Tn$
$\rightarrow E+Fn$
$\rightarrow E+digit\ n$
$\rightarrow T+digit\ n$
$\rightarrow F+digit\ n$
$\rightarrow digit+digit\ n$

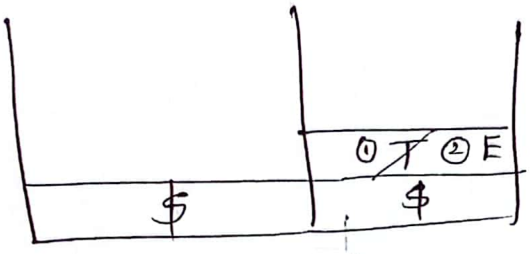
* shift
* Reduction ⟶ production এর head দিয়ে reduce করা

$E \rightarrow E + T$



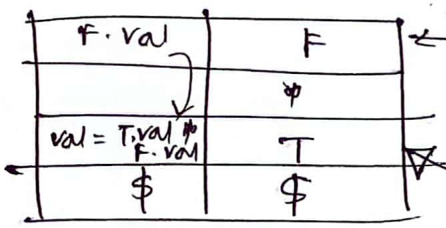
Now top - 2 = top

$E \rightarrow T$



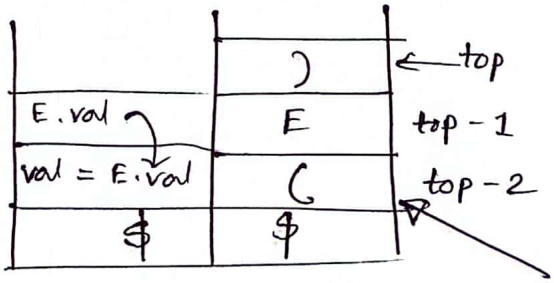
← nothing to do

$T \rightarrow T * F$



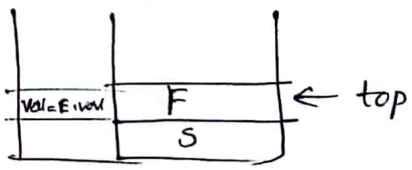
Now top

$F \rightarrow (E)$



Now, this is top

-) appeared on top of the stack
- Perform the semantic action ({ action })
- Reduce the production



as $F \rightarrow (E)$

Ex : 5.16

Problematic SDT

↳ Prefix বানাবো

(Ex : 4+5 ⟶ +45)

4*5 ⟶ *45

4*5 + 6