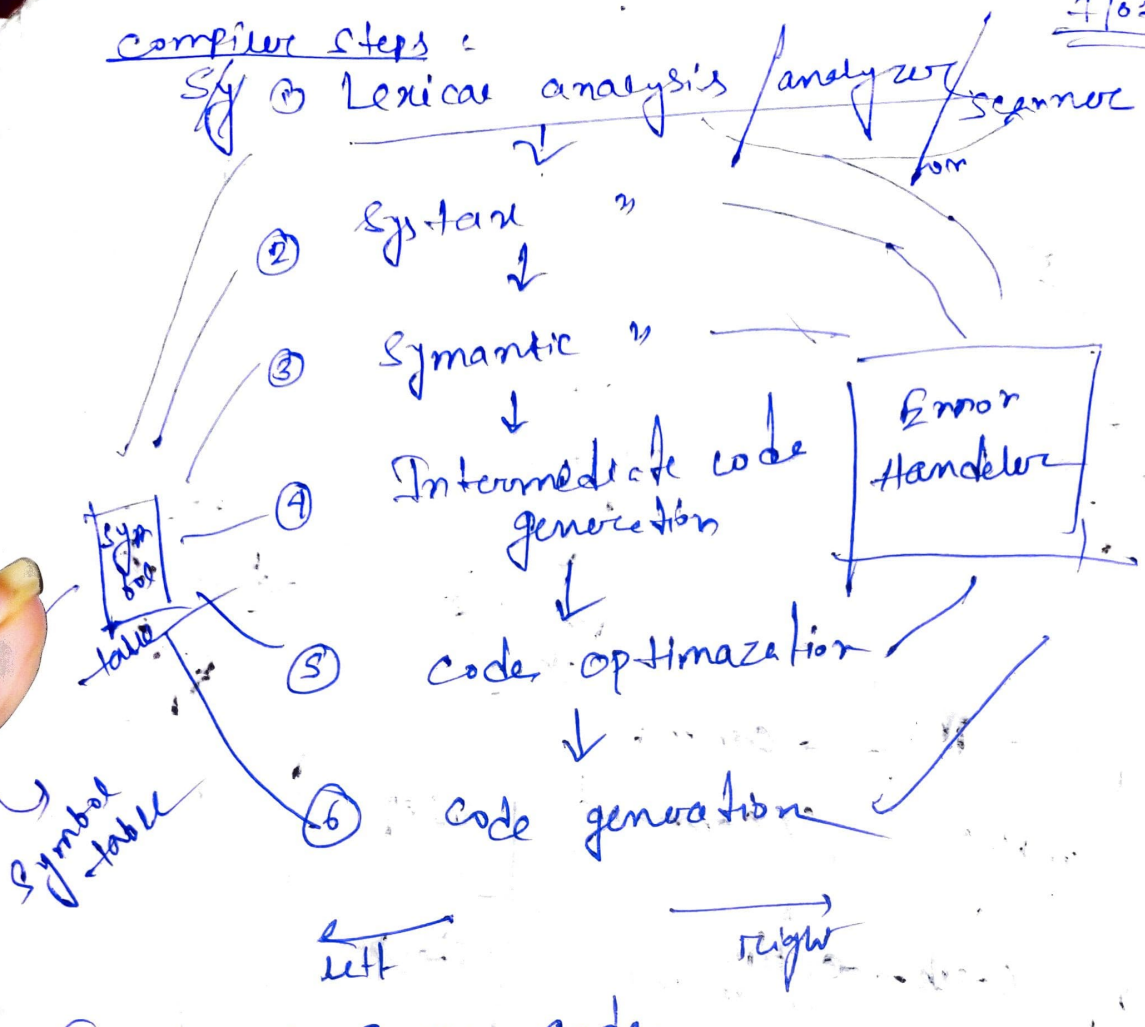
7/07/22

Compiler steps :



Input → source code

- Total = number 1 + number * 50

Token → sequence of character that have meaning

at a time one character

lexical analysis

Total = number 1 + number * 50

id 1 id 2 id 3 50

Scanning from left to right (→)

$id1 = id2 + id3 * 5$

→ output of the lexical analysis

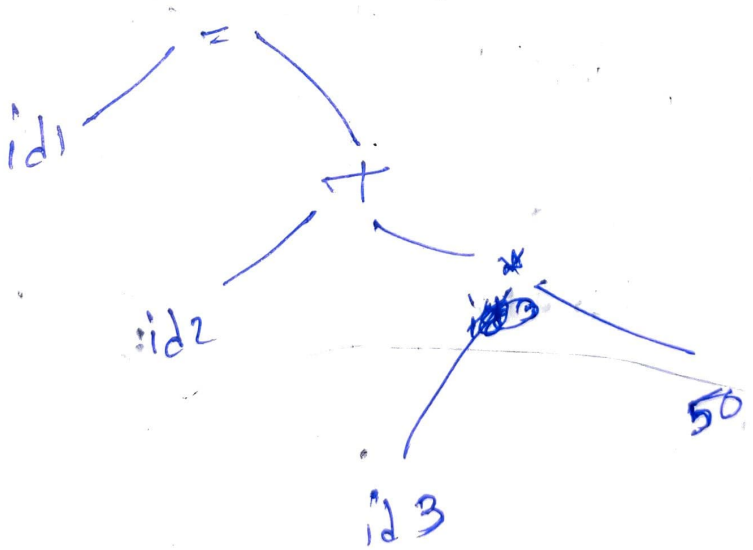
every phase connected with two phases.

Syntax analysis

→ Input is → token

$id1 = id2 + id3 * 50$

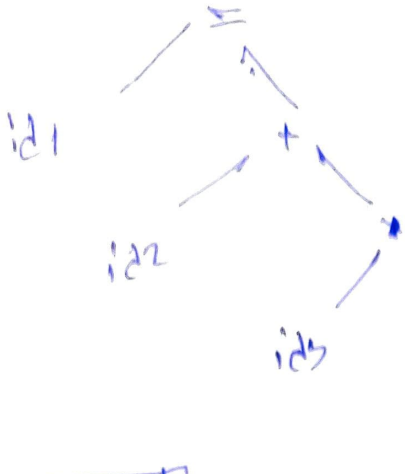
↓ syntax



parse tree

after syntax analysis the output is parse tree

↓ Symantic analysis



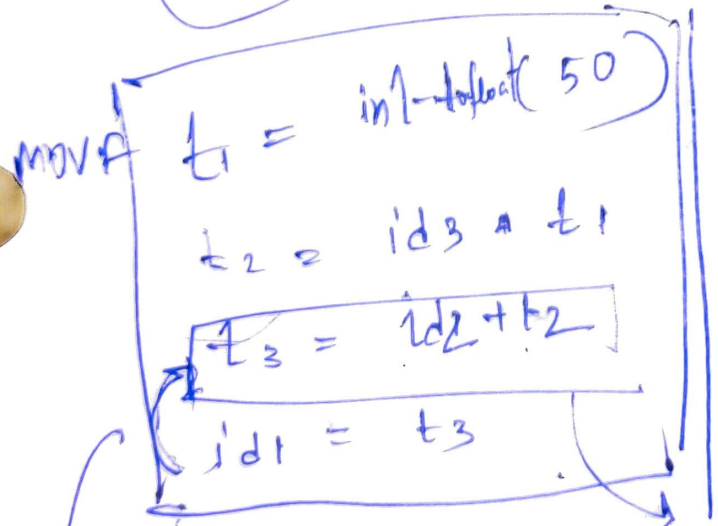
(Symantic analysis)

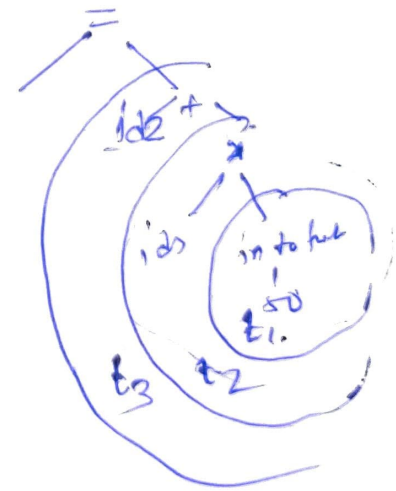
front-end → depend on source code.

Back-end → depend on intermediate code.

Intermediate code → Three address code

DAG





Intermediate code

Three address code

Format

x = y operator z

[3 operand]

Code optimization :- After

$t_3 = id3 * 60.0$
$id1 = id2 + t3$

2 steps

Machine level language → Binary

Assembler

Assembly level language

```
MOVF   // move the floating point num
MOVF   R2  #5.0
MULF   R2, id3      // multiplied with id3 and store in R2
MOVF   R1, R2
ADDF   R1, id2
MOVF
```

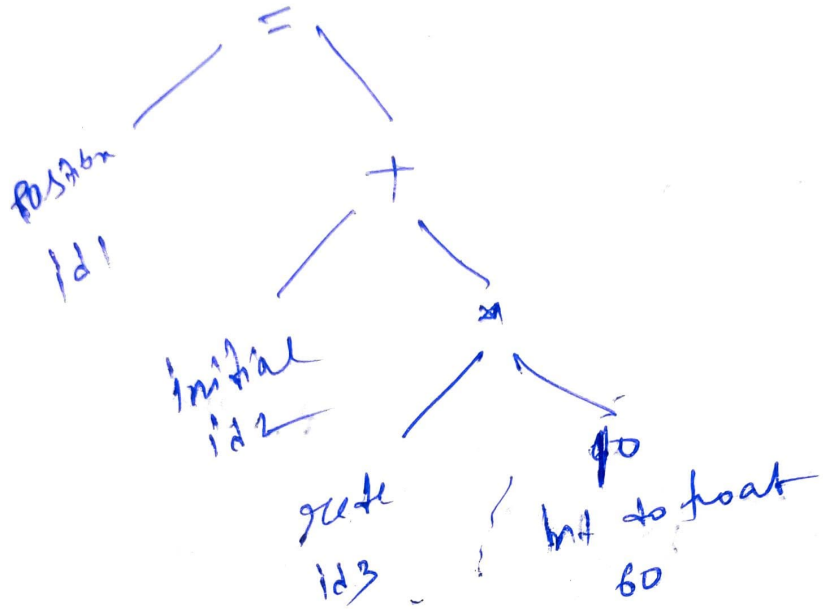
Binary

```
MOVE   R1  #60
MULF   R1, id3
ADDR   R1, id1
```

Position = Initial + rate * 60

id1 id2 id3 lexical

Parse tree



R_1 = int to float (60)

R_2 = rate * R_1
id3

R_3 = initial + R_2
id2

position = R_3
id1

Code optimization

R_3 = rate * 60.0

position = initial + R_3

Assembler Language

```
MOVF   R2   #60
MULF   R2, rate(#3)
MOVF   R1, R2
ADDF   R1, initial(#1)
```