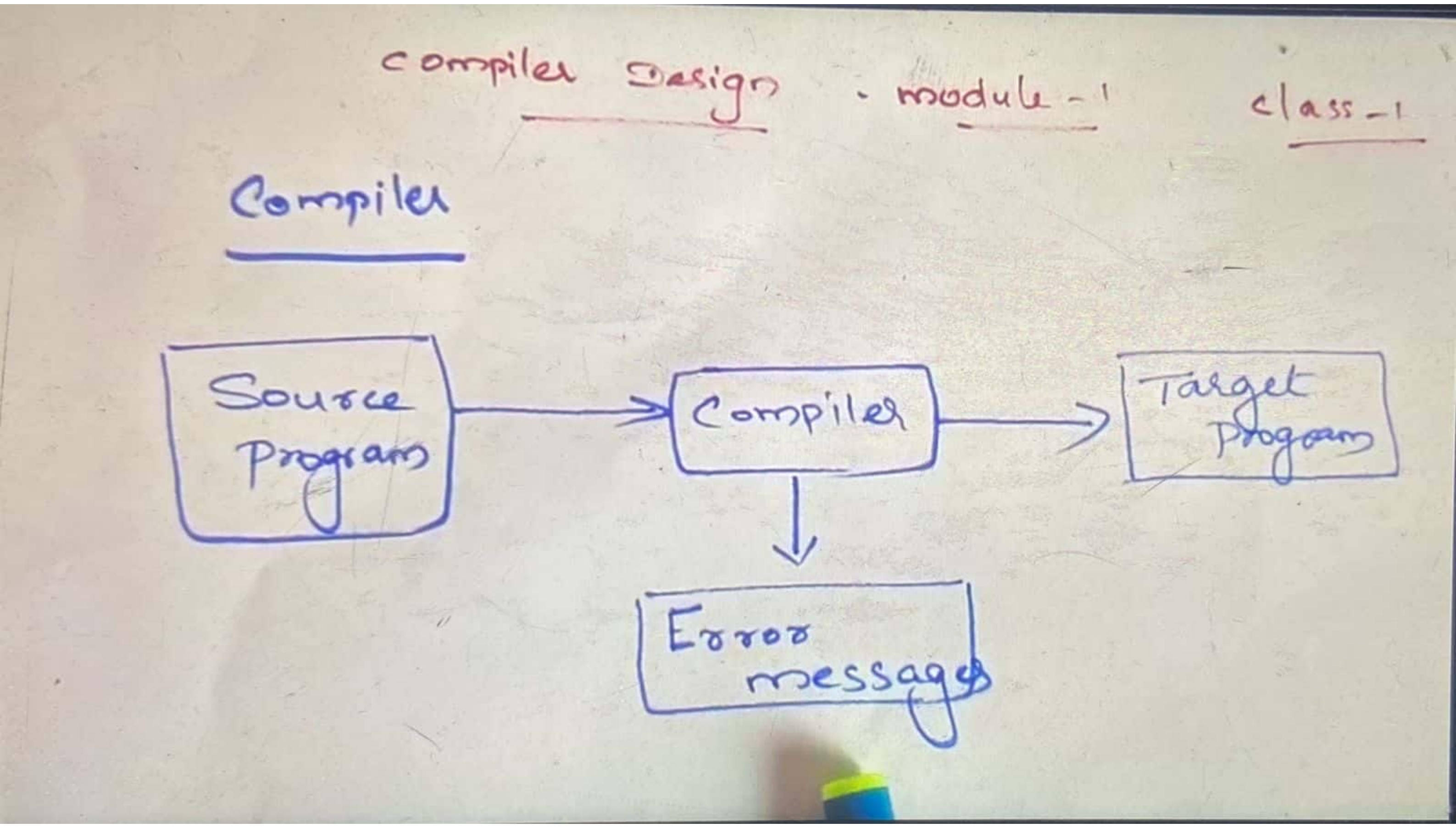
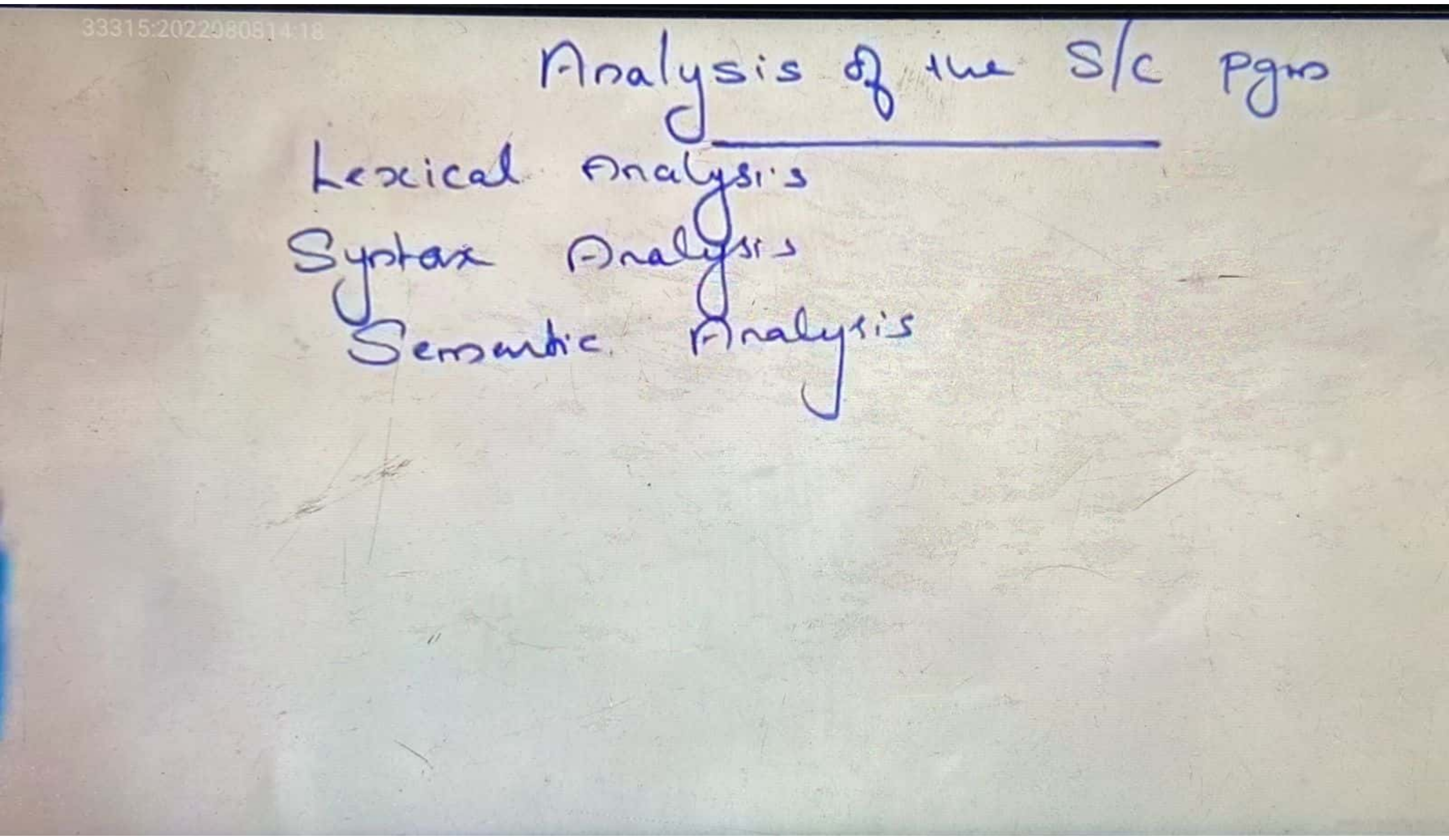


compiler Design . module -1 class -1

Compiler



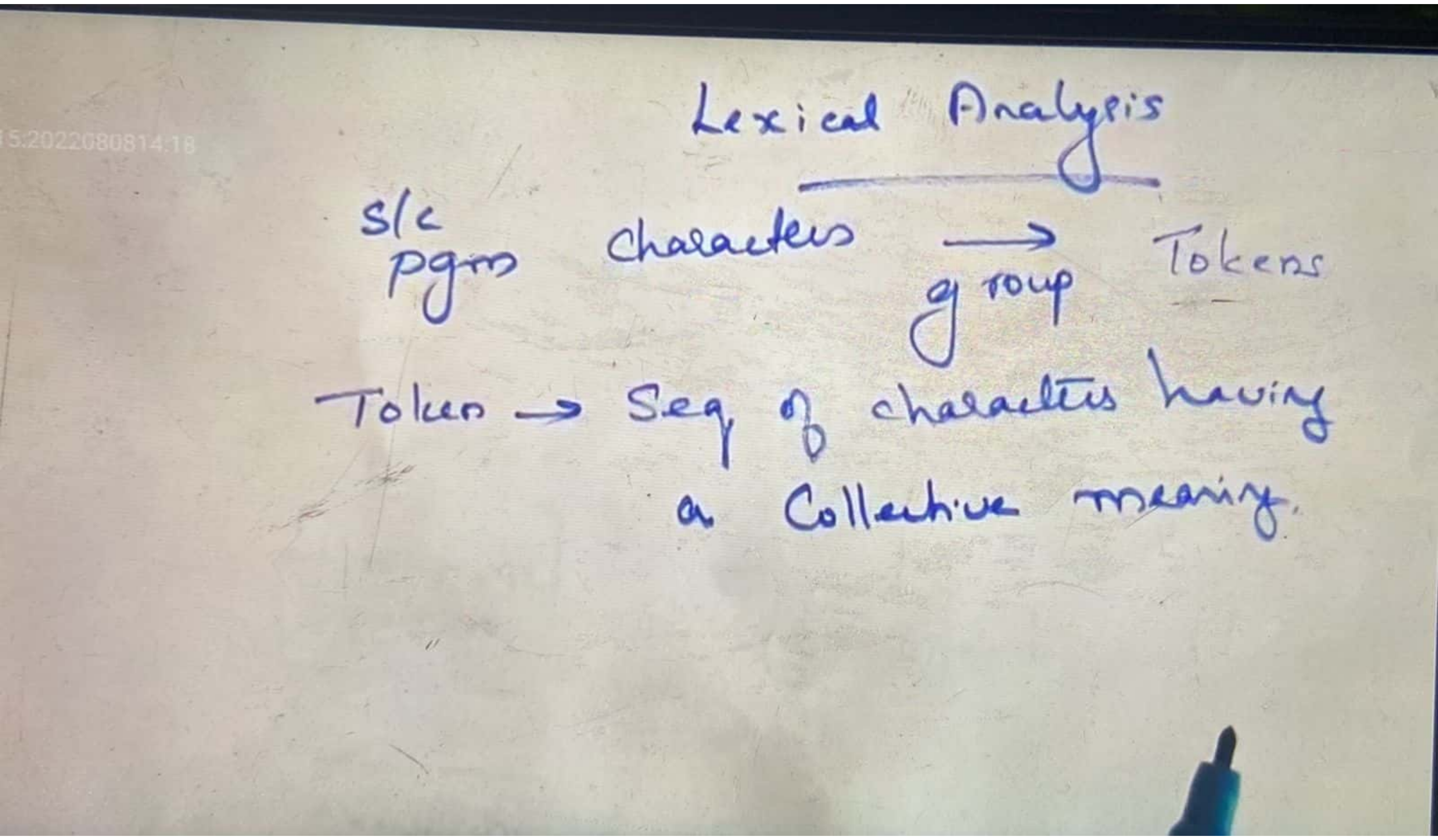
33315:2022080814:18

Analysis of the S/C Pgm

Lexical Analysis

Syntax Analysis

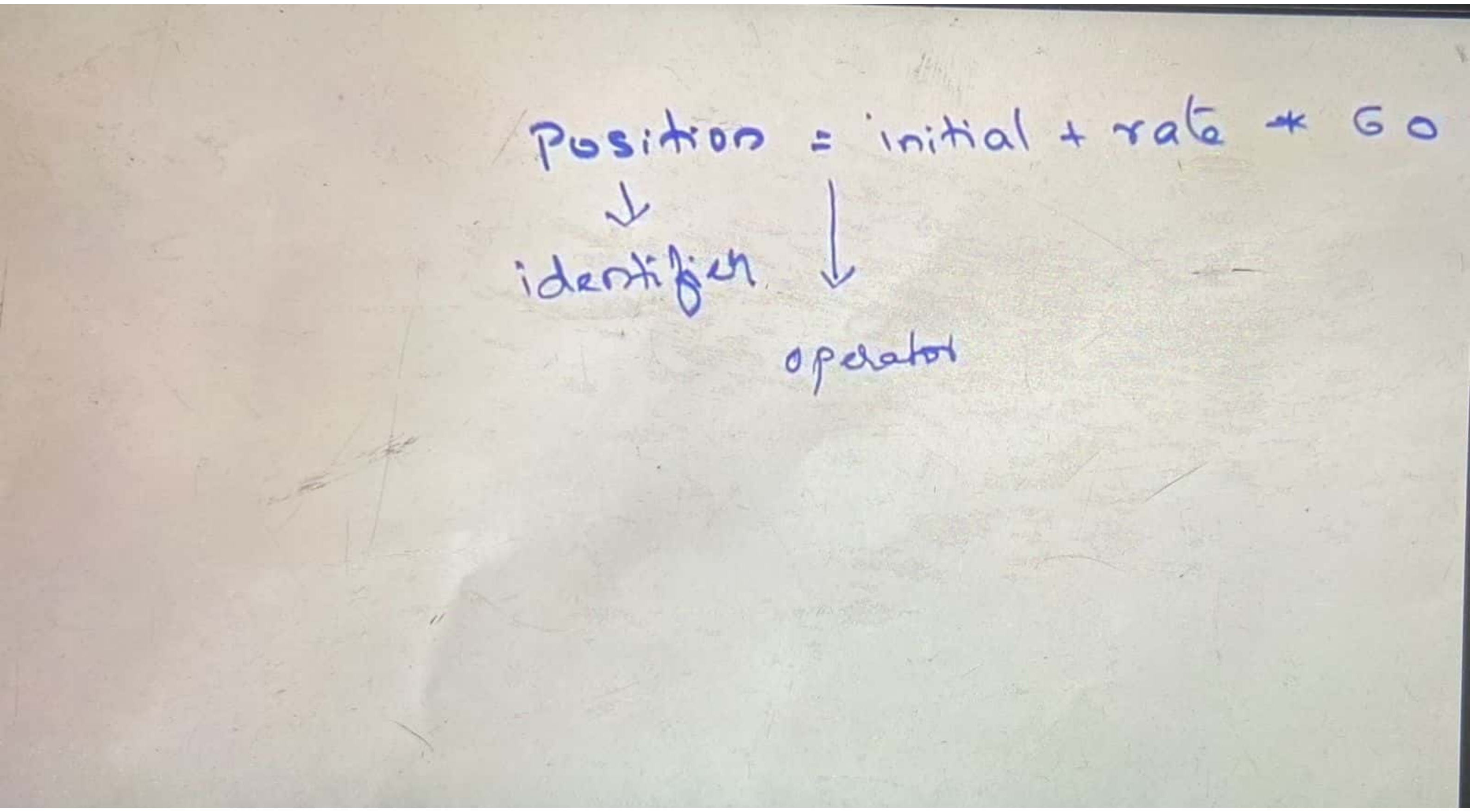
Semantic Analysis



Lexical Analysis

s/c
pgm characters $\xrightarrow[\text{group}]{}$ Tokens

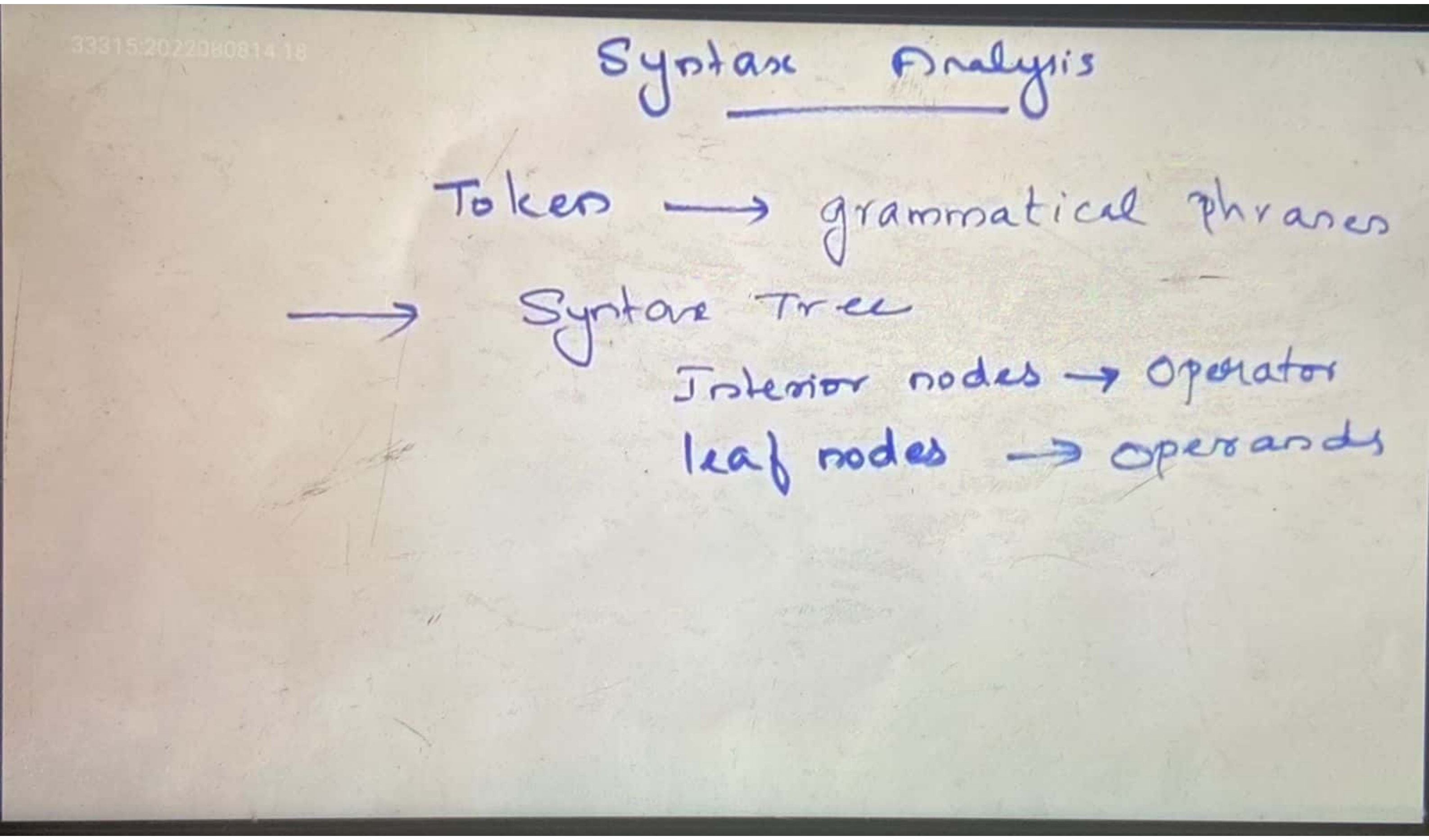
Token → Seq of characters having a Collective meaning.



Position = initial + rate * 60

↓ identifier

↓ operator



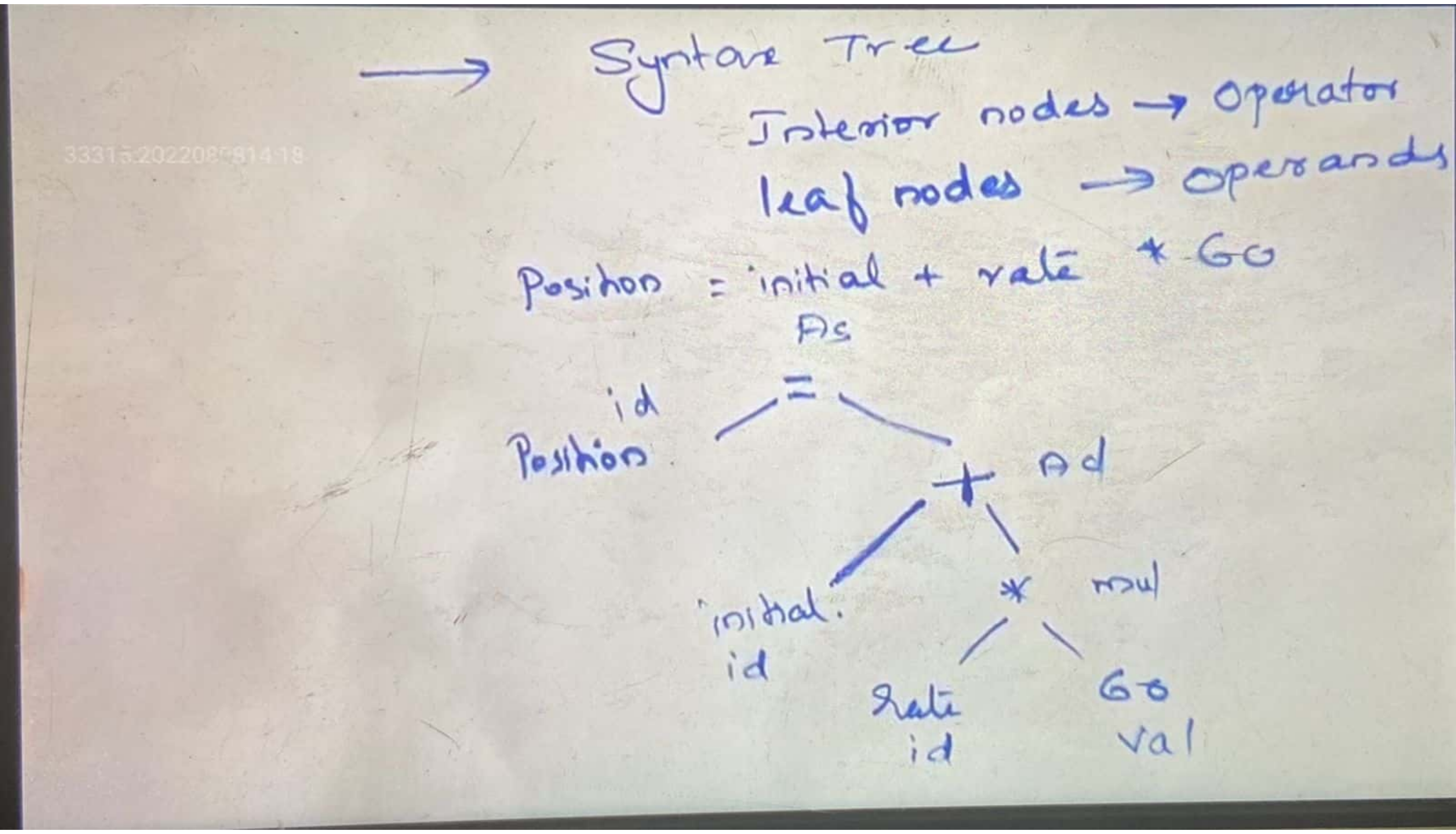
Syntax Analysis

Token → grammatical phrases

→ Syntax Tree

Interior nodes → Operator

leaf nodes → operands



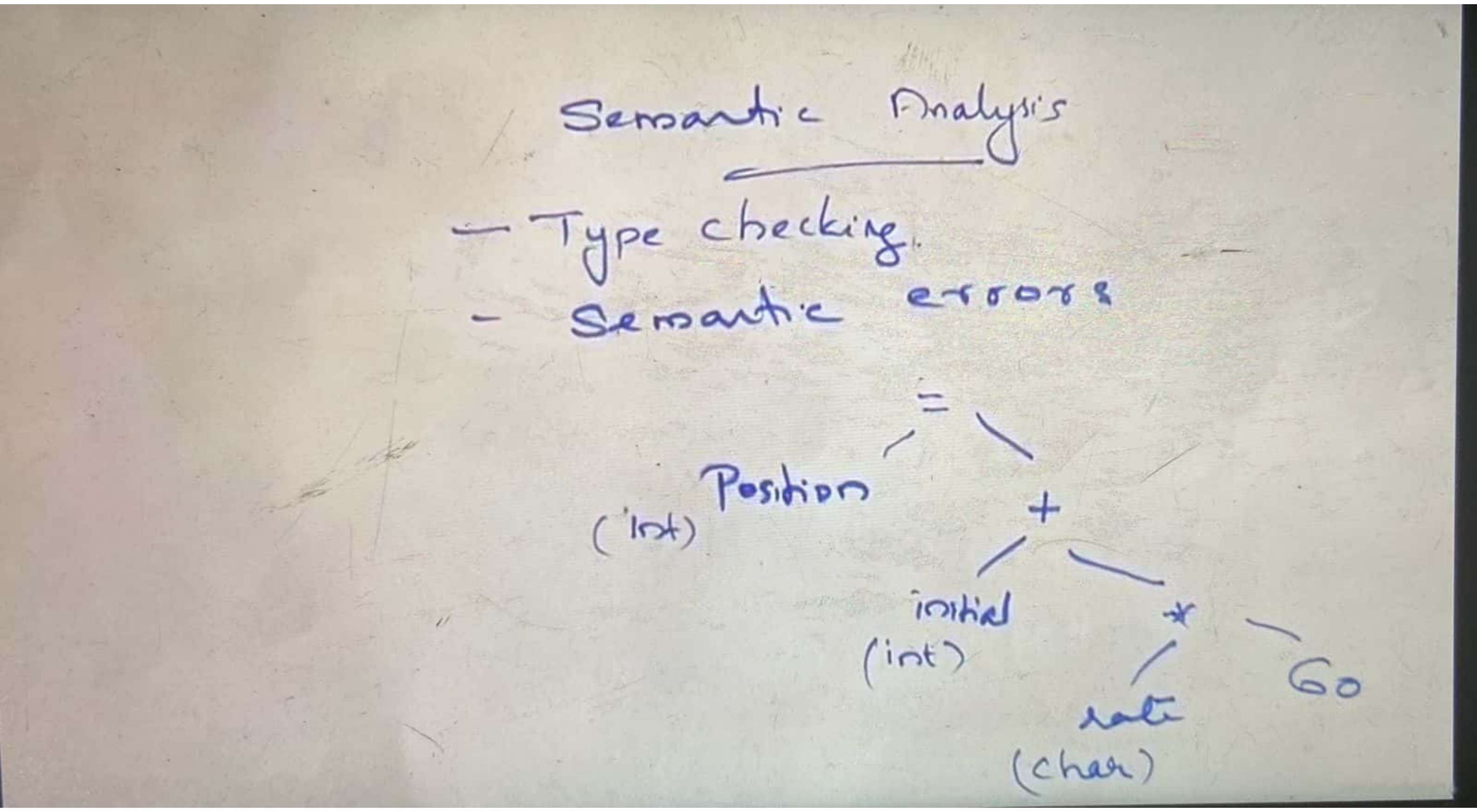
→ Syntax Tree

Interior nodes → operator

leaf nodes → operands

Position = initial + rate * 60

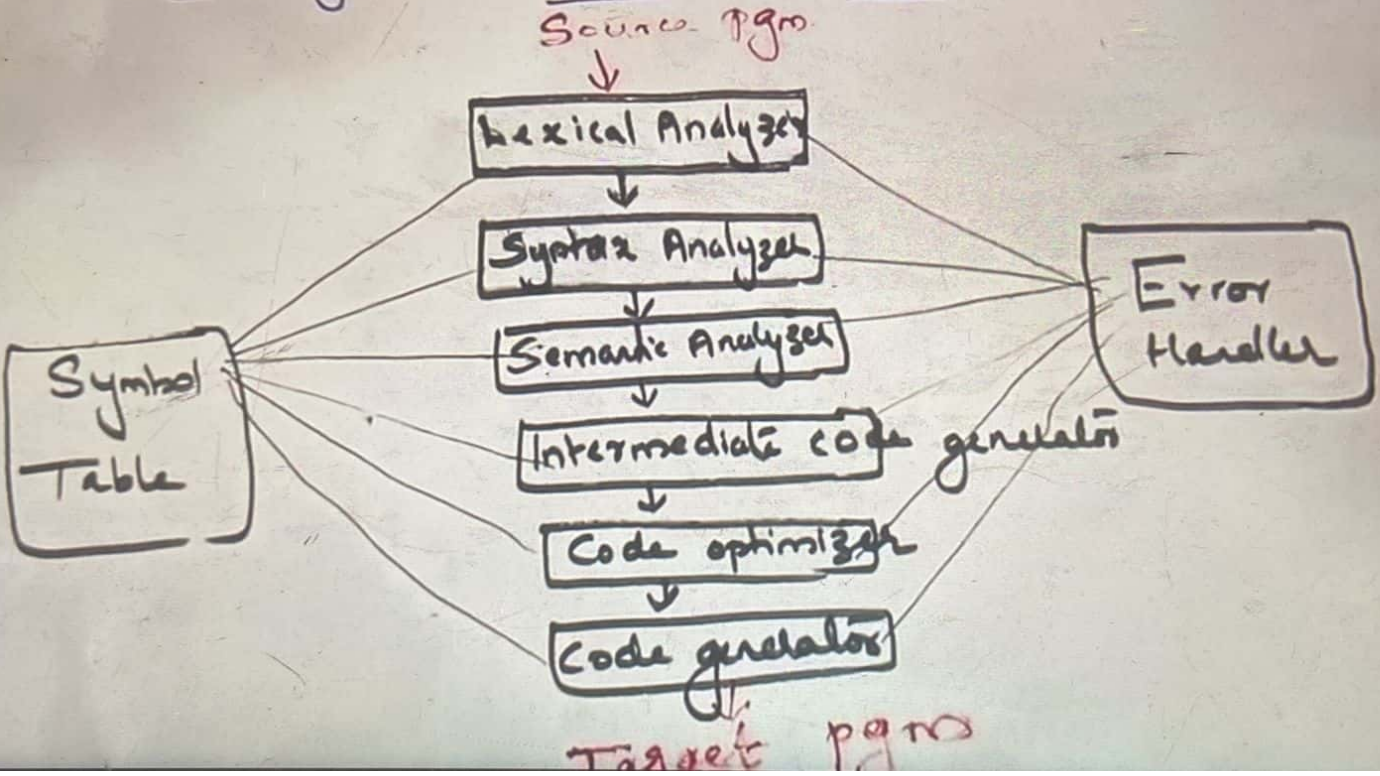
As
=
id
Position
+
Ad
initial
id
*
mul
Rate
id
60
val

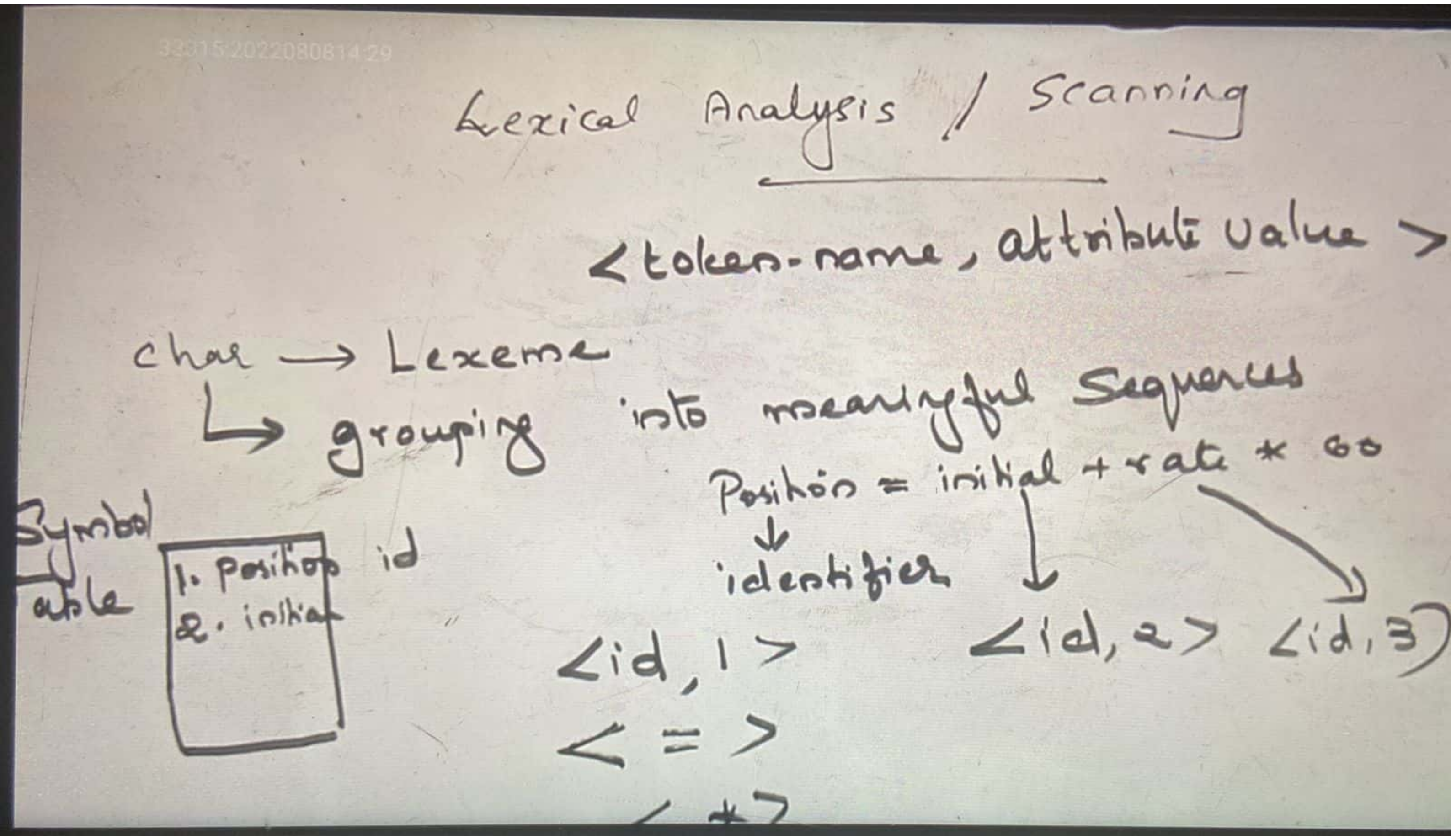


Semantic Analysis

- Type checking.
- Semantic errors

Compiler Design Module I class 2





Lexical Analysis / Scanning

<token-name, attribute value >

char → Lexeme

↳ grouping into meaningful sequences

Position = initial + rate * 60

Position ↓ identifier

<id, 1> <id, 2> <id, 3>

<=>

<*>

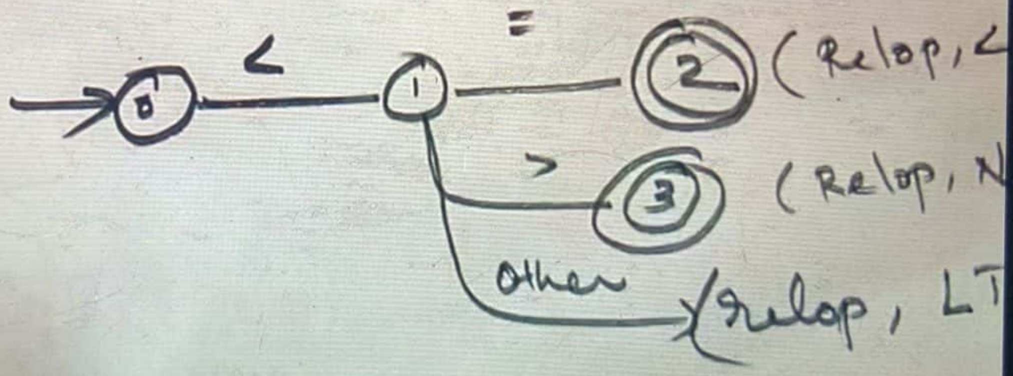
Symbol table

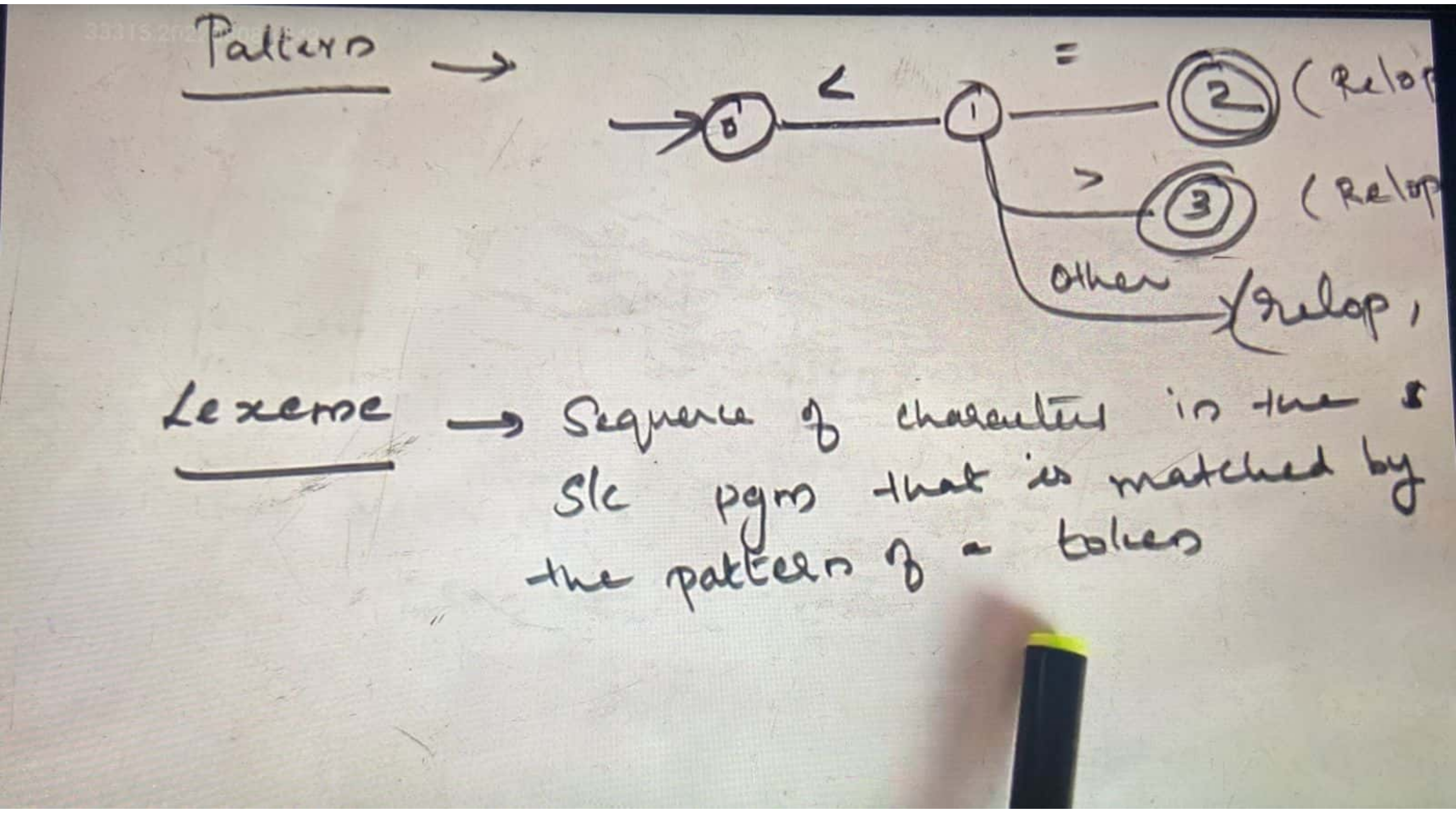
1. position id
2. initial

Token → seq of characters that can be treated a single logical entity.

eg) Id, keyword, operator, special symbols, constants.

Pattern →




0
<
1
=
2
(Relop
>
3
(Relop
other
(relop,

Pattern →

Lexeme → Sequence of characters in the s s/c pgm that is matched by the pattern of a token

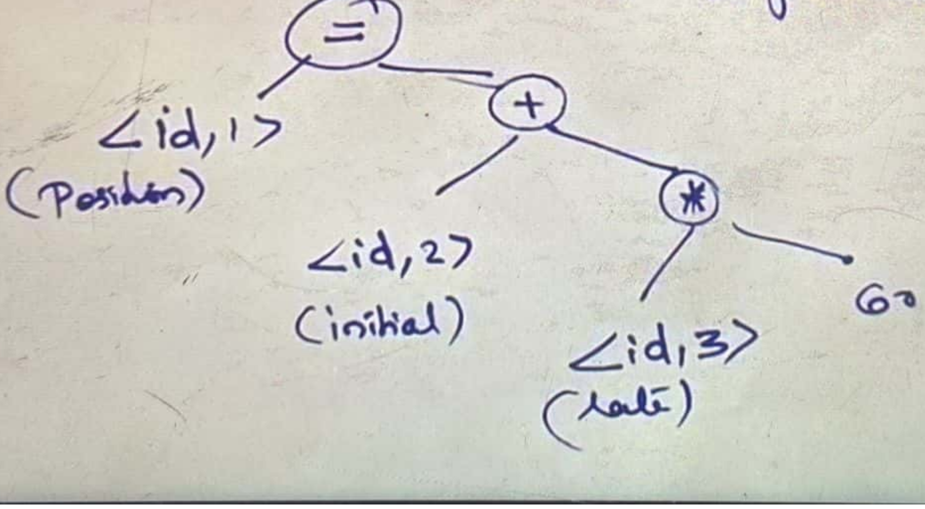
Compiler Design   module-1   class-3

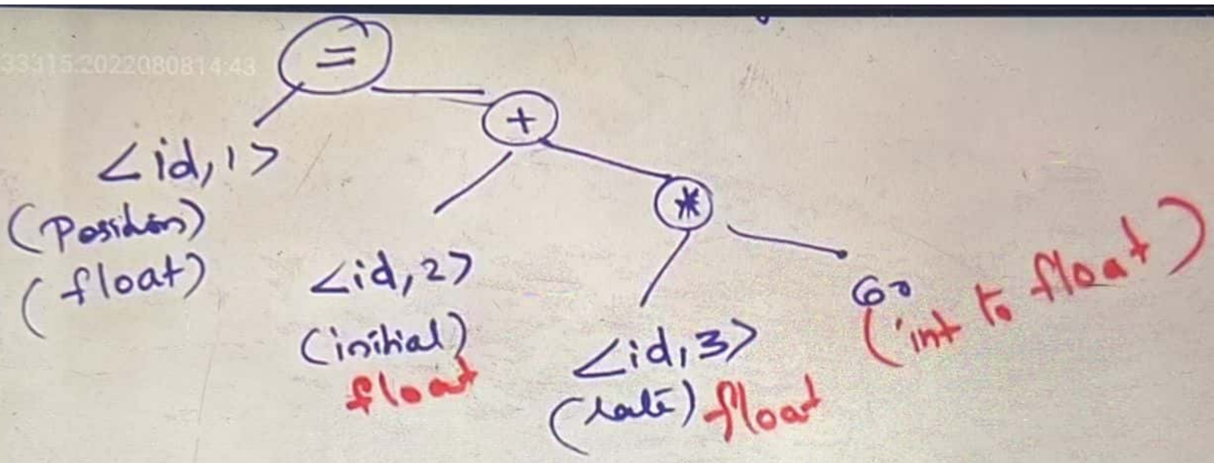
Syntax Analysis / Parsing

Syntax Tree

interior nodes
↓
operator

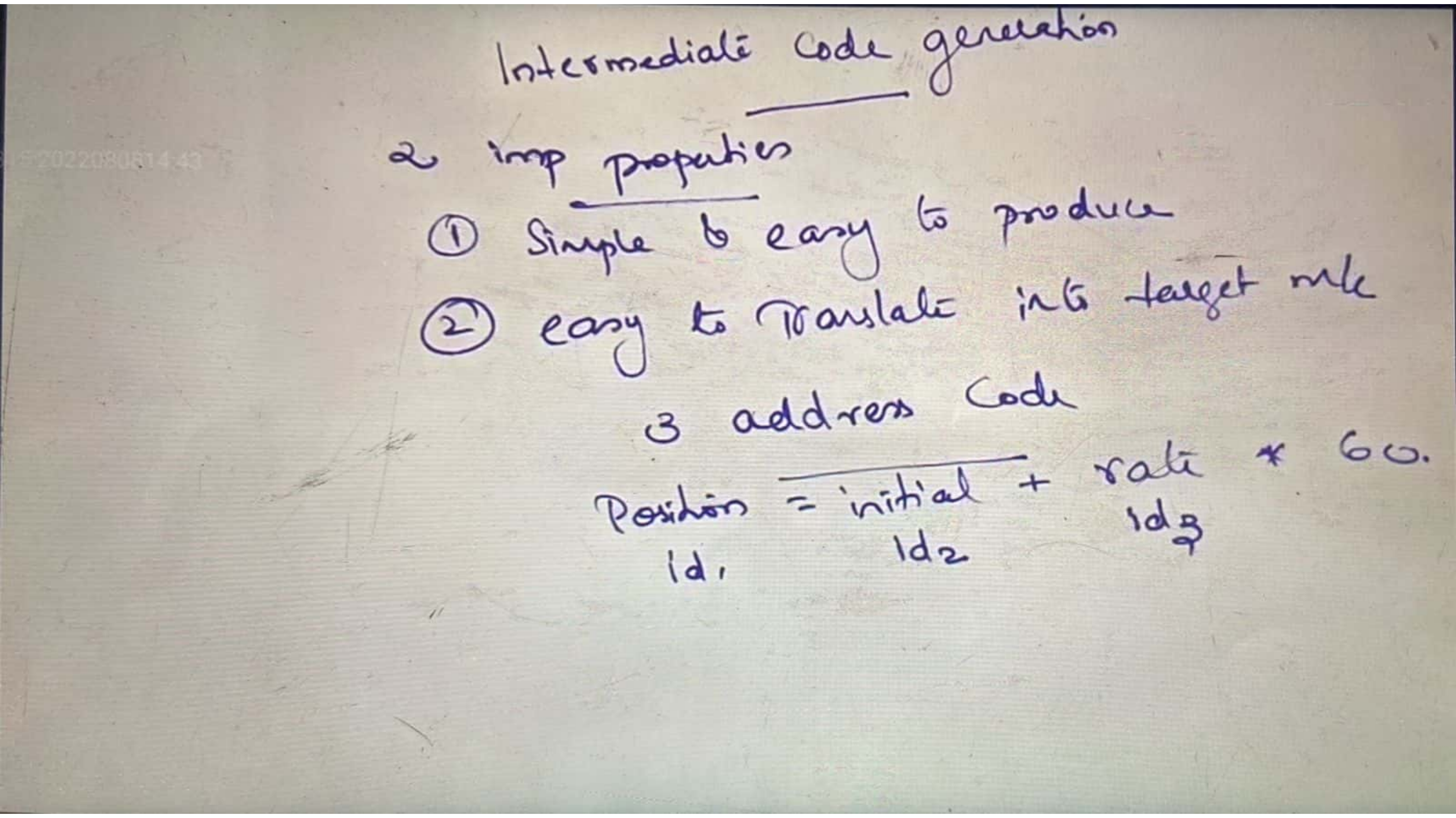
leaf nodes → Operands





Semantic Analysis

Type Conversion



Intermediate code generation

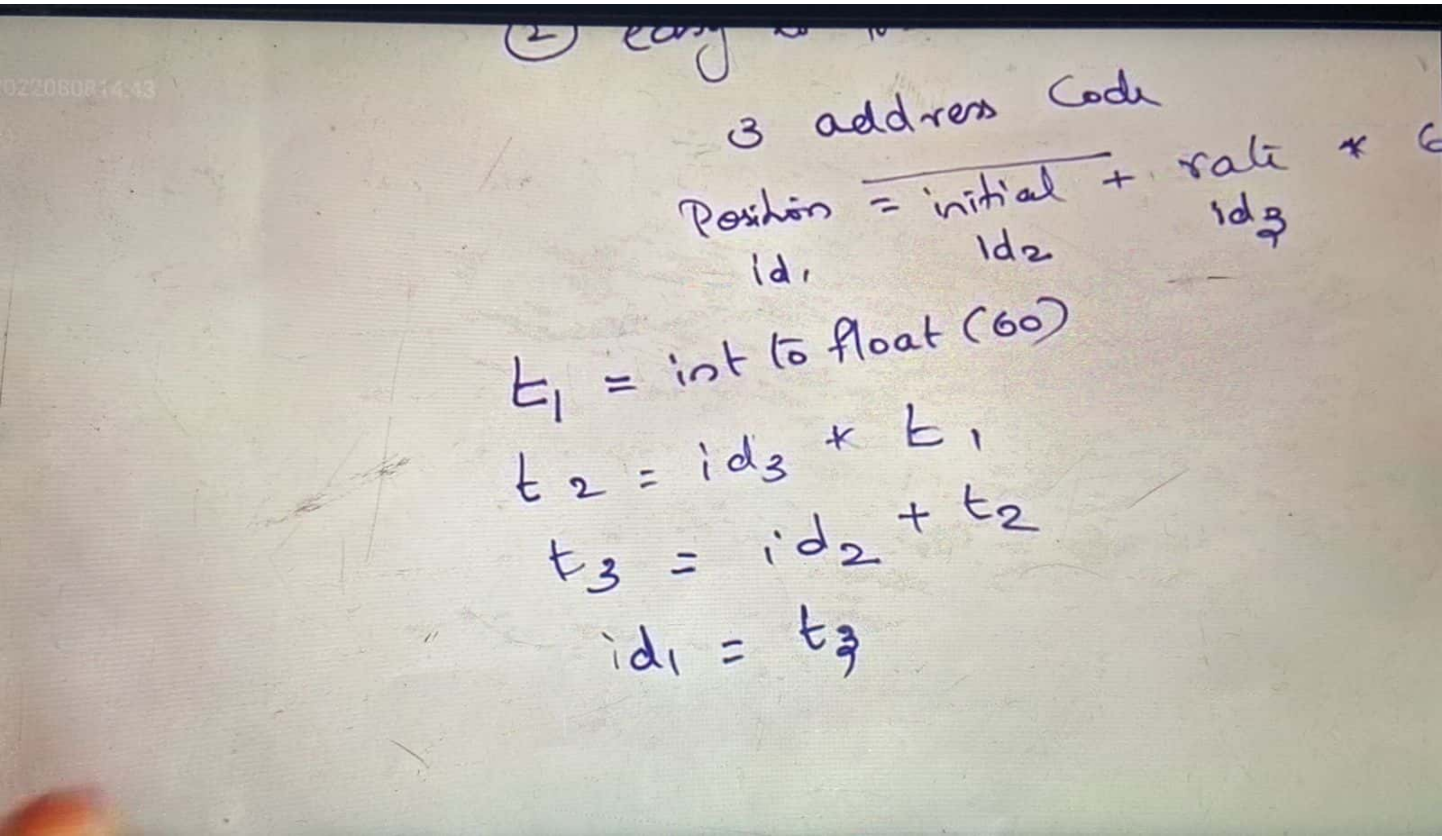
2 imp properties

① Simple & easy to produce

② easy to Translate into target m/c

3 address Code

$\underset{id_1}{\text{Position}} = \underset{id_2}{\text{initial}} + \underset{id_3}{\text{rate}} * 60.$



(2) easy to

3 address Code

Position = initial + rate * 6

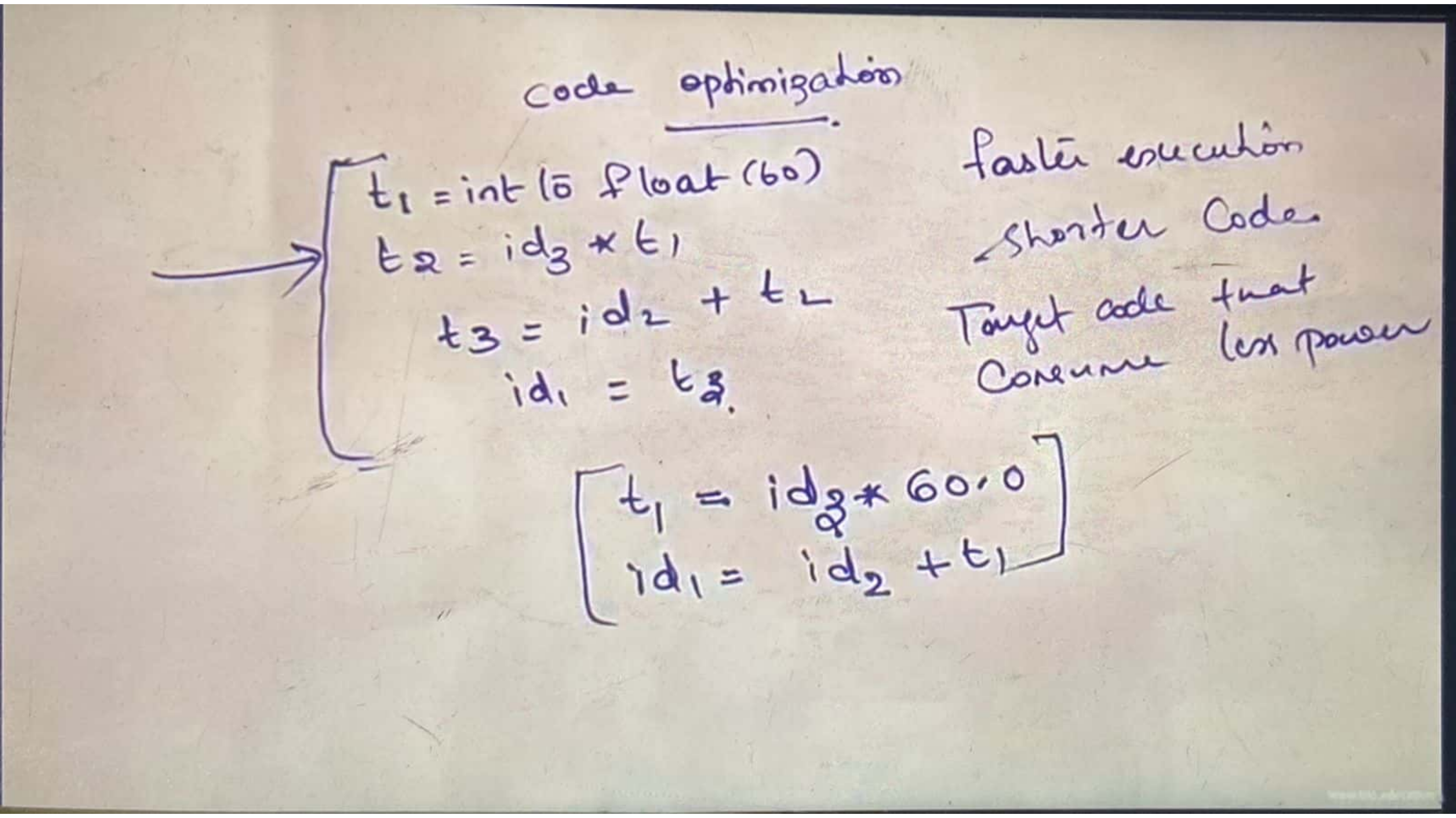
id_1 id_2 id_3

t_1 = int to float (60)

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$



code optimization

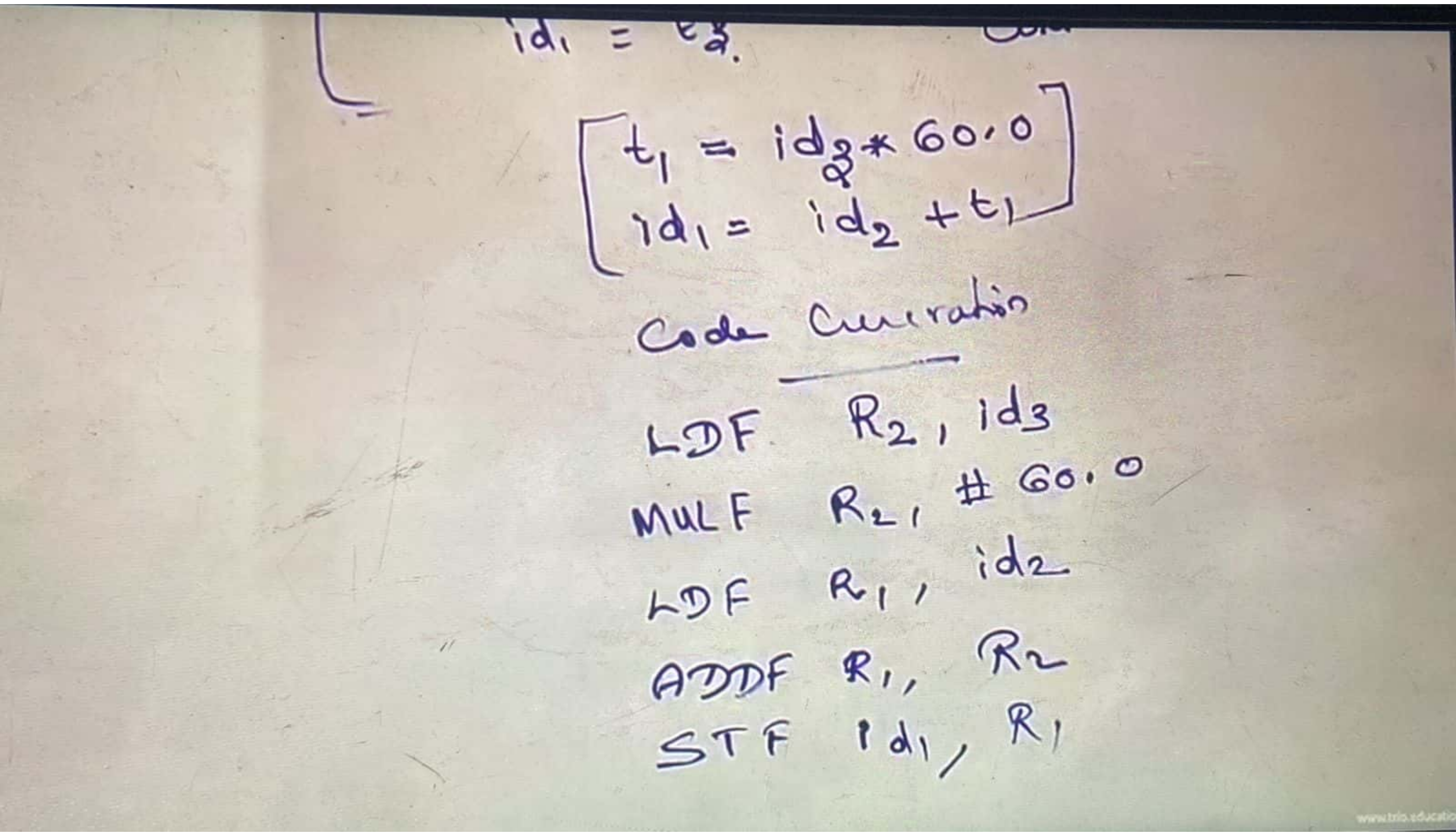
t_1 = inttofloat (60)
t_2 = $id_3 * t_1$
t_3 = $id_2 + t_2$
id_1 = t_3.

faster execution

Shorter Code

Target code that Consume less power

$t_1 = id_3 * 60.0$
$id_1 = id_2 + t_1$

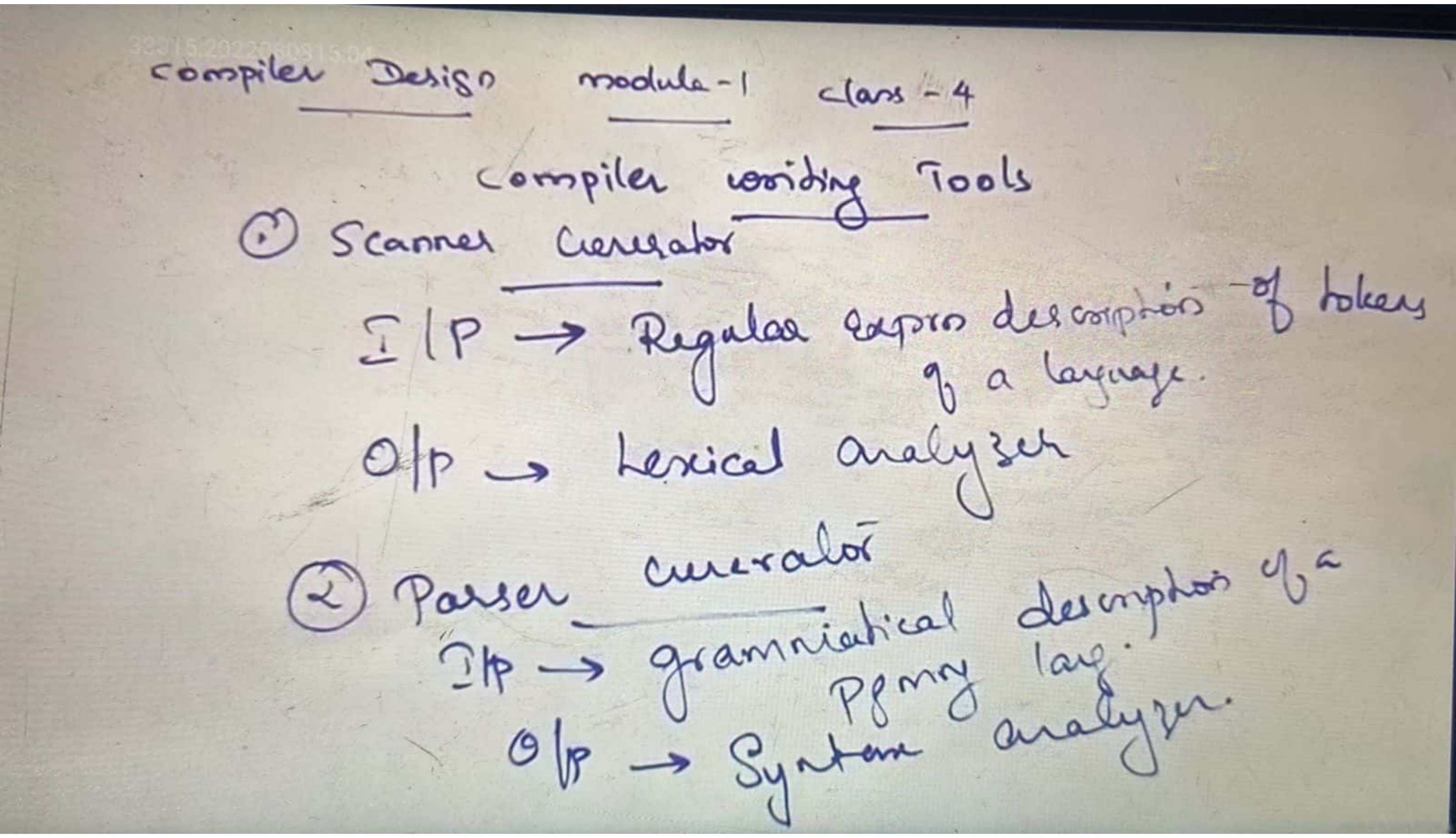


$id_1 = t_2.$

$$\left[\begin{array}{l} t_1 = id_3 * 60.0 \\ id_1 = id_2 + t_1 \end{array}\right]$$

Code Generation

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



compiler Design   module-1   class - 4

Compiler writing Tools

(1) Scanner Generator

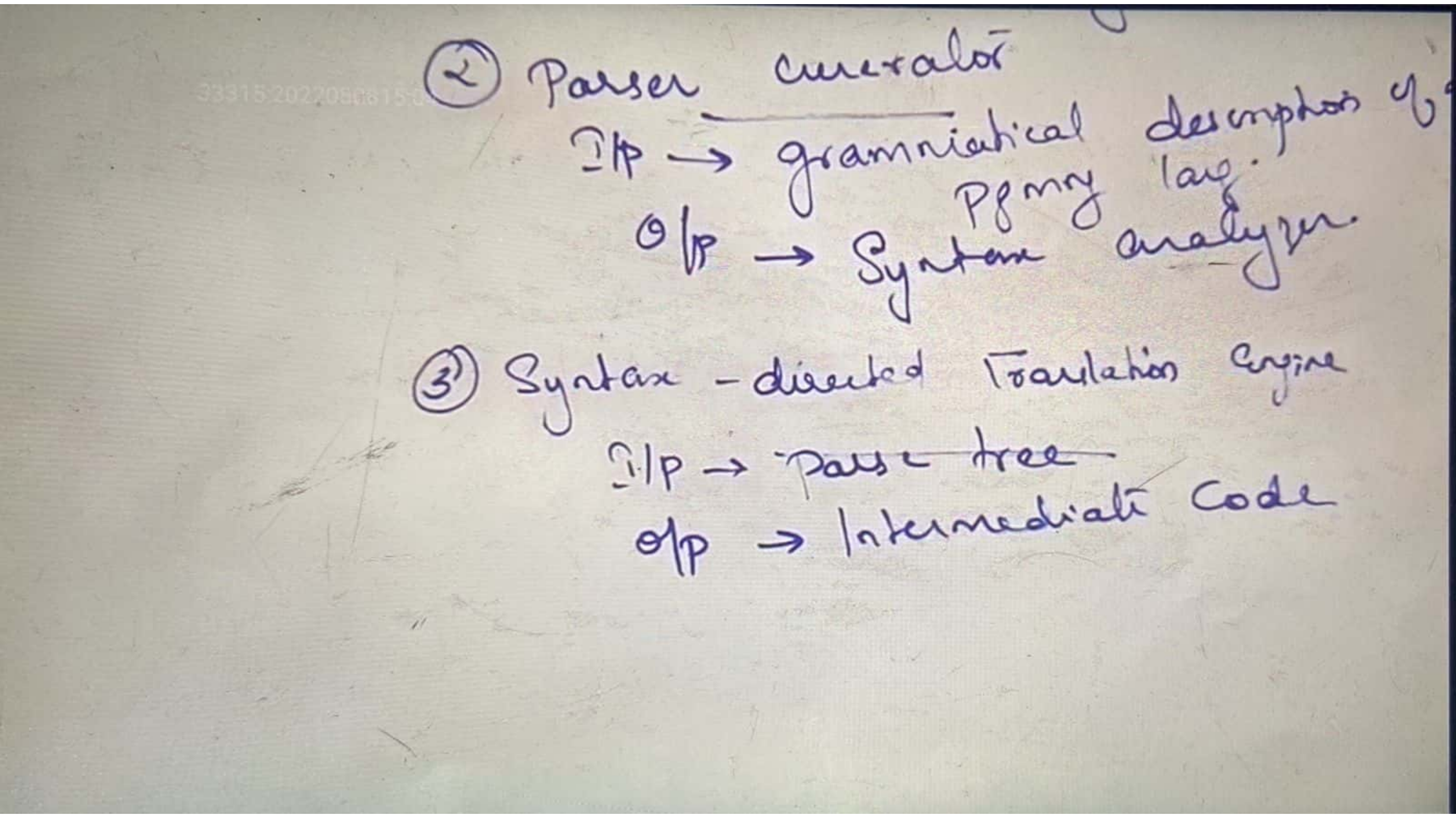
I/P → Regular expn description of tokens of a language.

O/P → Lexical analyzer

(2) Parser Generator

I/P → grammatical description of a pgmng lang.

O/P → Syntax analyzer.



② Parser generator

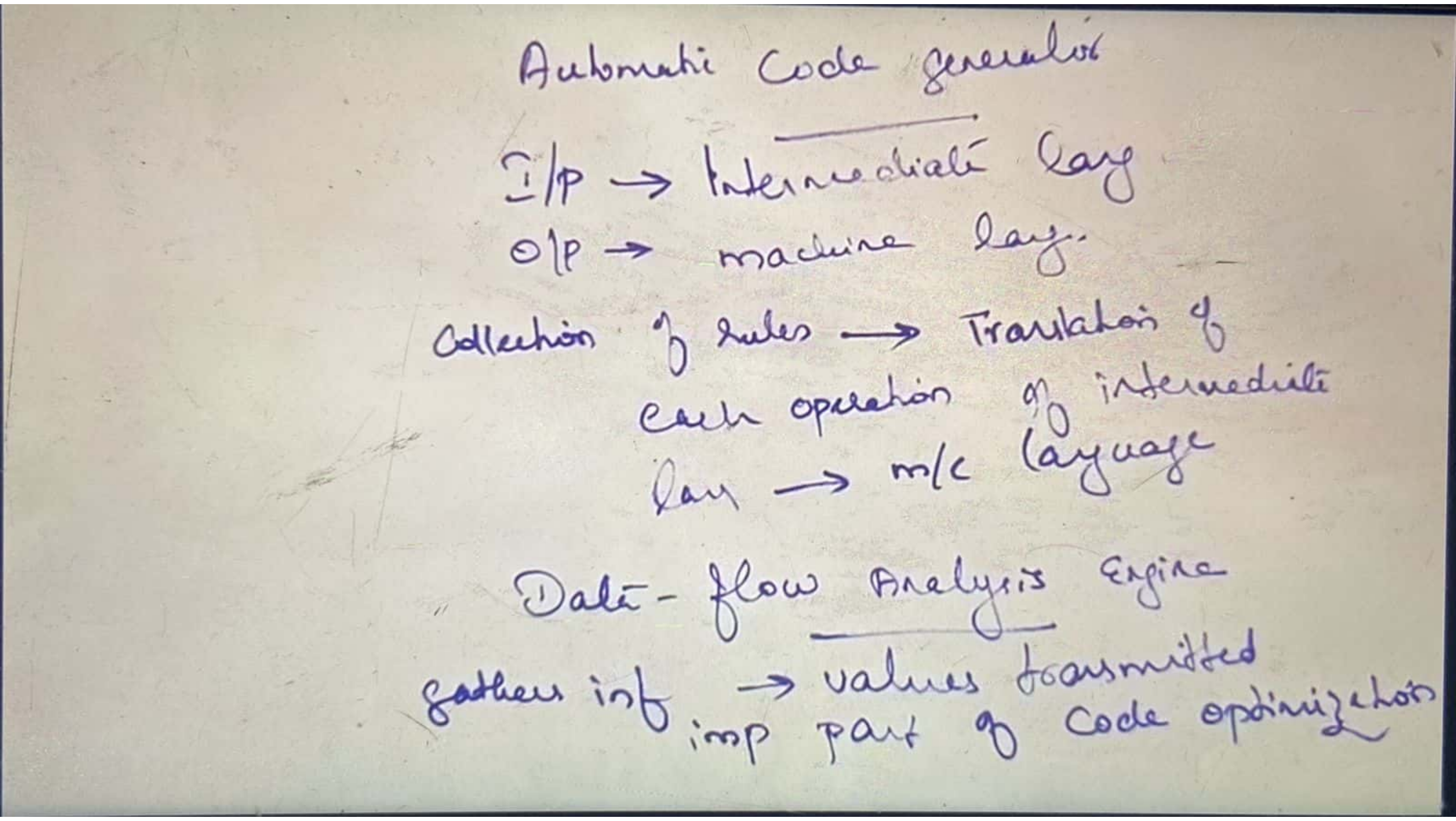
I/p → grammatical description of Pgmg lang.

O/p → Syntax analyzer.

③ Syntax - directed Translation Engine

I/p → parse tree

O/p → Intermediate Code



Automatic Code generator

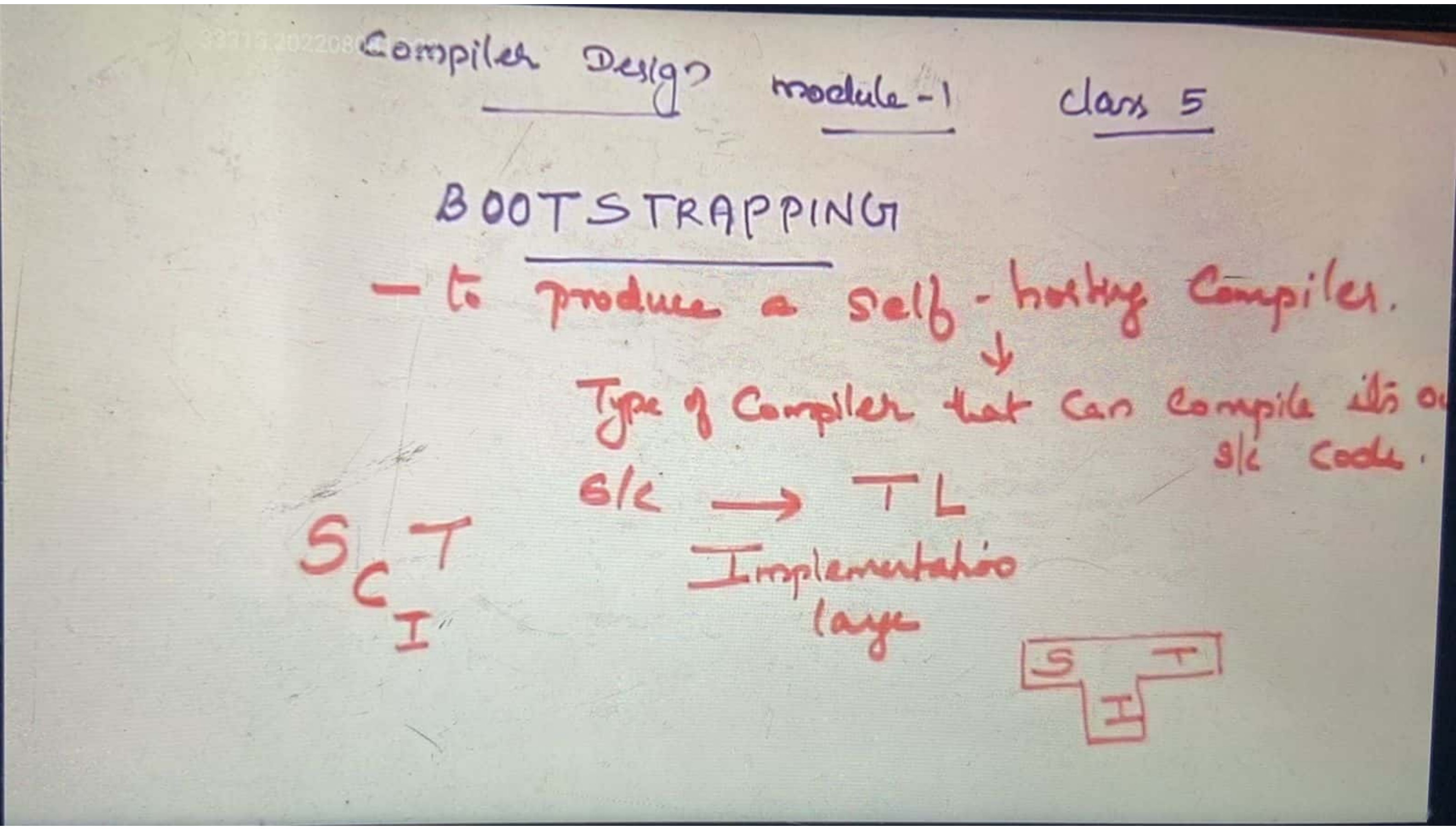
I/p → Intermediate lang

O/p → machine lang.

Collection of rules → Translation of each operation of intermediate lang → m/c language

Data-flow Analysis Engine

gather inf → values transmitted
imp part of code optimization



Compiler Design module-1 class 5

BOOTSTRAPPING

- to produce a self-hosting compiler.

↓

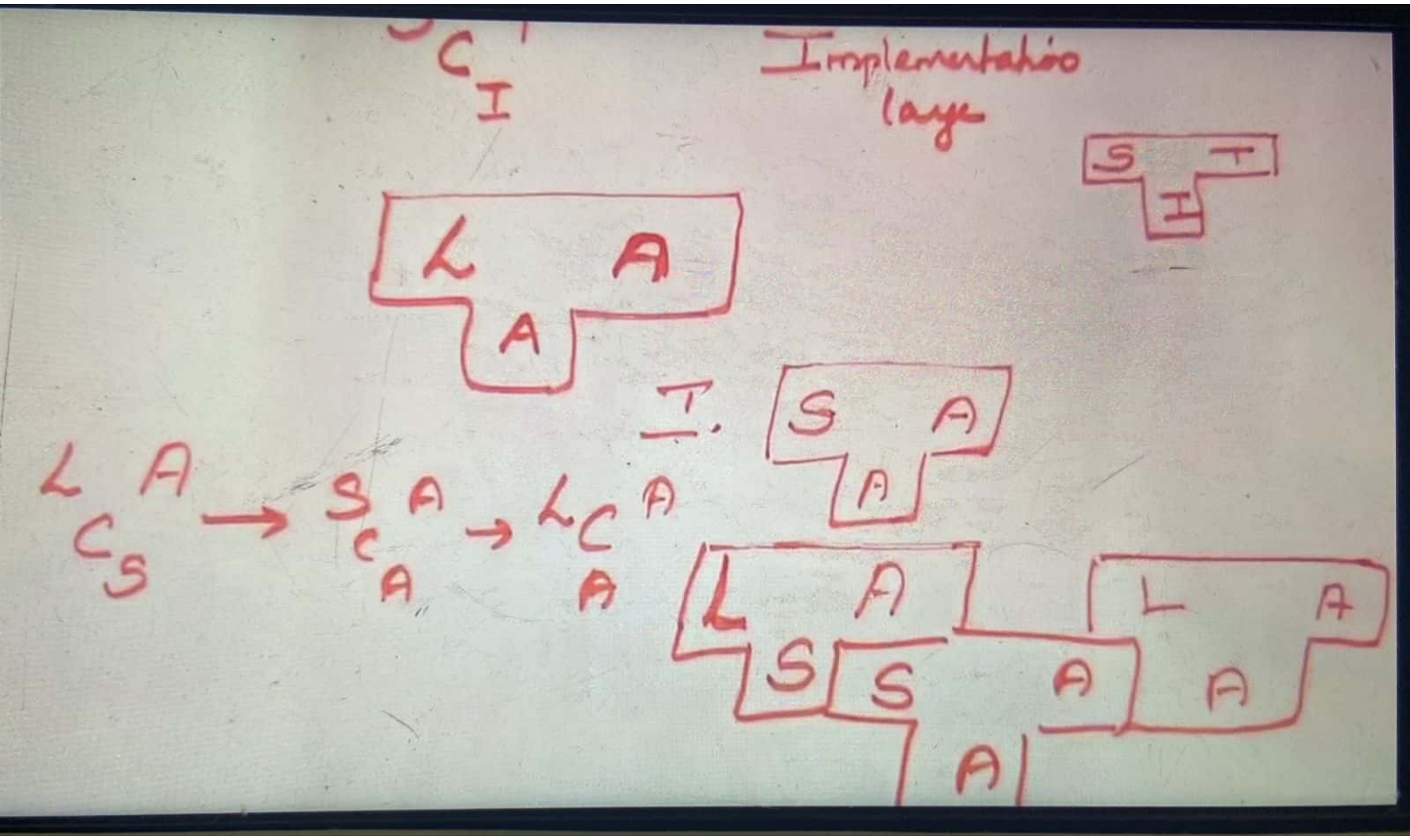
Type of Compiler that can compile its o

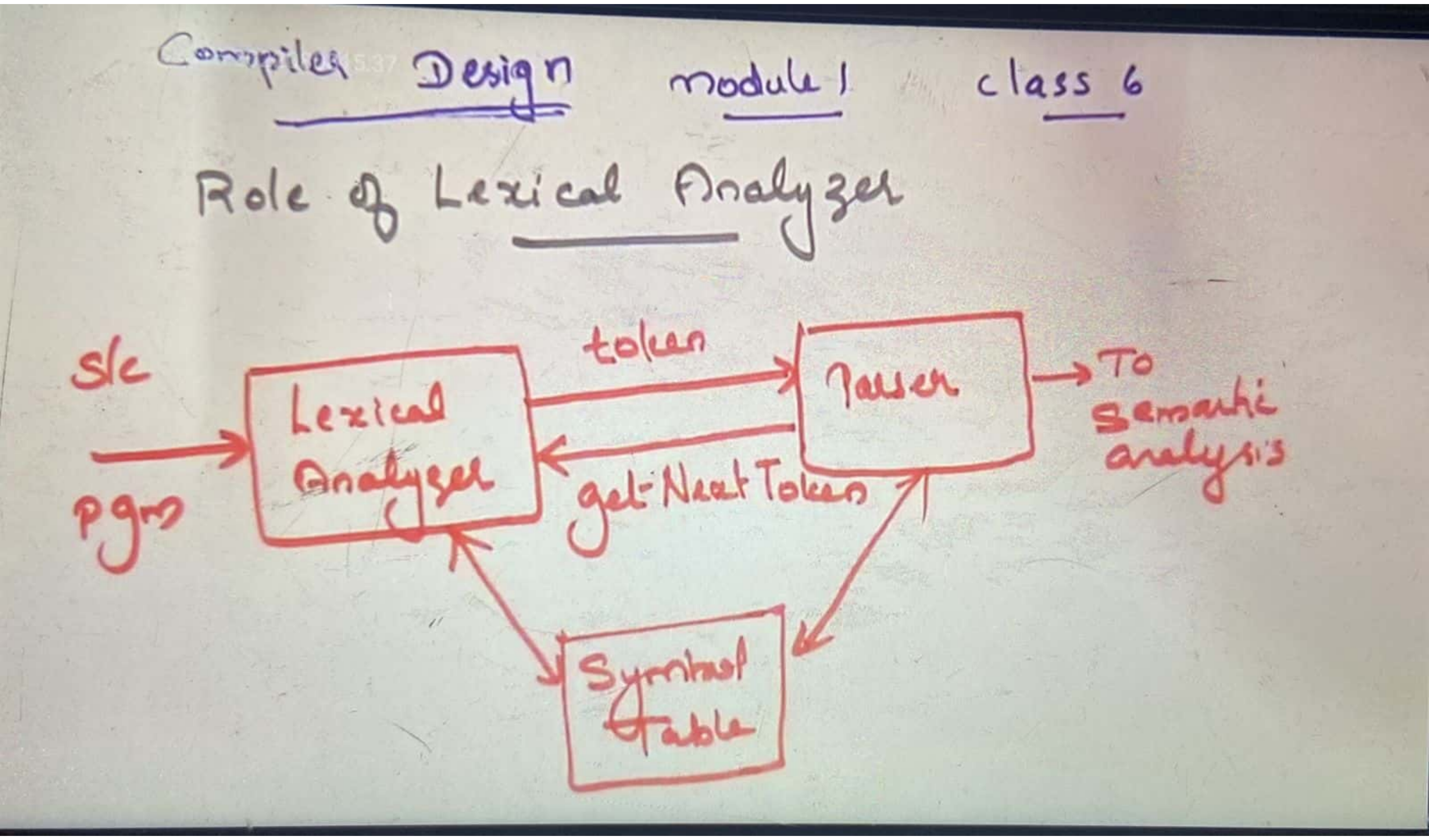
s/c code.

s/c → TL

Implementatio

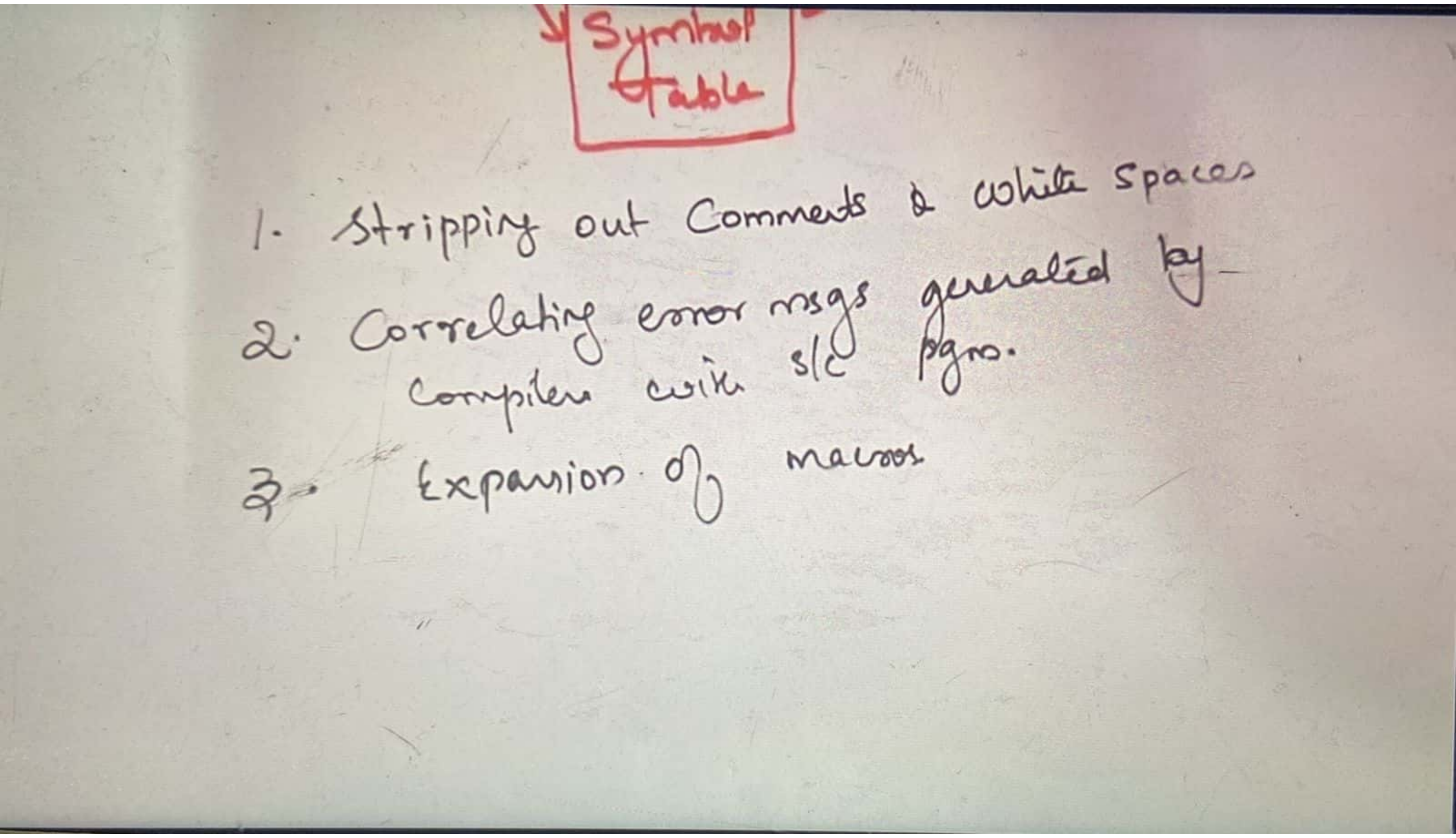
lang


C
I
Implementahóo
laye
S T
I
L A
A
T.
S A
A
L A
C S
→
S A
C
A
→
L C A
A
L A
S
S A
A
L A
A



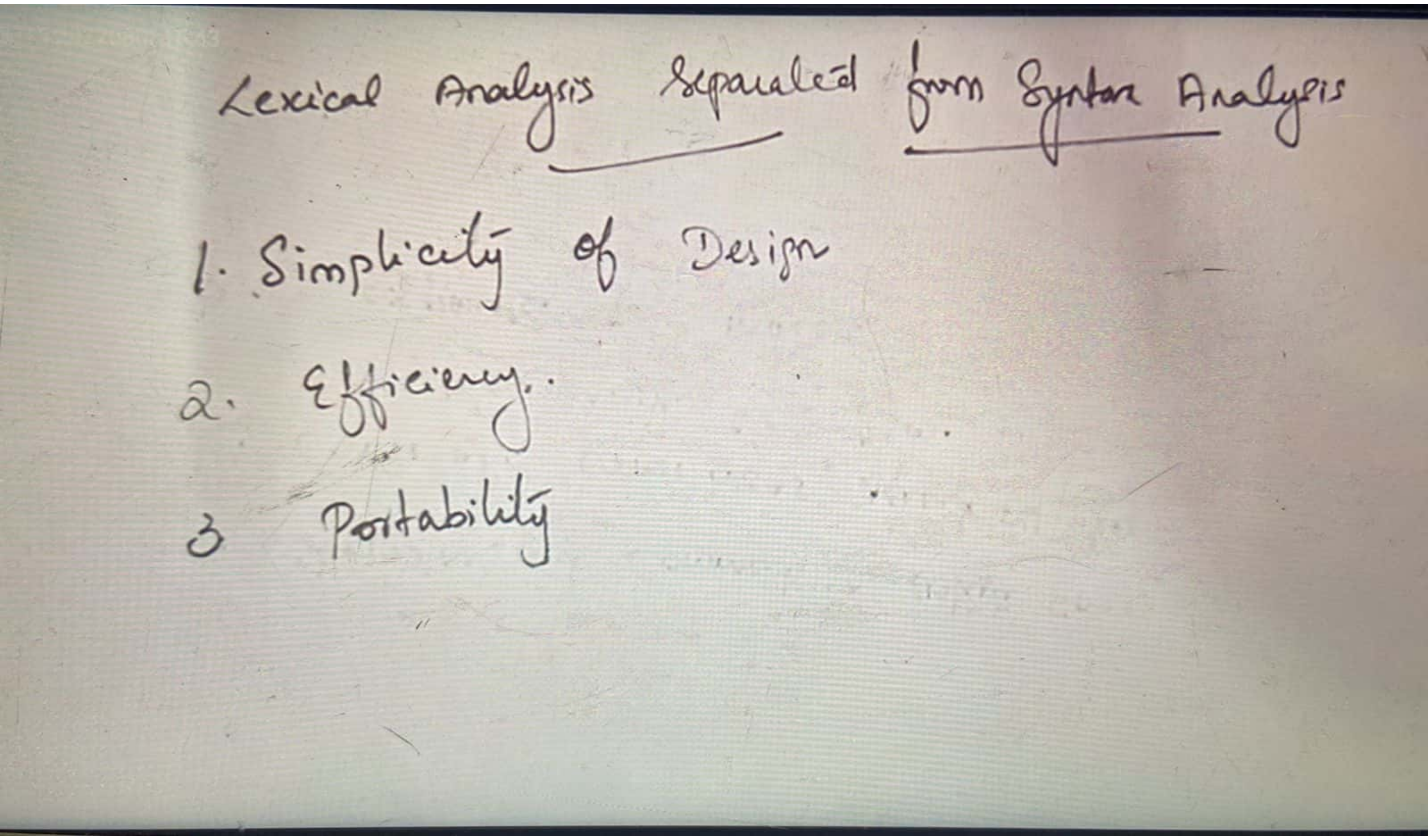
Compiler Design module 1 class 6

Role of Lexical Analyzer



Symbol table

1. Stripping out Comments & white spaces
2. Correlating error msgs generated by compiler with s/c pgm.
3. Expansion of macros

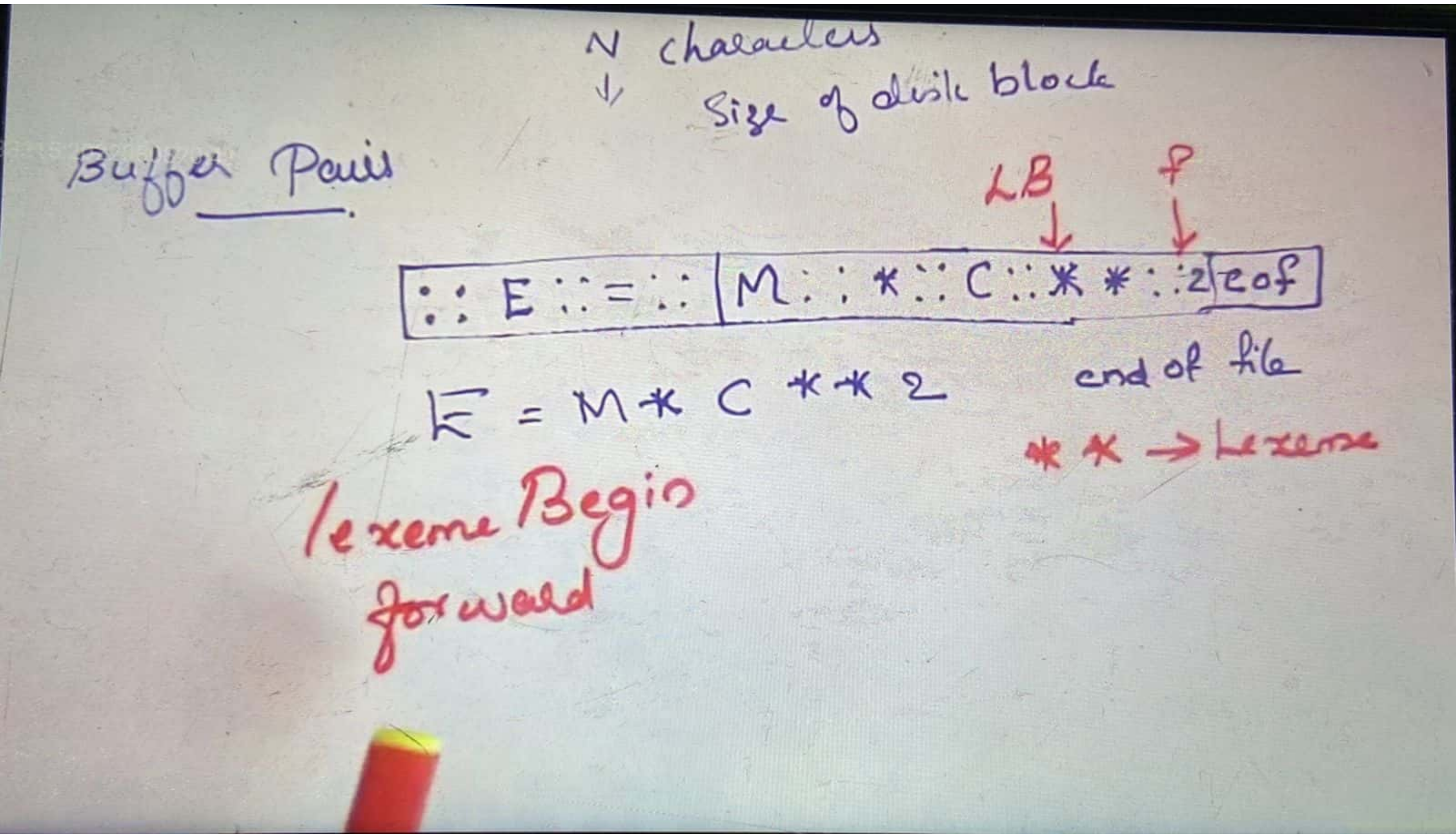


Lexical Analysis Separated from Syntax Analysis

1. Simplicity of Design

2. Efficiency..

3 Portability



N characters
↓ Size of disk block

Buffer Pairs

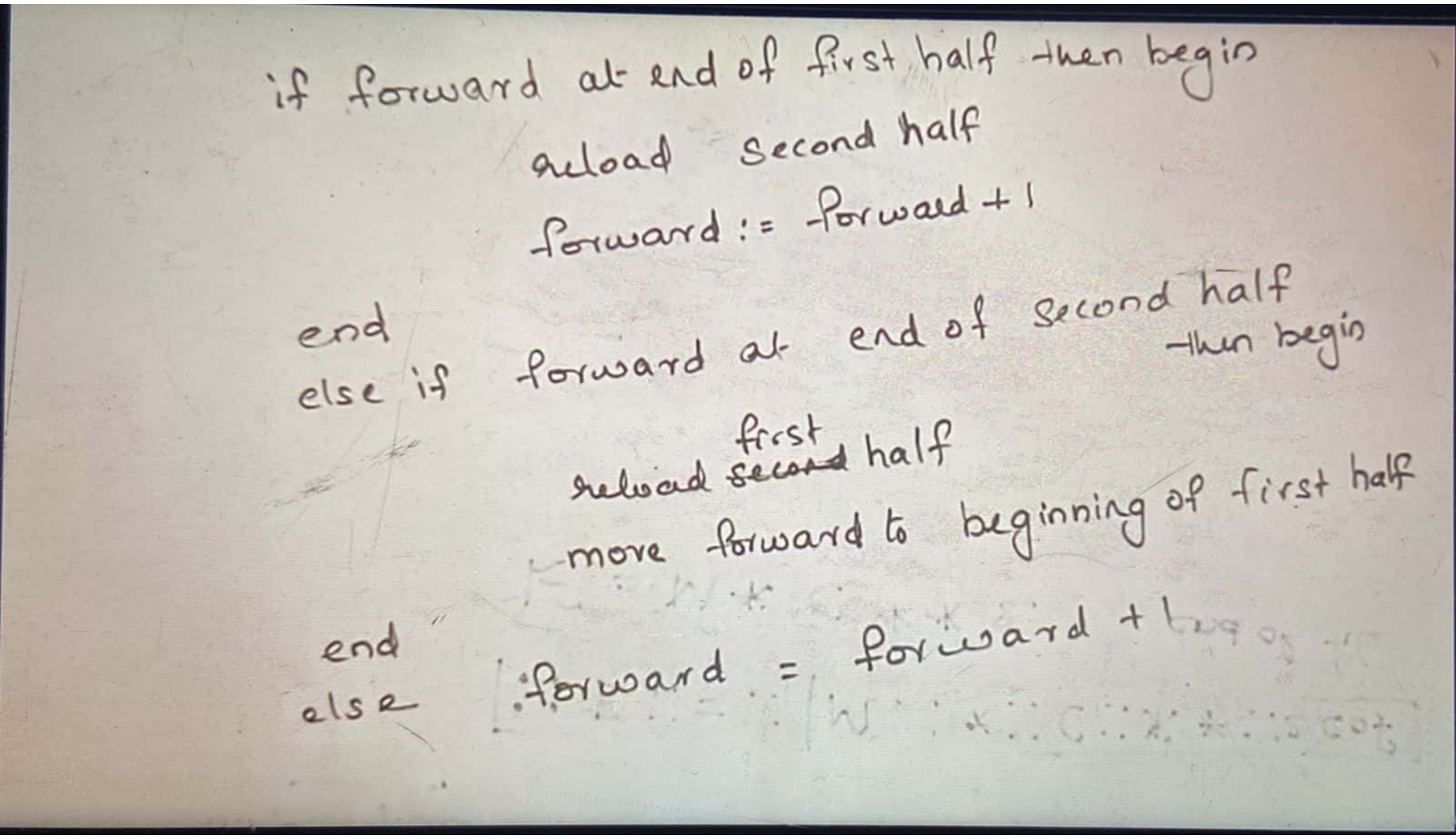
E = M * C * * 2

end of file

* * → Lexeme

lexeme Begin

forward



```
if forward at end of first half then begin
        reload second half
        forward := forward + 1
end
else if forward at end of second half then begin
        reload ~~second~~ first half
        more forward to beginning of first half
end
else    forward = forward + 1
```