1. $a = b + c * 60$;

Tokens are determined with the help of RE

1. Lexical Analysis :

$a = b + c * 60$;

⟨id, 1⟩ , ⟨=⟩ , ⟨id, 2⟩ , ⟨+⟩ , ⟨id, 3⟩ , ⟨*⟩ ⟨60⟩

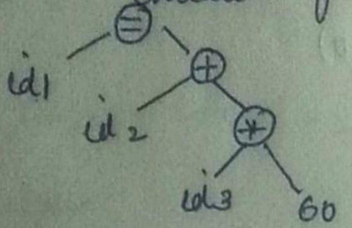
$a = b + c * 60$

id1 = id2 + id3 * 60 ;] – o/p of LA.

2. Syntax Analysis :

operator as internal node

children of node as arguments of operations



Parse tree / syntax tree is constructed on the basis of given grammar.

$$\left.\begin{array}{l} S \rightarrow id = E ; \\ E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow id \end{array}\right] \text{CFG}$$

S → Statement

E → Expression

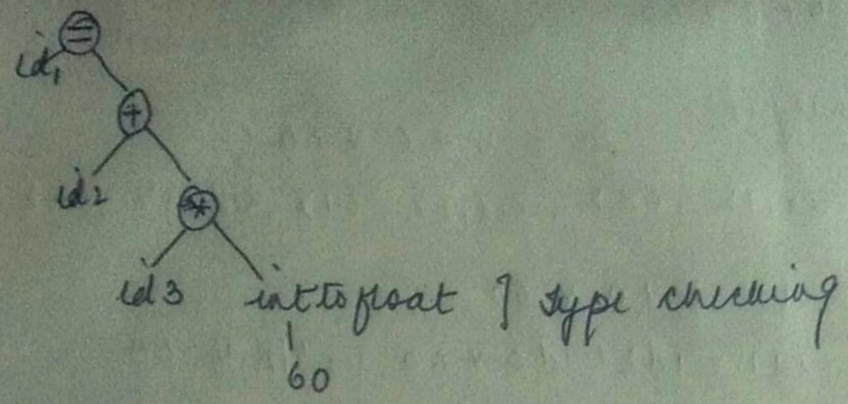
T → Term

F → factor

id → Identifier

3. Semantic Analysis : If b or c are floating point variables, then the constant value 60 must also be automatically type converted to 60.0 by inttofloat (60).

id



id1 = id2 + id3 * inttofloat (60);

4. <u>Intermediate-code Generation</u>

t1 = ~~id3~~ inttofloat(60)

t2 = ~~t1~~ id3 * t1

t3 = id2 + t2

~~t1~~ id1 = t3

5. <u>Code Optimization</u> :

~~intto~~ inttofloat (60) → 60.0

t1 = id3 * 60.0

id1 = id2 + t1 [t3 = id2 + t2 & id1 = t3] are combined

6. <u>Machine-Dependent Target Code Generation</u>

~~LDF Reg, id~~

~~STR id, Reg~~

~~ADD R1, R2~~

~~MUL R~~

- LDF R2 , id3 [Load id3 in register R2]

- MULF R2, R2, #60.0

i.e. R2 = R2 * 60.0

~~ADDF~~

- LDF R1 , id2 [Load id2 in R1]

- ADDF R1, R1, R2 [R1 = R1 + R2]

- ST id1, R1 [Store R1 to id1]

6. Machine-Dependent Target Code Generation

LDF R_2, id_3 [Load id_3 in R_2]

MULF R_2, R_2, #60.0 [$R_2 = R * 60.0$]

LDF R_1, id_2 [Load id_2 in R_1].

ADDF R_1, R_1, R_2 [$R_1 = R_1 + R_2$].

ST id_1, R_1 [store R_1 in id_1]

Q2 int i, j ;

i = i * 70 + j + 2 ;

i = i * 70 + j + 2 ;

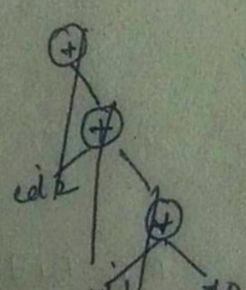
Symbol Table

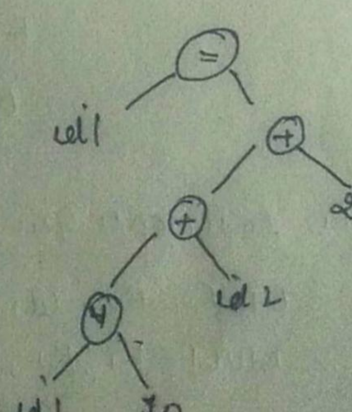
1	id	
2	id	
	----	

1) Lexical Analysis :

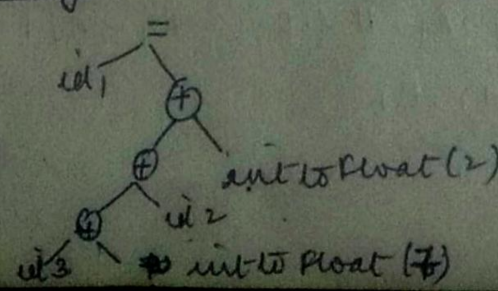
id,1 = id1 * ~~*~~ 70 + id2 + ~~id~~ 2 ;

2) Syntax Analysis :





3) Symantic Analyzer



4 ICG:

~~t_1 = int to float (2)~~

~~t_2 = int to float (70)~~

~~t_3 = id2 + t_1~~

t_1 = int to float (70)

t_2 = id1 + t_1

t_3 = t_2 + id2

t_4 = int to float (2)

t_5 = t_3 + t_4

id1 = t_5

~~4. MD Code Generation~~

5. Code optimizer

t_1 = id1 + 70.0

t_2 = t_1 + id2

t_3 = t_2 + 2.0

id1 = t_3

6. MD Code Generation

```
LDF   R1, id1.
MULF  R1, R1, #70.0      R1 = R1 + 70.0
LDF   R2, id2
ADDF  R1, R1, R2         R1 = R1 + R2
ADDF  R1, R1, #2.0       R1 = R1 + 2.0
STF   id1, R1
```

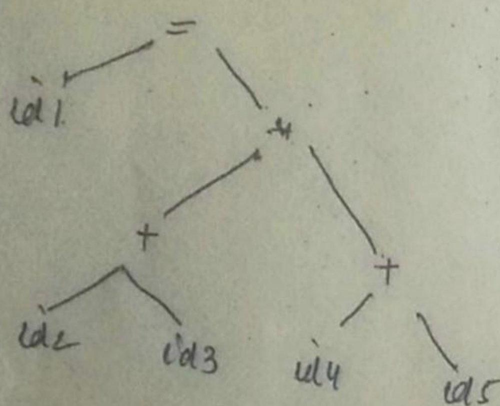
Q2 Show the o/p generated by each phase for the following expression: (8C)

$$x = (a+b) * (c+d)$$

1) <u>Lexical Analysis</u>:

$id1 = (id2 + id3) * (id4 + id5)$

2) <u>Syntax Analysis</u>:



3) <u>Semantic Analysis</u>:

Since there is no constant, no type casting is required. The o/p parse tree is ~~the~~ correct

4) <u>Intermediate Code Generator</u>:

$t_1 = id_2 + id_3$
$t_2 = id_4 + id_5$
$t_3 = t_1 * t_2$
$id_1 = t_3$.

5) <u>Code optimization</u>:

$t_1 = id_2 + id_3$
$t_2 = id_4 + id_5$
$id_1 = t_1 * t_2$.

5) ~~Code Optimization~~: Machine-Dependent Code Generator:

~~LD R_1, id_2~~

LD R_1, $id2$ [Load id_2 in Register R_1.]

ADD R_1, R_1, id_3. [$R_1 = R_1 + id_3$]

~~ADD R_2, id_2~~:

LD R_2, id_4. [Load id_4 in R_2]

ADD. R_2, R_2, id_5 ~~Add~~ [$R_2 = R_2 + id_5$]

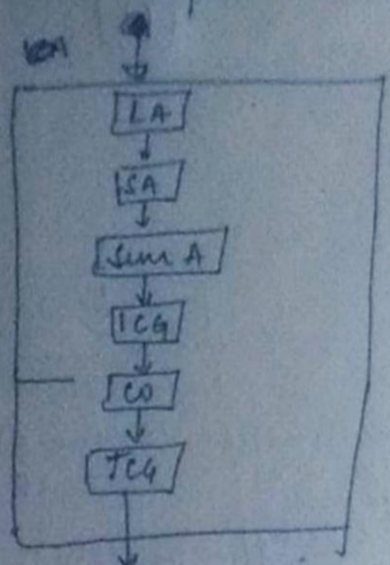
MUL R_1, R_1, R_2 [$R_1 = R_1 * R_2$]

ST id_1, R_1 [store R_1 in id_1]

The Grouping of Phases into Passes

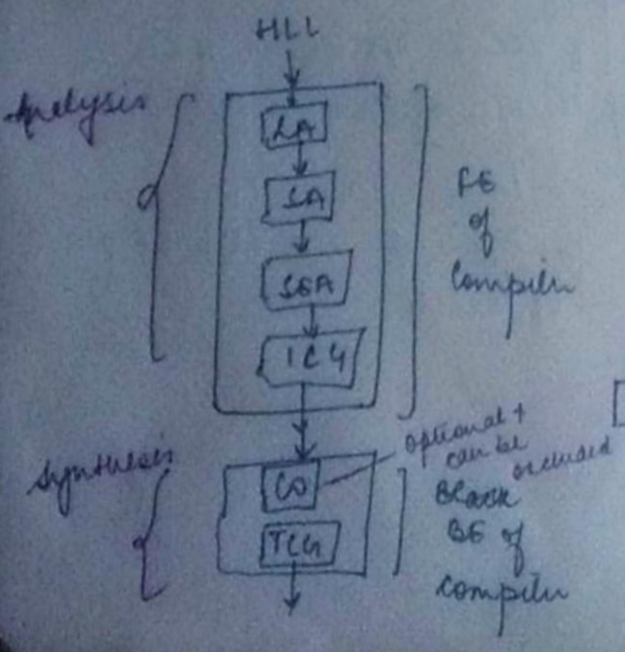
1. **Single Pass Compiler**: All the phases are grouped into one part

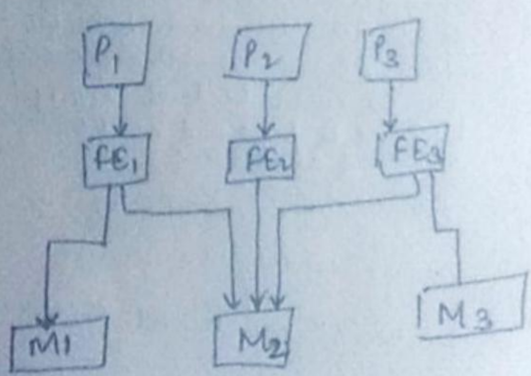
Source Program (HLL)

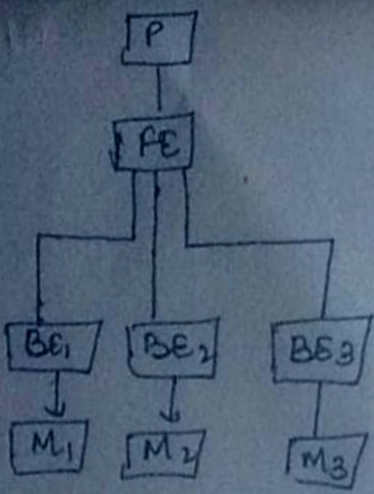


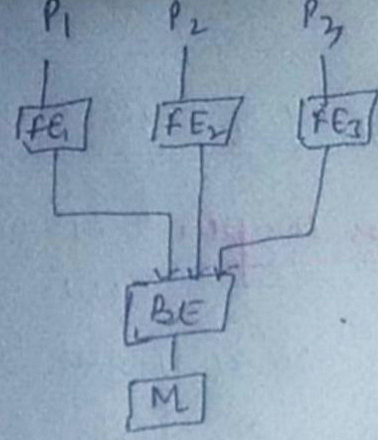
Target machine code. (Assembly code)

2. **Multi Pass Compiler**: If more than one phases are grouped in to two or more parts.









	Type of compiler	Advantage	Disadvantage
1.	Single pass compiler	• Faster than multipass compiler because there is no communication gap. • Components are closely related because they are on same machine • No need to keep track of intermediate file	• Consumes more memory space at runtime. • Difficult to handle error
	Multipass compiler	Consumes less memory space on run time ~~because either FE will be under operation or BE~~ because either backend or front end will be executing at a time. • Portability as FE is programming language dependent while BE is machine dependent. while FE & BE are independent of each other • Easy error handling	• Slower than single pass because of communication gap. • Components i.e. FE & BE are not related.