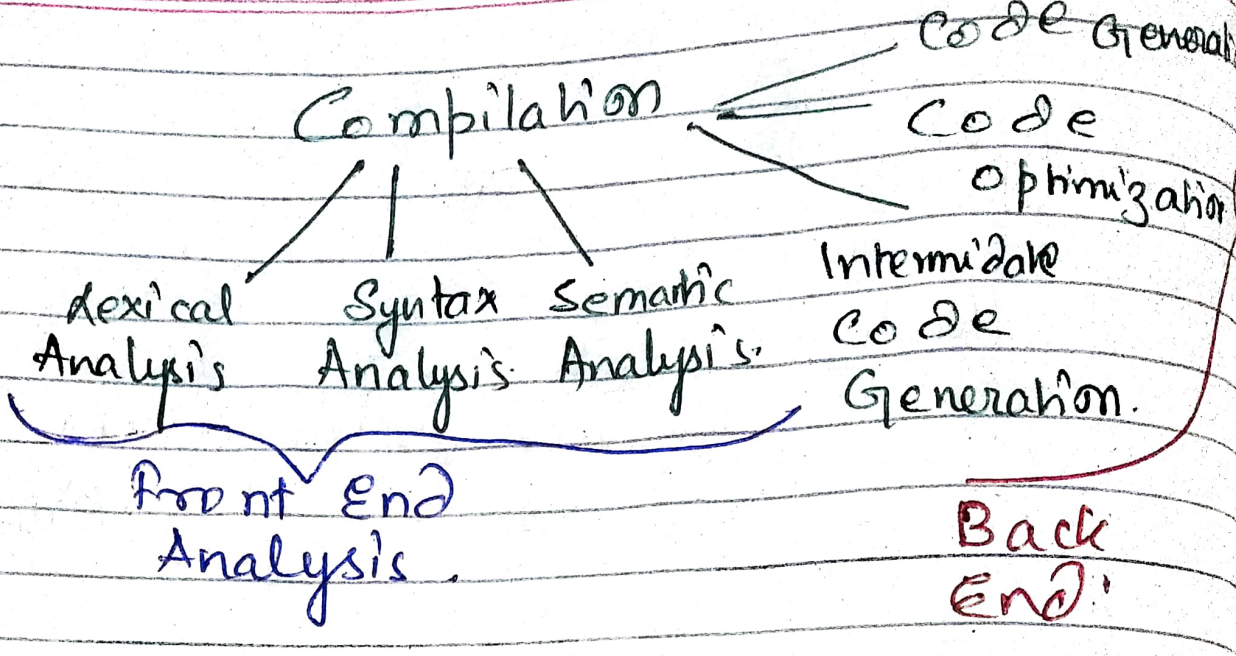
Sankhadeep Gir

Compiler Design

18/10/19.



- Source code broken down into Tokens.

⇒ **FRONT END ANALYSIS of Compilation:**

- **lexical analysis**

→ checks the validity of each word or token in the sentence. (or each keyword).
→ Output is token stream.

- **Semantic Analysis**

→ Helps us to come out of Ambiguous condition.
→ Output is Parse Tree.
E.g. → Scope of variable.

Syntax Analysis

→ Checks the syntax of a statement.

⇒ **BACK END ANALYSIS OF COMPILATION:**

- **Intermidiate Code Generation:—**

a = b + c * 10;

- Parsed through lexical Analyzer.

E.g. 1

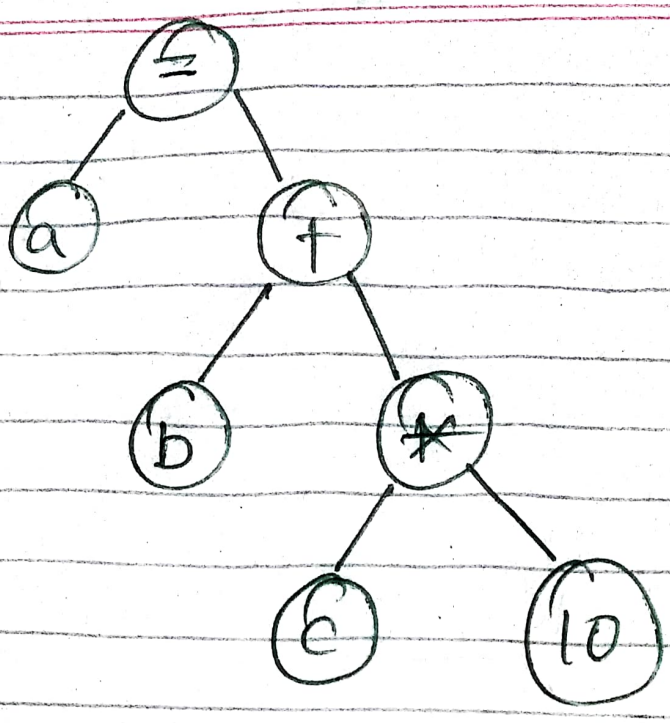
a	=	b	+	c	*	10	;
↓	↓	↓	↓	↓	↓	↓	↓
⟨id,1⟩	⟨=⟩	⟨id,2⟩	⟨+⟩	⟨id,3⟩	⟨*⟩	⟨10⟩	⟨;⟩

Token (output)

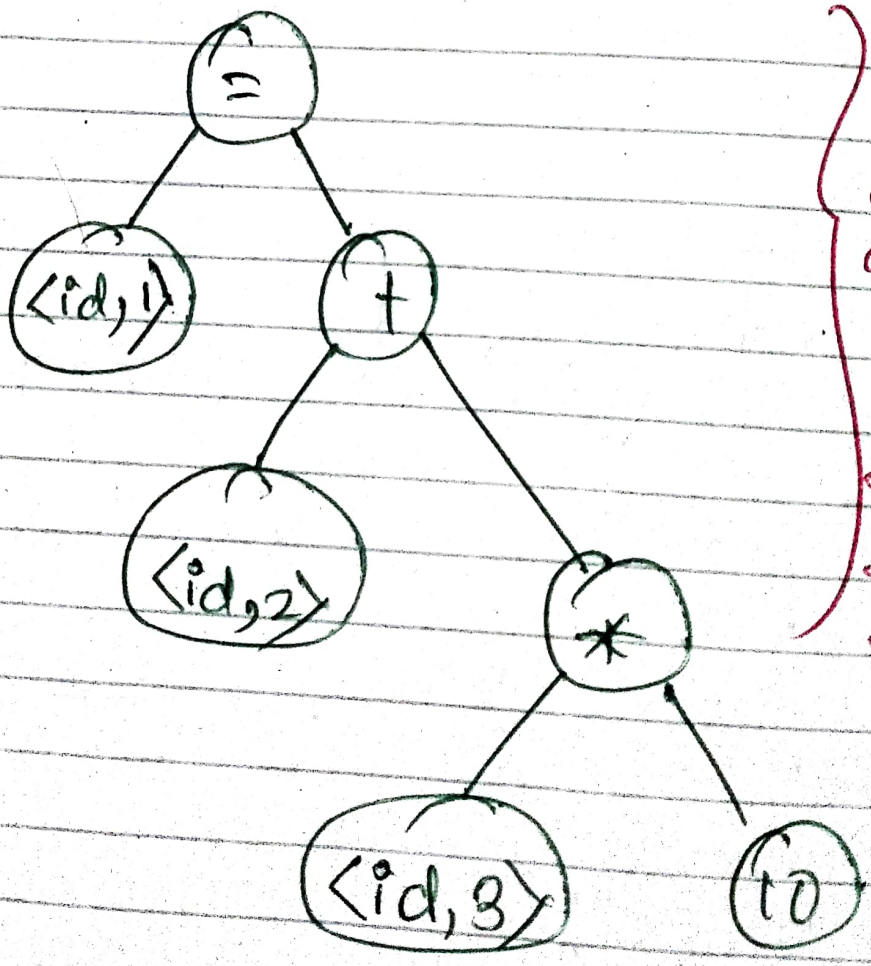
Generated

- Parsed through Syntax Analyzer

- Output will be parse tree.
- The output, i.e, tokens Generated after lexical Analyzer is parsed to it.



But in exam write :



Parse Tree as generated as output of Syntax Analyzer

(1) • Parent must have operator.

(2) • Left child should have variable, i.e., identifier.

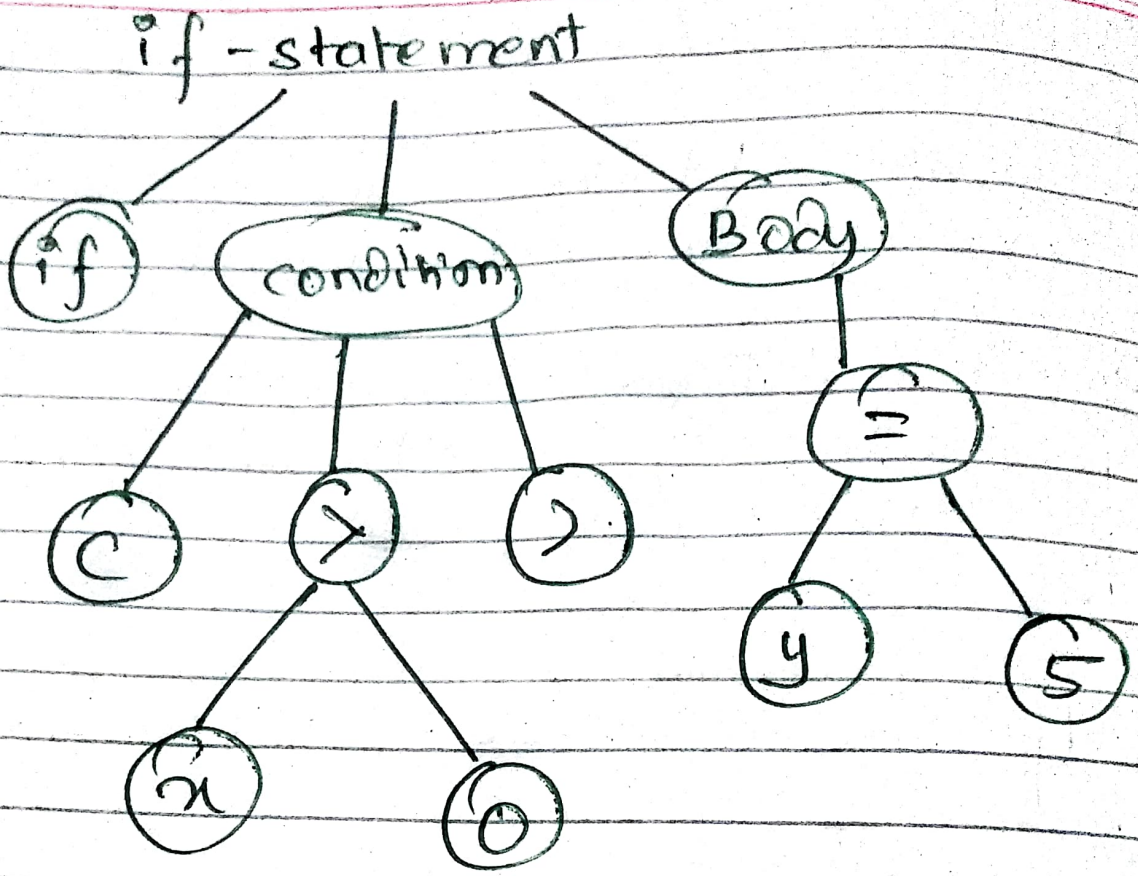
(3) • Right child can be anything or any expression include constant, operator or variable.

If these three point is not satisfied in parse tree then statement is false.

• Eg. 2

if | (|x|>|0|) | y| =|5| ;|.

<id,1> <id,2>



- Semantic Analysis

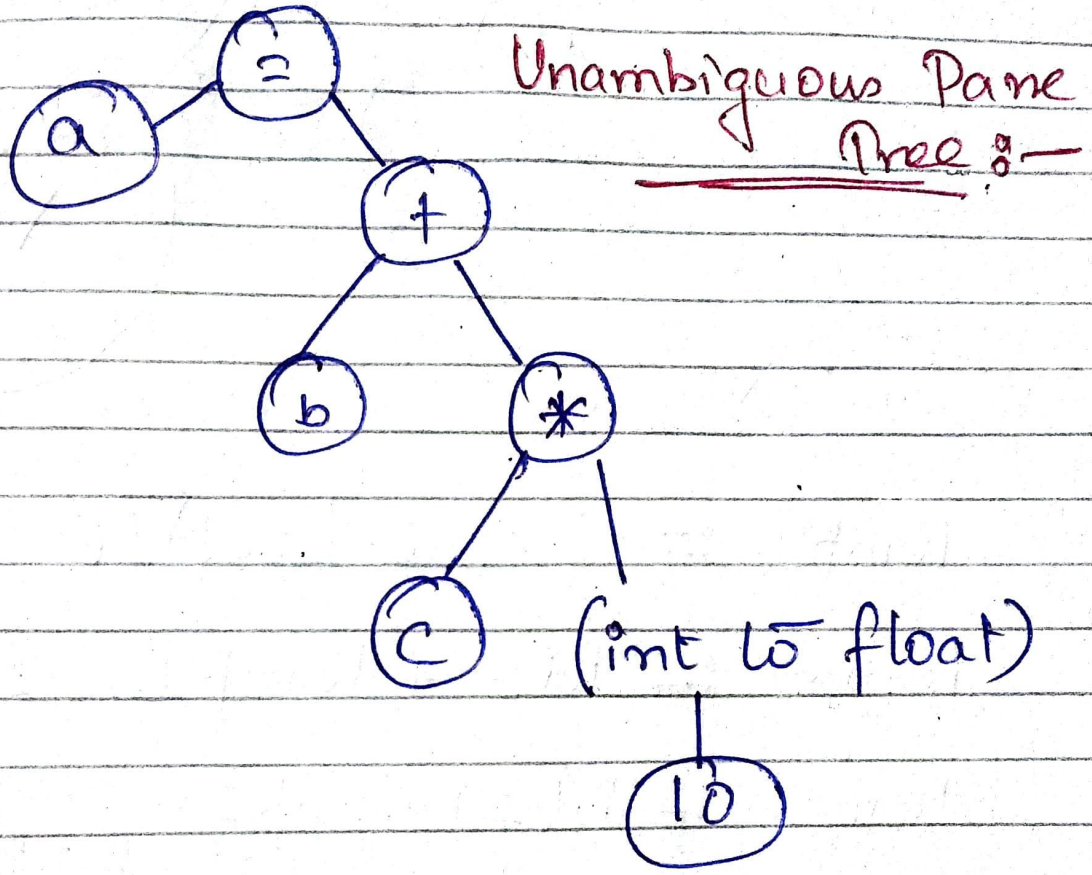
⇒ Takes cares of ambiguity in parse tree

e.g. → implicit type casting

- For eg. (1) there can arise an ambiguity that all the variable is float

and with that 10 an integer is added. Then parse tree becomes :-

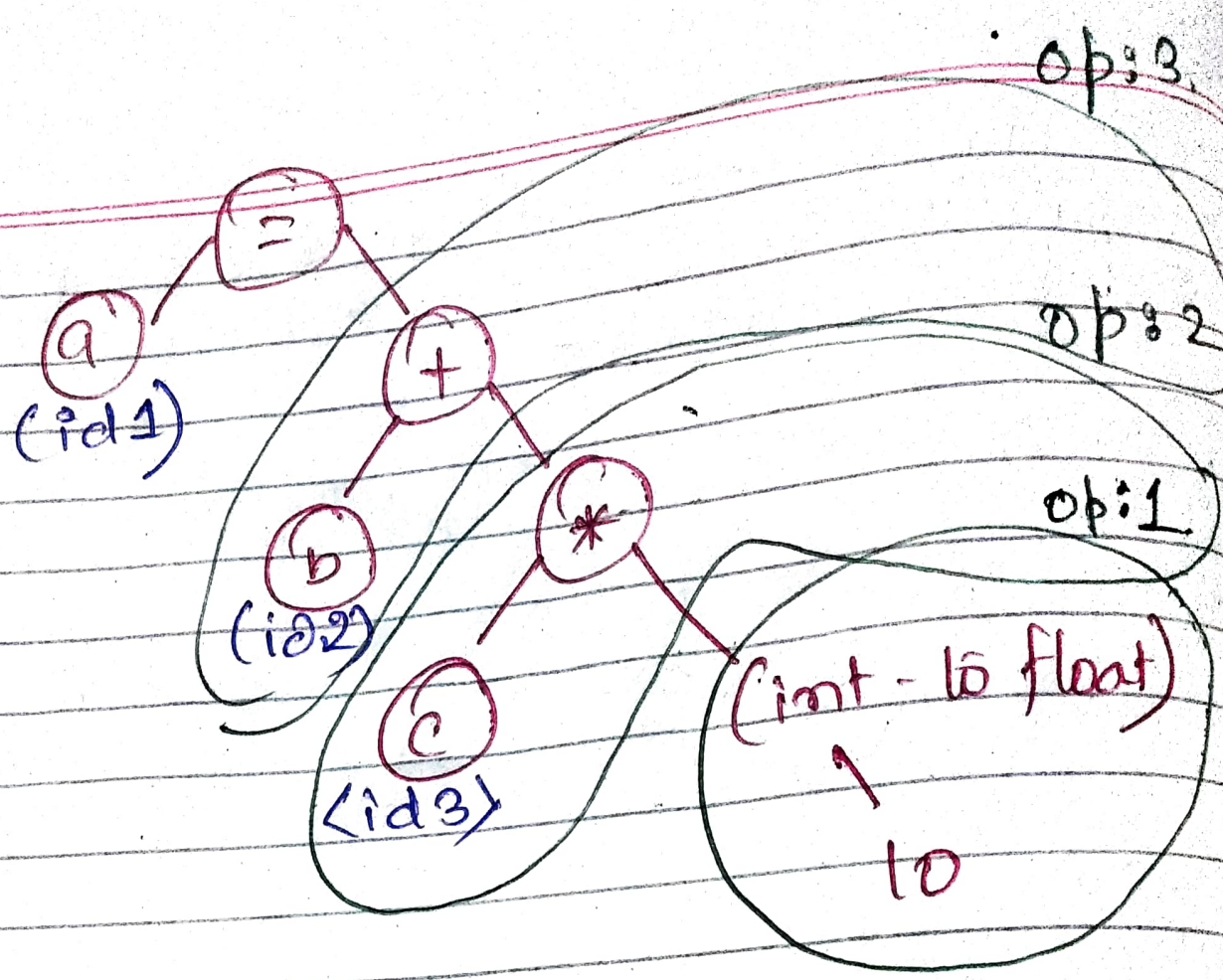
a = b + c * 10 ;



Unambiguous Parse Tree :-

Three Address Code :-

It helps to convert syntax tree (semantic parse tree) to assembly level code.



Registers

temp1 := (int-to-float)10

temp2 := id3 * temp1

temp3 := id2 + temp2

} not optimize code.

id1 := temp3

→ main memory

- **Code Optimization :—→**

Process of using minimum no. of statement to run a code. Reduce the amount of resources & time.

temp1 := id3 * (int to float) 10.

id1 := id2 + tem1

- **Symbol Table :—** is produced in the lexical analysis stage. It consists of info. about the operands including their scope, data type, storage class etc.

1	a	S.C.	D.T.
2	b		
3	c		
4	d		
5	e		

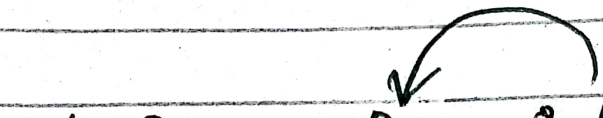
Advantage :— (i) Use of less no. of registers.
(ii) Time Complexity Reduces

- **Code Generation :-**

→ High level language is converted to low level language.

→ Resource Processors use Registers to store Data to speed up the efficiency.

Code :-



```
LDF     R1, id3
MULF    R1, R1, 10.0
LDF     R2, id2
ADDF    R2, R2, R1
STF     id2, R2.
```

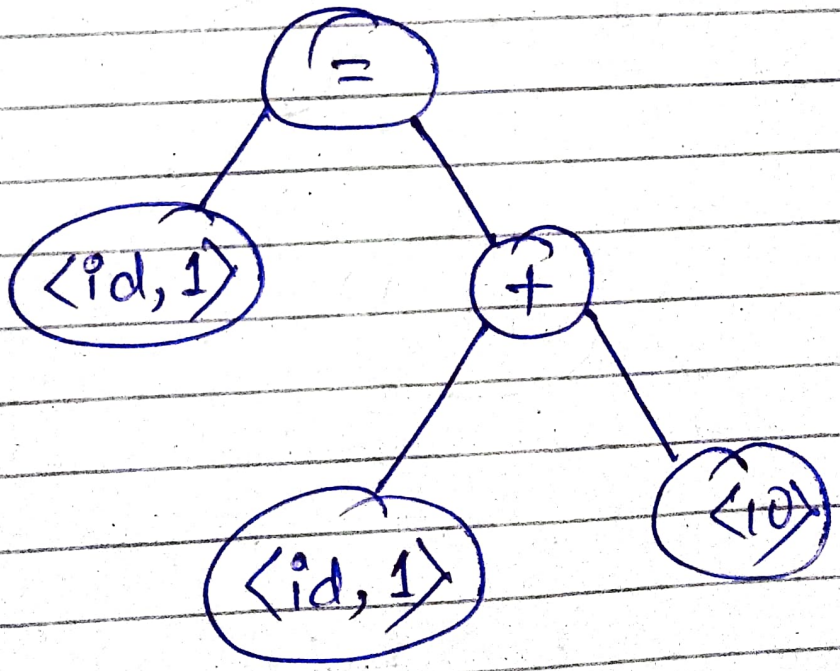
LDF = load Float
MULF = Multiplication Float

Problem : 1 :- x = x + 10 ;

Step 1 → Lexical Analysis.

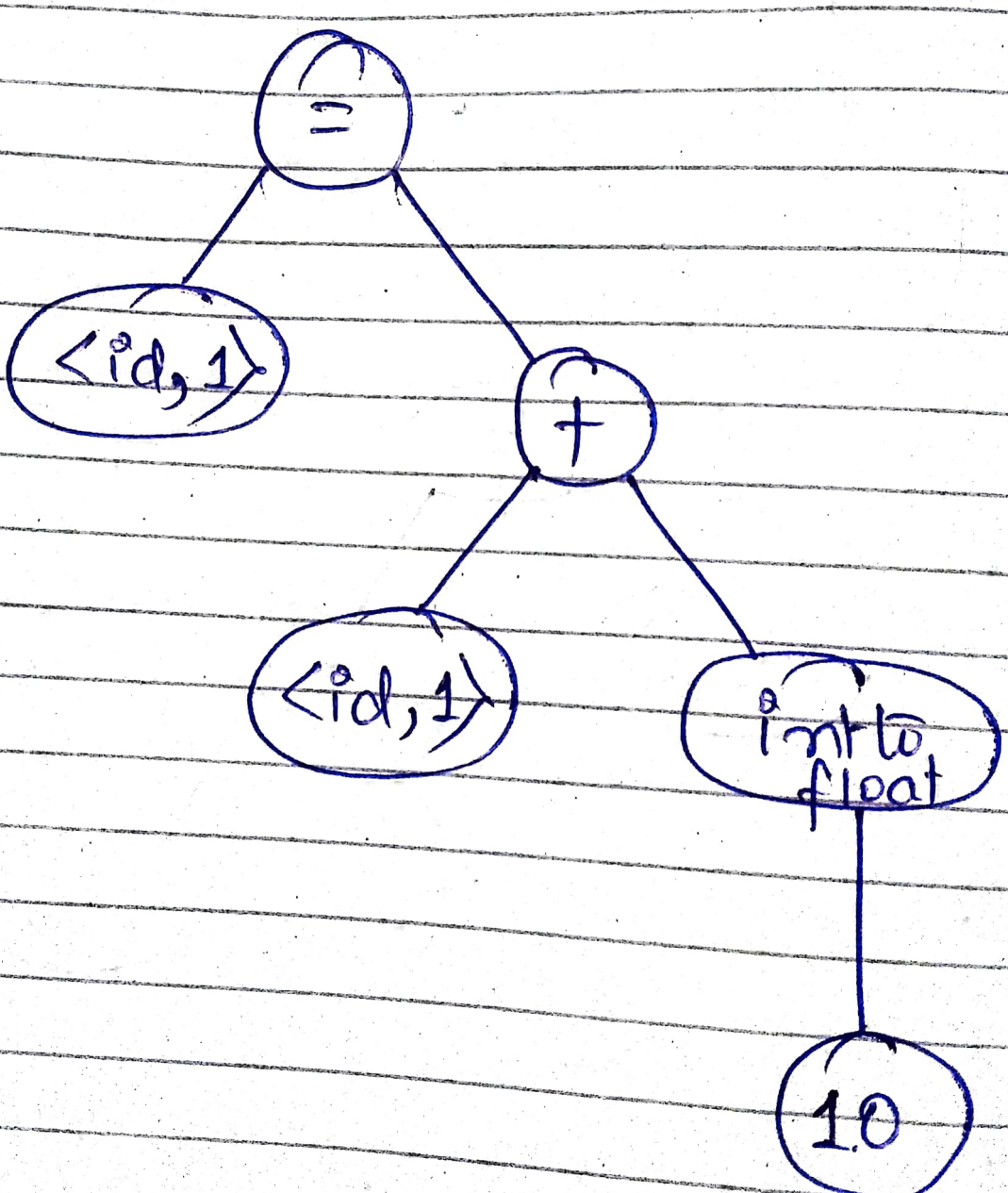
x	=	x	+	10	;
↓	↓	↓	↓	↓	↓
<id, 1>	<=>	<id, 1>	<+>	<10>	<;>.

Step 2 :- Syntax Analysis / Parse Tree



Step 3 :- Semantic Check.

If x = float or double then ambiguity arises means (float + integer), then not possible.



Step4 :- Intermediate Code.

temp1 := (into-to-float) 10'

temp 2 := . <id, 1> + temp 1

<id, 1> := temp 2.

Step5 :- Code Optimization

~~temp1 := <id,1> * (int-to-float) 10.~~

id1 := ~~temp1~~ . <id,1> + (int-to float) 10.

Step6 :- Code Generation.

LDF R1, id1.

ADDF R1, R1, 10.0.

STF id1, R1.