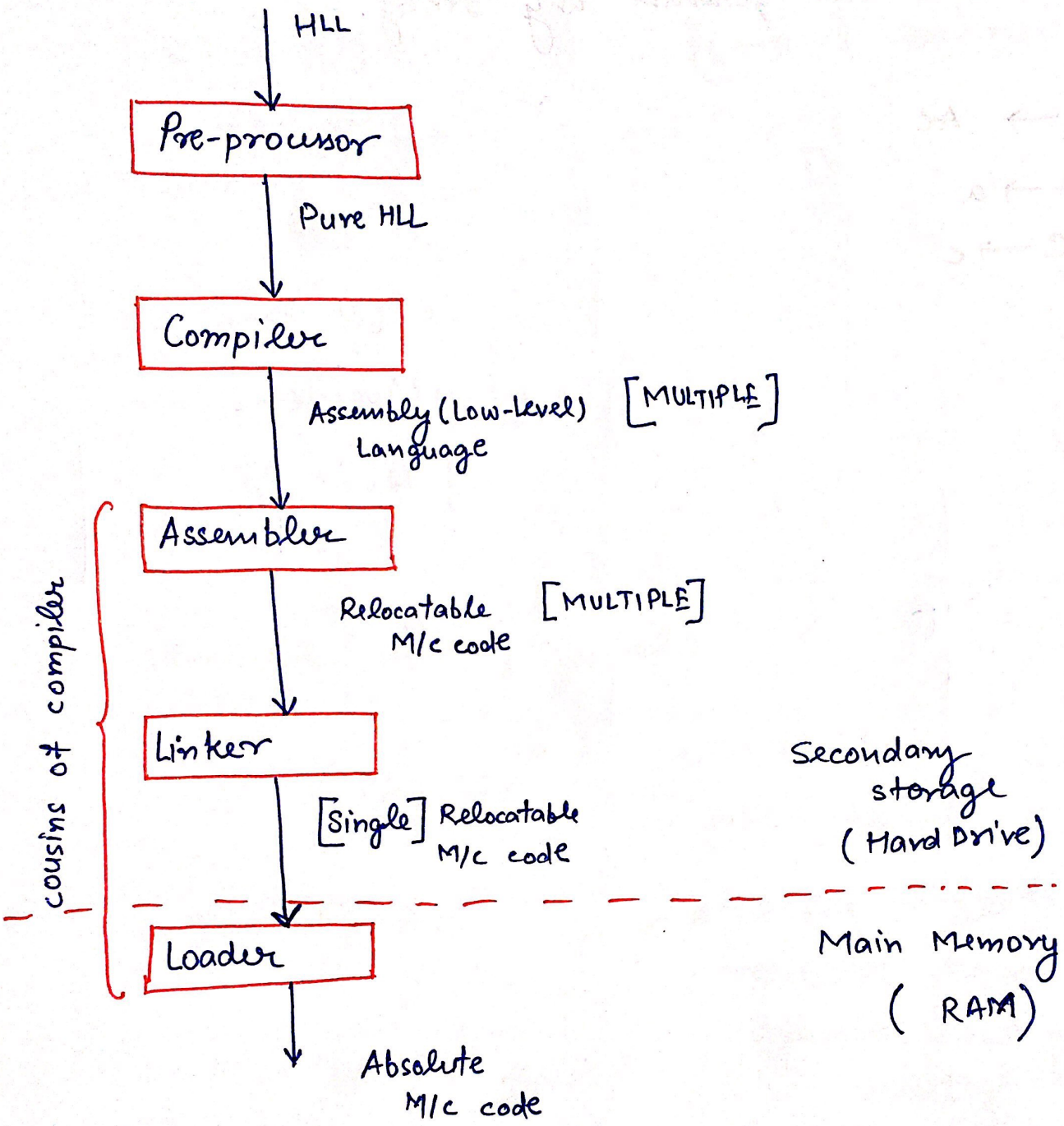
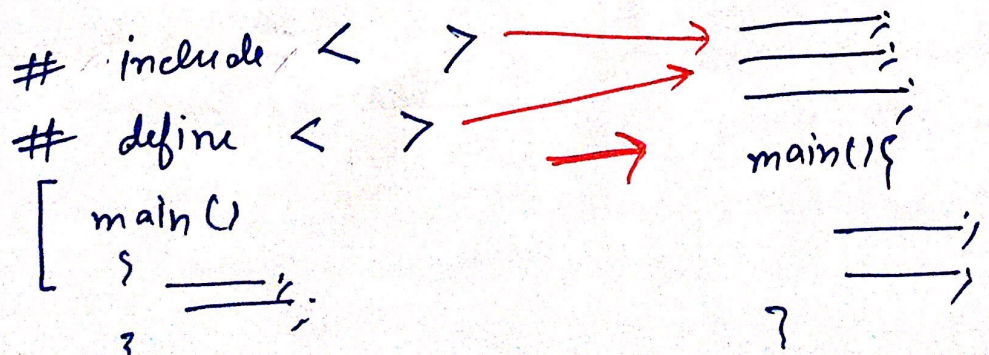
Introduction to compiler



(i) **Pre-processor:-** Convert HLL to Pure HLL i.e substitute header file with there code.



(ii) **Compiler:-** Converts Pure HLL to Assembly (Low-level) Language.

```
main()                          main:
{     x = a + b;      ⇒          MOV a, R0;
}                                MOV b, R1;
                                 ADD R0, R1;
                                 MOV R1, x;
```

(iii) **Assembler:-** Converts Assembly language to relocatable M/c code. Relocatable means independenty of type of memory.

MOV a, R0 ⇒

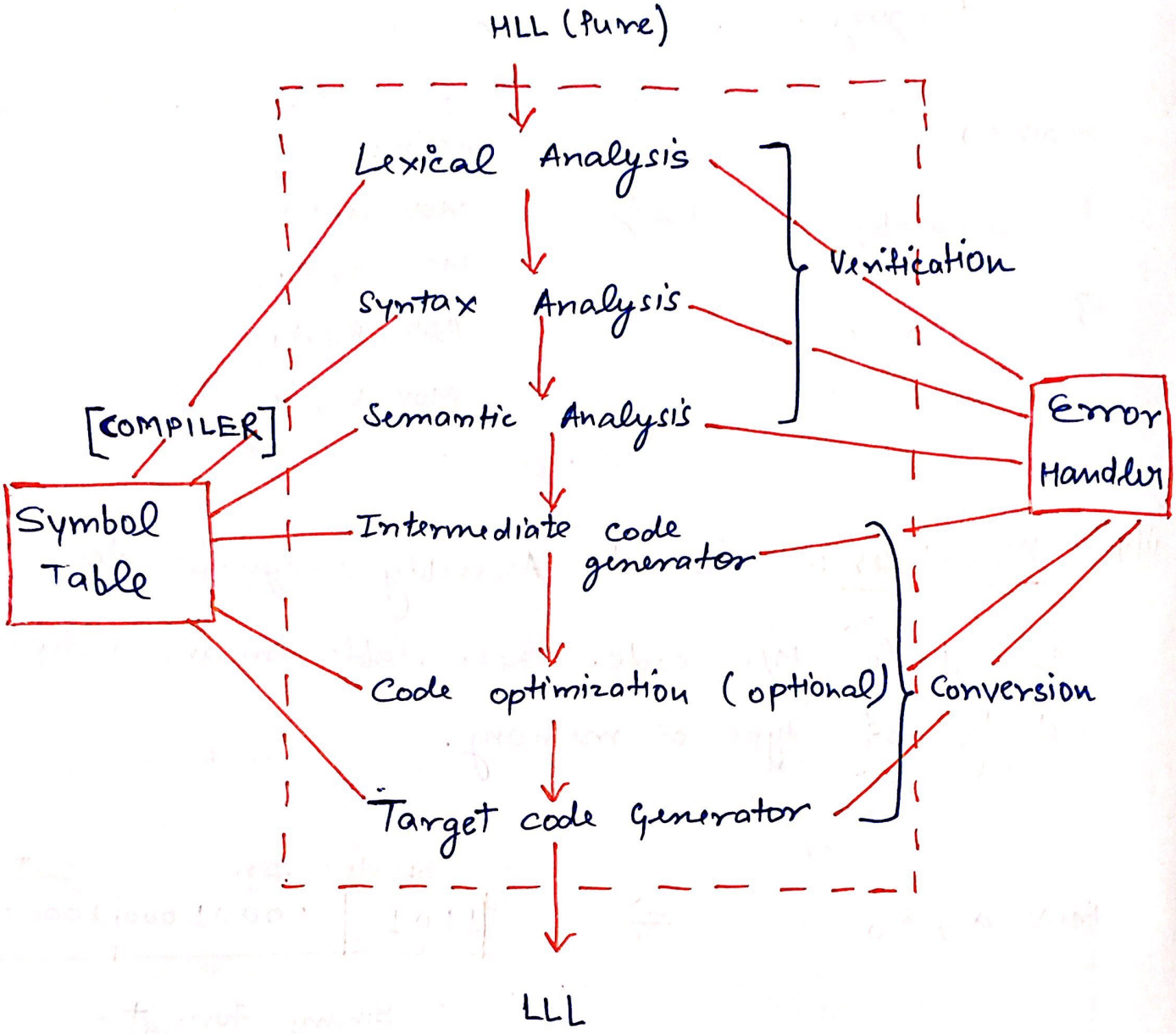
opcode	Src	Dest
1101	10001000	10001111

Binary format

(iv) **Linker:-** It links multiple M/c code file into a single M/c code (Relocatable)

(v) **Loader:-** It executable code LOADL into Main memory (RAM) from secondary memory

Phases of compilation :-



(i) Lexical Analysis :-

[HLL ⟶ tokens]

$x = a + b;$

id = id + id ;

tokens string

x	id
a	id
b	id

=	op
+	op

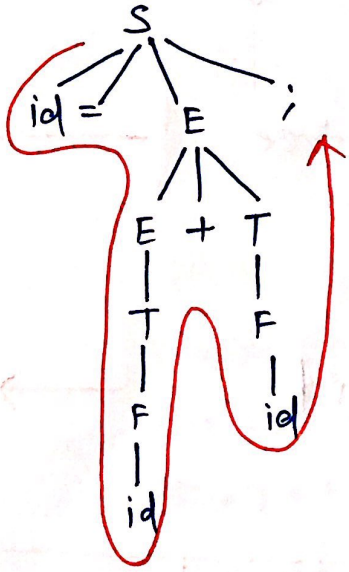
(ii) **Syntax Analysis :-** It verify whether the string of token is syntactly correct or not. It is done by Parser which use a CFG (rules) to generate Parse tree.

[token string ⟶ Parse tree]

If parse tree is deriving the same token string then I/p is syntactly correct, otherwise it raises Syntax Error.

P: $S \rightarrow id = E;$
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$

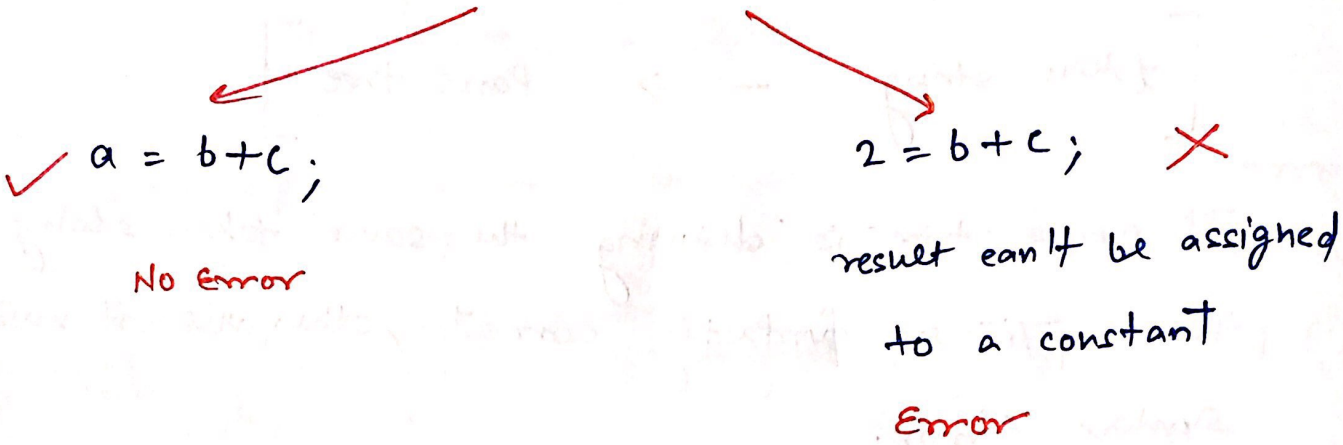
I/p: id = id + id ;



Since it derives same I/p string so it is correct

(iii) Semantic Analysis :- It modifies the parse tree and check whether it is meaningful ie it checks attributes of identifiers.

eg:



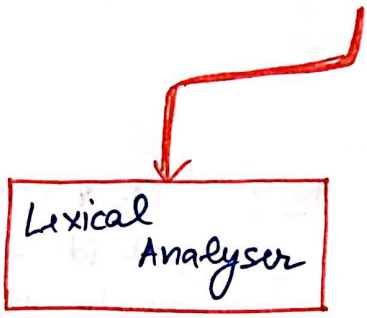
As, we can see both statements are syntactly correct but 2 = b+c; is not semantically correct ie meaningless.

[Parse tree ⟶ Modified Parse tree]

$id = id + id \longrightarrow id.^{type} = id_1.^{type} + id_2.^{type}$

S.no	id/func	V.name	V.Type
1	id	x	int
2	id	a	int
3	id	b	int

Eg:- $x = a + b * 60$



x — identifier — (id,1)

= — assignment op

a — identifier — (id,2)

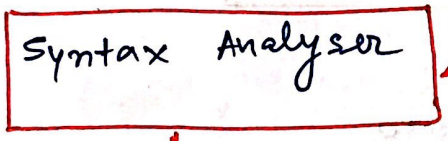
+ — add op

b — identifier — (id,3)

* — mul op

60 — int const

(id,1) = (id,2) + (id,3) * 60



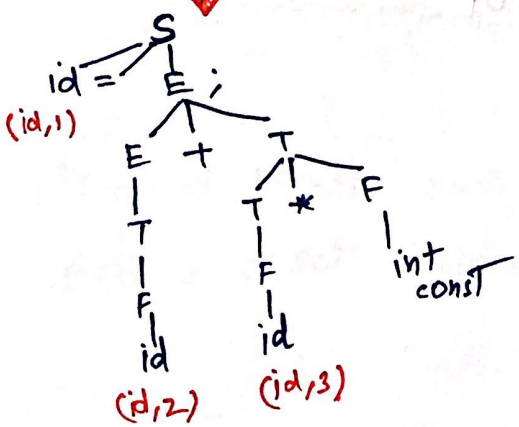
CFG:

S → id = E ;

E → E+T | T

T → T * F | F

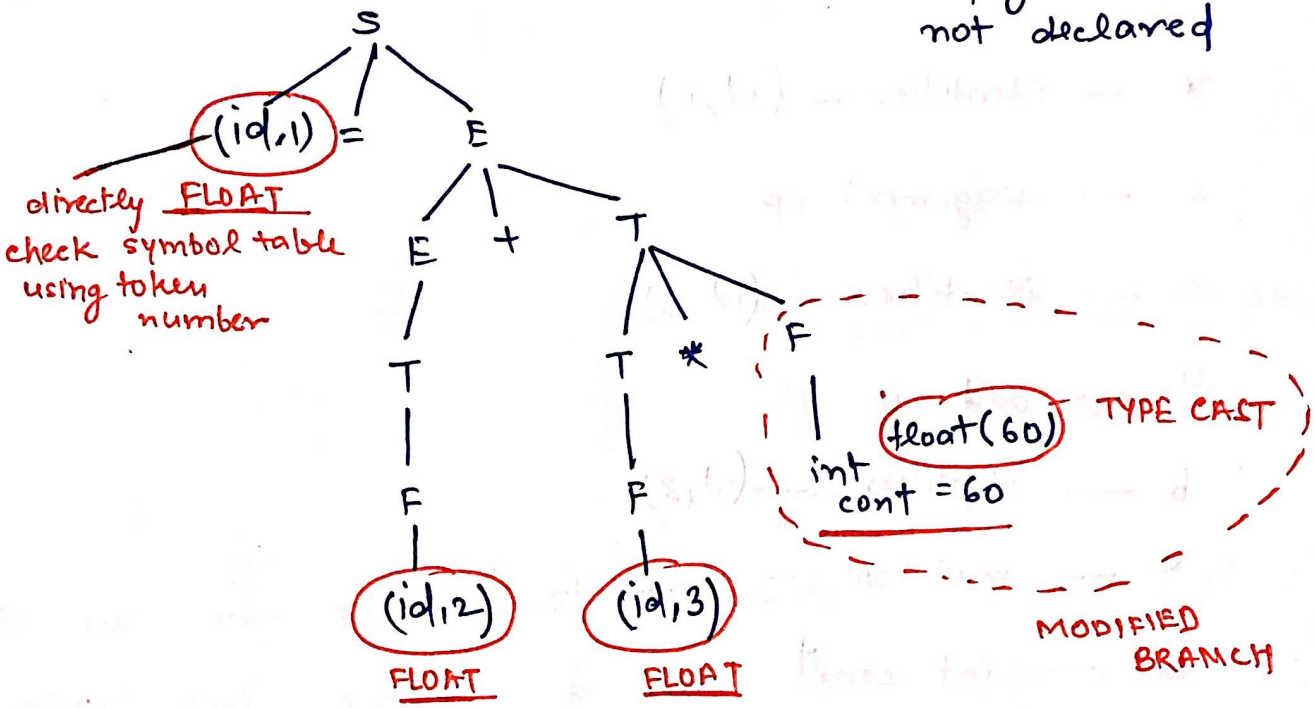
F → id | int const



if Parse tree possible then no Error

Semantic Analyser

NOTE: if TYPE column of id token is empty then var is not declared



(Float = Float + Float * Float)

Intermediate code Generator

$$t_1 = b * 60.0$$
$$t_2 = a + t_1$$
$$x = t_2$$

Code optimization

$$t_1 = b * 60.0$$
$$x = a + t_1$$

Target Code Generator

```
MUL   60.0 , R0
ADD   R0 , R1
MOV   R1 , x
```

```
MOV b, R0
MOV a, R1
```

NOTE:- Most of the time semantic analyser store into in Symbol Table, but if some undeclared variable comes than Lexical analyser store it with leaving all the fields empty and release a token for future reference.

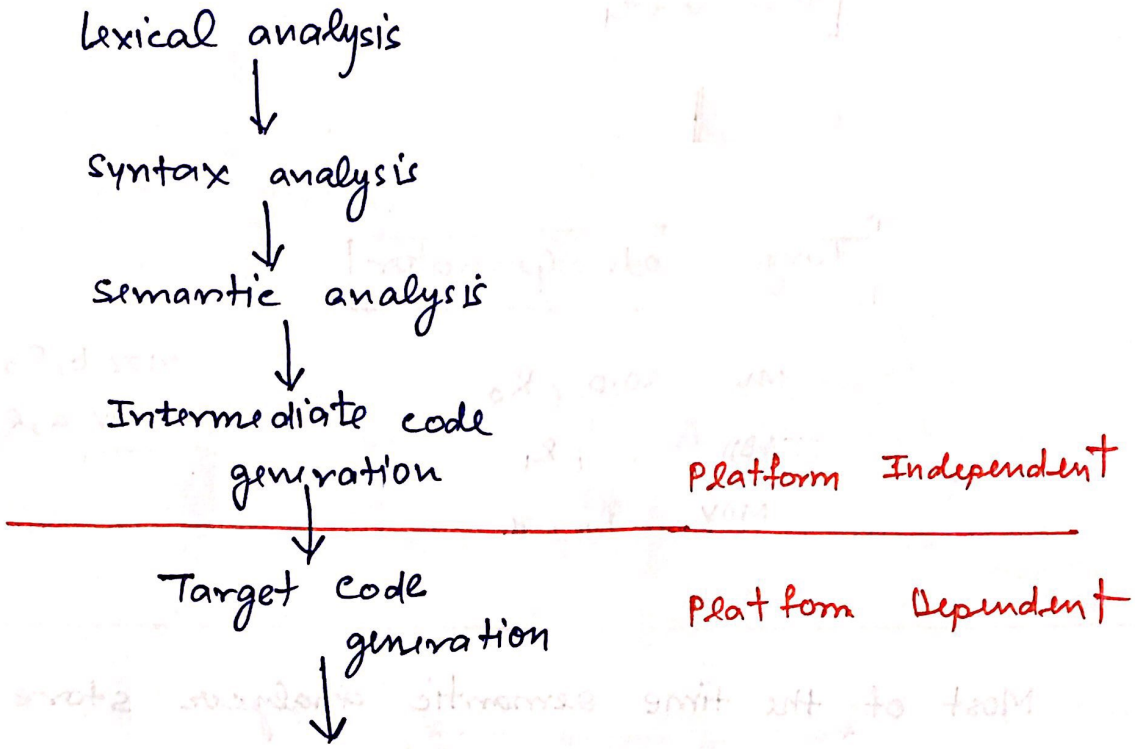
eg:- float x, a, b;

By semantic analyser ⇒

S.no	name	type
1.	x	float
2.	a	float
3.	b	float

NOTE:- Symbol Table only contains user defined variable, func, objects, interfaces.

- After compilation, o/p is M/c dependent code i.e it can only runs on the machine in which compilation is done.



= o/p of ICG : Platform Independent code

= o/p of TCG : Platform dependent Code

- So compiler o/p depends upon whether TCG is included or excluded.

eg.

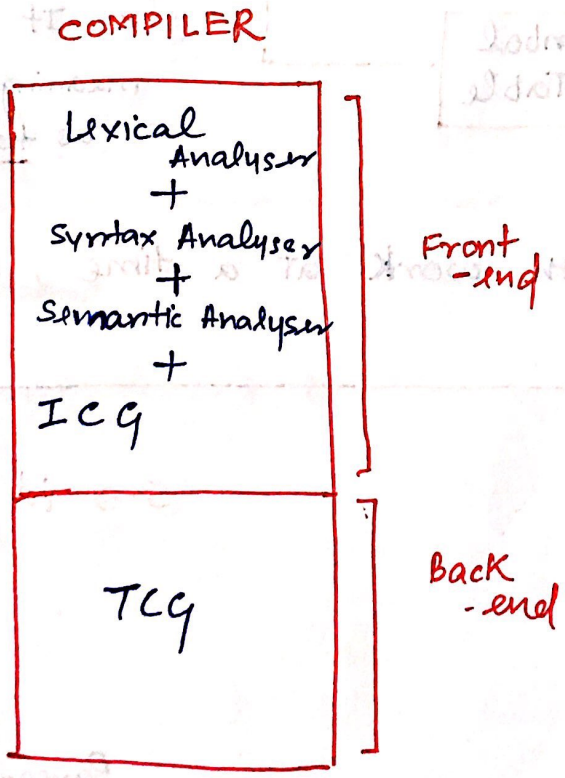
JAVA compiler :- Exclude TCG

Plate form independent

C/C++ compiler :- Include TCG

Plateform dependent

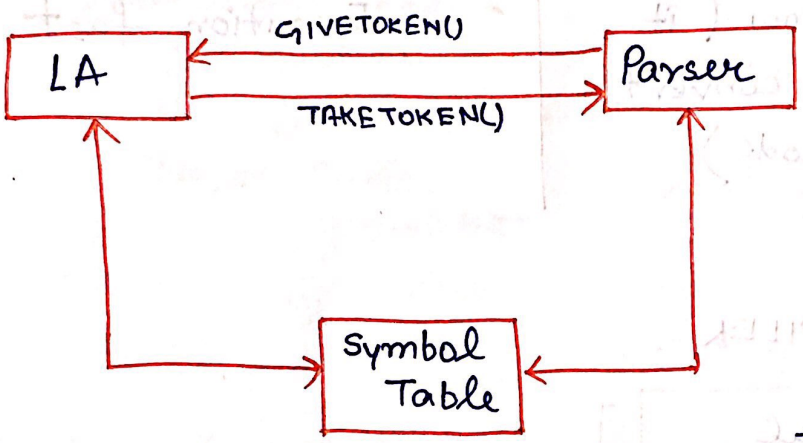
JAVA compiler	C/C++ compiler
- Half compilation	- Full compilation
- Exclude TCG	- Include TCG
- Plate form independent	- Plate form dependent
- Portable	- Not Portable
- Execution slow (it has to first convert into M/c code)	- Execution fast



- **Single pass compiler :-** Total compilation process (phases) at a time (Less time, more space)

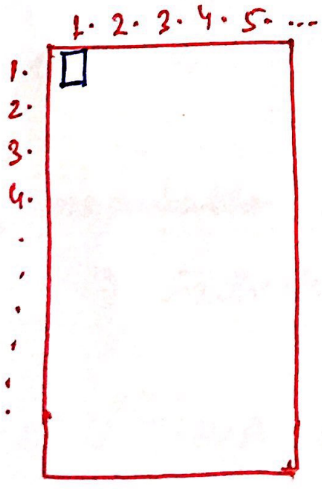
- **Multi passes compiler :-** compilation process at time frames. (More time, Less space)

Lexical Analyser :-



- LA is first phase of compiler, also known as scanner
- It divides into meaningful strings known as tokens

- LA and Parser both work at a time, side-by side,



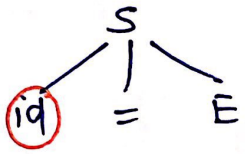
LA is at row = 1 & column = 1

$S \rightarrow id = E \mid AB \mid CD$

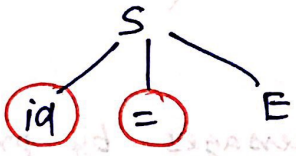
Parser at S

I/p :- id = E

- LA send I^st token (id,1) to parser
- Parser checks production for id alphabet

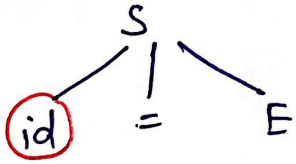


- Then Parser again ask for token from Parser



I/p :- Id + E

- LA send I^st token id,1
- Parser checks for id



- Now LA sends '+' token, But Parser Expect for '=' so Error is generated

---

- Types of tokens :-

(i) Identifiers

(ii) operators

(iii) Keywords

(iv) Constants

= Responsibilities of LA :-

(i) Dividing given program into tokens.

eg. int a, b;
(1) (2) (3) (4) (5)

Tokens: <int/1>, <a/2>, <,/3>, <b/4>
<;/5>

(ii) It will eliminate whitespace character (tab, new-line, gap)

(iii) It will eliminate comment lines {/* */}

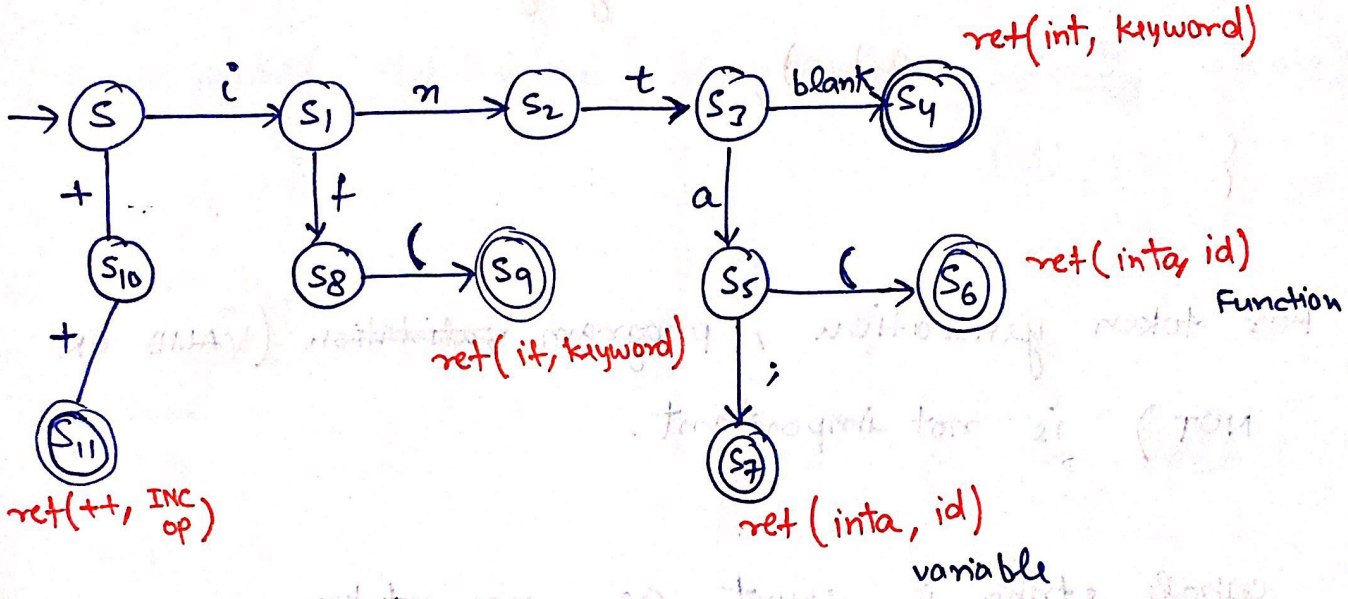
(iv) It will help in giving error messages by providing (row, column) number.

Q. Find number of tokens present in the following C program :

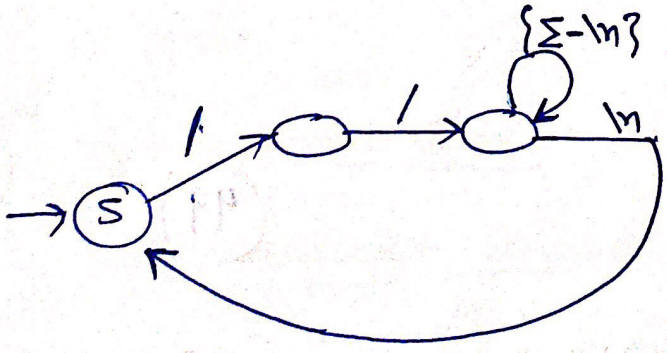
```
int main()
{
    /* finding max of a & b */
    int a=10, b=20, max;
    if (a<b)
        max=b;
    else
        max=a;
    return (max);
}
```

// **Token Creation :-** LA use FSA (finite state Automata) to create token.

Abstract DFA :-

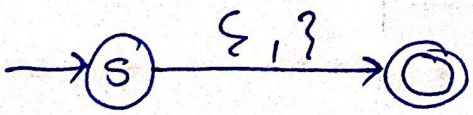


- DFA is provided to LA, which includes finite states. It includes every possible combination for identifying tokens.

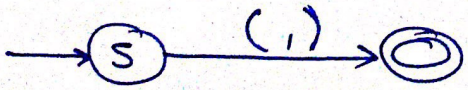


DFA for comment

'Σ' means any alphatet



DFA for brackets



DFA for paranthesis

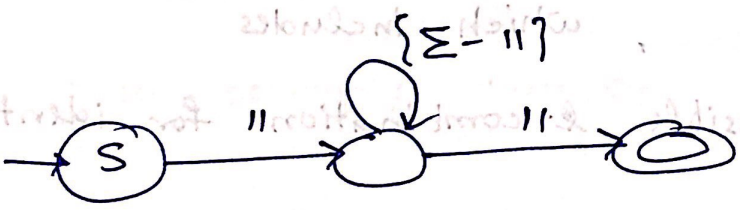
Q. Find no. of tokens for the following program:

```
int main()
{
x = a + b * c;
int x, a, b.
printf( " you & your family go to hell %d %d ",
                a,b,c);
}
```

(32)

For token generation, program . (VALID OR NOT) is not important.

whole string is count as one token.



Q. Find no. of tokens -

```
main()
{
x = a+b-c;
y = ++|++|-+ =|==| *|/* +++++ ----- == - =-
    *** +++ === *** /|+ - + -;
comment
printf( " %d , %d %d ", a, *a, **a, ***a, &&a);
}
```

(49)

3 — main()

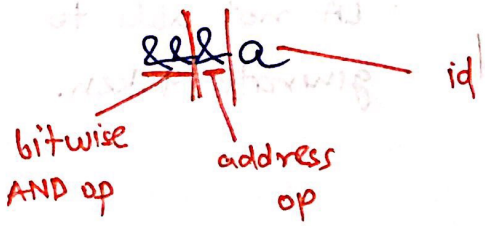
1 — {

8 — x = a + b - c;

9 — y = ++|++|--|-=|==|*|/* ++ ..... ***/|

5 — +|-|+|-|;

23 — printf("%d %d %d", a, *a, **a, &&a, ***a);



||*|a

= *(*(*(a)))

Q. Find no. of tokens :

```
3 main()
1 {
6     x = a+b;
13 [  y = xab - 123 abcdef *** ghi *** /*** comment
      x = ab + cd - ghabcd /***** abcd
1 }
```

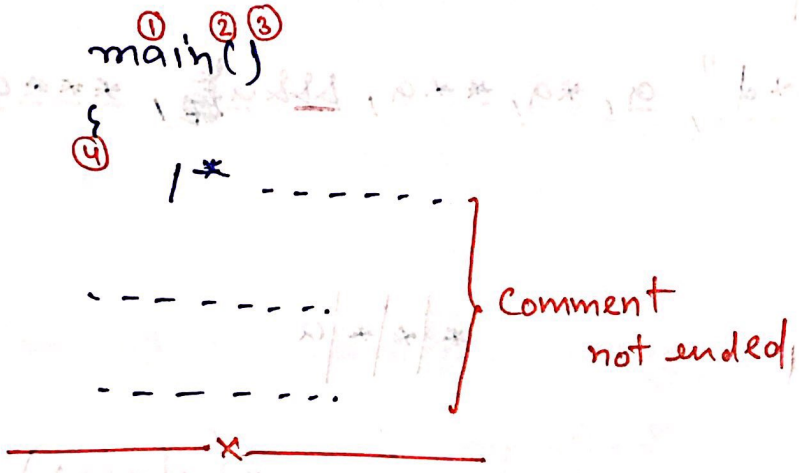
(24)

comment not counted

NOTE:-

Lexical Error :- When LA not able to generate token from given string.

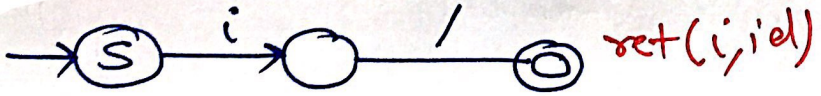
eg:- comment started but not ended and program is ended.

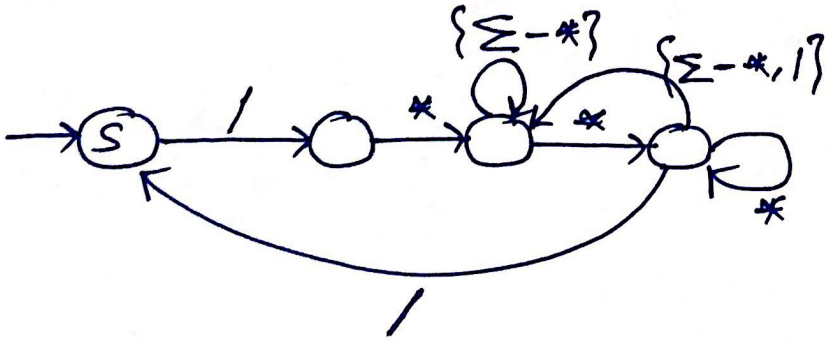


Q. Find number of token:

```
main()
{
    int x=10;
    i nt  x = 10;
    i /* comment */ nt  x=10;
}
```

(22)





Multiline comment

Q. which one of the following string is said to be token without looking next input char in C lang?

✗(a) return

✗(b) if

✗(c) main

✗(d) +

✓(e) ;

✓(f) ,

✓(g) (or)

Q. A lexical analyzer uses the following patterns to recognize three token T_1 T_2, & T_3

T_1 : $a?(b|c)^*a$

T_2 : $b?(a|c)^*b$

T_3 : $c?(b|a)^*c$

x? mean 0 or 1 occurence of x

I/p: bbaacabc ; which one of the following is the seqmence of tokens it outputs? which has longest matching?

(a) $T_1 T_2 T_3$

(b) $T_1 T_1 T_3$

(c) $T_2 T_1 T_3$

(d) ✓ $T_3 T_3$