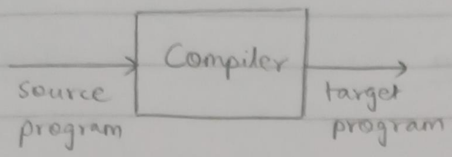
3|1|25

→ 

(C, C++) (Assembly, etc.)

Converts source language into target language.
It also checks the grammar of the source language and reports errors with appropriate error messages

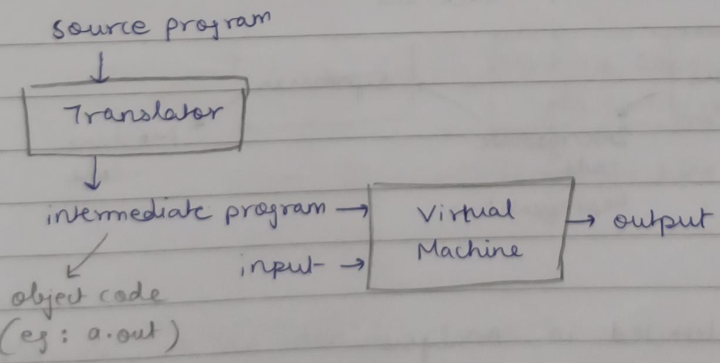
→ Interpreter → ~~compiles~~ executes line by line. Program execution stops when first error encountered.
used by PHP, Python, etc.

→ Compiler → takes entire program and compiles to an object code. Displays all errors at the end of compilation. (each error is found, recovered from and then move on until a threshold no. of errors)
used by C++, C, C#, etc.

(Differences table in slides)

→ Compiler is faster than Interpreter

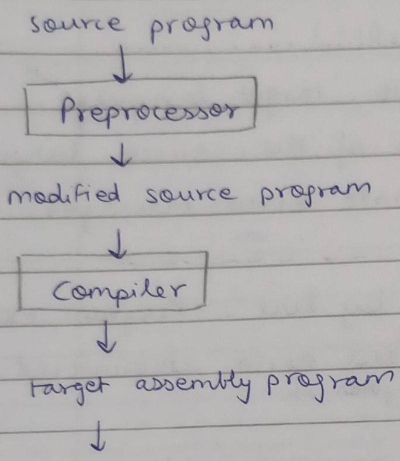
Hybrid Compiler



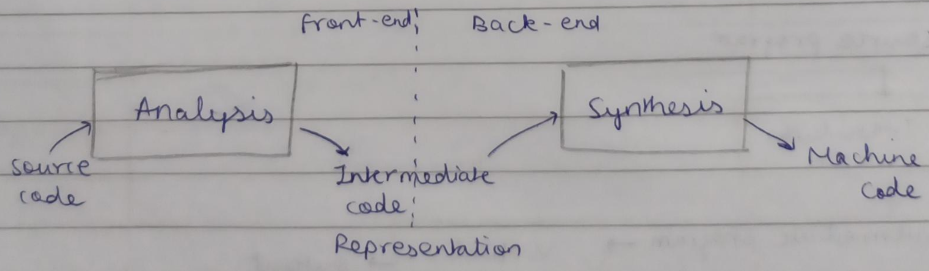
→ VM helps create a virtual environment for optimal code.

She said
everything [illegible]
at same [illegible]

Language Processing System

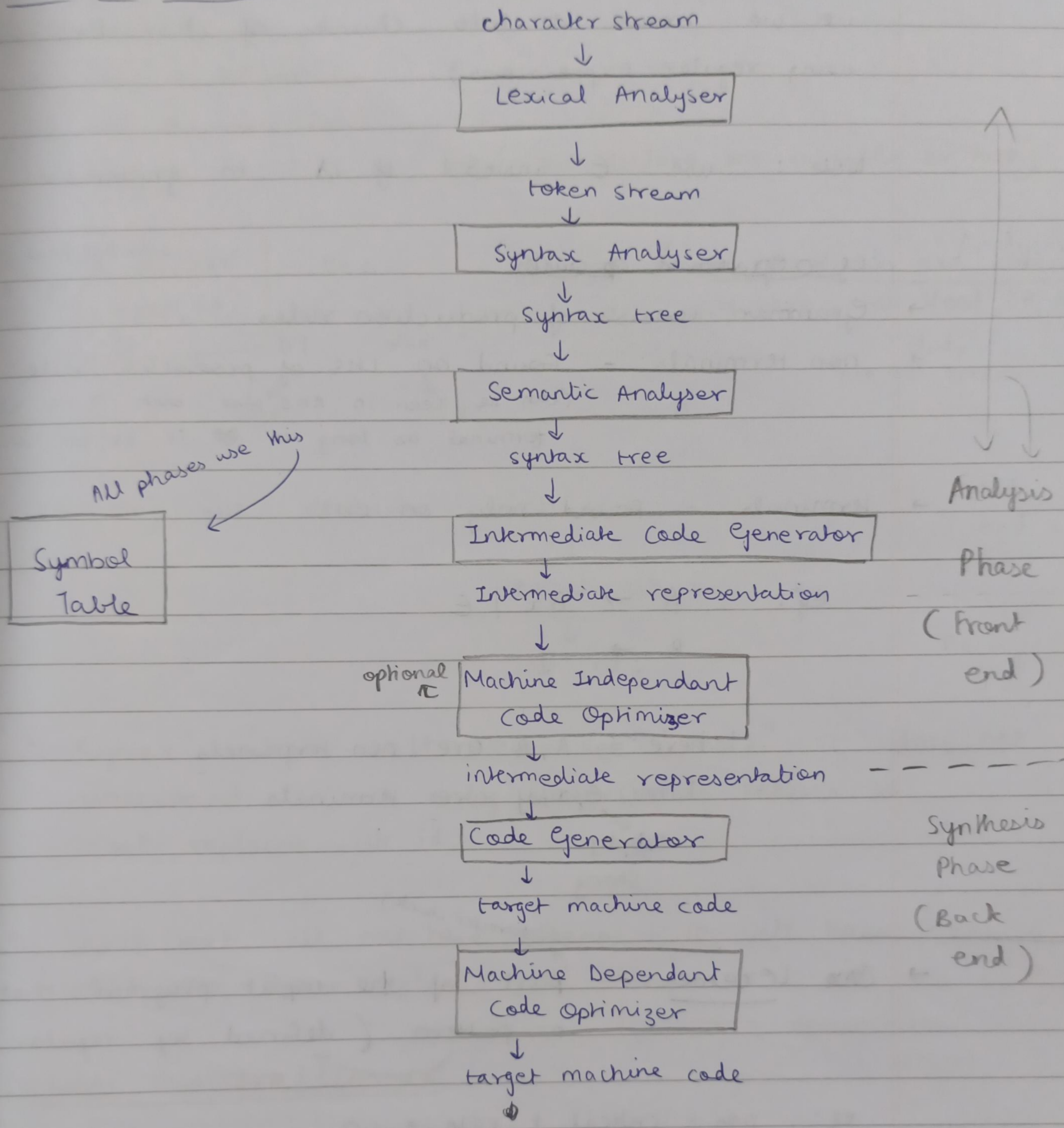


Structure of the compiler



→ Most errors detected in Analysis.

PHASES OF COMPILER


character stream
Lexical Analyser
token stream
Syntax Analyser
Syntax tree
Semantic Analyser
syntax tree
Intermediate Code Generator
Intermediate representation
optional
Machine Independant Code Optimizer
intermediate representation
Code Generator
target machine code
Machine Dependant Code Optimizer
target machine code
All phases use this
Symbol Table
Analysis Phase (Front end)
Synthesis Phase (Back end)

→ Variable is an identifier

↳ starts with a letter or undescore and followed by a letter or number.

→ Grammar is used to define rules

→ Regular expressions are used to define patterns.

→ In lexical Analyser we tokenize the program. i.e we break it into chunks of characters. (specified using regular expressions)

Note : use ϵ instead of λ in grammar

production head → rule
eg: $A \rightarrow Ba \mid \lambda$

→ Grammar consists of production rules

→ non terminals - Found on LHS of production rule (can be seen in RHS also but it is still non terminal as long as it is on any LHS)

→ terminals - Found only on RHS

eg:

$$A \rightarrow Bx \mid \epsilon$$
$$B \rightarrow y$$

here A, B are non-terminals

ϵ, x, y are terminals.

empty string

→ Lexemes - part of the input program that matches a pattern (defined by regular expression)

(character chunk)

eg: pos = critical + rate * 60

here, pos, =, critical, +, rate, *, 60 are all lexemes.

- input program contains lexemes. →
lexical analyser replaces lexemes with tokens which are understood by Syntax analyser

↳ (syntax analyser checks if syntax is correct based on grammar) eg: Grammar rules for Declaration of a variable

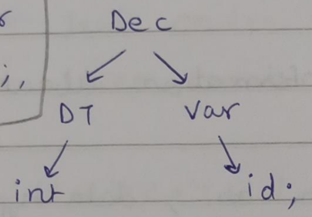
Dec → DT Var

DT → int | float

var → id; | id, id; → (either declare one variable or two)

non terminals - Dec, DT, Var

terminals - int, float, id;, id, id;



※ all terminals are tokens

eg: here, int, float, id; & id,id; are all tokens

lets say we declare int cost;

↓

Lexical analyser

↓

int id;

∴ Syntax analyser will never see "cost;" it does not understand it. it only understands tokens so cost; replaced with id;

∴ Input text will not have tokens, it will have lexemes which are then replaced with tokens.

LEXICAL ANALYZER / SCANNER

→ Identifies lexemes

→ Once it identifies a lexeme it checks which token needs to be generated corresponding to it.

eg: pos → id	+ → +	60 → num
= → =	rate → id	
critical → id	* → *	

→ Discards whitespaces and comments.

→ Extra info is needed about each token as in our eg many variables are converted to id, so they need to be differentiated.

→ Syntax of token : < token-name, ~~optional~~ attributes >
optional

→ Symbol table stores information about identifiers/variable
↳ it is a data structure

So it gives info such as, data type, size, scope, initial value, etc.

※ Symbol table stores only identifiers and no other lexemes

eg:

SI.No	Lexeme Names	data type
1	pos	float
2	critical	float
3	rate	float

Symbol table

→ Conversion of ~~tokens~~ lexemes to tokens

eg:

pos = critical + rate * 60

↓

Lexical analyser

↓

index in symbol table

<id, 1> <=> <id, 2> <+> <id, 3> <*> <num, 60>

↳ value of num

SYNTAX ANALYSER

→ Syntax Analyser imposes a grammatical restriction. i.e checks if the given input follows the grammar rules.

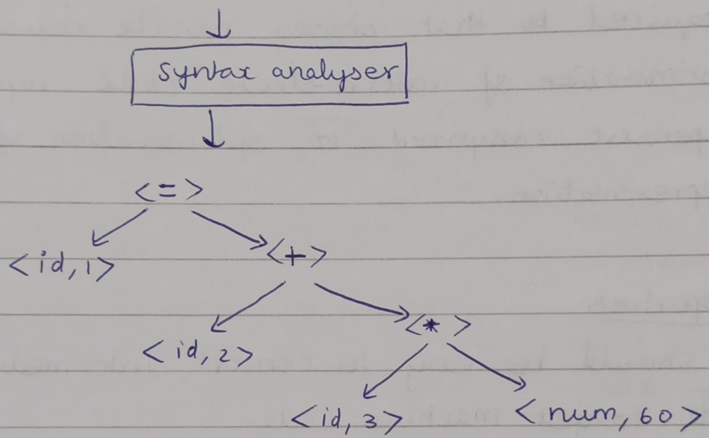
→ Syntax Tree : internal nodes → operators

leaf nodes → operands.

Only one syntax tree exists for a given input.

→ if there are errors it will not be able to generate the tree so it will know some error exists.

eg: <id,1><=><id,2><+><id,3><*><num,60>



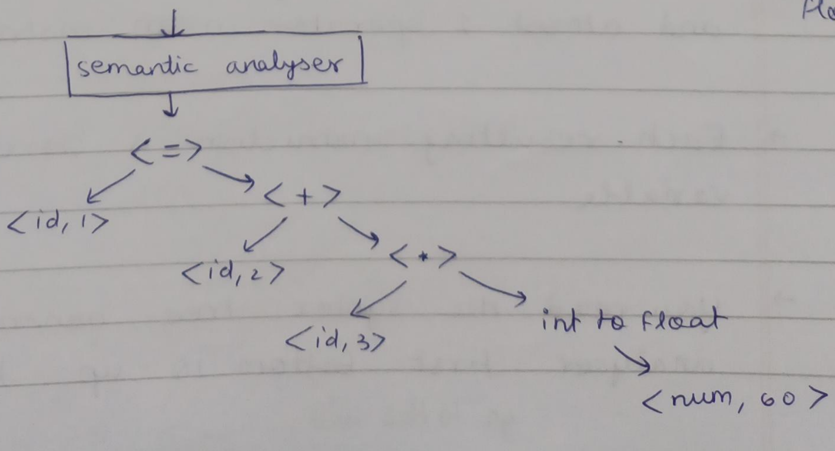
SEMANTIC ANALYSER

→ Type conversions are checked

→ Input is syntax tree

→ There may be multiple parses of semantic analyser.

eg: here all three variables are float so, 60 should also be float not int



INTERMEDIATE CODE GENERATOR

→ Say we have to ~~convert~~ translate from language L1 to L2 and L3 to L2 with intermediate generator

intermediate code

$$L_1 \rightarrow \underbrace{|R| \rightarrow L_2}$$

$L_3 \rightarrow |R|$ here, this backend of compiler can be reused to convert to L2

→ Required so that phases can be reused

→ Optimisation of intermediate code representation is less expensive compared to optimisation of target machine code representation.

Properties

① It should be easy to convert intermediate representation into target machine code.

② Easy to produce i.e. should be easy to convert from source language to intermediate code representation.

Three Address Code (TAC)

an intermediate code

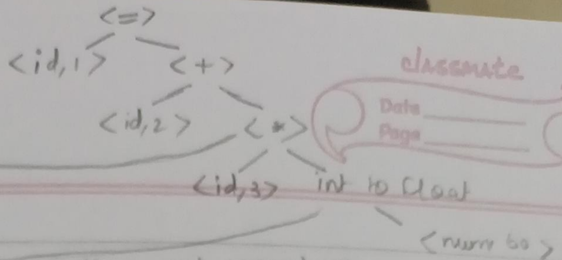
TAC instruction syntax → TAC contains assembly like code with atmost 3 operands and atmost 1 operator in an instruction

→ Each resulting instruction is stored in a temporary variable

→ You read the syntax tree generated by semantic analyser first bottom to up then left to right.

go to the bottom most operator

write eqn from left operand to right.



second bottom most operator
write the operands left to right i.e <id, 3> then t1

bottom most operator

eg: t1 = int to float (60)

t2 = id3 * t1

t3 = id2 + t2

saving or storing new value ← id1 = t3

↳ since operator at syntax tree is = so just write directly

- → last instruction in TAC is top of syntax tree
- → No optimisation while generating TAC

CODE OPTIMIZER

eg:

TAC

↓

Code optimizer

↓

removed previous t1 and instead write 60.0 ← t1 = id3 * 60.0

id1 = id2 + t1

removed t3

remove unnecessary temporary variables such that the instruction still follows TAC instruction ~~rules~~ syntax.

CODE GENERATOR

- → Optimised TAC instructions are input.

eg:

Load float

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

→

Q F = C * 1.8 + 32

Assumption: F & C are floating. Show various phases of compiler

Soln

F = C * 1.8 + 32 → lexemes

↓

Lexical Analyser

↓

<id,1><=><id,2><*>< num, 1.8 > <+> <num, 32> → tokens

just <1.8> is also okay

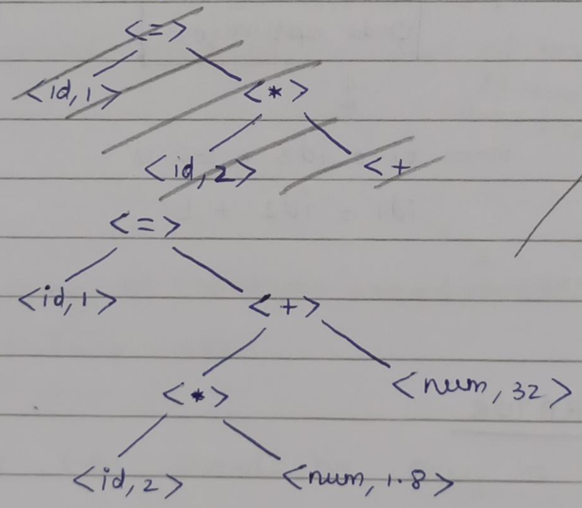
Symbol Table

index	lexeme	data type
1.	F	Float
2.	C	float

↓

Syntax Analyser

↓

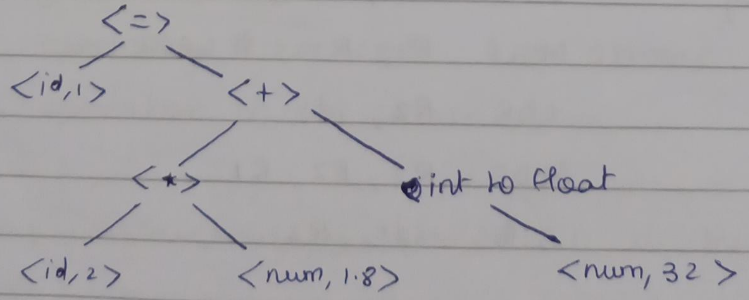


→ syntax tree

↓

Semantic Analyser

↓



→

P.T.O.

↓

Intermediate Code Generator

↓

```
t1 = id2 * 1.8
t2 = inttofloat (32)
t3 = t1 + t2
id1 = t3
```

↓

Code optimizer

↓

```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

↓

Code generator

↓

```
LDF   R1, id2
MULF  R1, R1, #1.8
ADDF  R1, R1, #32.0
STF   id1, R1
```

Q sum = a[i] + 20, consider array element takes 8 bytes. Consider sum & a as integers.

array variable syntax
arr[int] | arr[id]

sum = a[i] + 20

Symbol table

↓

lexical analyser

↓

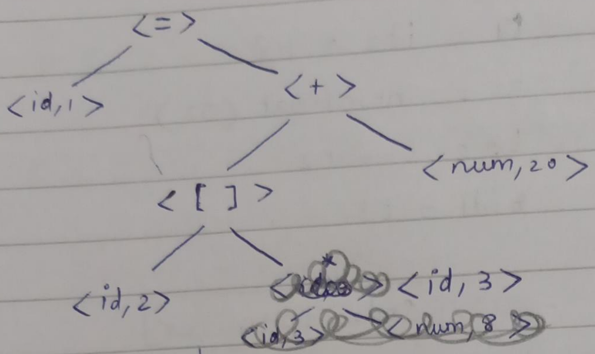
<id,1> <=> <id,2> <[> <id,3> <]> <+> <num,20>

here operations are =, array indexing & +
precedence of array indexing is highest (so written at bottom)

↓

Syntax analyser

↓



↓

semantic analyser

↓

same tree

↓

Intermediate code generator

↓

t1 = id3 * 8 → to get offset of the i^th element i * 8 bytes

t2 = id2 [t1] → not the same as C program it means offset not index.

t3 = t2 + 20

id1 = t3

↓

Code optimser

↓

t1 = id3 * 8

t2 = id2 [t1]

id1 = t2 + 20

↓

CODE GENERATOR

↓

LD R1, id3

MUL R1, R1, #8 → not sure if correct

LD R2, id2(R1)

ADD R2, R2, #20

ST id1, R2