7/5/25 L1

Compiler Construction

CSE - 415, 416

Target for generation

code optimization

intermediate program (.asm)

Lab - Hard

Theory - Easy.

Source code

Lexeme

Parsing

Lexical Analysis → Token

Type id lp Rp

Syntax Analysis

↓

Check wether Grammar is correct

Semantic Analysis

↓

Logical correctors

```
int main()
{
    printf("Hello 57E");
    return 0;
}
```

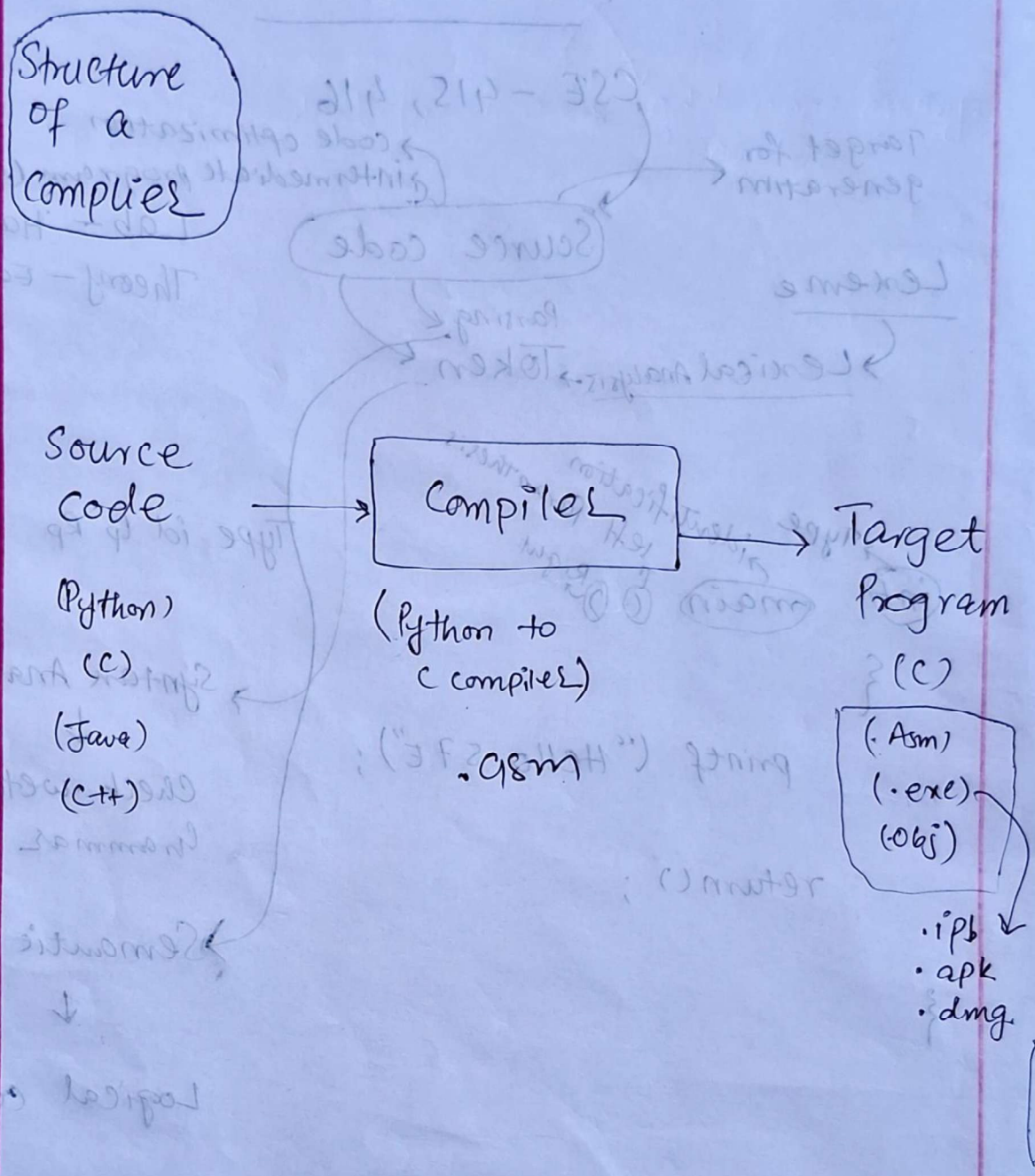
int - Type

main - identification

(- left Paranthesis

) - Right

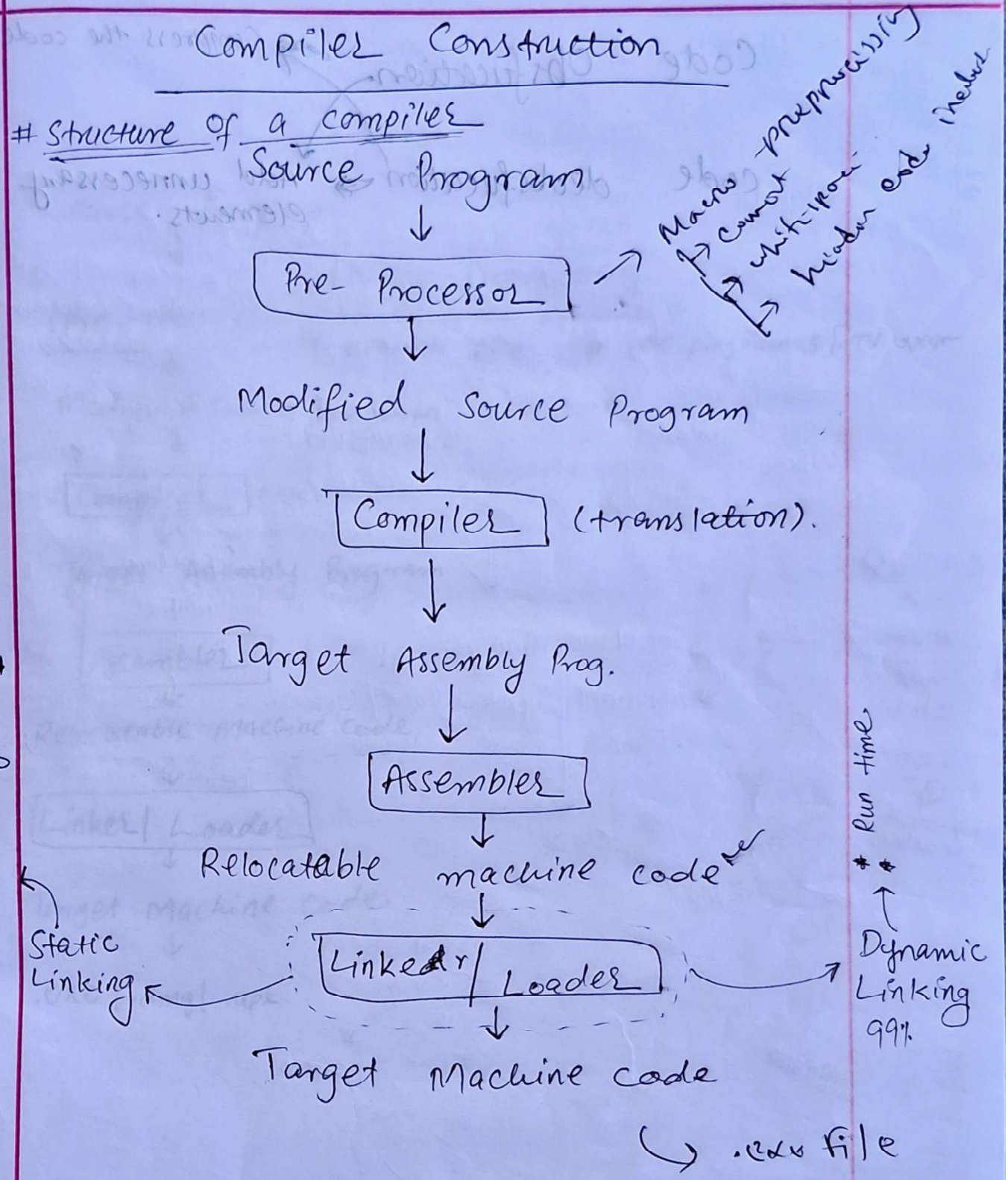
Structure of a Compiler



Nahin Ul Sadad

Compiler Construction

Structure of a compiler



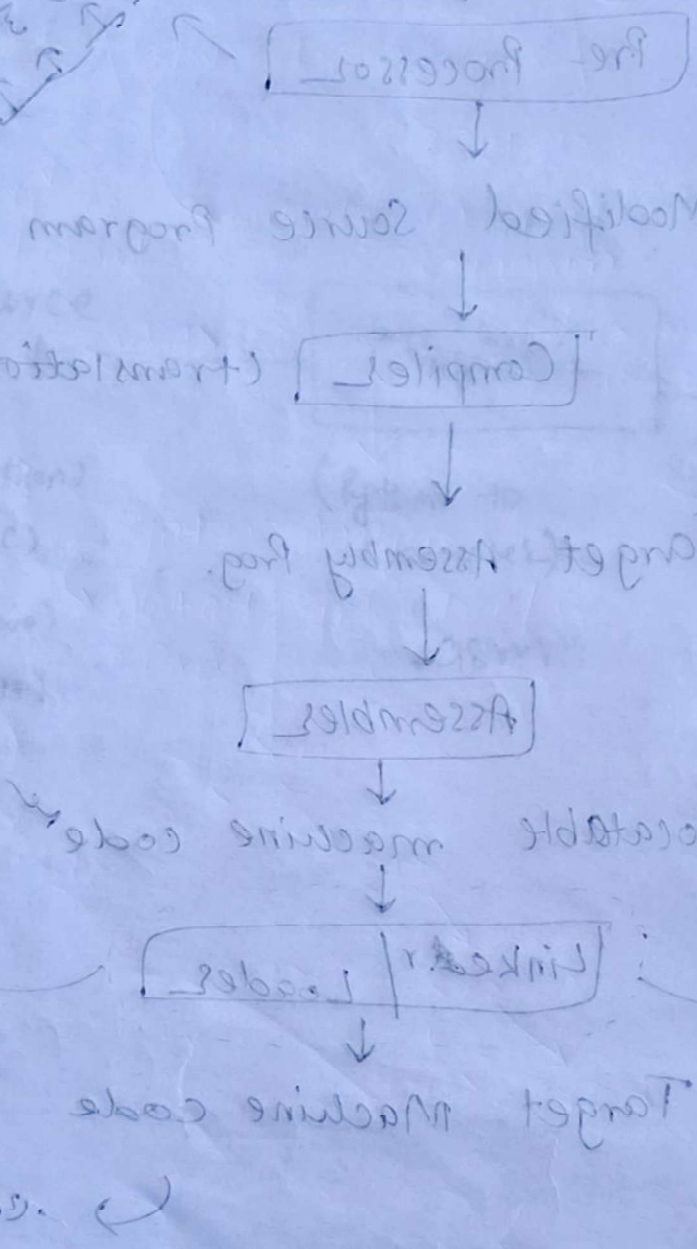
Exam
*Compile takes time
*Updating tough
*File size is big.

* run time

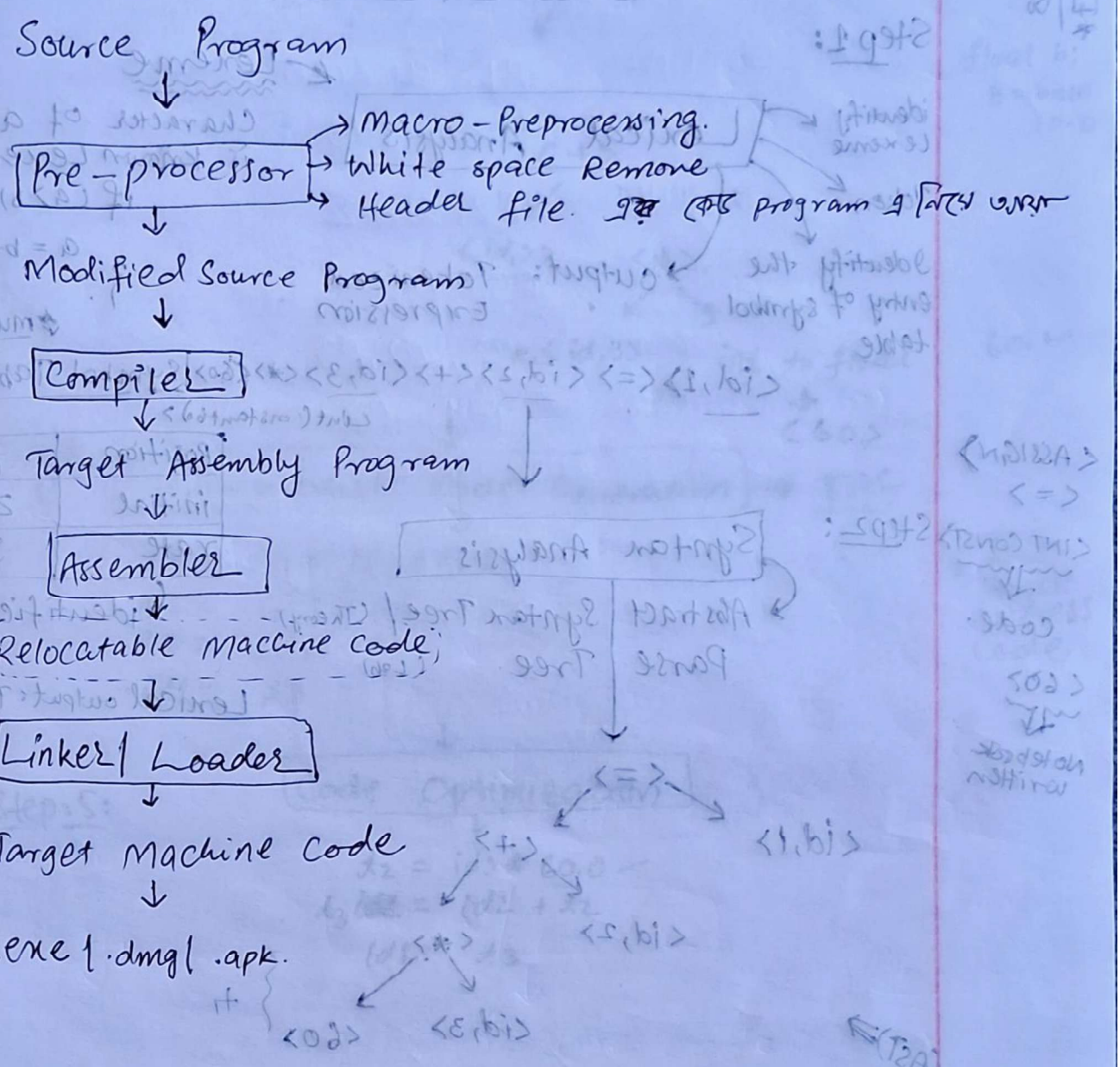
Code Obsfucation

- Compress the code
- Add unnecessary elements.

code → deobsfucation



14/5/25 L2



* dependency code- where the environment needed to run the program.

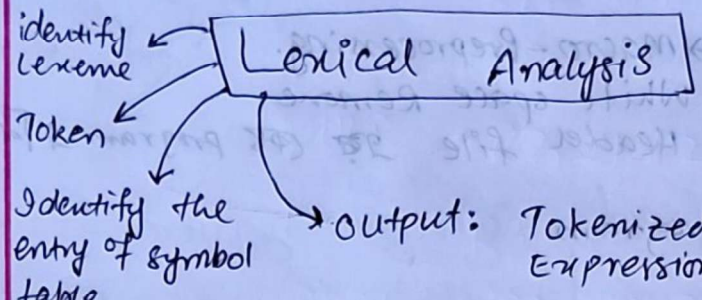
Show the phases/stages of a compiler for the given input/code snippet/exp

Phases of a Compiler

*Exam 8/10 mark

position = initial + rate * 60

Step 1:



Lexeme ← meaningful character of a string is known Lexeme.

```
if (a>b) {
    a = b+10;
}
```

must (3 marks)

<id,1><=><id,2><+><id,3><*><60> | Symbol Table

<int(constant 60)>

position	1
initial	2
rate	3

identifies only

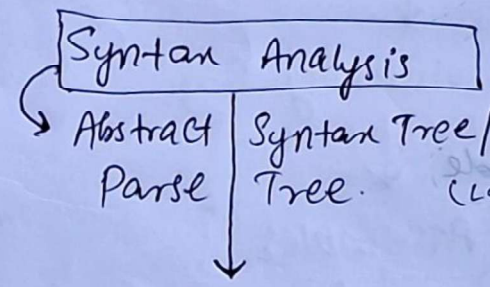
Lexical output: Tokenized Expr.

<ASSIGN> <=>

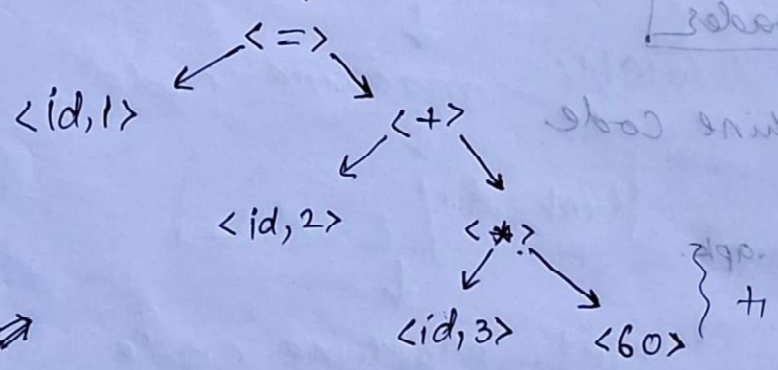
<INT CONST> code.

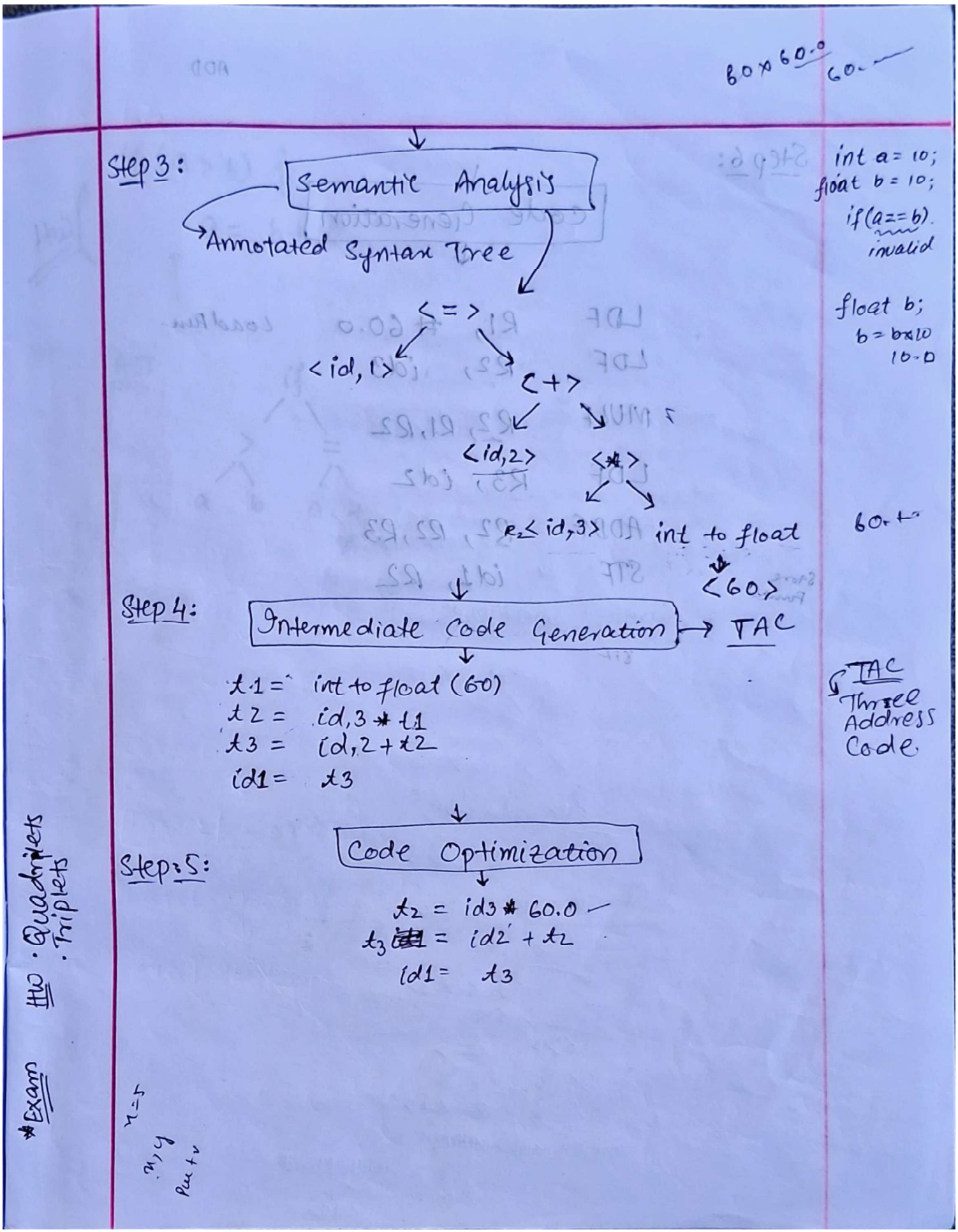
<60> Notebook written

Step 2:



(AST)




Step 3:
Semantic Analysis
Annotated Syntax Tree
<=>
<id,1>
<+>
<id,2>
<*>
<id,3>
int to float
<60>
Step 4:
Intermediate Code Generation
TAC
t1 = int to float (60)
t2 = id,3 * t1
t3 = id,2 + t2
id1 = t3
Step 5:
Code Optimization
t2 = id3 * 60.0
t3 = id2 + t2
id1 = t3
int a = 10;
float b = 10;
if(a==b).
invalid
float b;
b = b*10
10.0
60.0
TAC
Three Address Code
*Exam
HW
Quadruplets
Triplets

ADD

Step 6:

Code Generation

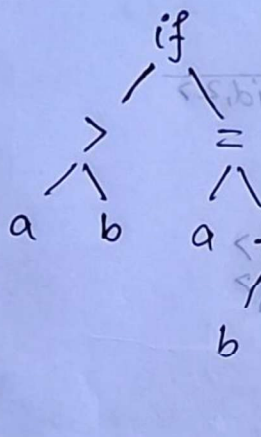
```
LDF   R1, #60.0     Load Fun
LDF   R2, id3
MULF  R2, R1, R2
LDF   R3, id2
ADDF  R2, R2, R3
STF   id1, R2
```

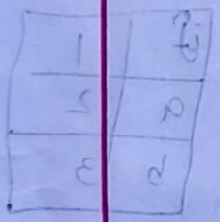
Store Func

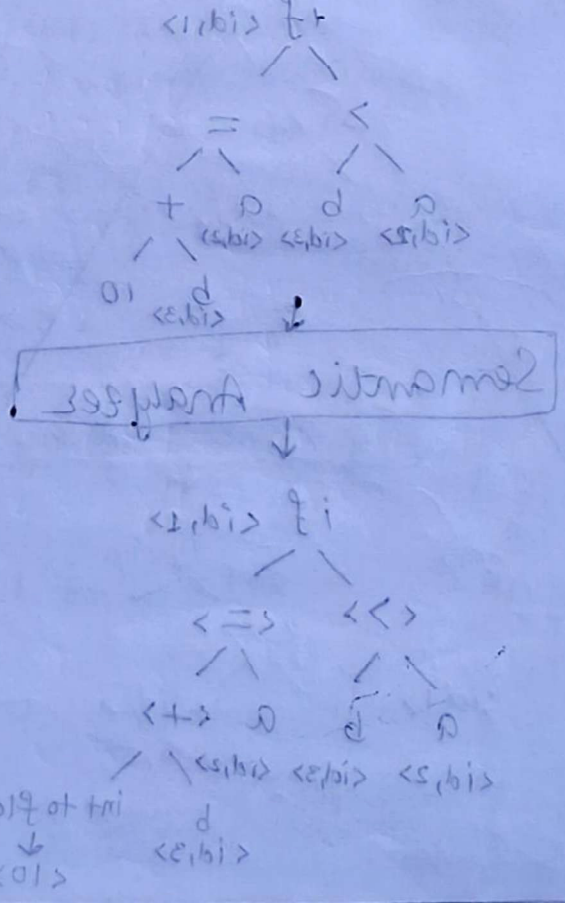
HW

```
if (a > b) {
    a = b + 10;
}
```

AST







18/05/25 HW

```
if(a > b) {
    a = b + 10;
}
```

Step 1

Lexical Analysis

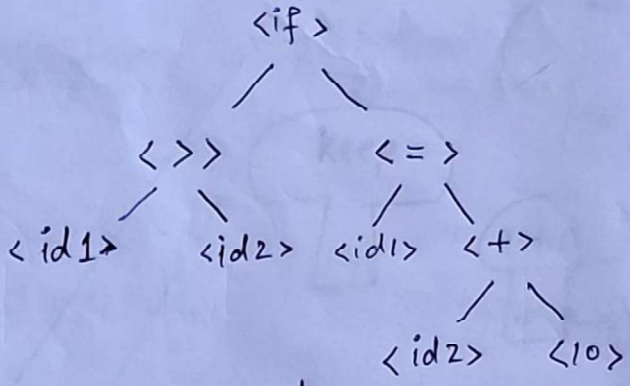
<if> <(> <id1> <>> <id2> <)> <{ >
<id1> <=> <id2> <+> <10> <;> <}>

Symbol Table

a	1
b	2

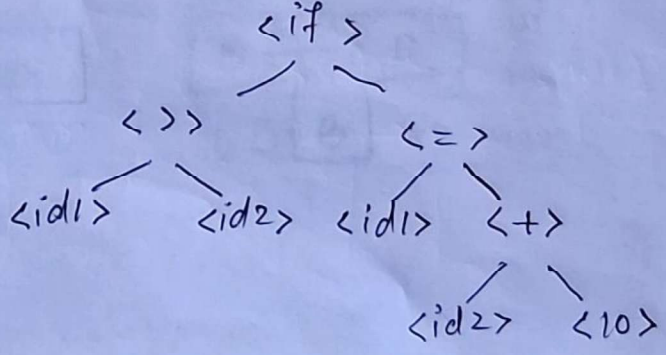
Step 2

Syntax Analysis



Step 3

Semantic Analysis



Step 4 **Intermediate code Generator**

↓

① $t1 = id1 > id2$
② IF $t1$ = TRUE goto L4
③ EXIT
④ $t2 = id2 + 10$
⑤ $id1 = t2$
⑥ goto L3

↓

Step 5 **Code optimization**

↓

① IF $id1 > id2$ goto L3
② EXIT
③ $t2 = id2 + 10$
④ $id1 = t2$
⑤ goto L2

↓

Step 6 **Code Generation**

↓

```
LDF   R1, id1
LDF   R2, id2
CMP   R1, R2
JG    L1 - ASM
L2:   EXIT  L1 - ASM
```

```
LDF   R3, id2
ADDF  R3, R3, #10
STF   id1, R3
JMP   L2
```

HW

Three Address Code

There are 3 representations of the TAC namely,

1) Quadruple
2) Triples
3) Indirect Triples.

Quadruple: It is a structure which consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and results is used to store the result of the expression.

Example: Consider the expression

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

Because (-) is binary (used for subtraction like a-b), but uminus is unary, affecting just one variable (-c, -x, etc).

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

Operator (op)	Arg 1	Arg 2	Result.
uminus	c		t1
*	b	t1	t2
uminus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples: This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So it consist of only three fields namely op, arg1 & arg2.

Example: Consider expression

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

$t1 = \text{uminus } c$

$t2 = b * t1$

$t3 = \text{uminus } c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

	Index	Operator (op)	Arg 1	Arg 2
t_1	0	uminus	c	
t_2	1	*	b	0
t_3	2	uminus	c	
t_4	3	*	b	2
t_5	4	+	1	3
t_6	5	=	a	4

Indirect Triples: This representation makes use of pointer to the listing of all references to computations which is made separately & stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example: Consider expression

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

Index	Op	Arg1	Arg2	#	Statement
14	uminus	c		0	14
15	*	b	14	1	15
16	uminus	c		2	16
17	*	b	16	3	17
18	+	15	17	4	18
19	=	a	18	5	19

18/05/25 L3

VIVA

AOT (Ahead of Time) → Code blocks

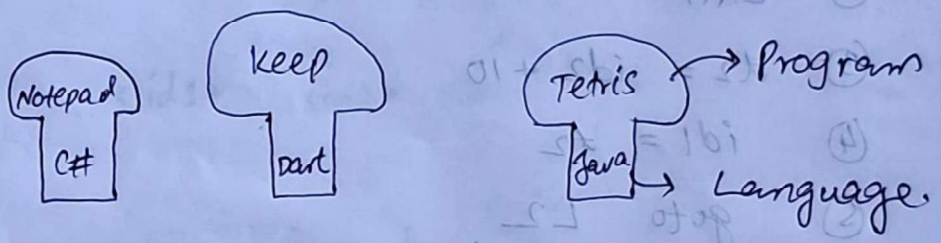
JIT (Just In Time) → Run time when needed compiler.

Hybrid Compiler

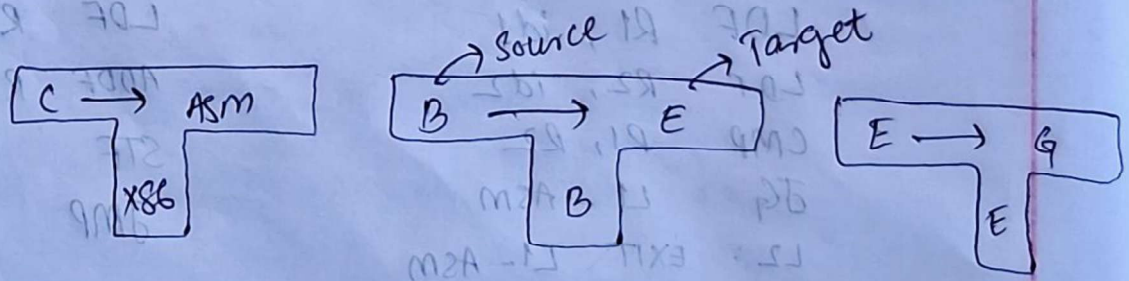
Tombstone Diagram

Different graphical process of a computer is Tombstone

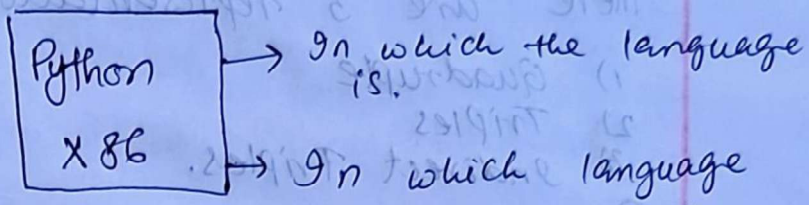
1) Program



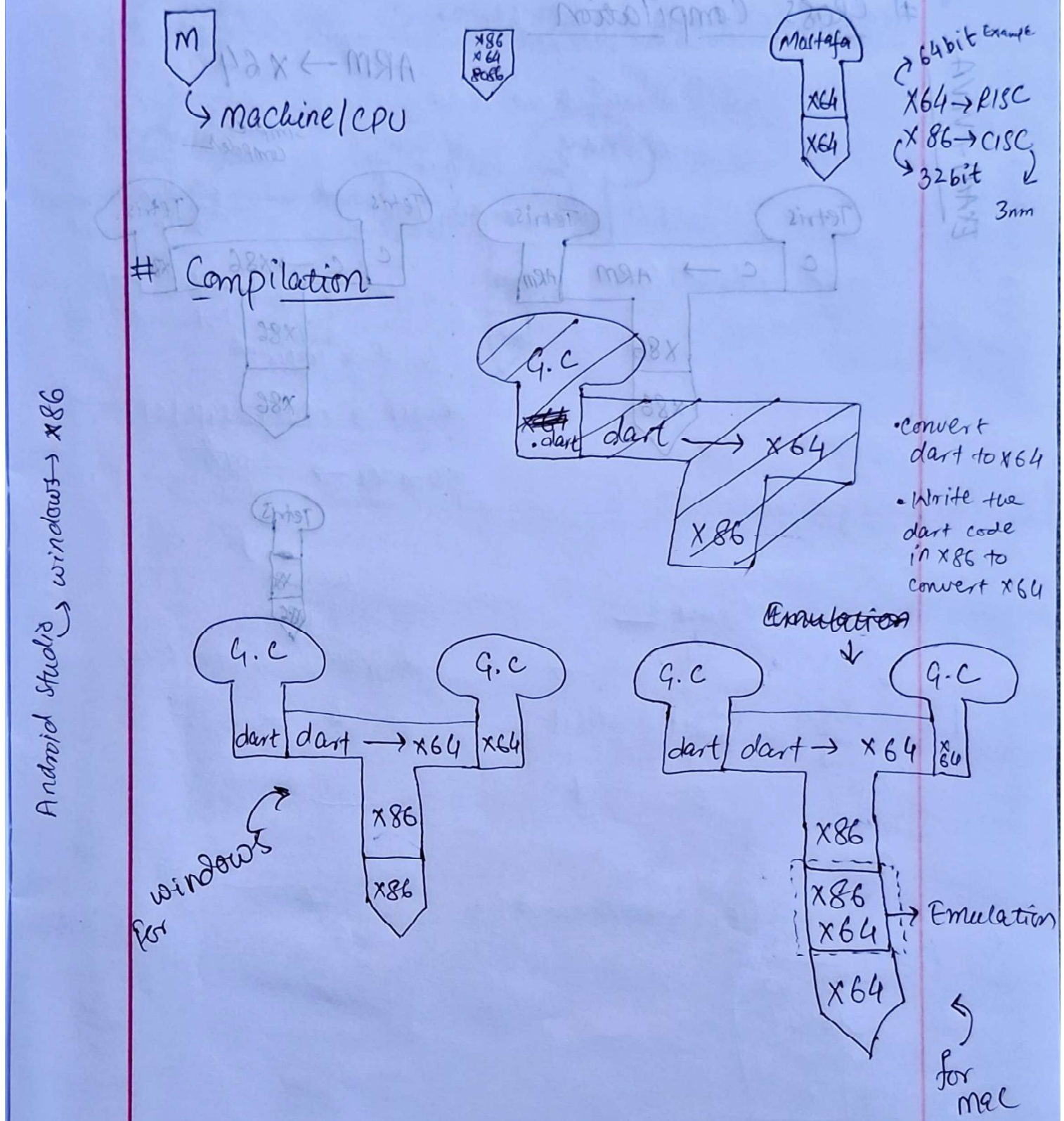
2) Compiler



3) Interpreter

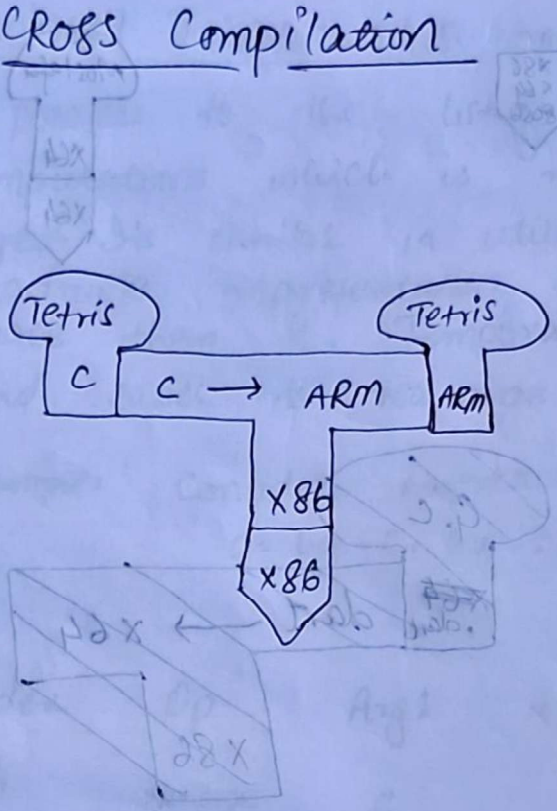


21/05/25 L4 Tombstone Diagram



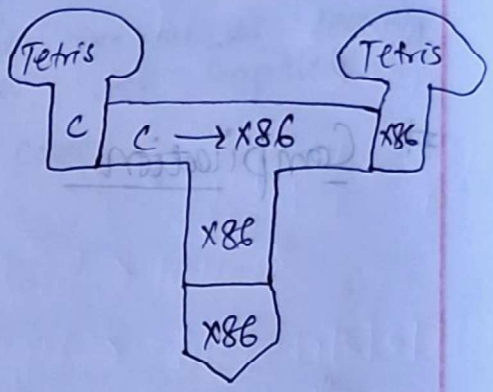
EXAM + VIVA

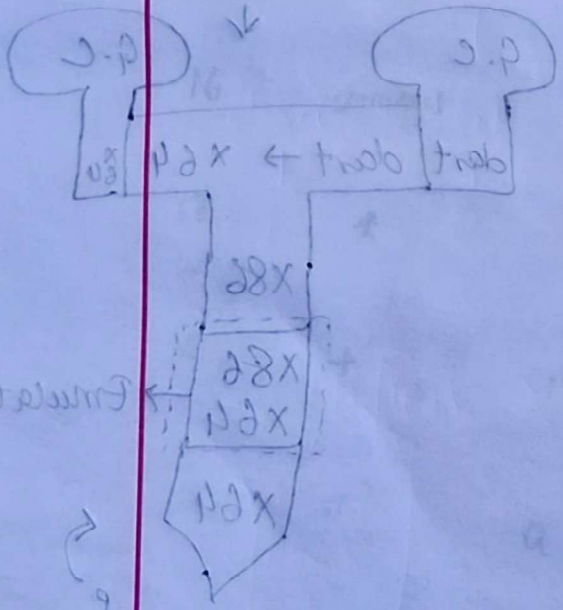
Cross Compilation

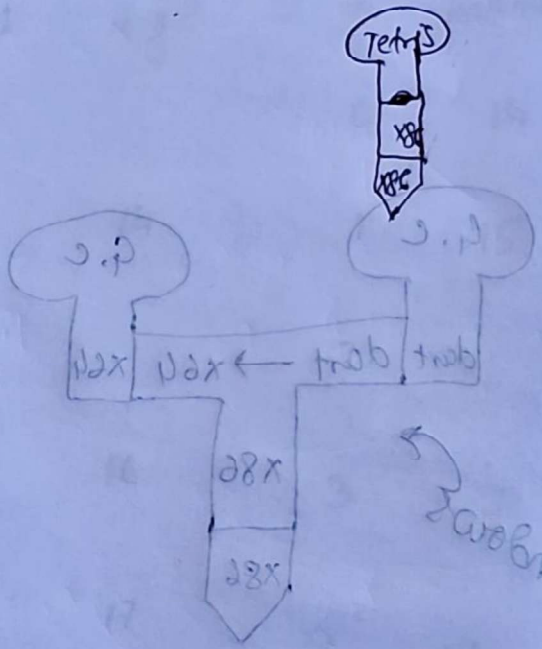


ARM → x64

simple compiler



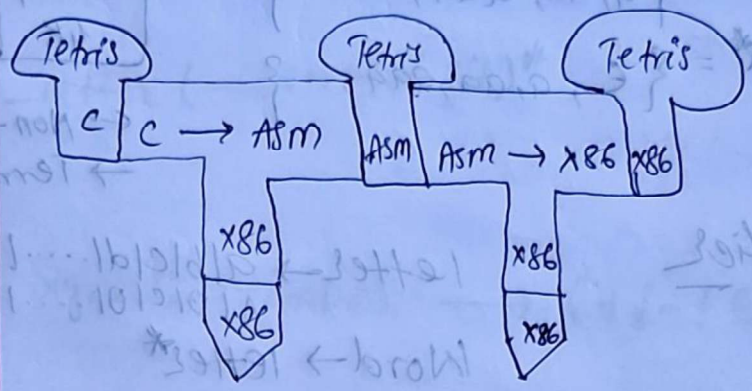




CT Date 28/5/25

Topic :- Upto Tombstone Diagram

Two Stage Compilation



25/5/25 L5 LAB3 26/5/25

CFG = Context Free Grammar

Regular Expression & CFG

$a^+ = \{a, aa, aaa, \dots\}$

$a^* = \{\epsilon, a, aa, aaa, \dots\}$

Ak47	✓
-Ak	✓
-k_4	✓
4K	X

← Non-Terminal
→ Terminal

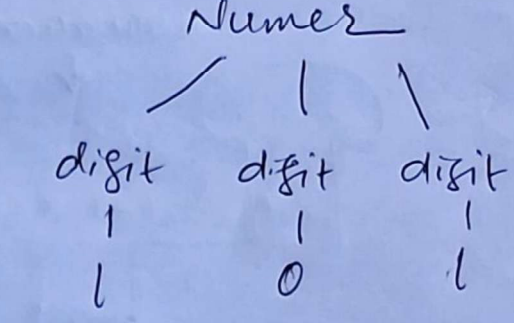
Valid Identifier

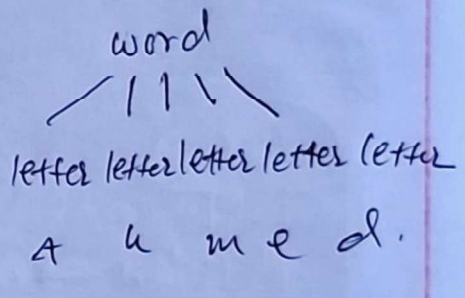
letter → a|b|c|d|....|z|
A|B|C|D|....|Z| →

Word → letter*

digit → 0|1|2|3|4|5|6|7|8|9

Number → digit*





Identifiers → letter*(letter/digit)*

AK47	
AK	✓
K4	✓
4K	X

- letter → a|b|c|d|----- |z|-|
A|B|C|D|----- |Z|-|

if (@)
- letter (- letter @ digit)*

- letter (- letter | digit)*

Signed_Number → (+|-|ϵ) Number

- 101 (valid).

Sign/Number*
5+55-4
invalid.

valid invalid
- 101.

- 101.86

Float_Numb → (+|-|ϵ)(digit)*.(digit)*+

Optional_Float_Num → (+|-)? (digit)* ((digit)+)|ϵ)
(.(digit)+)?

same both correct

SDP
SOT

$6.673E^{-34}$

Scientific_Optional_Float = Num →

$(+|-)?(digit)^* ((.(digit)^+)|\epsilon)$

$((E \wedge (+|-)?(digit)^+)|\epsilon)$

Q validate the expression

-101.75

digit $[0 \rightarrow 9]$

Signed_Float_Num → $(+|-)?(digit)^*(.(digit)^+)?$

$100/20$ Expr → $(digit)^* / (digit)^+$

While

WHILE → while

FOR → for

IF → if

LP → (

RP →)

FOR → for

) → Paranthesis

} → Brace

] → Bracket

Lexical → Structure

Tokenization

```
double main () {
    float c = 12;
    int b = 10.0;
    while (1) {
        a = b + c;
    }
    return 5;
}
```

Pattern
Type → double | int | float
Main → main
LP → (
RP →)
LBR → {
_letter → [A-Z a-z _]
digit → [0 — 9]
id → _letter (_letter | digit)*
Assign → =
num → (+ | - | ϵ) digit+ (. (digit)+)?
Semi → ;
Wlhile → while
ADDOP → +
RBR → }
Return → return

CFG 2

Special case:

decl-stmt → Type id assign num semi;

decl-stmt_int → INT ID ASSIGN_NUM

Token Name	Lexeme
~~Type~~ TYPE	double, float, int
~~MAIN~~ MAIN-~~FUNC~~ FUNC	main
LP	(
RP	)
LBR	{
ID	c, b, a
ASSIGN	=
~~INT~~ NUM	12, ~~1,5~~ 12,1,5, 10.0
SEMI	;
WHILE	while
ADDOP	+
RBR	}
RETURN	return

18/06/25 L6

First & Follow sets

A → Non Terminal
a → Terminal.

① $S \rightarrow abc \mid def \mid ghi$

$S \rightarrow abc$

$S \rightarrow def$

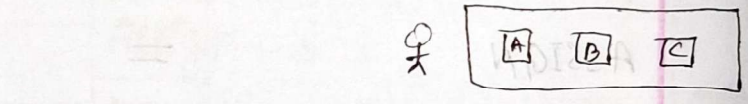
$S \rightarrow ghi$

First(S) = First(abc) ∪ First(def) ∪ First(ghi)
= {a} ∪ {d} ∪ {g}.

First(S) = {a, d, g}

---

② $S \rightarrow ABC$
$A \rightarrow a \mid b \mid \epsilon$
$B \rightarrow c \mid d \mid \epsilon$
$C \rightarrow e \mid f \mid \epsilon$



FIRST(S) = FIRST(ABC)
A → ϵ, B → ϵ, C → ϵ

FIRST(A) = {a, b, ϵ}.
FIRST(B) = {c, d, ϵ}.
FIRST(C) = {e, f, ϵ}.

= {ϵ, a, b, c, d, e, f}

FIRST(S) = {ϵ, a, b, c, d, e, f}.

$E \rightarrow TE'$

$E' \rightarrow *TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow \epsilon \mid +FT'$

$F \rightarrow id \mid (E$

$FIRST(E) = FIRST(TE')$ $FIRST(E) = \{id, (\}$.

$FIRST(T) = FIRST(FT')$ $FIRST(T) = \{id, (\}$.

$FIRST(F) = \{id, (\}$

$FIRST(T') = \{+, \epsilon\}$

$FIRST(E') = \{*, \epsilon\}$.

PRACTICE

$S \rightarrow ACB \mid Cbb \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

FIRST(S) = FIRST(ACB) ∪ FIRST(Cbb) ∪ FIRST(Ba)
= {d, g, h, ε} ∪ {h, b} ∪ {g, a}.
= {d, g, h, b, a, ε}.

FIRST(A) = {d, g, h, ε}

FIRST(B) = {g, ε}

FIRST(C) = {h, ε}

FIRST(S) = { FIRST(ACB)

✗ FIRST(A) = {d, B}.

FIRST(B) = {g, ε}.

FIRST(C) = {h, E}.

start symbol → S → ACD
C → a|b.

① Follow of start symbol is $.

② Follow (A) = {a, b}.

Follow (D) = {$}

S → ACD
C → a|b.

Follow (A) = {a, b}.

Follow (D) = {$}.

S → aSbS | bSaS | ∈

aSbS | bSaS | ∈

F(S) = {b, $, a}

S → AaAb | BbBa

A → ∈

B → ∈

Follow (A) = {a, b}.

Follow (B) = {b, a}.

$S \rightarrow ABC$

$A \rightarrow DEF$

$B \rightarrow \epsilon$

$C \rightarrow \epsilon$

$D \rightarrow \epsilon$

$E \rightarrow \epsilon$

$F \rightarrow \epsilon$

Follow (A) = FIRST (B)

= FIRS (C)

= FOLLOW (S)

= $\{\$\}$

23/06/25
27/06/25

L7

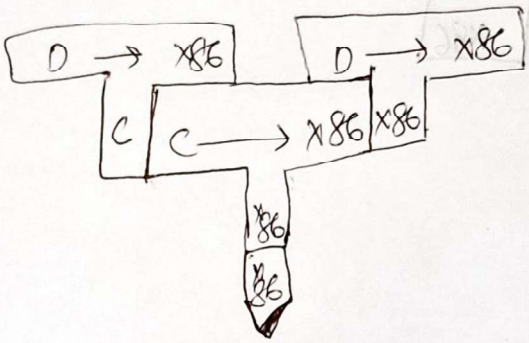
Syntax Analysis

Tombstone ** Bootstraping

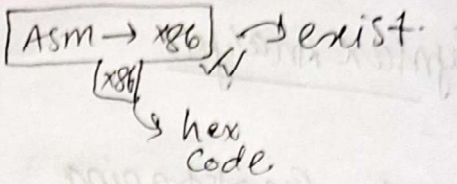
Half bootstraping

Full "

Half

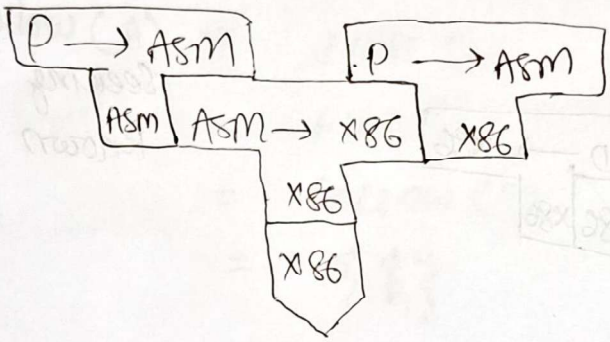


seeking help from known lang.

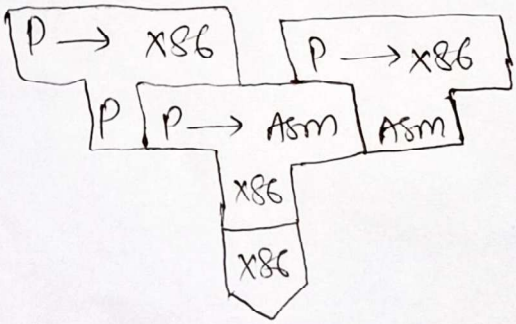


Full Bootstraping

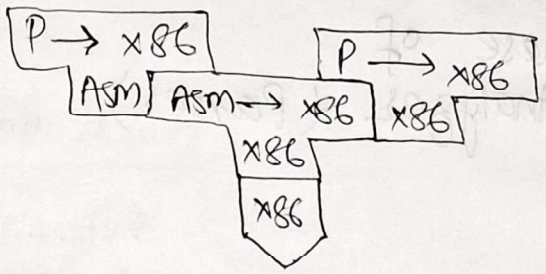
Step 1



Step 2



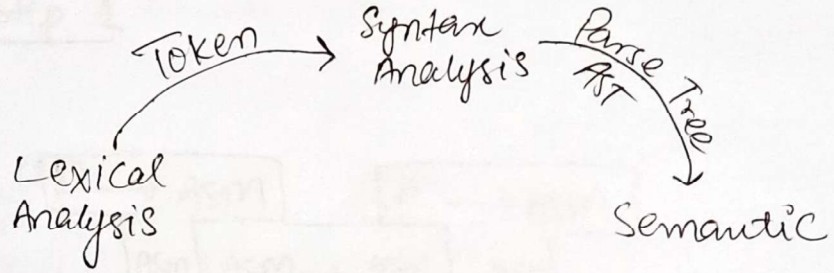
Step-3



Syntax Analysis

→ 2nd Phase of
→ Syntax Analyzer (Parser)

* Final Exam
20–25 Mark



6 Phase of compiler
Syntax Analysis
B-10 → 20-25 →

Types of Parser

Top down

- Recursive Descent Parser
- Backtracking Parser
- Predictive Parser / Non-R.D.P / Non-Backtracking Parser / LL(1)

Bottom up Parser

- LR Parser
  - LR(0)
  - SLR(1)
  - LALR(1)
  - CLR(1)
- operation Precedence Parser

Q. • Parse the input string with the given grammar with recursive parser

Recursive Descent Parser

Grammar:

$S \rightarrow cAd$

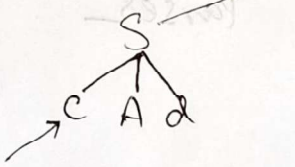
$A \rightarrow abla$.

String: cad

Buffer ↓ Temp. Storage

Soln:

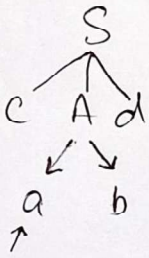
Step 1: Buffer: cad



Accept move to next step.

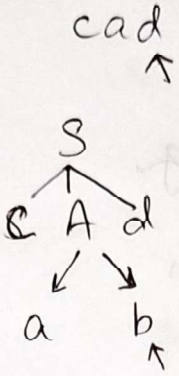
Step 2:

cad



Accept move to next step.

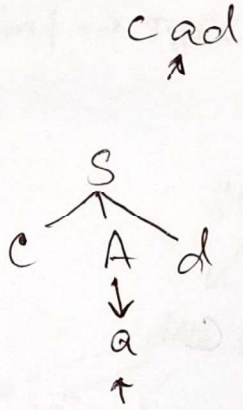
Step3 :



Not Accepted.
(The parser will backtrack)

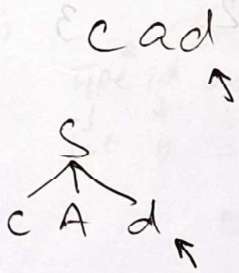
Backtracking Parser.

Step 4 :



Accept move to next step.

Step 5 :

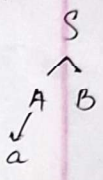


Accepted

Non-Recursive Descent Parser

LL(1)

- Left-to-Right scanning
- Left most derivation
- 1 look ahead



$$S \xrightarrow{1} aSa \mid \underset{2}{bS} \mid \underset{3}{c}$$

First(S) = {a, b, c}

Follow(S) = {a, $}

Predictive Parsing Table

	a	b	c
S	1	2	3

L8
08/08/25

L8

Q Write app. grammer and make/Draw parse tree?

LL(1)

* Make the parsing of LL(1)

① $S \rightarrow asbS \mid bSaS \mid \epsilon$ (① ② ③)

First(S) = { a, b, ϵ }

Follow(S) = { b, a, $ }

LL(1) Parsing table

	a	b	$
S	1/3	2/3	3

Not parseable ✗

② $S \rightarrow aSa \mid bS \mid c$ (① ② ③)

First(S) = { a, b, c }

Follow(S) = { a, $ }

LL(1) Parsing Table

	a	b	c	$
S	1	2	3	

Parseable ✓

③ $S \xrightarrow{①} iCtSS_1 \mid \xrightarrow{②} a$

$S_1 \xrightarrow{③} eS \mid \xrightarrow{④} \epsilon$

$C \xrightarrow{⑤} b$

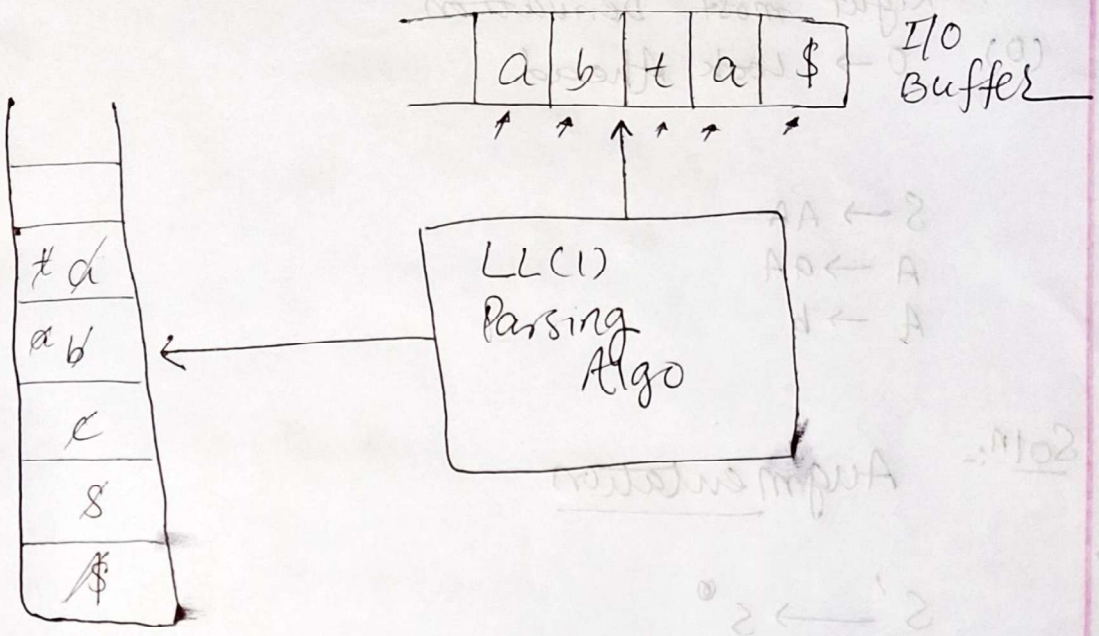
	First	follow
S	{i, a}	{e, $}
S_1	{e, ϵ}	{e, $}
C	{b}	{t}

LL(1) Parsing Table

	i	a	e	t	b	$
S	1	2				
S_1			3|4			4
C					5	

→ Not LL(1) Parseable ✗

How LL(1) Parser decides wether a string accepted or rejected?



$S \longrightarrow abC \mid ab$

$C \longrightarrow ta$

String: abta

9/7/25 | L9

*Validate the grammer with LR(0)

LR(0) Parsing

(L) Left to Right Scanning
(R) Right most Derivation
(0) 0 → Look Ahead

Regular grammer

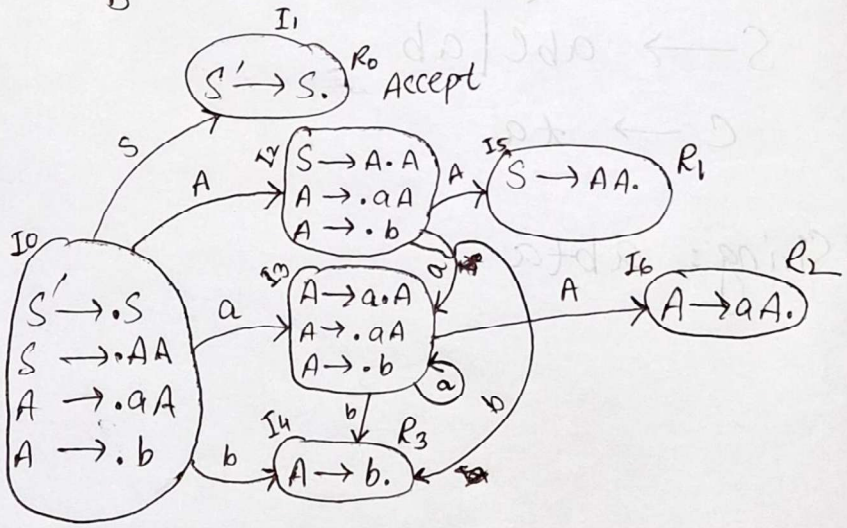
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

Soln:- Augmentation

Augmented grammer

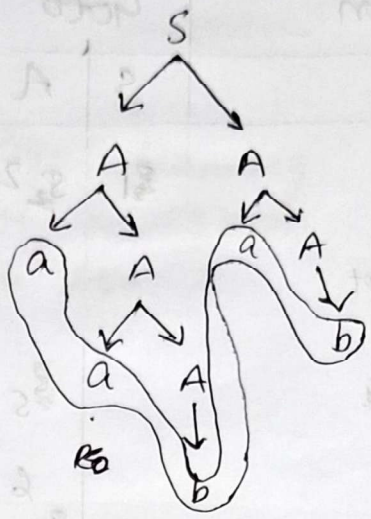
$S' \longrightarrow S$ ⓪
$S \longrightarrow AA$ ①
$A \longrightarrow aA$ ②
$A \longrightarrow b$ ③

Initial state

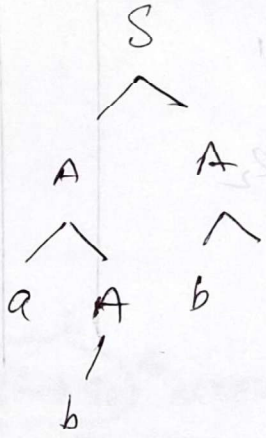


States	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3	R_3		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		

Q Validate the given input string by LR(0) parser?



aabab



abba

16/07/25 L10

LR(0)

abab

Stack	Input	Action
$0	abab$	Shift S_3
$0a3	bab$	Shift S_4
$0a3b4	ab$	Reduce by R3
$0a3A6	ab$	Reduce by R_2
$0A2	ab$	Shift S_3
$0A2a3	b$	Shift S_4
$0A2a3b4	$	Reduce by R3
$0A2a3A6	$	Reduce by R2
$0A2A5	$	~~Shift~~ S_5 Reduce by R_1
$0S1	$	Accept

16/07/25

L10

* validate the string abba

abba

Stack	Input	Action
$0	abba$	Shift S_3
$0a3	bba$	Shif S_4
$0a3b4	ba$	Reduce by R_3
$0a3A6	ba$	Reduce by R_2
$0A2	ba$	Shift S_4
$0A2b4	a$	Reduce by R_3
$0A2A5	a$	Reduce by R_1
$0S1	a$	Not accepted.

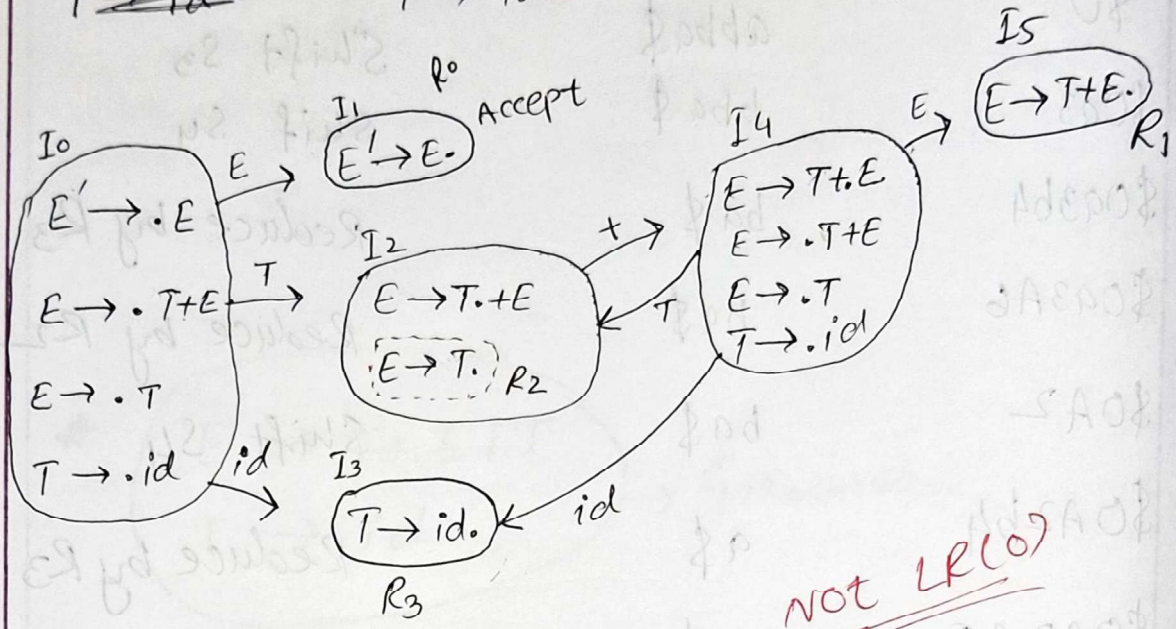
* Check wether the given grammar is LR(0) or Not?

~~E → E+T | T~~ ① ②

~~T → id~~ ③

$E \to T+E \,|\, T$ (1, 2)

$T \to id$ (3)



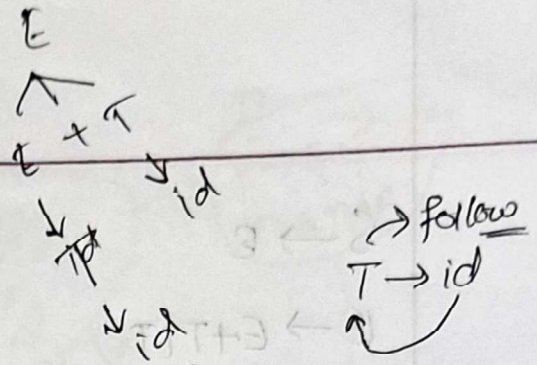
NOT LR(0)

LR(0) Parsing Table:

States	Action			GOTO	
	+	id	$	E	T
I_0		S_3		1	2
I_1			Accept		
I_2	S_4/R_2	R_2	R_2		
I_3	R_3	R_3	R_3		
I_4		S_3		5	2
I_5	R_1	R_1	R_1		

23/07/25 L11

SLR(1) Parsing



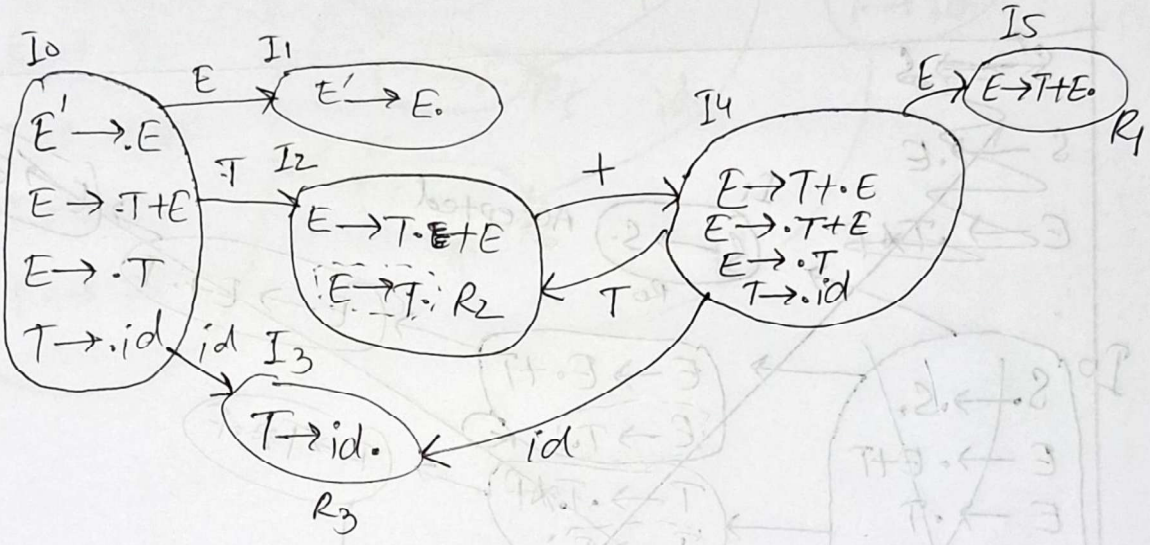
follow (T) = { +, $ }
Follow (E) = { $ }.

$E \rightarrow T+E \mid T$

$T \rightarrow id$

input : id+id

same here



State	Action			GOTO	
	id	+	$	E	T
I_0	S_3			1	2
I_1			Accept		
I_2		S_4	R_2		
I_3		R_3	R_3		
I_4	S_3			5	2
I_5			R_1		

$S \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

$S \rightarrow E$

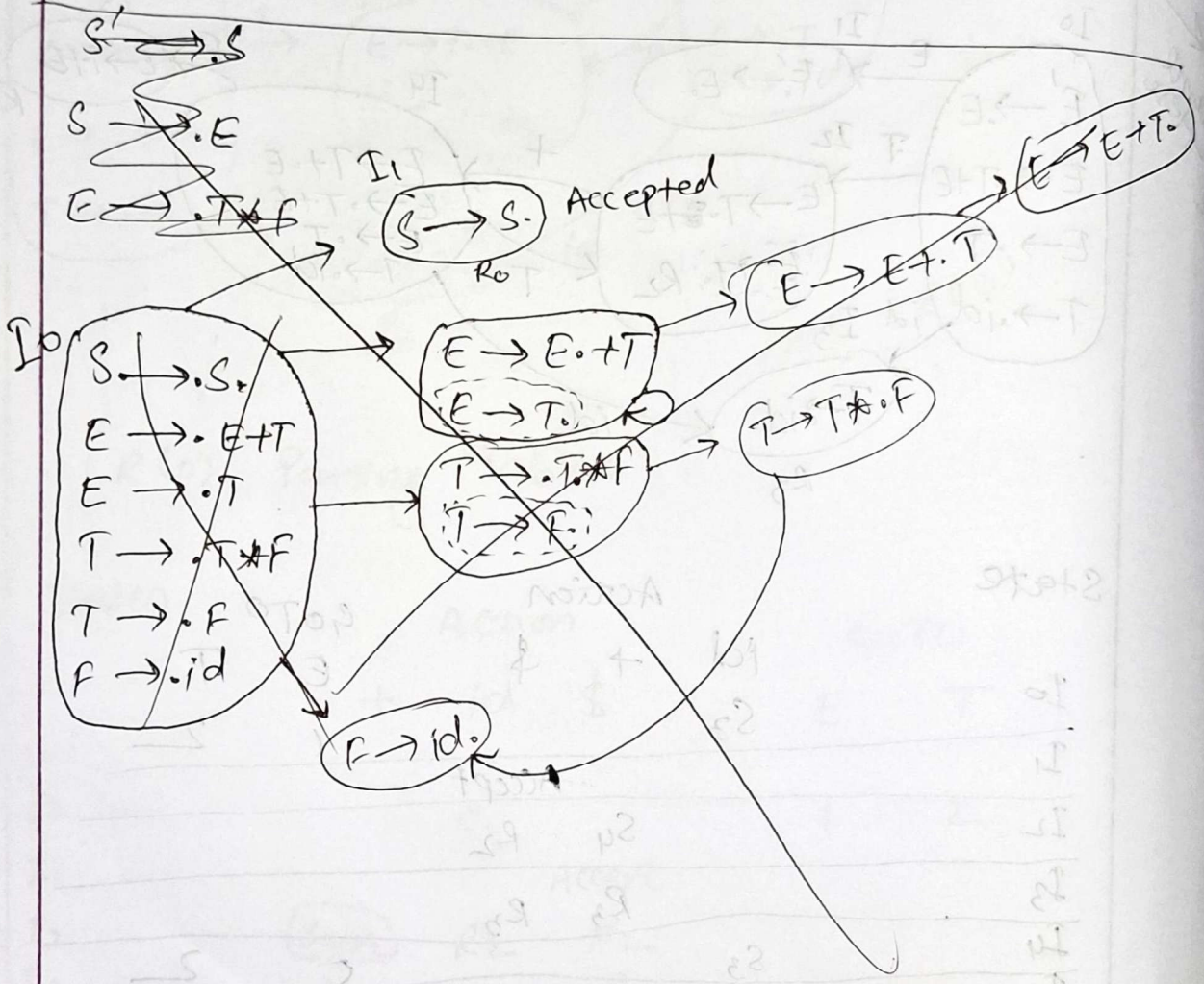
$E \rightarrow E+T$

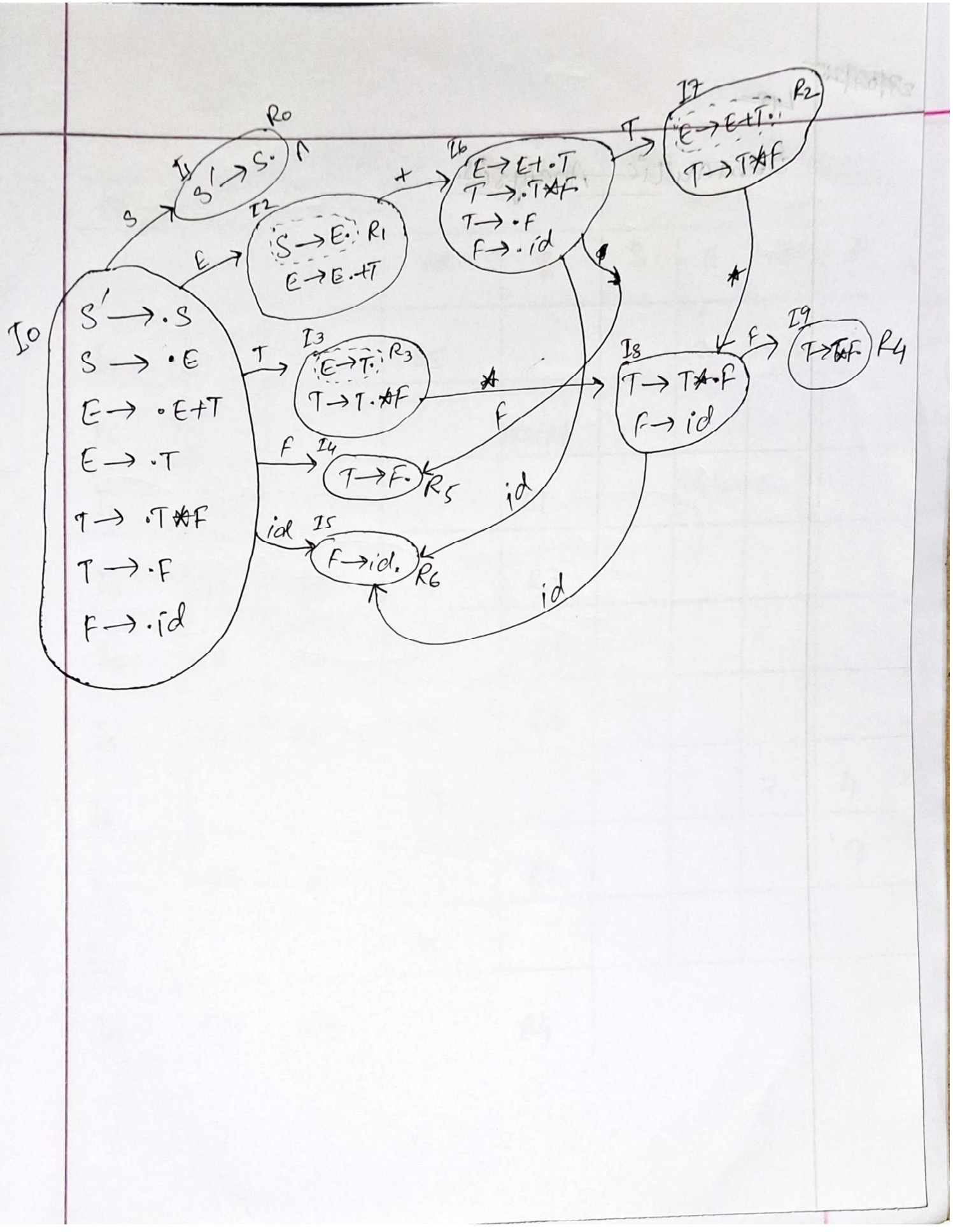
$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow id$




I0
S' → .S
S → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .id
S
E
T
F
id
I1
S' → S.
R0
I2
S → E. R1
E → E.+T
+
I6
E → E+.T
T → .T*F
T → .F
F → .id
I7
E → E+T. R2
T → T*F
I3
E → T. R3
T → T.*F
*
I8
T → T*.F
F → id
I9
T → T*F. R4
I4
T → F. R5
I5
F → id. R6

SLR(1) Parsing Table

States	Action				GOTO			
	+	*	id	$	S	E	T	F
I_0			S5		1	2	3	4
I_1				Accept				
I_2	S6			R1				
I_3	R3	S8		R3				
I_4	R5	R5		R5				
I_5	R6	R6		R6				
I_6			S5				7	4
I_7	R2	S8		R2				9
I_8			S5					
I_9	R4	R4		R4				

27/07/25 L12

SDD → Syntax Directed Defination.

Semantic Analysis.

Semantic Rule.

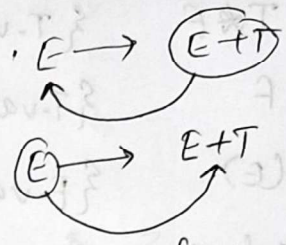
$$E \rightarrow E+T \qquad \{ E.val = E.val + T.val \}$$

$\$\$ \quad \$1 \ \$2 \ \$3$

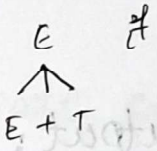
↑ attribute

Synthesized attribute

Inherited "



take from parent.



30/7/25 L-13

Syntax directed definition

SDD → How the expressions are evaluated/ calculated.

$E \rightarrow E+T$	$\{E.val = E.val + T.val\}$
$E \rightarrow T$	$\{E.val = T.val\}$
$T \rightarrow T*F$	$\{T.val = T.val * F.val\}$
$T \rightarrow F$	$\{T.val = F.val\}$
$F \rightarrow (E)$	$\{F.val = E.val\}$
$F \rightarrow$ digit	$\{F.val = digit.lexval\}$

Output: Data flow within parse tree.

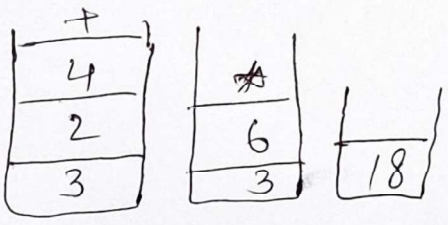
Syntax Directed Translation

SDT → what is done & when

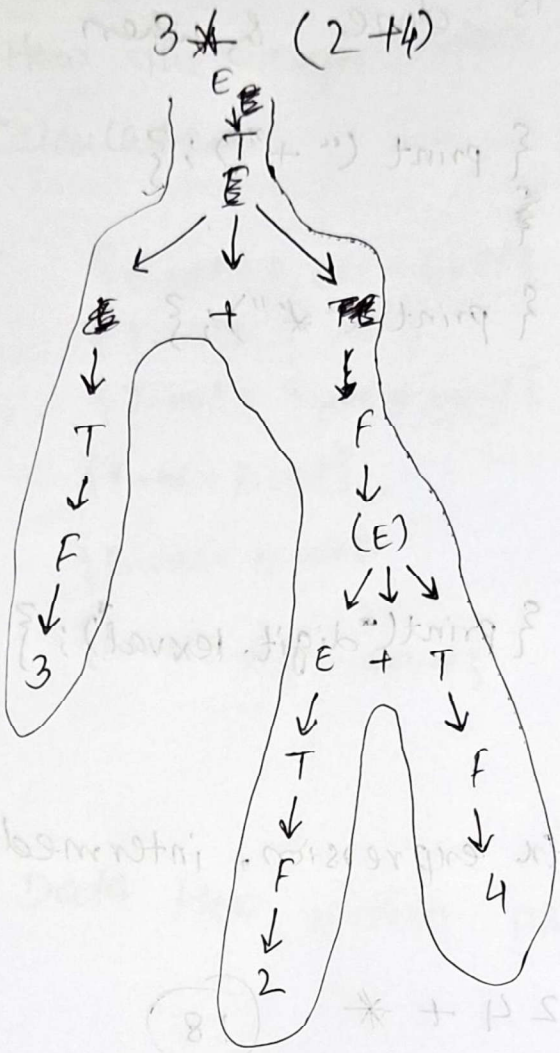
$E \rightarrow E+T$	{ print ("+"); }
$E \rightarrow T$	
$T \rightarrow T*F$	{ print ("*"); }
$T \rightarrow F$	
$F \rightarrow (E)$	
$F \rightarrow digit$	{ print("digit.lexval"); }

Output : Postfix expression, intermediate code

output : 3 2 4 + * (18)



SDD & SDT Differences

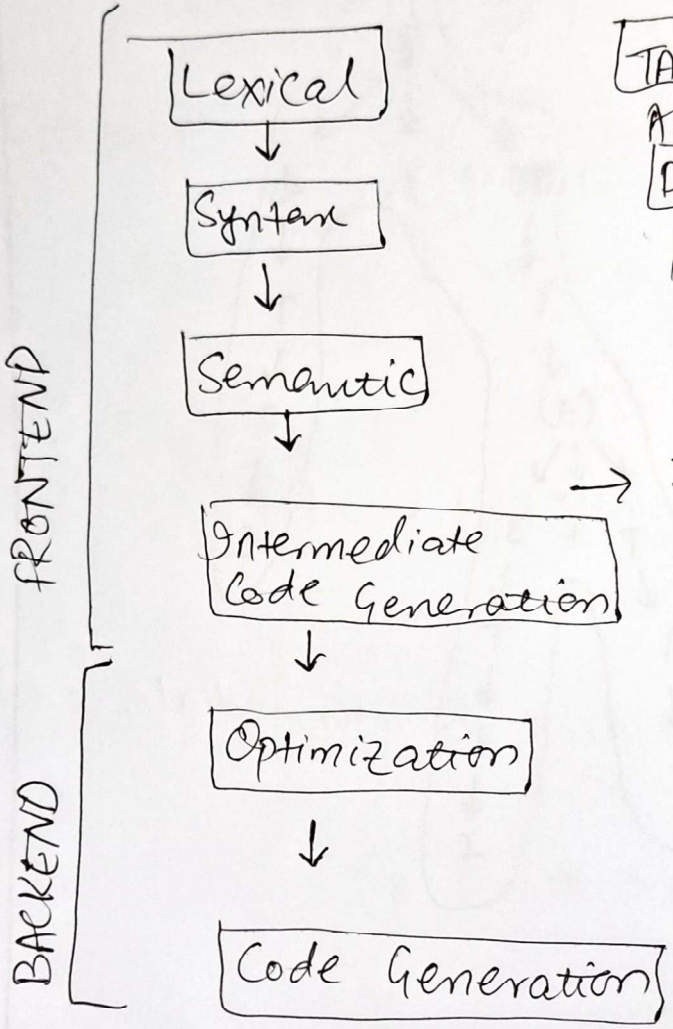


EXAM * SDD VS SDT

LL(1)
LR(0)
SLR(1)) 18 marks.

3/8/26 L-14

Intermediate Code Generation



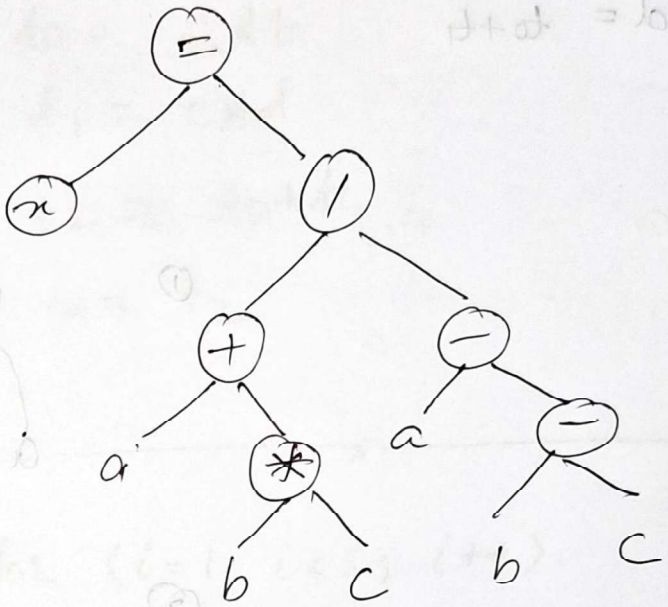
TAC
AST
DAG ★
Postfix notation.

→ 70%

DAG → Directed ~~Assig~~ cyclic Graph

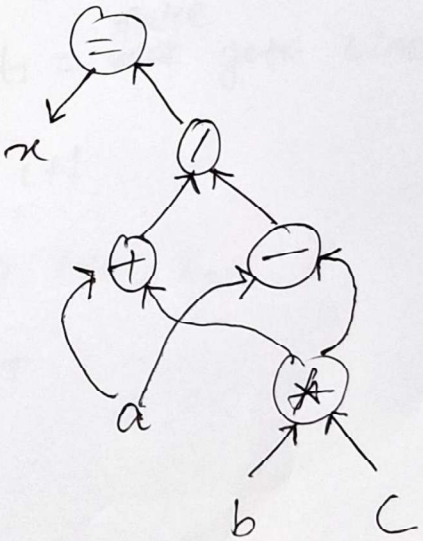
$x = (a + b * c) / (a - b * c)$

TAC



DAG

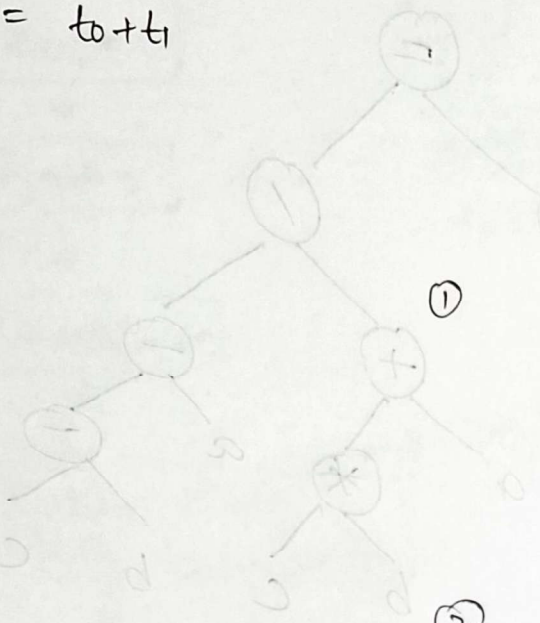
EXAM



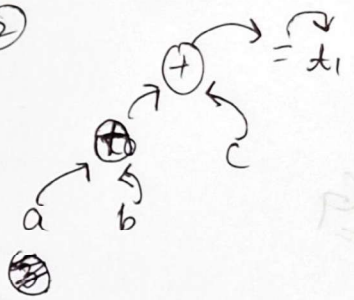
$t_0 = a+b$

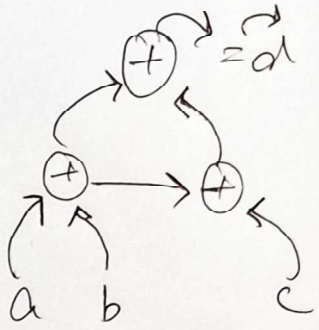
$t_1 = t_0 + c$

$d = t_0 + t_1$



① 

② 

③ 

TAC

$x = (a * b) + (c * d)$.

$t_0 = a * b$

$t_1 = c * d$

$t_2 = t_0 + t_1$

$x = t_2$

---

for (i=1; i<5; i++).

① $i = 1$

② $t_1 = (i < 5)$

③ if t_1 = ~~true~~ false goto Line 6

④ $i = i + 1$

⑤ goto Line 2

⑥ EXIT

Quadruple.

$$x = (a * b) + (c * d)$$

Operator	Arg 1	Arg 2	Result
*	a	b	t_0
*	c	d	t_1
+	t_0	t_1	t_2
=	t_2		x

Triples

#	operator	Arg 1	Arg 2
0	*	a	b
1	*	c	d
2	+	(0)	(1)
3	=	(2)	

56 → CT 2 EXAM

always Platform independant

VIVA

Code optimization

1. Compile time Evaluation
2. Common Sub-Expression
3. Code Movement
4. Dead code

(1)

Constant folding :-

Area (Circle) = πr^2

= 22/7 $\times r^2$ ✗

= 3.1415 $\times r^2$ ✓

Compile time evaluation

Constant Propagation :-

x = 5 * Area + 6

= 5 * 100 + 6.

(2)

$S_1 = 4 * i$

$S_2 = a[S_1]$

$S_3 = 4 * 1$

$S_4 = 4 * i$

$S_5 = n$

$S_6 = b[S_4] + S_5$

$S_6 = b[S_1] + S_5$

(3)

```
for (i=0; i<10; i++){
    print(i);
    a = c+d;
}
a = c+d;
```

move

(4)

```
i = 3
if (i == 1){
    ⋮
}
```

→ Dead code

Peephole optimization

↳ It is done on low level language.

Redundant Load Store:-

```
mov   R0, b
ADD   R0, c
MOV   a, R0      <- Redundant
MOV   R0, a
ADD   R0, c
MOV   d, R0
```

a = b+c
d = a+c → d = b+c+c.

↳ d = a+b+c

$x^2 = x * x$

$2x = x+x$ shift left

$x/2 =$ ~~x/2~~ shift right.