COMPILER CONSTRUCTION

Compiler
It is a program that translates one programming language into another programming language.

Programming language code (source code) → input of compiler
↓
Compiler
↓
Programming language code (target code) → output

H.L.L → High Level language
⇓
Compiler
⇓
L.L.L → Low Level language (typically assembly)

Assembler
Assembly code ⇒ Assembler ⇒ Machine code

Types of Compilers
1) L.L.L → Decompiler → H.L.L
2) H.L.L → Transpiler → H.L.L
↳ also called source to source compiler

Interpreter
→ line by line translation (statement by statement)

Compiler vs Interpreter
→ If you want fast compilating, a compiler is more suitable
→ If you want more portability, an interpreter is more suitable

Compiler vs Interpreter

Compiler	Interpreter
→ Whole code translation	→ line by line (on the fly)
→ Compile once & run always	→ translate on every run (overhead bcz you are translated each line & executing it)
→ Faster ↳ bcz already compiled ↳ lots of optimizations	→ Slower
→ Shows all errors but is vague in its telling ↳ as it tries to fix errors (as it takes assumptions)	→ stops on the first error (precisely tells the error)
→ less portable (compiled differently on different OS (and hardware))	→ more portable (can run on different OS (translating code on fly))
→ Less RAM	→ More RAM

Assembler vs Compiler
→ Assembler is a compiler BUT it translates assembly to machine code
→ Compiler can compile into other languages as well

Just-in-time Compiler
→ on the fly
→ stores blocks of code in cache (functions, loops)
→ interpreter + compiler
→ interpretes some lines & compiles loops, functions etc.

Phases of Compiler

stream of characters,

→ input: character stream of source code

→ (before this pre-compiler or pre-processor)

lexical analyzer

- → scanner (also known as scanning phase)
- → does regular expressions
- → finds lexeme & tokenizes them
- → it finds error regarding individual token (not btw relations e.g 'a6 kill' is not valid variable name)
- → Lexeme: smallest group of characters which are meaningful for compiler

→ input: stream of tokens (these tokens need to be in order)

Syntax analyzer

- → CFG
- → we use CFG in syntax analyzer
  - ↳ finds syntax related errors (no open/closing, broken etc)

→ input: syntax tree

Semantic Analyzer

- → we use CFG in semantic analyzer
  - ↳ finds semantic error (int has float value)
  - ↳ extends syntax tree or/and symbol table
- → annotated syntax tree made here

SYMBOL TABLE

→ input: annotated syntax tree

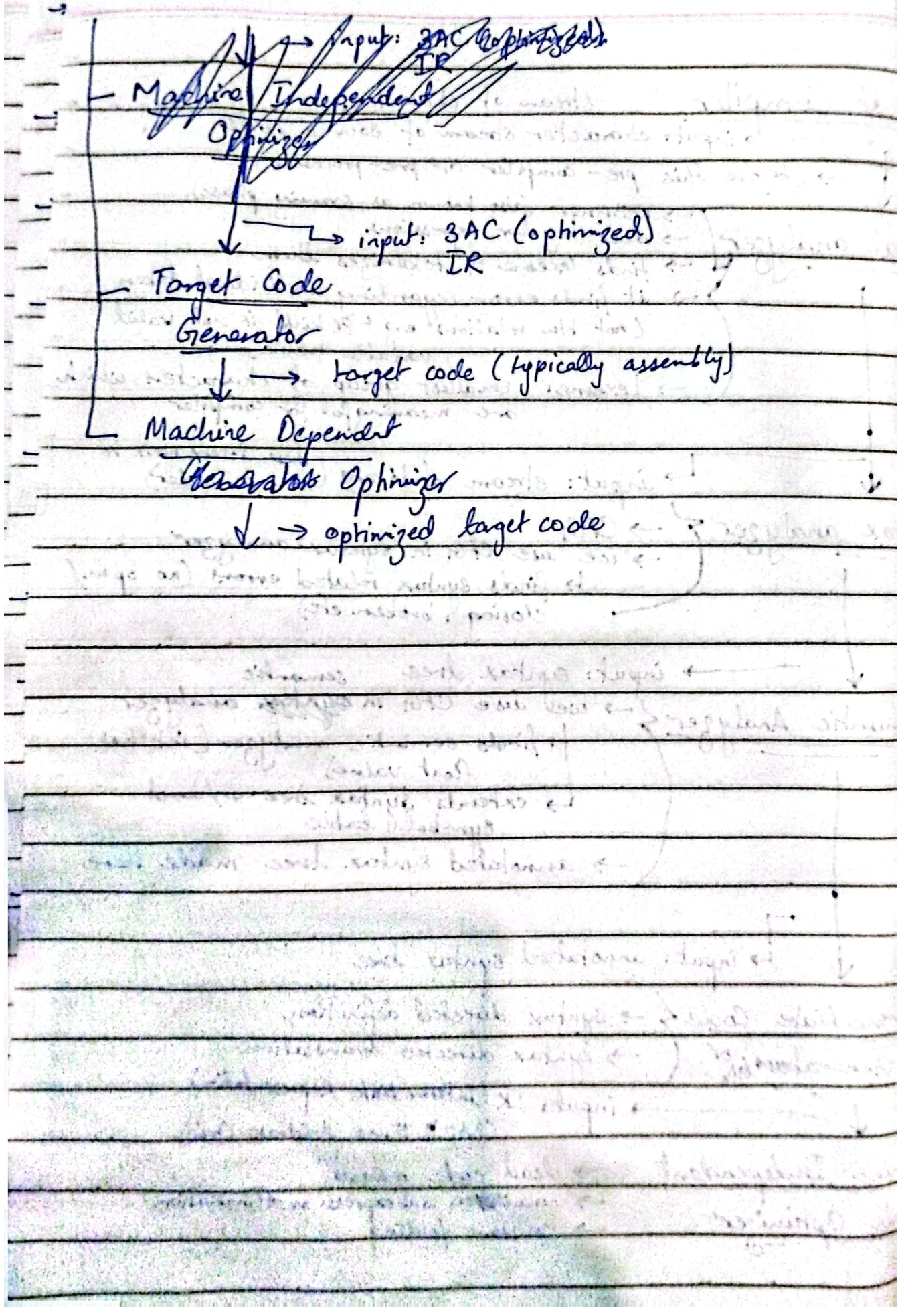
Intermediate Code Generator

- → syntax directed definition
- → syntax directed translation

→ input: IR (Intermediate Representation) (3AC: Three Address Code)

Machine Independent Code Optimizer

- → dead code removal
- → common subexpression elimination
- → constant folding


Machine Independent Optimizer
input: 3AC (optimized) IR
Target Code Generator
input: 3AC (optimized) IR
target code (typically assembly)
Machine Dependent Optimizer
optimized target code

Lexical Analyzer

Phase: stream of characters → Lexical Analyzer → stream of tokens

Specifications:
Regular Expressions (RE)

What Errors Are Produced?
↳ related with individual lexemes
↳ e.g. 87 kills → "illegal identifier name"

Phase 1:

e.g. sum = (i+j) * 10

all are lexemes

Token: is a tuple which consists of two things:
< token-name, attribute-value >
(optional)

↳ this is optional
↳ is index to the symbol table entry

Symbol Table: is a data structure used by all phases of compilation. The phases can store, update & extract info from a symbol table

e.g. sum = (i+j) * 10 ← for this example

Tokens for this will be:

<id, 1> <=> <(> <id, 2> <+> <id, 3> <)> <*> <10>

↳ index to the symbol table entry
↳ identifier

Symbol Table

1	sum	...
2	i	...
3	j	...

→ Lexical analyzer removes spaces

check if type conversion int to float is to be done in semantic analyzer

Syntax Analyzer

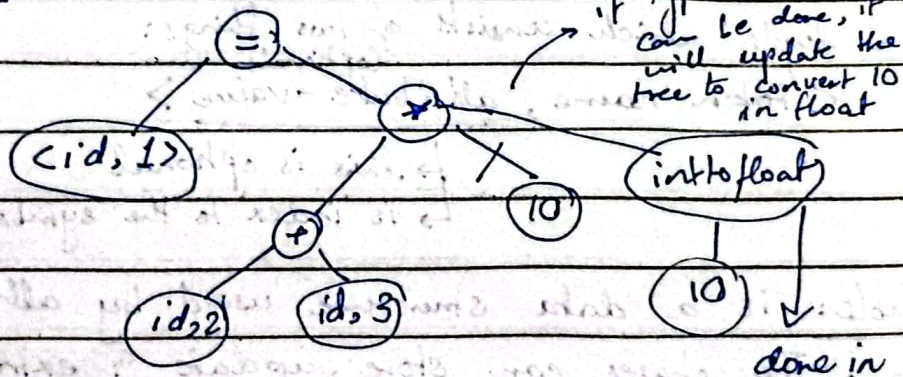
Phase:	Specification:
↓ stream of tokens	Context-Free Grammar (CFG)
Syntax Analyzer	Errors:
↓	↳ Incorrect parenthesis
syntax tree	↳ incorrect key
	↳ e.g for () { } ⇒ is an error
	for (;;) { } ⇒ is correct

Tokens:

e.g <id, 1> <=> <(> <id, 2> <+> <id, 3> <*> <10>

Phase 2:

Syntax Tree:



if type conversion can be done, it will update the tree to convert 10 in float

done in

Semantic Analyzer

↳ updates the symbol table by adding more info in it
↳ updates type & scope in symbol table & in the tree

Phase:	Specification:
↓ syntax tree	Context-Free Grammar (CFG)
Semantic Analyzer	Errors:
↓ annotated syntax tree	↳ Type ⇒ e.g int i = 3.5 ↳ "illegal assignment"
	↳ Scope

Phase 3:

Type / Scope related variables caught here

Intermediate Code Generation

Phase:

annotated syntax tree
↓
intermediate code generator
↓
Three address code

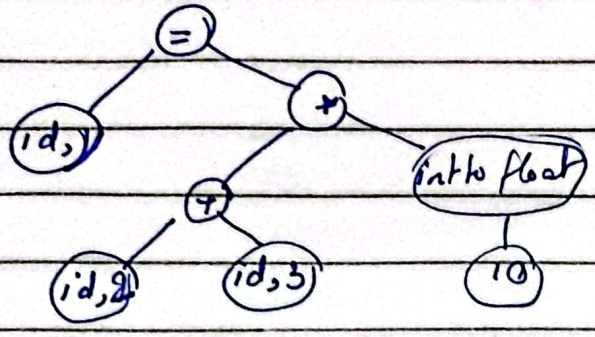
Specifications

↳ syntax directed definition
↳ syntax directed translation
(basically CFG with some code of rules attached with the grammar symbols or productions)

Errors:

↳ does not produce many errors

e.g



Phase 4:

$t_1 = id_2 + id_3$

$t_2 = \text{inttofloat}(10)$

$t_3 = t_1 * t_2$

$id_1 = t_3$

Advantages:

↳ can do optimizations which are machine independent

Machine Independent Optimizer

Phase:

3AC
↓
machine independent optimizer
↓
optimized 3AC

→ use dead code removal algorithm
→ constant folding algorithm
→ common subexpression elimination algorithm

e.g $t_1 = id_1 + id_3$
$t_2 = inttofloat(10)$
$t_3 = t_1 * t_2$
$id_1 = t_3$

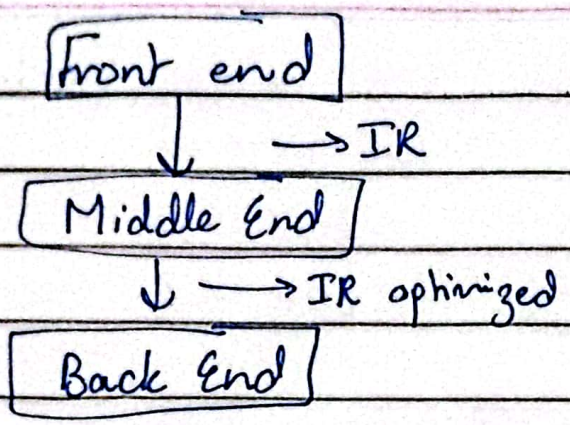
Optimized code:

$t_1 = id_1 + id_3$
$id_1 = t_1 * 10.0$

FE { Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate code Generator } Programming language dependent

ME [Machine Indep Optimizer] independent

BE { Target code Generator → Machine Dependent Optimized } Machine dependent



→ if two languages have same IR, their backends will be same 4 so any backend built will support both languages

→ middle-end 4 backend development is then not required

→ LLVM-IR is supported by many backend

Source Code

↓

Pre-processor

↓

pre-processed source code

Used for:

(1) Macro Expansion

→ #define E 2.71

→ #define square(x) (x.x)

e.g & i = square(3);

i = 3 x 3;

but if square(3+7)

=> i = (3+7 x 3+7) // incorrect answer

Solution:

#define square(x) ((x).(x))

→ ((3+7).(3+7)) // correct solution

(2) #include <stdio.h> → including libraries

(3) #ifdef __linux__ → code related to diffent OS's

// linux specific code

#elif _WIN32_

// windows specific code

#else

#endif

what is unresolved ref, relative address ??

Linker/Loader Object file → used for modular development ??

a.c ⇒ a.o

b.c ⇒ b.o

d.c ⇒ d.o] → unresolved reference, relative address

several object files

↓

Linker

↓

single executable