Prepared and Written By
RAZEEN AHMED

CSE-420

[COMPILER DESIGN]

HANDWRITTEN NOTE

Computer Science and Engineering
BRAC University

Github: github.com/razeen
LinkedIn: linkedin.com/in/razeenahmed

COMPONENTS OF COMPILER :-

Symbol Table :- It is a table the variable which gets initialized is stored

scope - the variable last stored.

LEXICAL ANALYSER :- variables = identifiers.

traverses through the entire

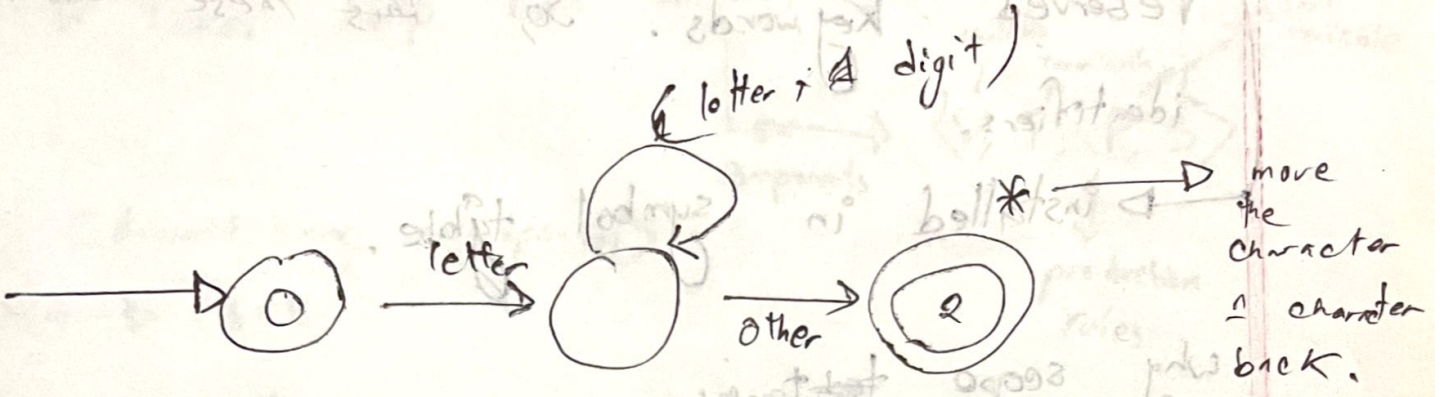
→ statements characters by characters until it finds a meaningful word.

→ task is to extract meaningful words.

→ It's need to know which set of characters are accepted.

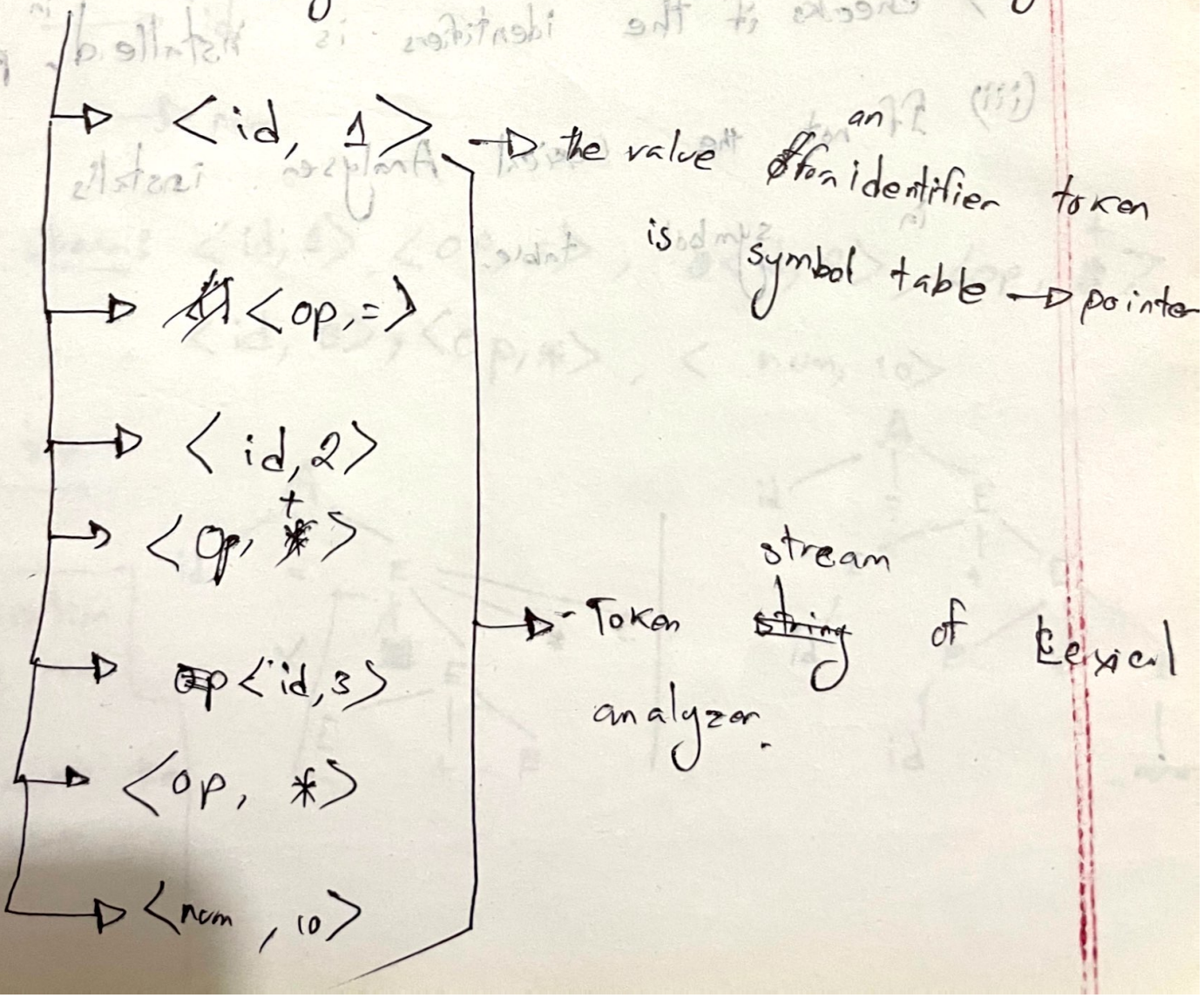
→ RE helps lexical analyzer to identify meaningful words.

* identifier have to start with a letter.



TOKEN : < name, value >

↳ general format token in lexical analyzer



if, for, int, float, except, while these all are reserved keywords. So, ~~this~~ these can't be identifiers.

→ installed in symbol table.

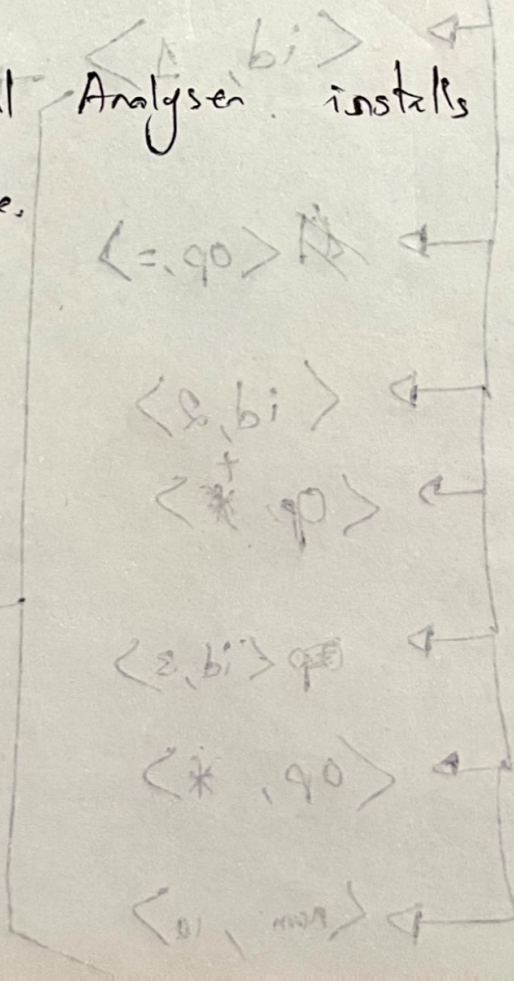
why scope ~~tack~~ tracks :-

Features :-

(i) Generates the token stream

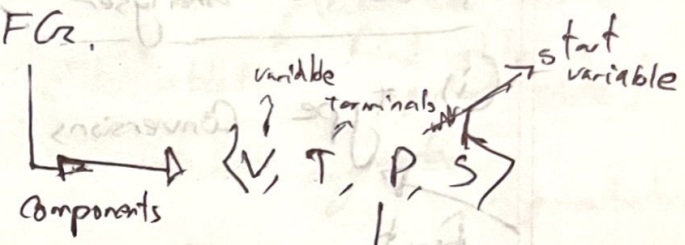
(ii) checks if the identifiers is installed in symbol table previously.

(iii) If not the Lexical Analyser installs the identifiers in symbol table.



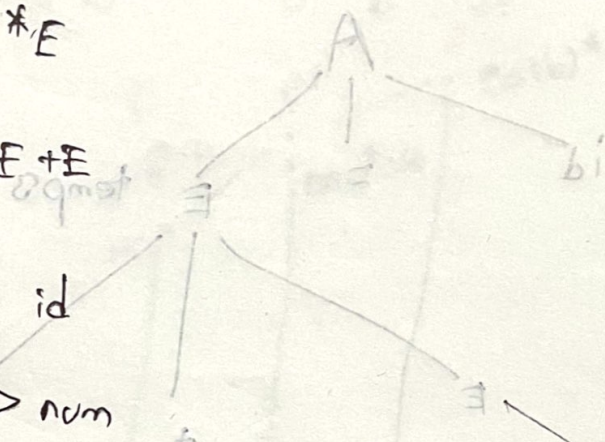
Syntax Analyzer :-

⟶ Takes help from CFGs.



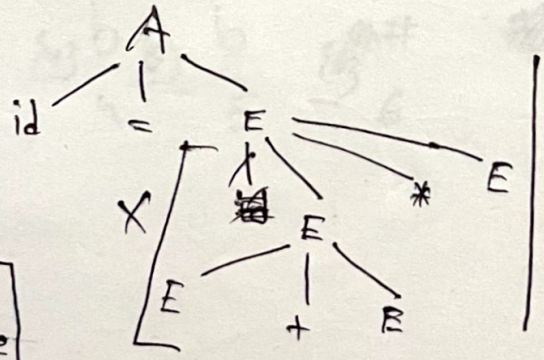
CFGs :- C doesn't have any operators

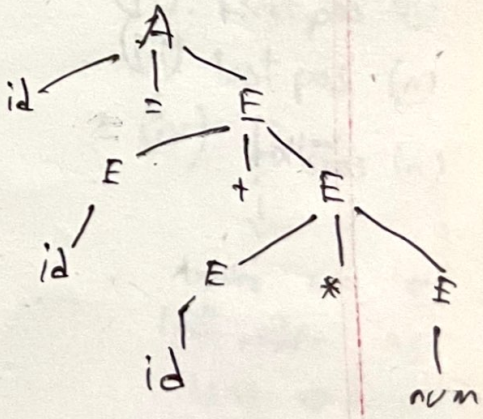
1. A ⟶ id = E
2. E ⟶ E * E
3. E ⟶ E + E
4. E ⟶ id
5. E ⟶ num



Token stream: ⟨id, 1⟩, ⟨op, =⟩, ⟨id, 2⟩, ⟨op, +⟩ ⟨id, 3⟩, ⟨op, *⟩, ⟨num, 10⟩

Parse tree :-





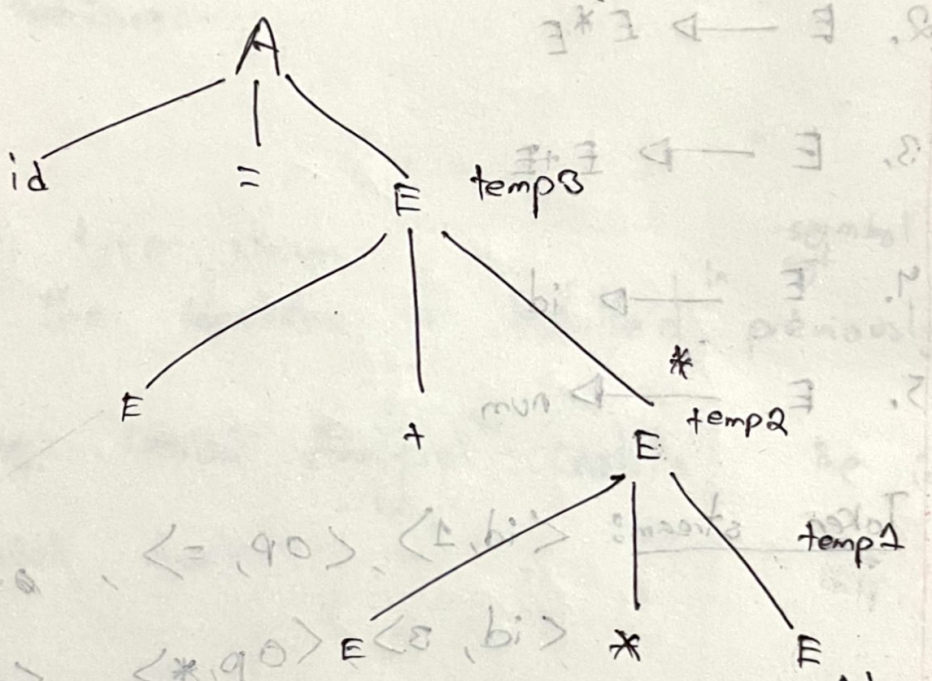
multiplication should come last to make the precedence highest

Sematic

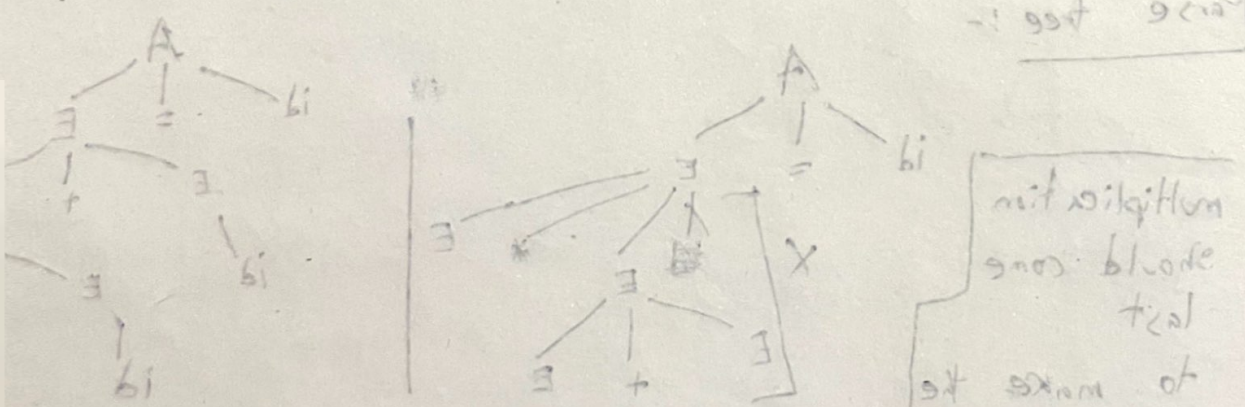
Semantic analyser

(i) Type Conversions (ii) Type checking

first executes the most bottom level of parse tree.



int to float



Razeen

05/10/23

CONVERSION OF RE to DFA

(a|b)*abb

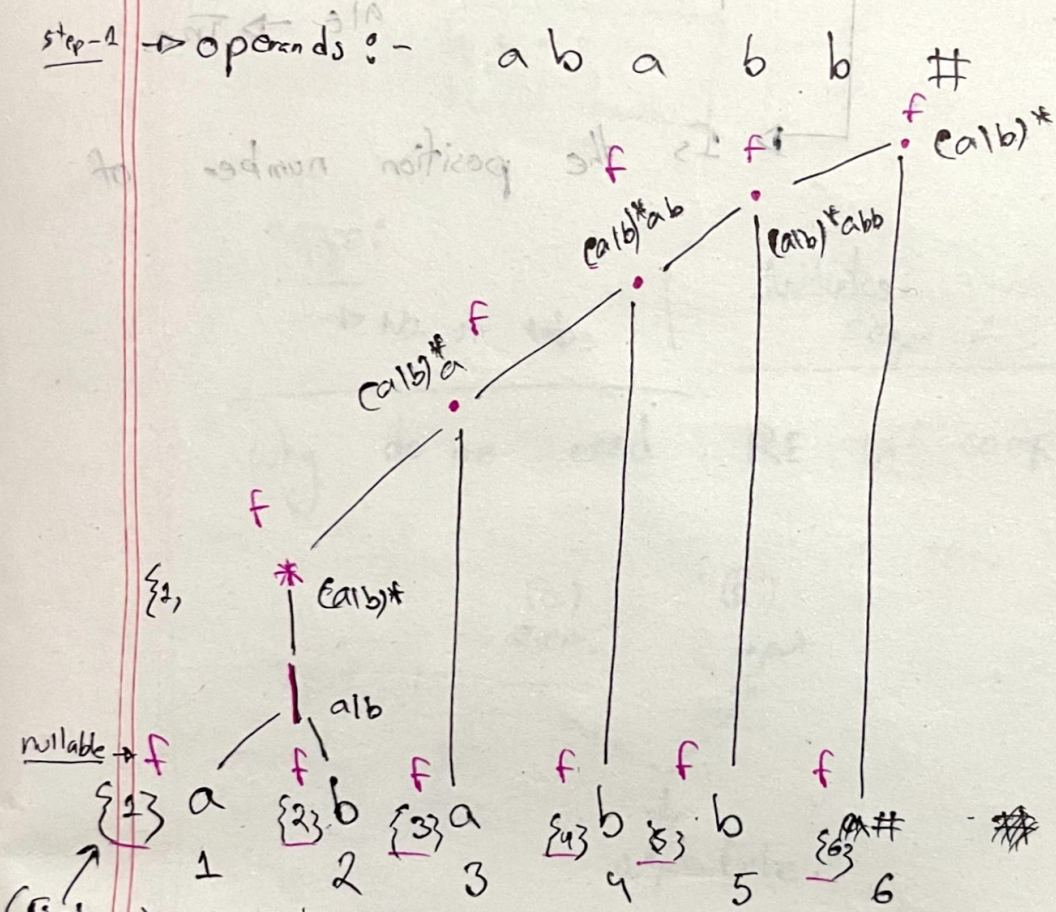
(a|b)*abb#

→ RE augmented / augmentation

(so we can make final state a important state)

SYNTAX TREE :-

step-1 → operands :- a b a b b #



operators

*, +	→ star closure
.	→ cat (concatenate)
|	→ or

precedence

highest precedence

() → not operator (used for grouping purpose)

Annotate syntax tree

(i) nullable (n) → node

(ii) firstpos (n)

(iii) lastpos (n)

(iv) followpos (n)

↓

takes non ε leaf nodes as leaf nodes.

a ε a b b #
1 2 3 4 5

2X (skip it)

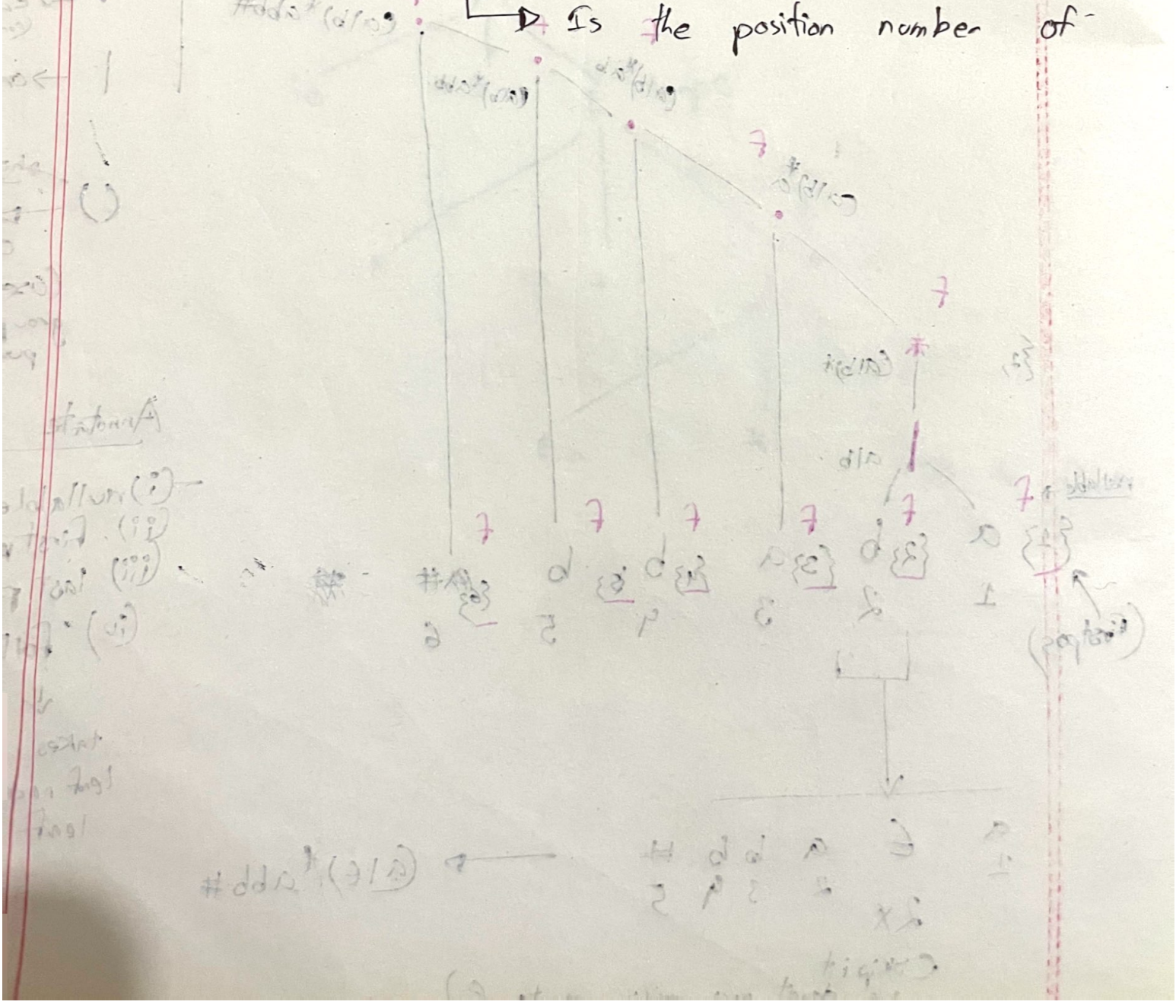
→ (a|ε)*abb#

Annotate syntax tree :

(i) nullable (n) → all the leaf node will be
if char :- False
if ∈ :- True

(ii) firstpos (n) → a|b → false
a|∈ → True

(iii) ~~for~~ lastpos (n) → Is the position number of



12/8/23

PRACTICE SESSION (CONVERSION TO DFA)

$a^+(bc)?\ |\ b^*$

$\Sigma^+ = \Sigma^* - \epsilon$

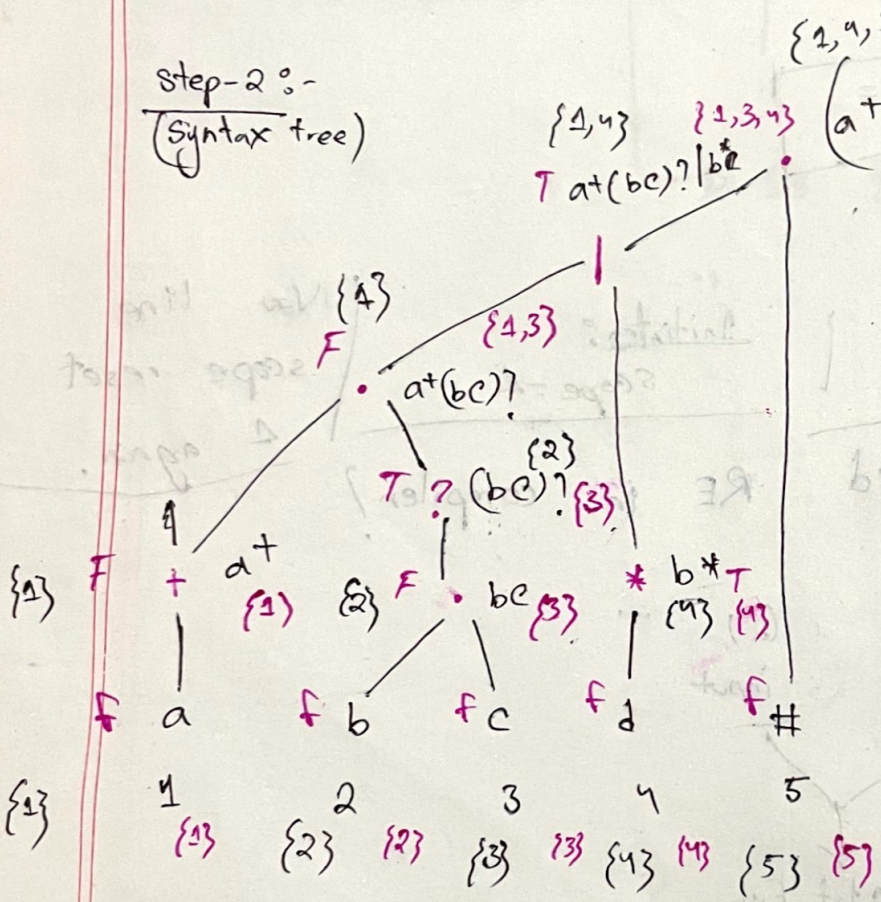
$r? = r|\epsilon$ | ? = operator comes once or doesn't come at all.

Step-1 :-

$(a^+(bc)?\ |\ b^*)\#$ (augmentation)

Step-2 :-

(Syntax tree)



{1,4,5} $(a^+(bc)?\ |\ b^*)\#$ {1,3,4,5}

F

if the left child is nullable

First pos → left (for the unary operator f.p. will be its childs f.p)

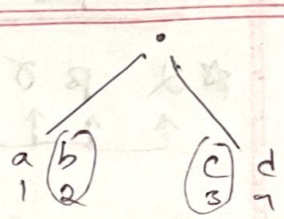
Right pos → Right (for the unary operator L.P. will be its childs lp.)

(if the right child is nullable)

(follow pos will only be on the leaf nodes)

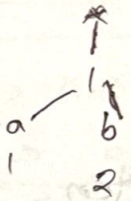
Symbols	Node (n)	followpos (n)
a	1	{1,2,5}
b	2	3
c	3	5
b	4	4, 5
#	5	∅

cat node



Star node

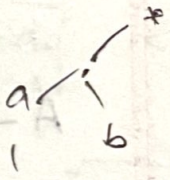
* (a|b)*



aabbaa

(ab)*

abab ab



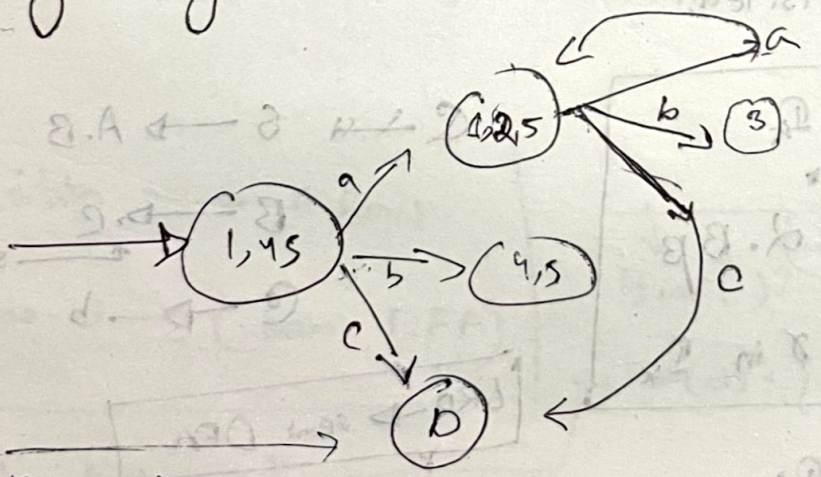
• → when there's two child

left child first pos is followed by right child's last post

(a + (be)? | b*)#

1 2 3 4 5

any string can start with 1, 4 5



Eve can't any string would start for the with 'c'!

Left to Right scan

Rightmost derivation

Rightmost

14/10/23

Simple Left Right (SLR) Parsing

* $\alpha \ \beta \ \gamma$ — strings of grammar symbols

LR Parsing Algorithm

input → $a_1 \ldots a_2 \ldots a_n \$$

$\$$: The input string ended.

* item

[dot : no one's been visited]

$A \rightarrow .XYZ$ — dot in the production body. So, it is an item

$A \rightarrow XYZ.$ — dot in the end reduced forms of the items.

. → It specifies the symbol upto which we have visited.

CLOSURE (I) = I_k

1. I is in I_k
2. I : $A \rightarrow \alpha . B\beta$

   $B \rightarrow . \gamma$ in I_k

I_0 → $S \rightarrow A.B$

$B \rightarrow .C$ — still needs to be visited

$C \rightarrow .b$

LR0 → semi DFA

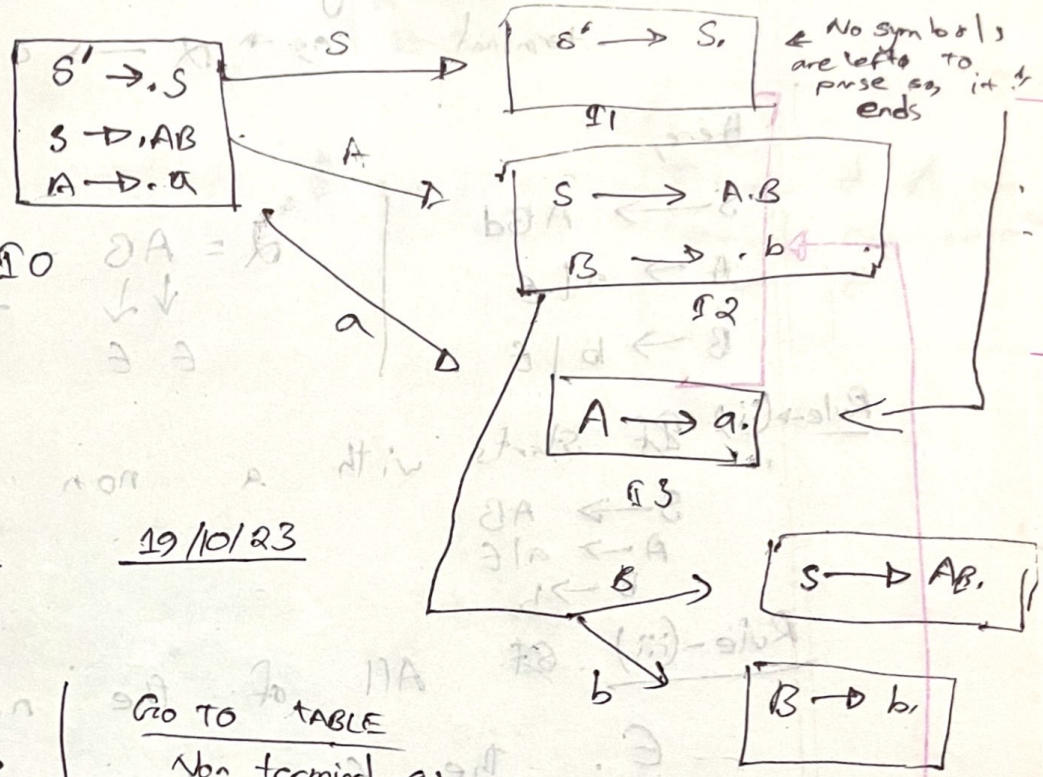
GOTO (I_k, X) = I_j

I : $A \rightarrow \alpha . X\beta$ —X→ I' : $A \rightarrow \alpha X . \beta$

Step-1:

Augmentation, introduce a new start state

0. $S' \rightarrow S$
1. $S \rightarrow AB$
2. $A \rightarrow a$
3. $B \rightarrow b$



SLR PARSING-2

19/10/23

Actions table
Only terminals as column

GO TO TABLE
Non terminal as column.

If
6 states → 6 Rows

Slide-13 → (semi DFA)

Functions (applied in CFG)

FIRST( ) — the first input combination string that can be generate from α)

FOLLOW( )

First(α) = {a, b

α = A B
↓ ↓
a b

S → AB
A → a | ε
B → b

F.

S → ABd
A → a | ε
B → b

α = ABd

First FIRST( ) :-

(i) If non terminal begins with a terminal first will be terminal e.g. → $\alpha \longrightarrow cB$

Here,

$S \longrightarrow ABd$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$\alpha = AB$

$\downarrow\downarrow$

$\epsilon \ \epsilon$

$\rightarrow$ First

FIRST = $\{a, b, \epsilon\}$

Rule→(ii) If starts with a non terminal

$S \longrightarrow AB$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b$

Rule-(iii) If All of the non terminal gives ϵ. the ϵ will also include in the first.

e.g.

Sentential form → each statement in the derivation process

e.g. S => AB => Ab = ab

'ε' can never be in FOLLOW( )

FOLLOW( ):-

FOLLOW(A) → { b }

S → AB
A → a
B → b

Rule(1):
FOLLOW(S) → { $ }
- start symbol follow will be $ sign.

∴ Q. ☆ Follow of non terminal can never be ε.

Rule(2):

S → ABd
A → a
B → b

FOLLOW(A) = { b }

S → ABde
A → a
B → b | ε

FOLLOW(A) = { b, d }

S → A Bde
α β

FOLLOW(A) = β first ( )
↓
FIRST(B)

Rule(3):

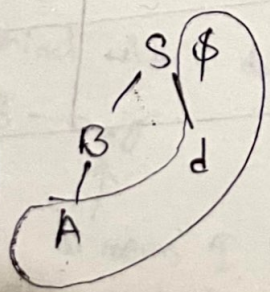
S → A
A → a

FOLLOW(A) = { $ }

look at the head of the production rule then the follow of that head is the one we need.

If β is ε, then follow of any non terminal

S → Bd
B → A
A → a



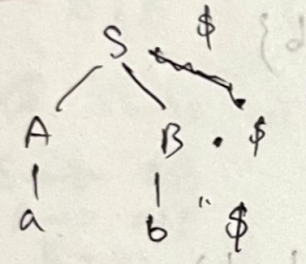
ACTION (TABLE)

(i) Shift → only terminal transition, states, will be written {13, 15}

(ii) reduce → check if there • at the end except S state 14, 15, 13

(iii) error →

(iv) accept

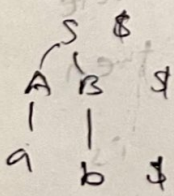


$ is the next look ahead symbol

0. S' → S
1. S → AB
2. A → a
3. B → b

if it is the state where the • is at the end of S

	Action (terminals)			GOTO (map the non terminal)		
	a	b	$	S	A	B
0				1	2	3
1			accept			
2						4
3		R2				
4			R1			
5			R3			



26/10/23

LR(1) Parser

Two types of LR parsers :→ (i) Simple LR parser (SLR
(ii) Complex LR parser /(CLR/LR)

Why do we use CR ~~parse~~ parser instead of SLR parser?

→ SLR(1) uses LR(0) automation $A \to \alpha.$ .. no lookahead symbol

LR(1) = LR(1) =

$A \to \alpha., a$ (core; a = lookahead symbol)

two functions:
(i) Closure functions
(ii) GOTO function.

CLOSURE:

Take an input item 'I' and gave us a state as an output

i_k CLOSURE(I) = i_k

(k is an integer)

RULES

(i) I is in i_k

(ii) If $I: A \to \alpha . B\beta$

look at the non-terminal after the dot

then we need to add the production rule of the non-terminal with a dot infront of the production body

$\#B \to .\gamma$

it means I haven't got visited all ~~rest~~ of the st... of the production body.

e.g. →

$S \overset{*}{\Rightarrow} \delta A a x$

$\Rightarrow \delta \alpha B \beta a x$

lookahead of B = FIRST(βa)

(13/10/23)

BOTTOM UP PARSING

F → id

↑ handle

Sentential form: each line of a derivation process

sentence: the last in the input string is sentence

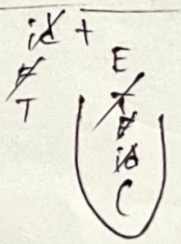
S => aBF
S => abBF
S => abbF
a => abbc

sentential form

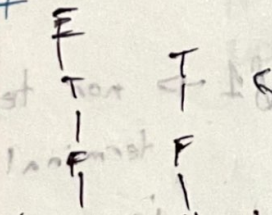
sentence

→ If we cannot find a handle within a portion which we have traversed then we have to traverse to add more & more input string.

E → T
T → F
F → id
E → E+T



#



In this situation Two production rules matches and is working as a handle. So, which one should I choose?

Ans:- Always choose the longest sequence from the stack.

If such a situation arrives where I can't understand whether the parser will reduce the current handle to the appropriate or prod head or ~~will it~~ shift one/more input symbo to look for a bigger match.

→ (Shift reduce conflict)

can't decide

(shift the input symbol

→ Shift reduce the input symbol)

PROBLEM TO SOLVE FROM BOTTOM UP PASSING :- (i) Shift-reduce conflict.
(ii) reduce-reduce conflict.

☞ In BOTTOM UP PARSING we are doing ~~right~~ rightmost derivation in reverse order.

★ If Something is working as a handle & and there are multiple production rules to reduce, ~~We can't choose~~ which matches with the handle. (We can't decide on a single handle to reduce)

⟶ Reduce-reduce conflict.

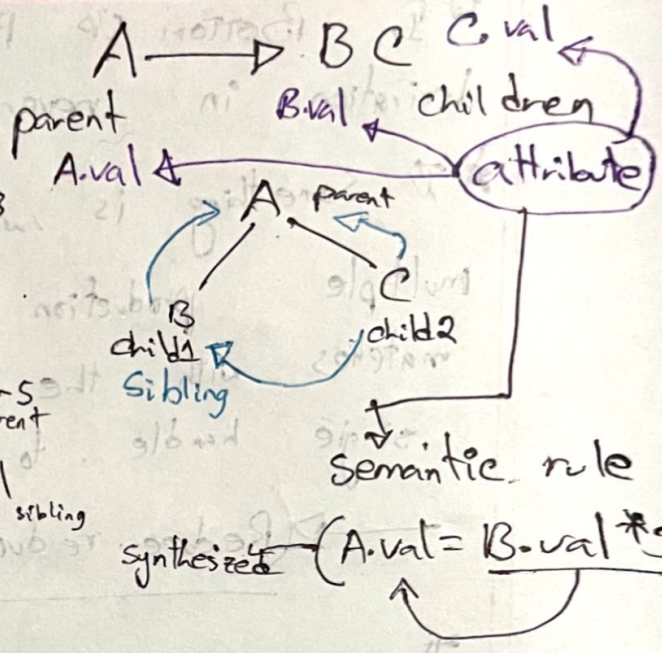
the process of changing or repeatedly eliminating handles is handle pruning.

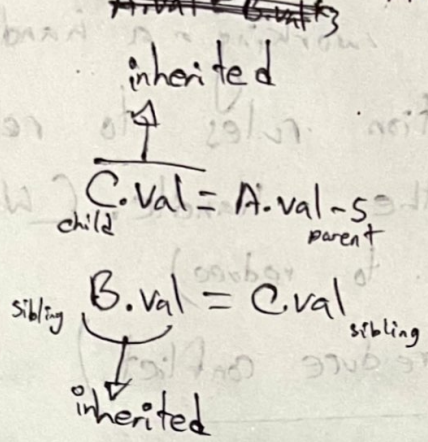
until the start symbol itself becomes a hand

16/11/23

SEMANTIC ANALYZER

(i) Type conversion
(ii) Type checking

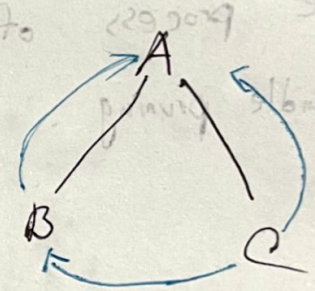




when we get a attribute from a child is synthesized.

A.val = A.val

can be both synthesized/inherited



SYNTAX DIRECTED DEFINITION:

PRODUCTION RULE	SEMANTIC RULE
(i) S → E	S.val = E.val
(ii) E → E1 + T	E.val = E.val + T.val
(iii) E → T	E.val = T.val
(iv) T → T1 * F	T.val = T1.val * F.val
(v) T → F	T.val = F.val
(vi) F → digit	F.val = digit.lexval

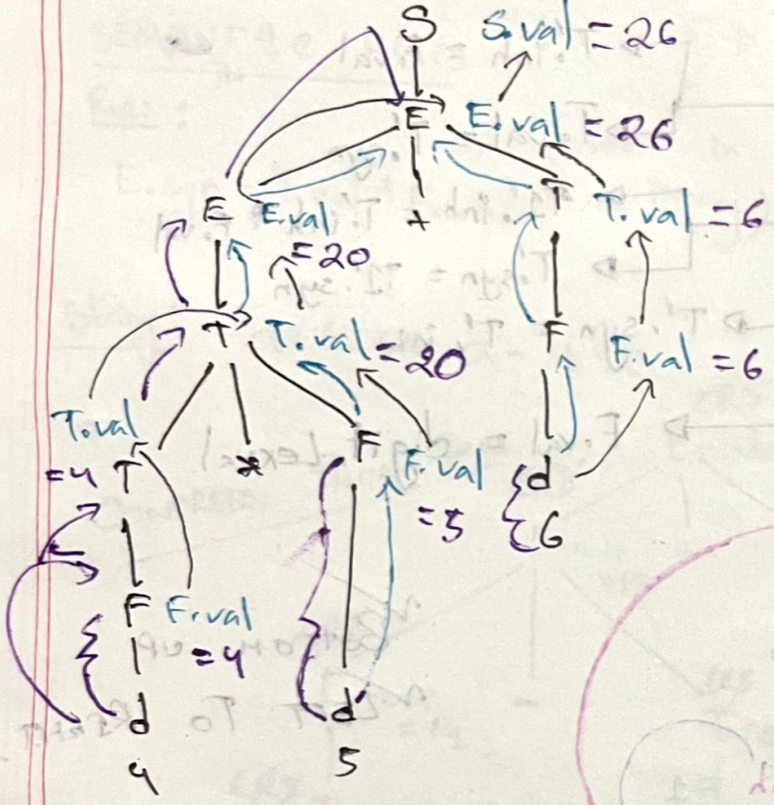
★ when a value comes from siblings or parent it is inherited

★ when a value comes from a child it is synthesized

"S Attributed SDD"

(Every attribute is synthesized) → come from ~~from~~ child

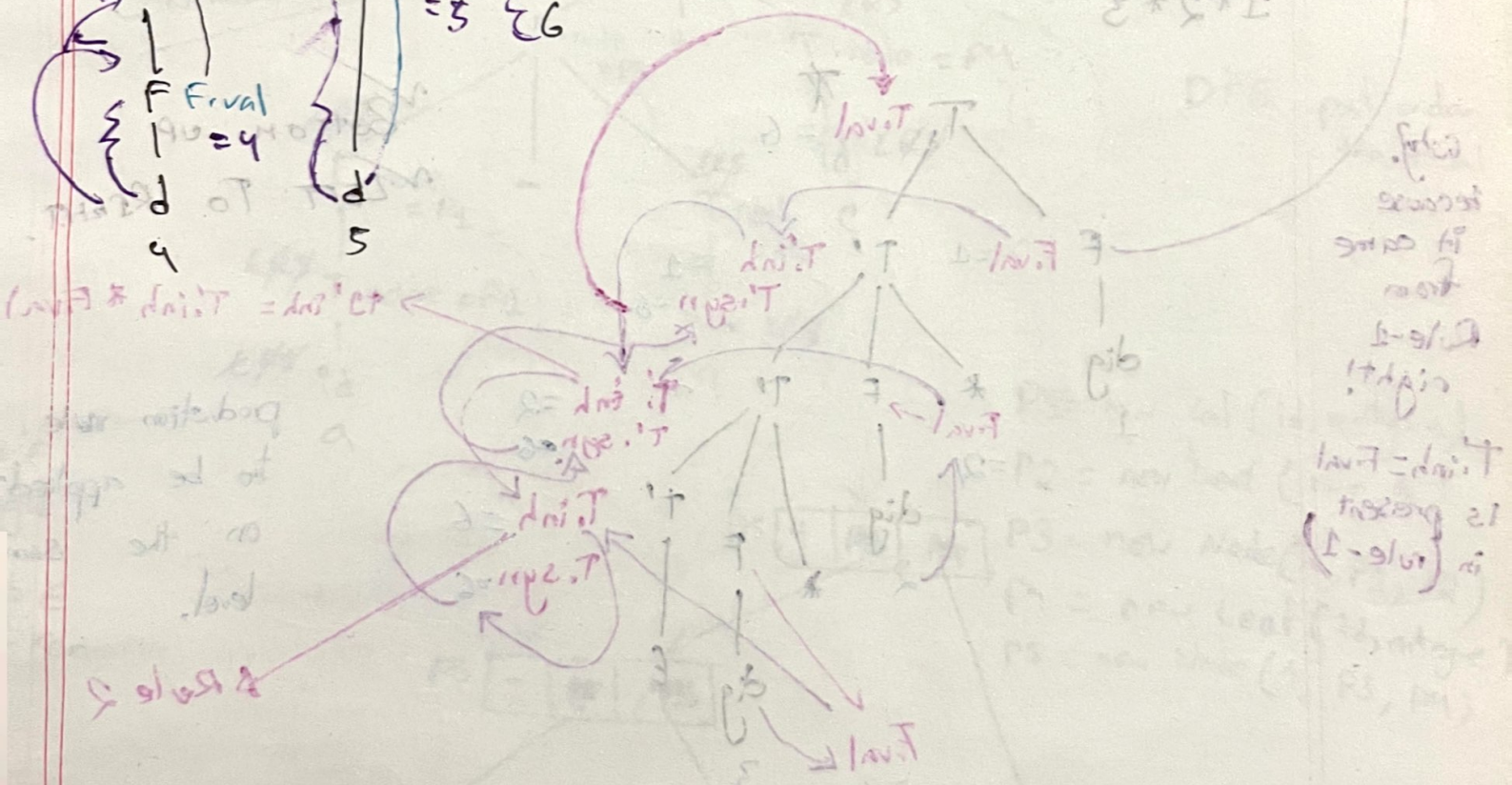
4* 5+6 DRAW The the "annotated parse tree"



"bottom up strategy"

=

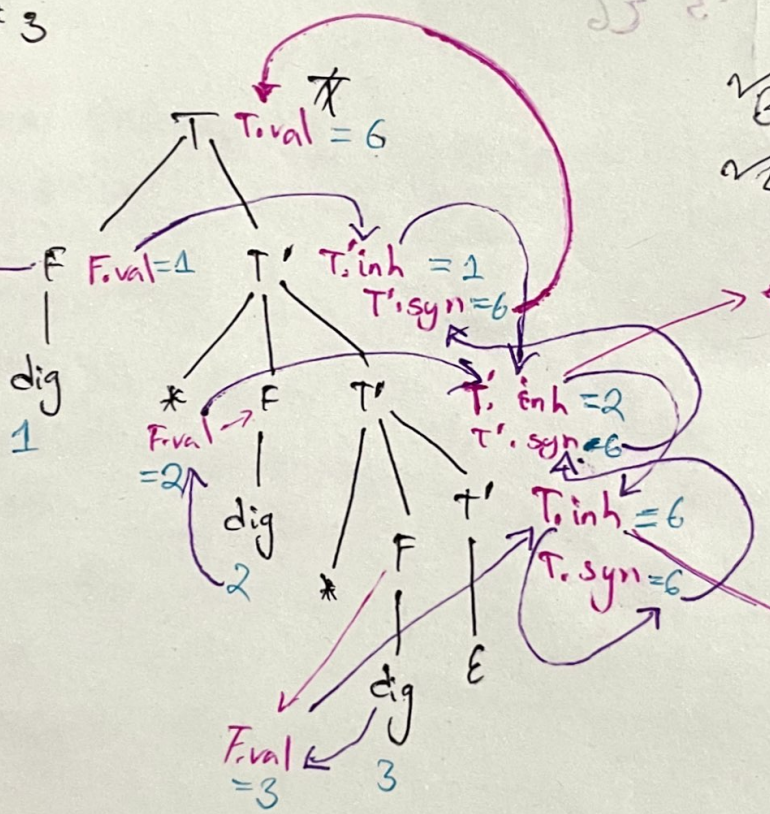
FOR THE ARROWS DEPENDENCY GRAPH.



L ATTRIBUTED SDD

PRODUCTION RULE:	SEMANTIC RULE:
(1) $T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
(2) $T' \rightarrow *FT1'$	$T1'.inh = T'.inh * F.val$ $T'.syn = T1'.syn$
(3) $T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
(4) $F \rightarrow digit$	$F.val = digit.Lexval$

1*2*3



✓ BOTTOM-UP

✓ LEFT TO RIGHT.

T1'.inh = T'.inh * F.val

production rule has to be applied on the same level.

Rule 2

why. because it came from Rule-1 right!

T'.inh = F.val is present in (rule-1)

whenever we apply a rule then we have to apply that rule everywhere

SYNTAX DIRECTED DEFINITION

VS

SDT (Syntax Directed Translation)

they are always in curly braces.

SEMANTIC RULES

RULE :

$E.syn = T.val * F.val$

(used in theory)

$E \rightarrow T * F$

SEMANTIC ACTIONS

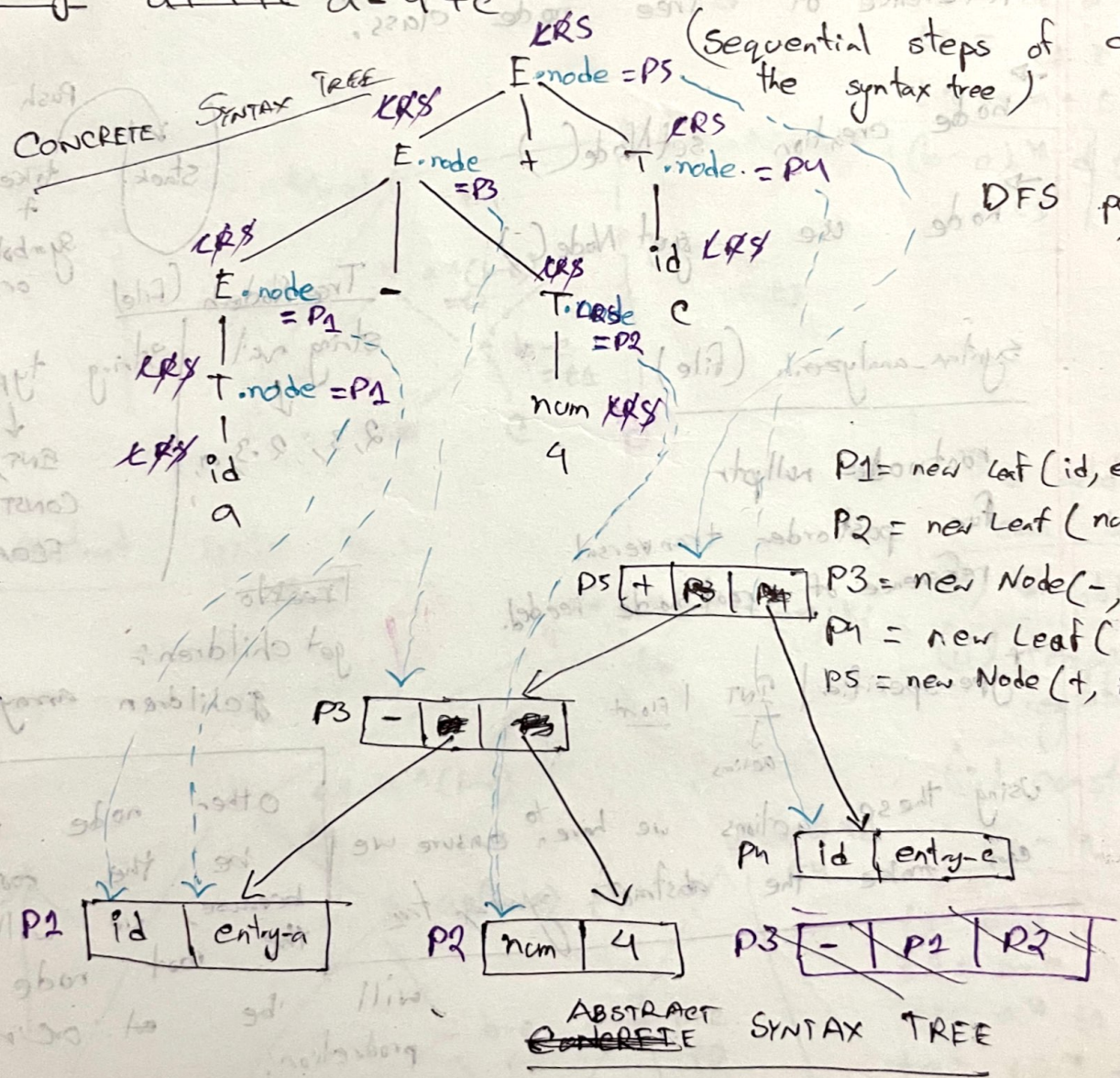
ACTION:

E.node = new Node (*, T.node, F.node)

(used in program)

String: ~~a-4+c~~ a-4+c

(Sequential steps of drawing the syntax tree)



ABSTRACT ~~CONCRETE~~ SYNTAX TREE

DFS post order traversal

* $E \rightarrow F$ no single.

* $F \rightarrow \epsilon$ no marker

(i) Parse tree (concrete syntax tree)

(ii) Abstract Syntax tree

(iii) Directed a Acyclic graph (DAG)

DAG :-

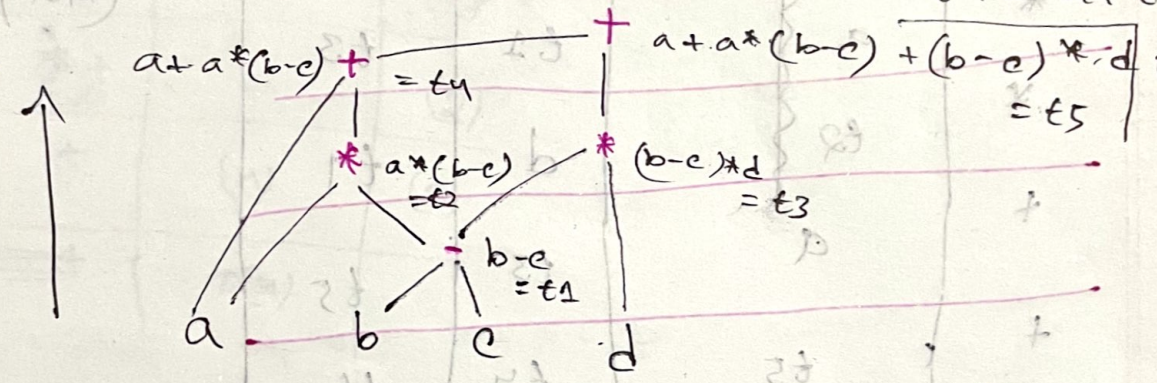
String :- a + a * (b - c) + (b - c) * d

Priority :-

( )

*

\+ -



Three address code :-

t1 = b - c

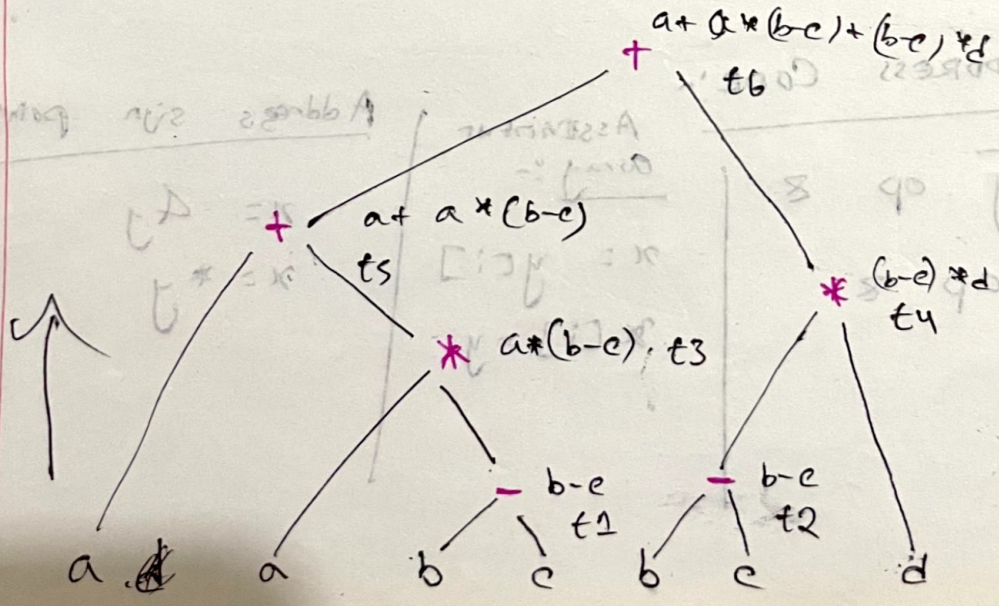
t2 = a * t1

t3 = t1 * d

t4 = a + t2

t5 = t4 + t3

Abstract syntax tree :-



Three address code can be :-

(i) Variable

(ii) temporary register e.g. - t1, t2

(iii) constant

eg. - 4, 5, 2.5

Three address code :-

t1 = b - c

t2 = b - c

t3 = a * t2

t4 = t2 * d

t5 = a + t3

t6 = t3 + t4

→ Three address code are stored in some data structures

The data structures are:-

QUADRUPLES:-

OP	left arg 1	right arg 2	result
-	b	c	t1
-	b	c	t2
*	a	t1	t3
*	t2	d	t4
+	a	t3	t5
+	t5	t4	t6

· FROM Abstract syntax tree (AST)

THREE ADDRESS CODE:-

ASSIGNMENT

$x = y$ op z

$x =$ op z

$x = y$

ASSIGNMENT array :-

$x = y[i]$

$x[i] = y$

Address sign pointer

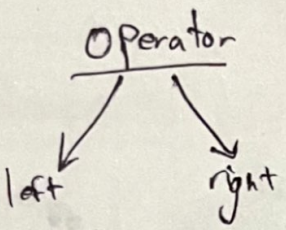
$x = 4y$

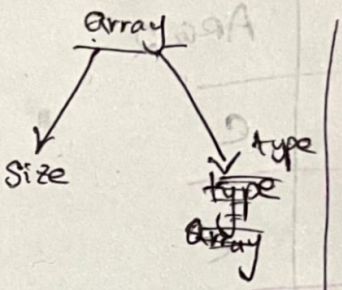
$x = *y$

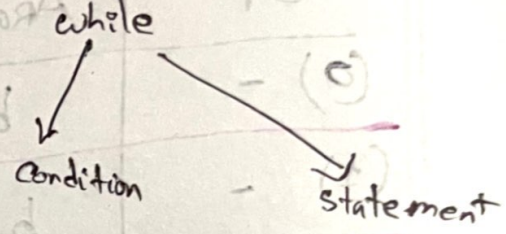
TRIPLES :-

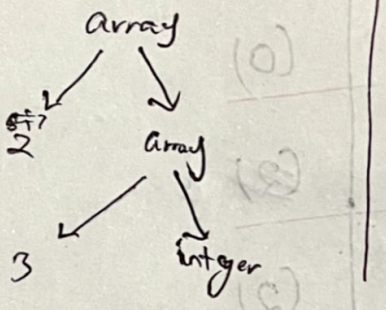
	OP	left ARG 1	right ARG 2
(0)	-	b	c
(1)	-	b	c
(2)	*	a	(0)
(3)	*	(1)	~~(2)~~ d
(4)	+	a	(2)
(5)	+	(4)	(3)
~~(6)~~	~~+~~	~~b (4)~~	

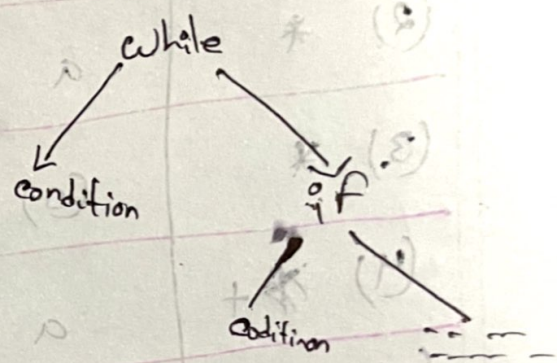
Abstract syntax tree











array(2, array(3, int))

In ^ Type conversion we use graph thats why we have to use recursion.

(Exercise - 6.3.1 Pg-401) Very Imp. (most)

Book Pg→ 396 (grammar) (most)

$D \rightarrow T\ id\ ;\ D \mid \epsilon$ (T: type, D: declaration)

$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$

$B \rightarrow int \mid float$

$C \rightarrow \epsilon \mid [num]\ C$

(modify the grammar) ← (Imp most)

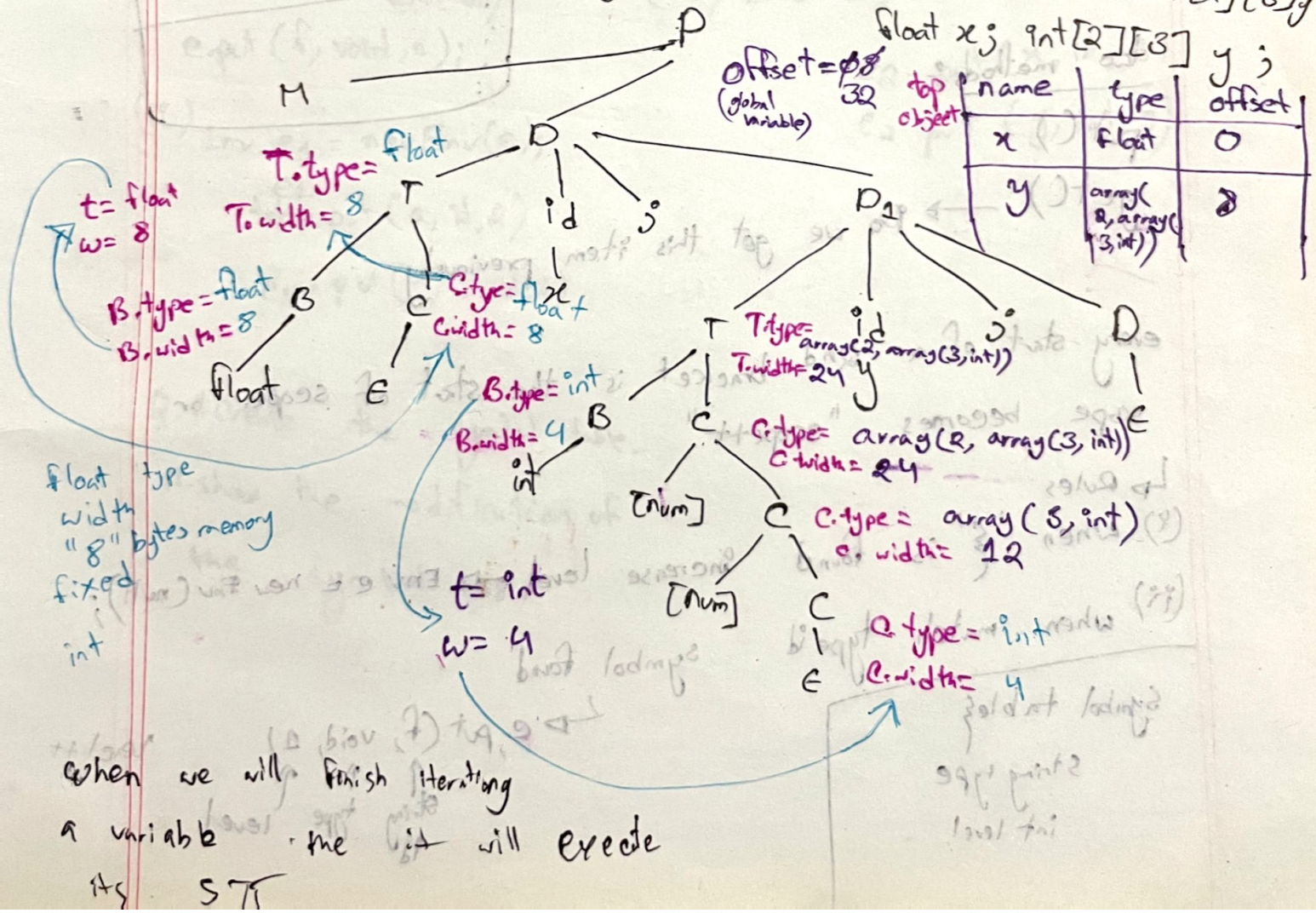
12/07/23

Q. Determine the type and relative address of the following declarations: ~~float x; int[2][3] y;~~

float x; int[2][3] y;

name	type	offset
x	float	0
y	array(2, array(3, int))	8

offset = ~~0~~ 32 (global variable) top object



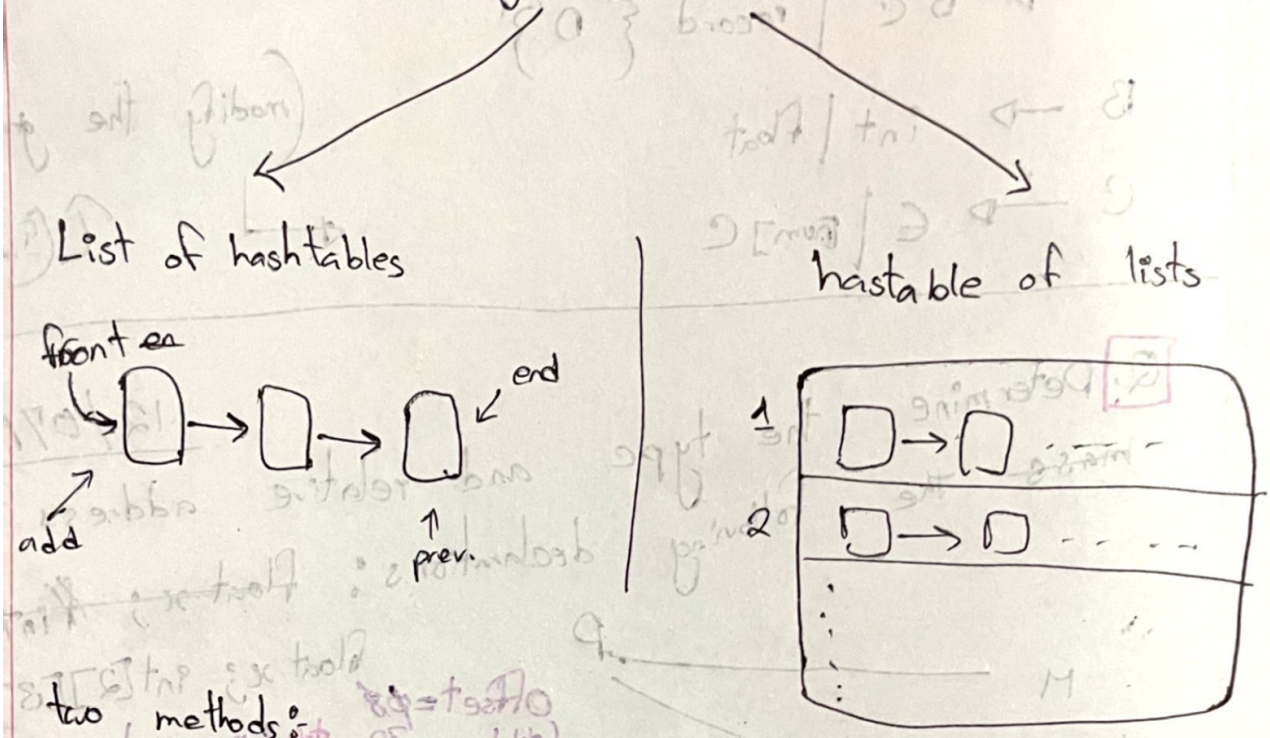
float type
width "8" bytes memory
fixed
int

when we will finish iterating a variable the id will create its ST

SYMBOL TABLE & SCOPING

there we will construct the symbol table

symbol table construction



two methods:-

put()	x
get()	→ Do we get this item previously

every start of a 2nd bracket is the start of scope. or

scope becomes "scope++"

↳ Rules

(i) when { is found increase level → Env e = new Env(null);

(ii) when match type id symbol found

↳ e.put(f, void, 1) level++

String id, type, level

Symbol table{

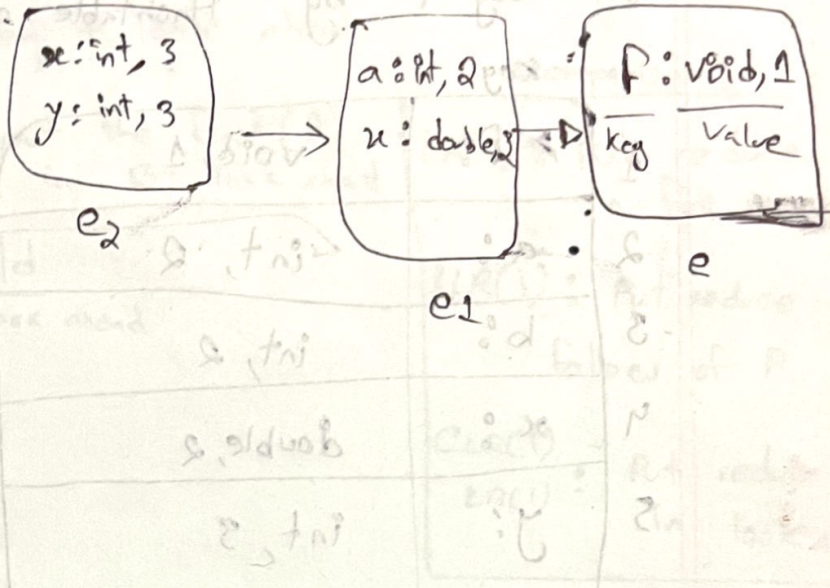
String type

int level.

When the scope will increase? exactly

List of hashtables

```
Void f (int a){
    double x;
    while(...){
     int x,y; a=5;
     ....
    }
   }
   Void g(){
    f(3);
   }
```



(i)
Env e = new Env(null)
e.put (f, void, 1);

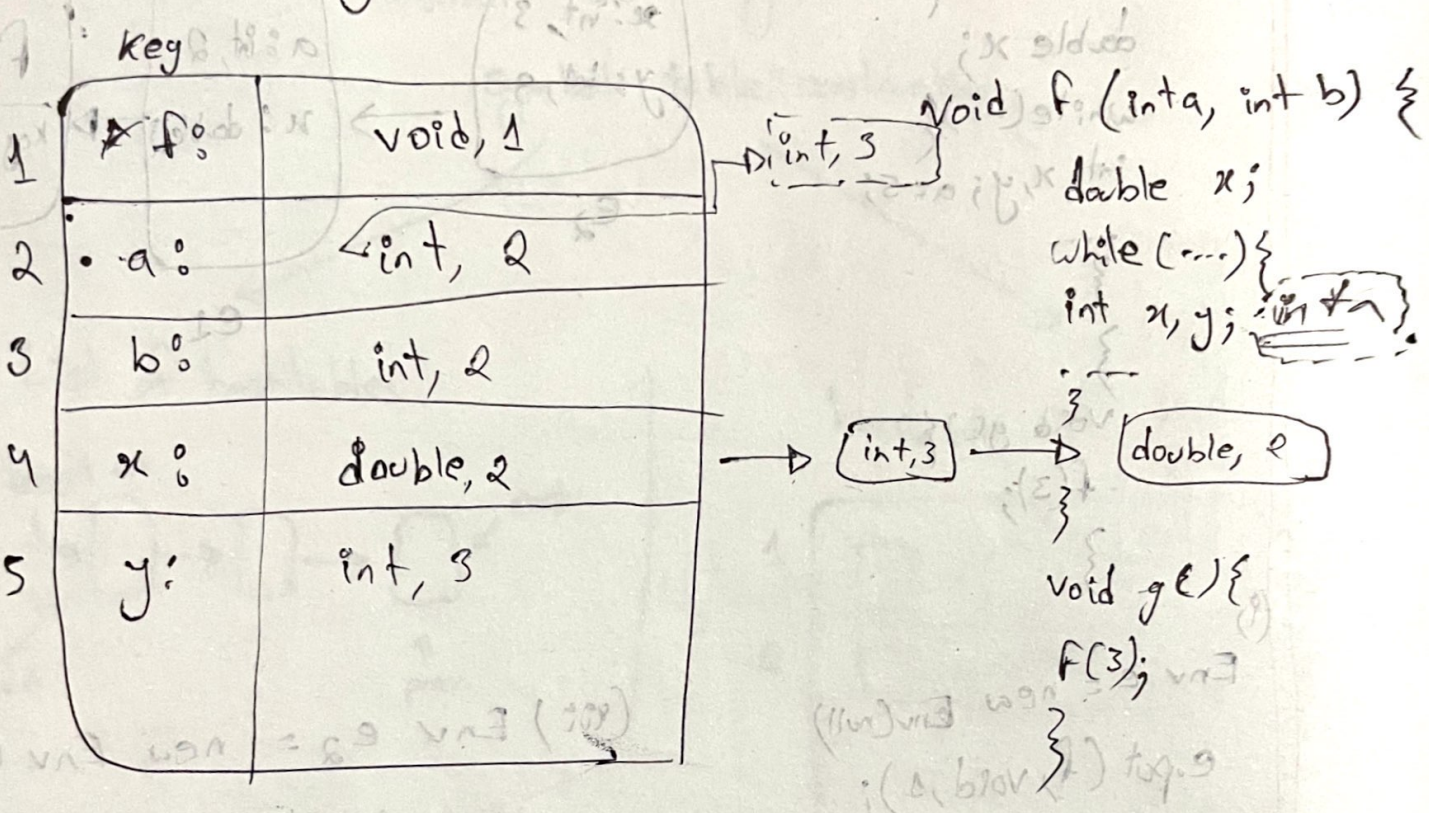
(ii) Env e_1 = new Env(e);
e_1.put (a, int, 2)
e_1 . put (x, double, 2)

(iii) Env e_2 = new Env (e_1)
e_2. put (x, int, 3)
e_2 put (y, int, 3)

Q. #Create the symbol table.
#. show the modification of the symbol table.

Hashtable of Lists

Q. Draw the symbol table operations ~~for~~ for the following declarations in the body f by Hashtable of lists approach.



1. scope entry → level ++
2. variable declare → check type id and check the levels
3. Variable use → check if ..
4. scope exit

different level if same type and it throws error?

LR(1) Parsing

LR(1) item = LR(0) item + look ahead.

Rules:-

(i) $A \rightarrow \alpha . B \beta$, c/d — put the first(β) in B's look ahead

c/d = look ahead

 # 1st case:-

$A \rightarrow \alpha . B \beta$, c/d

$\beta \rightarrow a$ — take the first of β

$B \rightarrow \gamma$, a — putting the β first() in as B's look ahead

 # 2nd case:-

$E \rightarrow BB$
$B \rightarrow cB \mid d$

→ what is the first() of B? → what B? because → after B, IB arrives

first(B) = c/d

$E \rightarrow . BB$

$B \rightarrow . cB \mid . d$, c/d

So,

 # 4th case: $\beta = \epsilon$

$E' \rightarrow . E$, $\$$

$E \rightarrow . BB$, $\$$

$\$$ was the look ahead

 # 3rd case:- no Non terminal !

$A \rightarrow \alpha . B$, c/d

$B \rightarrow \gamma$, c/d

$\beta = \epsilon$

then look ahead will simply move down

Comparison:-

LR(0): Put reduce in full row

SLR(1): Put reduce in follow of P

CLR(1) / LR(1): Put reduce only in look ahead

State	Action			GOTO	
	c	d	$	S	C
I_0	s3	s4		1	2
I_1			accept		
I_2	s6	s8			5
I_3	s3	s4			7
I_4	r3	r3			
I_5			r1		
I_6	s6	s8			9
I_7	r2	r2			
I_8			r3		
I_9			r2		