Compiler Design

Topics

- Basics of Compiler
- Lexical Analysis
- Syntax Analysis(Parsing)
- Syntax Directed Translation(SDT)
- Intermediate code generation
- Code Optimization
- Runtime Environment

GATE Weightage

4-8 Marks

Books

Ravi Sethi & Ullman

Compiler Design

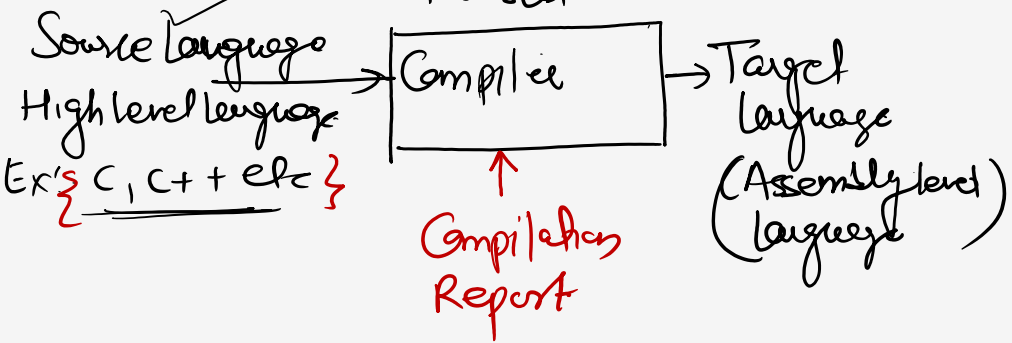
Today Class Topics

-Introduction of Compiler
-Difference Between Compiler and Interpreter
-Language Processing System
-Phases of Compiler & Type of Compiler
-Lexical Analysis
-Pattern, Lexeme and Tokens
-Count no. of Tokens
-Different Types of Errors

Compiler Design

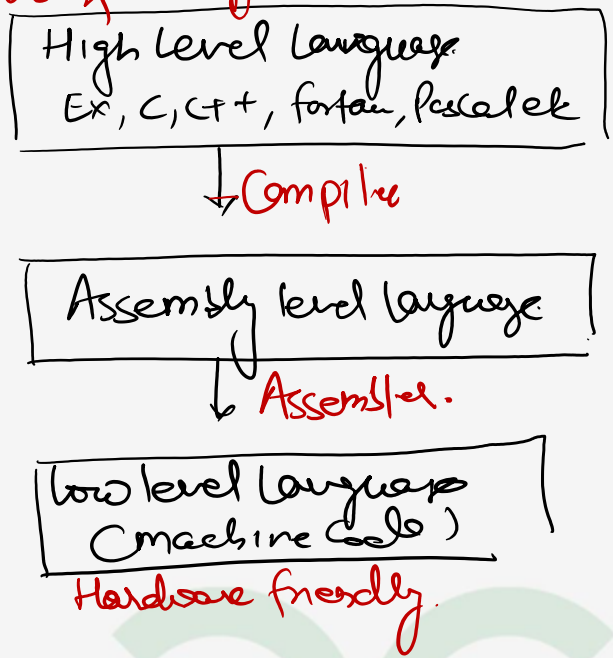
Compiler:

(Software)
Translator



Why do we need Compiler?

user friendly



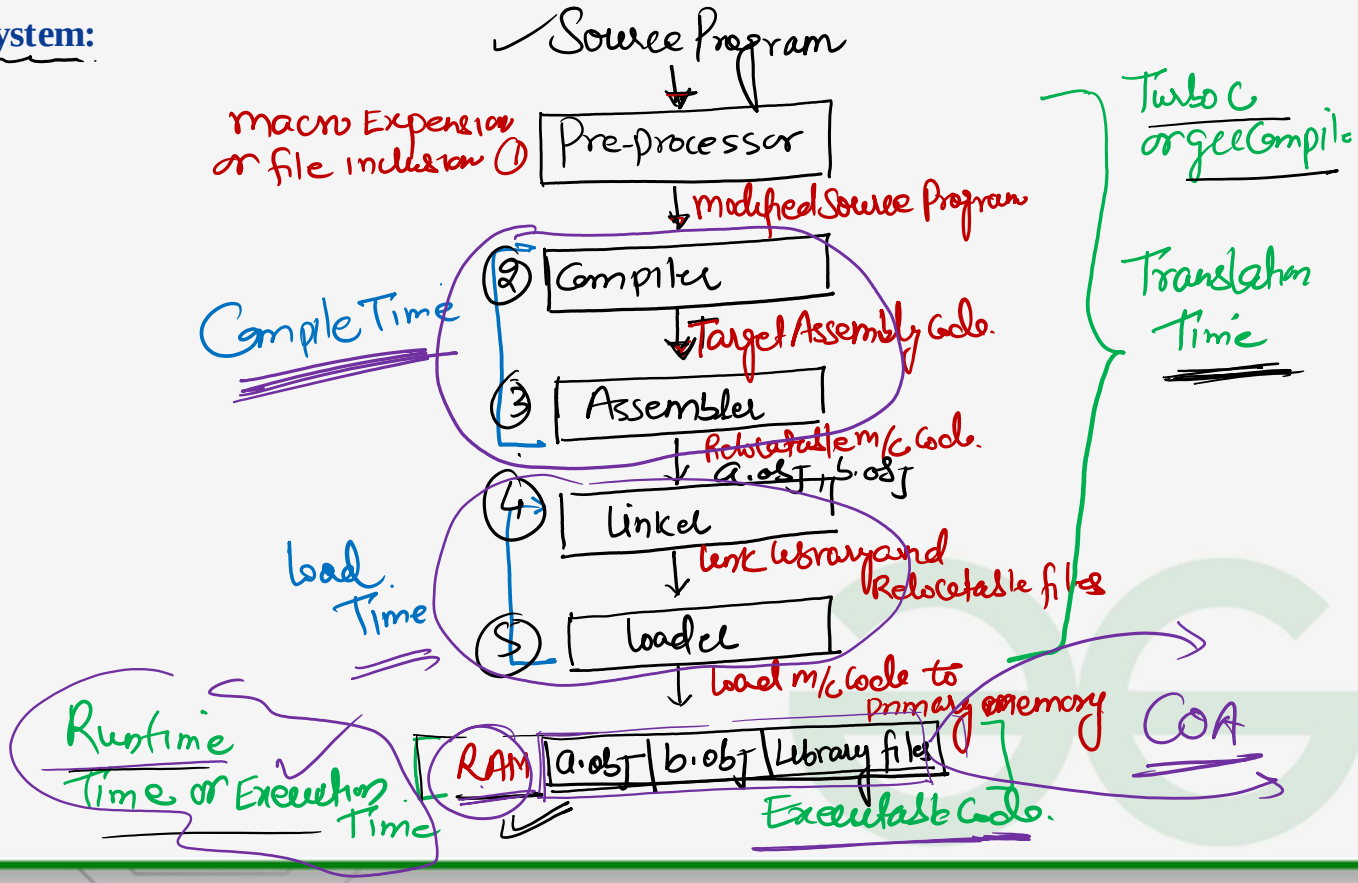
Compiler Design

Difference b/w Compiler and Interpreter:

Compiler	Interpreter
1) one full go	1) line by line go
2) faster	2) Slowed
3) more memory	3) less memory
4) poor debugging	4) Better debugging

Compiler Design

Language Processing System:



#include <std ... -> **Compiler Design**

Macro expension or file inclusion

```
#define Square(x) x*x
void main()
{ int y, x=8;
  y = 64 | Square(x); x*x;
  printf("%d",y); //64
}
```

=>

```
y = 64 | x*x;
y = 64 | 8*8;
y = 8*8, = 64
```

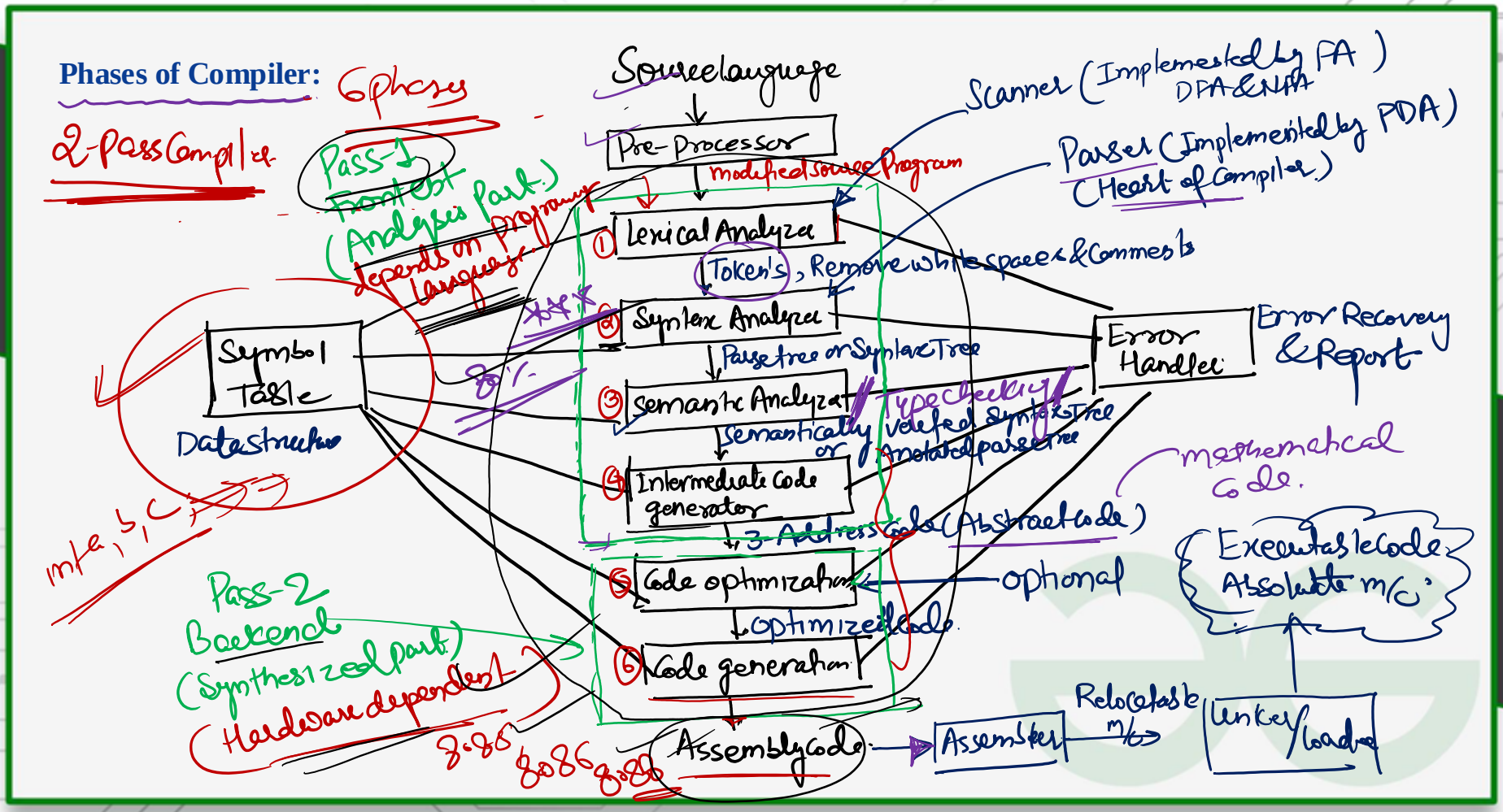
object files:

a.obj

```
#include <stdio.h>
void main()
{ int a,b,c;
  c = add(a,b);
  printf("%d",c);
}
```

b.obj

```
int add(int x, int y)
{ int z
  z = x+y;
  return z;
}
```


Phases of Compiler:
6 phases
2-pass Compiler
Pass-1
Front end
(Analysis part)
depends on programming language
Source language
Pre-Processor
modified source program
Lexical Analyzer
Tokens, Remove whitespaces & Comments
Syntax Analyzer
Parse tree or Syntax Tree
Semantic Analyzer
Semantically verified Syntax Tree or Annotated parse tree
Intermediate code generator
3-Address Code (Abstract code)
Code optimization
optional
optimized code
Code generation
Assembly code
Symbol Table
Data structure
Error Handler
Error Recovery & Report
Scanner (Implemented by FA) DFA & NFA
Parser (Implemented by PDA)
(Heart of compiler)
Type checking
mathematical code
Executable code
Absolute m/c
Relocatable m/c
Assembler
Linker/loader
Pass-2
Backend
(Synthesized part)
Hardware dependent
8086
8086 8086

Types of Compiler:

1) Single Pass
2) two Pass
3) Multipass

Nokia, samsung, motorola, iphone

Charger → Adv! Faster

Disadv: more space

goto 15 FeedBack

Single pass

Source language (HLL)

1, 2, 3, 4, 5, 6

Compiler

Assembly code

int a, b, c

c = a + b + 4 + 0

1, 2, 3, 4, 5, 6

RAM, it take more memory

Synthesize

Analysis Part 1, 2, 3, 4

RAM it take less memory

2 Pass Compiler

Source language (HLL)

Pass 1 — Analysis Part — Frontend

Pass 2 — Synthesize Part — Backend

Assembly code

Adv: ① less memory ② portable

Disadv: Slow

Portable. C L_1, CPP L_2, C# L_3

FE_1, FE_2, FE_3 → BE (M_1)

(or)

L_1 FE → BE_1 (M_1), BE_2 (M_2), BE_3 (M_3)

Output of Every Phases of Compiler: –

Compiler Design

Compiler Design

Compiler Design

FA (DFA)

for( )
if( )

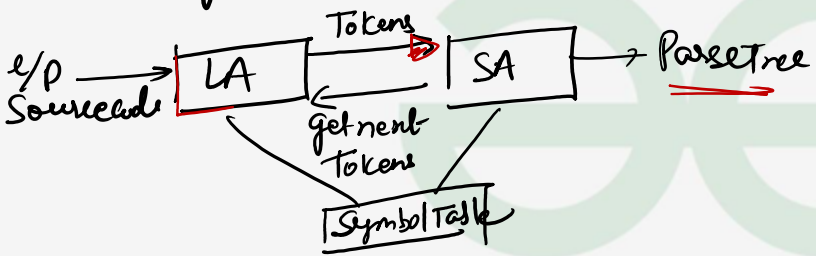
int
float --

(1) Lexical Analysis:-

Imp

The Lexical analyzer scannes every character of Source code and the following will be done by it

- Divide into tokens (Keyword, Identifier, operator, Constant etc)
- Ignore Comments (/* ---- */)
- Ignore white space characters such as Blank spaces, tab spaces and new line character etc.
- Counts the no. of lines in the Source Program
- it creates a Symbol table which contains the information about the variables and their Attributes
- it also identify lexical errors if any error in any line no. and Column no. Then Report to Error Handler → user
- it interacts with Syntax Analyzer



Compiler Design

$L = \{A, B, C \ldots a, b, \ldots\}$

$D = \{0, 1, 2 \ldots 9\}$

Token, Pattern and Lexeme:

Pattern:- Set of rules which describes the token's, what is the keyword, operator, constant, identifier etc.

Lexeme:- Meaningful group of character in a source Program that is "longest match" by Pattern and return token

Token:- is a Smallest unit of Program

Ex:

Token	Lexemes	Pattern
Keyword	Const, if, for, int, float	Const, if, for, int, float
Relational operator	<, <=, =, >, >=, <>	< or <= or > or >= or = or <>
identifier	Pi, Count, D2,	letter followed by letter and digit $L(L+D)^*$
Number	2.612, 5424 etc.	Any numeric Constant
Literal	"Hello India"	Any character between " and "

2, 22222

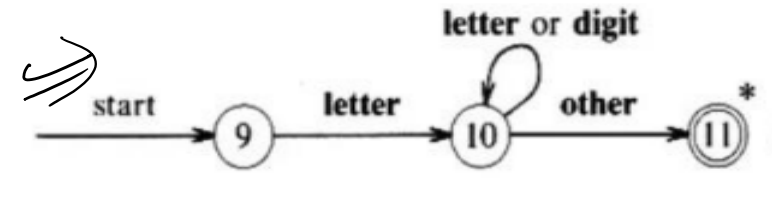
int a; int A1;

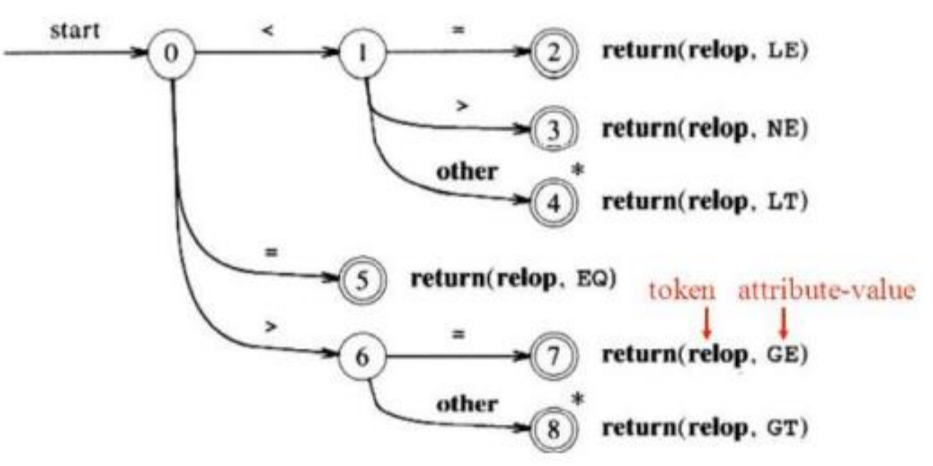
int 1A;

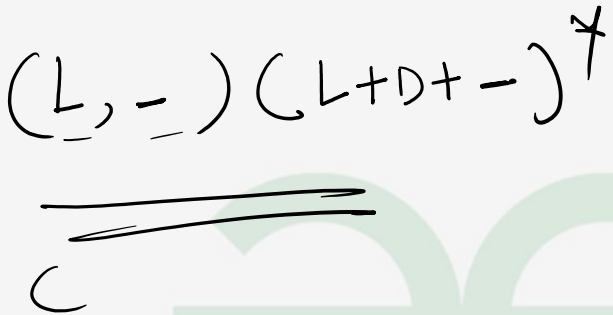
2 2222

Compiler Design

Recognization of identifier, Keyword and operators:







Compiler Design

Which of the following strings can definitely be said to be tokens without looking at the next input character while compiling a Pascal program?

I. begin II. program III. <>

(a) I (b) II (c) III

(d) All of the above

Compiler Design

int int a

Count no. of Tokens:

1) int | inta | ;
① ② ③

2) int | x | , | y | ; | ++
① ② ③ ④ ⑤

3) int | aaaa | ;
① ② ③

4) int | a | a | a | ; |
① ② ③ ④ ⑤

5) if | (| a > b |) |
a | = | a | ++ | ⇒ 13

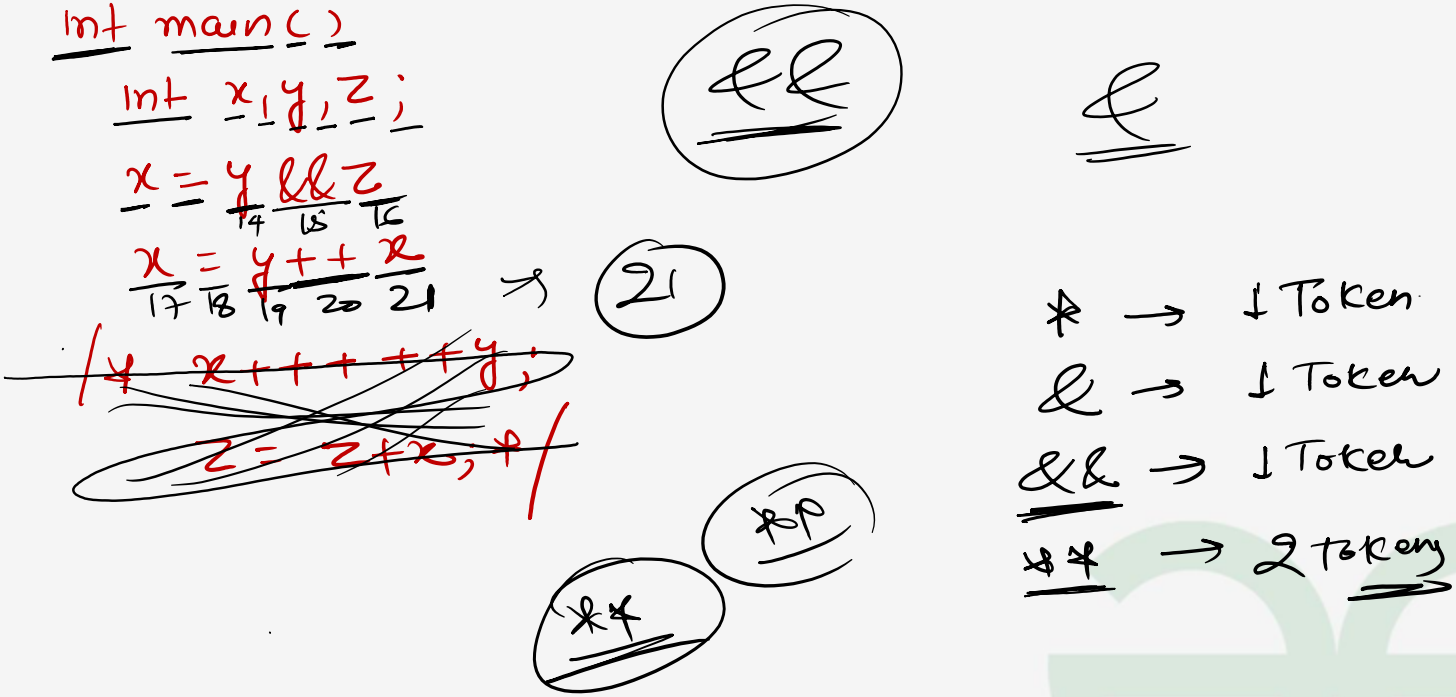
else
b | = | b | -- |

<< ee <=

6) for | (| i = 0 | ; | i < 10 | ; | i++ |) | ⇒ 13
① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬

7) printf | (| "GATE CSE 2025" |) | ; |
① ② ③ ④ ⑤

Compiler Design



Compiler Design

Compiler Design

Ques

The number of tokens in the following C statement is printf("i=%d, &i=%x", i,&i);

18

(A) 13

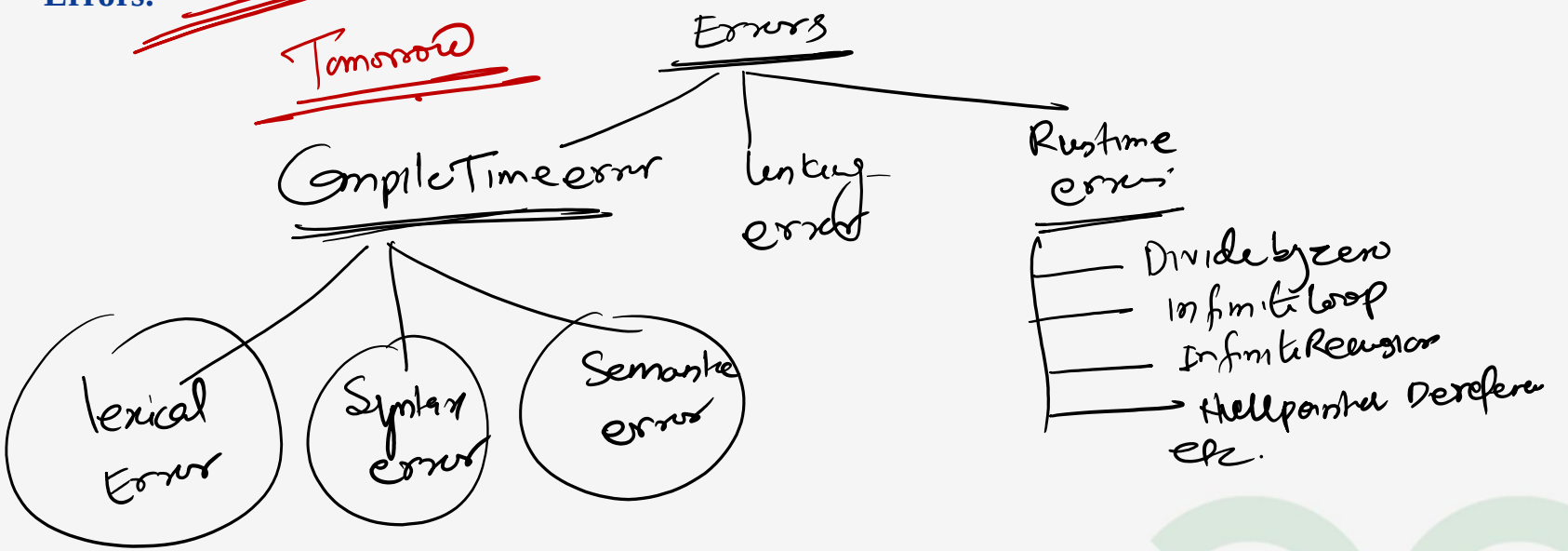
(B) 6

(C) 10

(D) 9

Compiler Design

Errors: VIMP Tomorrow



Compiler Design

lexical error: if the scanned group of character are not match.

V Imp

next class

Compiler Design

Compiler Design

Compiler Design

Consider line number **3** of the following C-program.

```
int main() {                    /*Line 1 */
    int I, N;                   /*Line 2 */
    fro (I=0, I<N, I++);            /*Line 3 */
}
```

Identify the compiler's response about this line while creating the object-module:

A. No compilation error
B. Only a lexical error
C. Only syntactic errors
D. Both lexical and syntactic errors

Thank You !