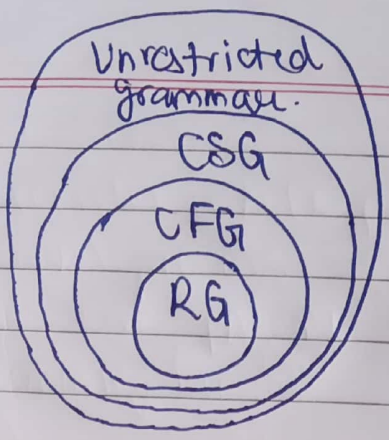


Regular Grammar :-

- least powerful.
- Finite length grammar.
- 4-tuples (V, T, P, S)
- Two Form → left linear R.G.
  → right linear R.G.

(1) Left linear Regular Grammar

S → Aaa
S → BBa
S → Xaaa

* Non-terminal on the left most side.

(2) Right linear Regular Grammar

S → aaAA
S → aBaB
S → aXaaXB

* Non-terminal on the right most side.

Grammar	4 tuple
Regular Grammar	4 tuple
context free Grammar	4 tuple
Finite Automata.	5 tuple
Machines	6 tuple
context sensitive Grammar	7 tuple.
Push down Automata	7 tuple
Turing Machine	7 tuple
Linear Bounded Automata	8 tuple.

Grammar :- (4 tuple)

V T P S → Starting symbol

V → Non-terminal or variable

T → Terminal Symbol

P → Production Rule.

RG < CFG < CSG < UG

1. Terminal (a-z) (0-9)
2. Non-terminal or variable (A-Z)

* Compiler :-
→ compiler executes whole program at a time.
→ Software development.
→ eg. C, C++, Java, PASCAL, Basic, cobol.
(Programming lang.)

* Interpreter :-
→ line by line execution.
→ eg. Python, Perl, ASP.NET, Ruby (Scripting lang.)
→ Web based language.

COBOL → Common Business Oriented Language.

Basic → Beginner All Purpose Symbolic Instruction Code.

Q) S → aAA
S → aab
A → a

Right linear R.G.

Q) S → aaAa
S → AA
S → B
A → a
B → b

Left and Right linear R.G.

Q) S → aAa
S → aBa

Neither left nor right linear R.G.

Q) $S \to aaAaA$
$B \to bb$

Left linear R.G.

Q) $B \to b$
$A \to aAa$
$A \to aa$

Neither left nor right R.G.

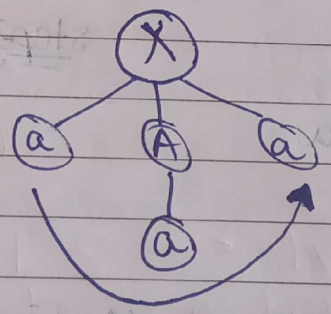
Q) $A \to aAA$
$B \to BBE$

Both left and right R.G.

Derivation Tree :-

* Starting Node → Root Node.
* Vertex → Non-terminal
* Leaf Node (leaves) → Terminal.

eg. $X \to aAa$
$A \to a$



aaa.

(1) Top-down derivation tree
(i) Start with root node.
(ii) Goes down toward leaf nodes (leaves)

(2) Bottom up derivation tree.
(i) Start with leaves.
(ii) Towards root node.

① Left linear Derivation Trees:

$S \to aAaA$

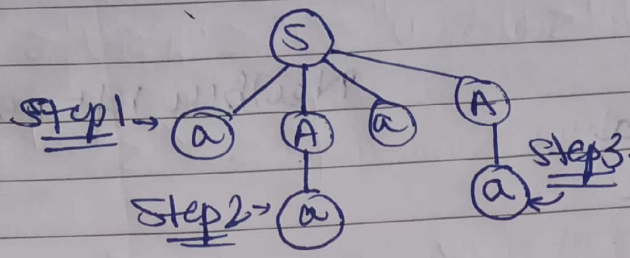
$A \to a$

$S \to aaaA$

$A \to a$

$S \to aaaa$

$A \to a$.



aaaa.

② Right linear Derivation Trees:

$S \to aAaA$

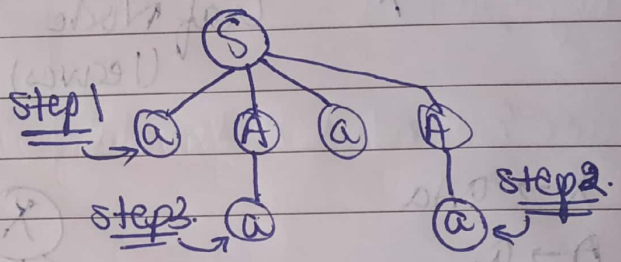
$A \to a$

$S \to aAaa$

$A \to a$

$S \to aaaa$

$A \to a$



aaaa.

Ambiguity Grammar (Ambiguous)

① Left linear derivation tree.
② Right linear derivation tree.
③ Left and right linear derivation tree.

} Any one

$S \to A \times A$

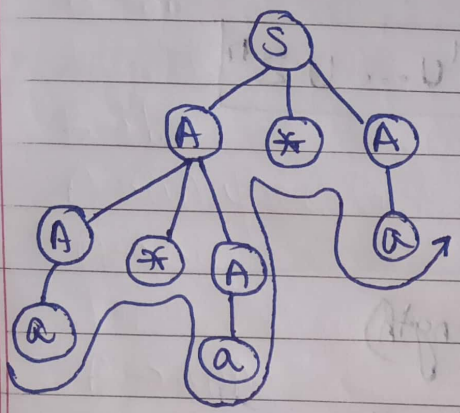
$A \to A + A$

$A \to A * A$

$A \to a$

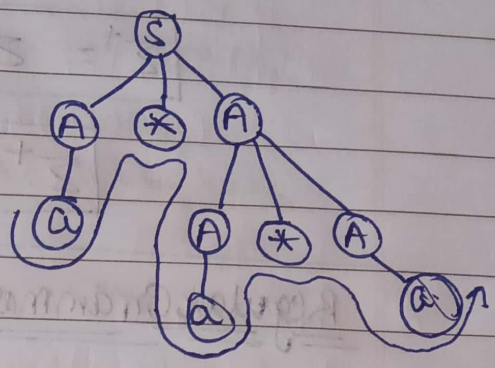
$B \to b$

Left linear.



a*a*a.

Right linear.



a*a*a.

★ Power of Sigma :-

$\Sigma = (a, b)$

① $\Sigma^0 = \{\}$ or ϕ or $\in$ or NULL

② Σ^1 = only one value. (set of all strings of length 1)

③ Σ^2 = set of all strings of length 2.

④ Σ^3 = set of all strings of length 3.

⑤ Σ^4 = set of all strings of length 4.

⑥ Σ^n = set of all strings of length n.

★ Kleene closure (*) :-

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots \cup \Sigma^n.$$

* Kleene plus :- (+)

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4 \cup \dots \cup \Sigma^n$$

$$\Sigma^+ = \Sigma^* - \Sigma^0$$

Regular Grammar :- (Fixed length)

V T P S → starting symbol

V → Variable or non-terminal

T → Terminal

P → Production Rule

① Left linear Regular grammar →

S → Aaa

A → b

② Right linear Regular grammar →

S → aaA

A → b.

③ Both type →

S → AaA

A → b

* Regular language is accepted by regular grammar ~~called regular grammar.~~

$a^* = (\text{empty}, a, aa, aaa, \dots)$

$a^+ = (a, aa, aaa, aaaa, \dots)$

$\Sigma = (a, b)$

$L^1 = (a, b)$

$L^2 = (aa, ab, ba, bb)$

$L^3 = (aaa, aab, aba, abb, baa, bab, bba, bbb)$

$L^4 = (aaaa, bbbb, \ldots)$

* Properties :

1. Union –

If we have 2 regular (grammar) language L_1 and L_2, then union of $L_1 \cup L_2$ are regular language.

$L_1 = (a, b)$

$L_2 = (b, c)$

$L_1 \cup L_2 = (a, b, c)$.

2. Complement –

If we have a regular language L, then complement of L' is a regular language.

$L_1 = (a, b)$ $\quad L_1' = (b, a)$

$L_2 = (a, c)$ $\quad L_2' = (c, a)$.

3. Concatenation – (.)

If we have 2 regular languages L_1 and L_2, then $L_1 \cdot L_2$ is also a regular language.

$L_1 = (a, b)$

$L_2 = (a, c)$

$L_1 \cdot L_2 = (aa, ba, ac, bc)$

4. Kleene closure – (*)

If we have 2 regular languages L_1 and L_2, then L_1^* and L_2^* are also regular languages.

$L_1 = (a, b)$ $\quad L_1^* = (\phi, a, aa, ab, bb, ba, \ldots)$

$L_2 = (a, c)$ $\quad L_2^* = (\phi, a, ac, cc, ca, \ldots)$

5. Intersection – (∩)
If we have 2 regular languages L_1 and L_2, then intersection of $L_1 \cap L_2$ is also a regular language.
$L_1 = (a, b)$
$L_2 = (a, c)$
$L_1 \cap L_2 = (a)$.

* Precedence :
1. Kleene closure (*)
2. Concatenation (.) → [complement / Intersection]
3. Union (lowest)

Alphabet :-
Alphabet is finite set of symbols
= (a, b, c, d, e, f)

Language :-
Language can be finite or infinite length.

L = (a, b, aa, ab, ba, bb)
L = (a, b, c, d, e, f, ...)

String :-
String is finite sequence of symbols
(ab), (01), (abc)

Length of String :-

Minimum → 0 (zero)
Maximum → ∞ (Infinity)

Set of String (or String length) :-
S → a String length = 1
S → aa String length = 2
S → aaa String length = 3

$\delta \rightarrow \phi$, empty, $\{\}$ string length = 0 (zero)

* No. of symbols.

Example of Recursive (Unrestricted Grammar) :- [accepted by Turing Machine]

① AaaA → AaA
AaA → AaaaA
aA → Aaa
A → a

Example of context Sensitive Grammar (CSG) :- [accepted by LBA]

aA → Aaa
A → aA
aA → AaaA
A → a.

* $[|LHS| \leq |RHS|]$

Example of context Free Grammar (CFG) :- [accepted by PDA].

① A → aa
A → AaA
A → Aaaa
A → Aaaaa

② A → aAa
A → aAA
A → AaAA
A → a.

* $[|LHS| = 1]$

Normal Form

- CNF (Chomsky Normal Form)
- GNF (Greibach Normal Form)

(1) ~~Chomsky~~ Greibach Normal Form (~~CNF~~ GNF)

(i) Non-terminal represent to empty symbol.

$A \rightarrow \epsilon$

(ii) Non-terminal represent to terminal.

$A \rightarrow b \quad A \rightarrow c.$

(iii) Non-terminal represent to 2 non-terminals

$A \rightarrow BC \quad A \rightarrow BD.$

$A \rightarrow AB$

(2) ~~Greibach~~ Chomsky Normal Form (CNF)

(i) Non-terminal represent to empty symbol.

$A \rightarrow \epsilon$

(ii) Non-terminal represent to terminal.

$A \rightarrow b \quad A \rightarrow c$

(iii) Non-terminal represent to terminal followed by non-terminal.

$A \rightarrow aB$

$A \rightarrow aBC$

$A \rightarrow aBCD$

Q) $A \rightarrow d$

$A \rightarrow \epsilon$

$A \rightarrow Aa$

$A \rightarrow a$

(Regular grammar)

Q) $A \to a$
$A \to \epsilon$
$A \to BC$
(CNF)

Q) $A \to a$
$A \to b$
$A \to aaA$
$A \to \epsilon$
(Regular Grammar)

Q) $A \to a$
$A \to \epsilon$
$A \to aBCDEFGHIJ$
(GNF)

Q) $A \to a$
$A \to \epsilon$
$A \to BC$
(CNF)

Q) $A \to \epsilon$
$A \to b$
$A \to aBC$
(GNF)

Context Free Language :-

(1) Union →
If we have two context free languages L_1 and L_2, then union of $L_1 \cup L_2$ are CFL.

(2) ~~Concate~~ Complement →
If we have two CFLs L_1 and L_2, then ~~union~~ of L_1' and L_2' are not CFL always.

③ Concatenation →
If we have 2 CFLs L_1 and L_2, then concatenation of L_1 and L_2 are CFL.

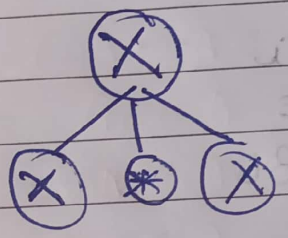
④ Kleene Closure →
(Followed by CFL).

⑤ Intersection →
(Not followed by CFL).

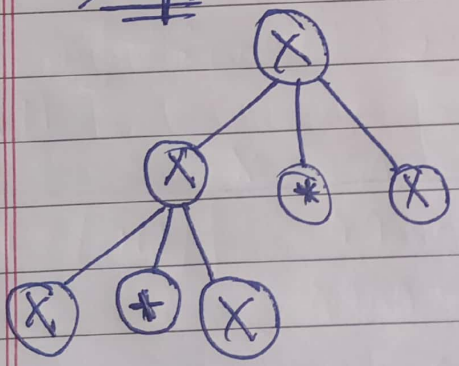
Q) Find out right most derivation Tree :-
Final string "a+a*a"

$X \rightarrow X+X$
$X \rightarrow X$
$X \rightarrow X*X$
$X \rightarrow a$

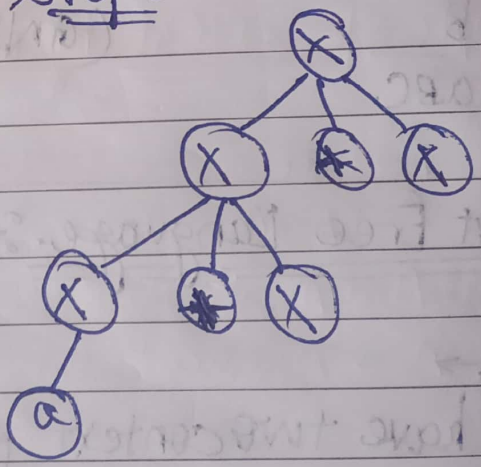
Step 1



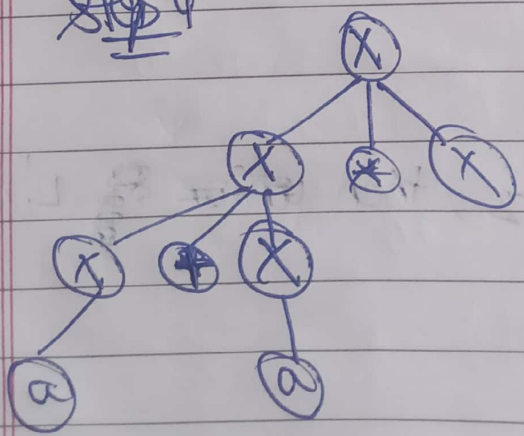
Step 2



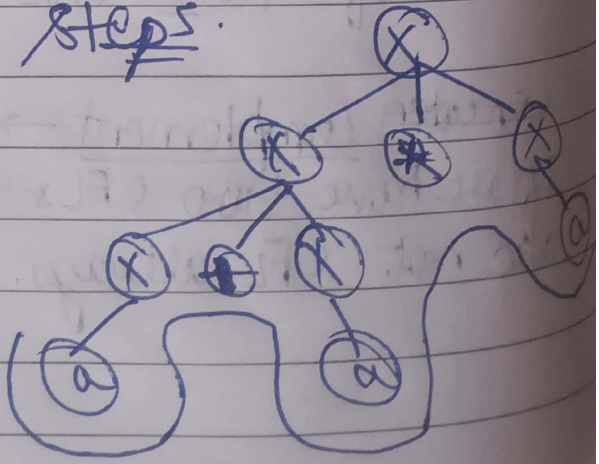
Step 3



Step 4

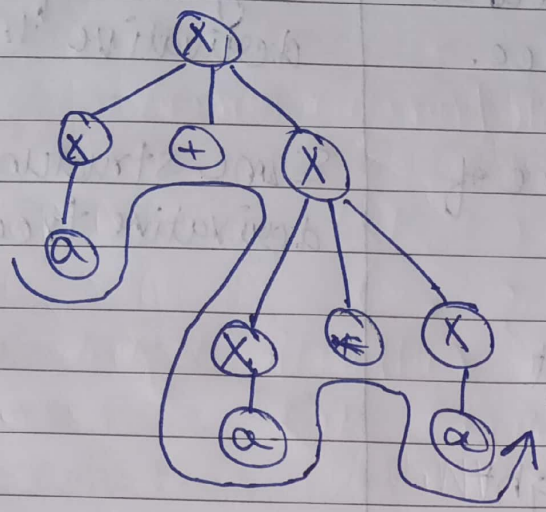


Step 5



Q) Find out leftmost derivative tree :-

Final string : [a+a* a]



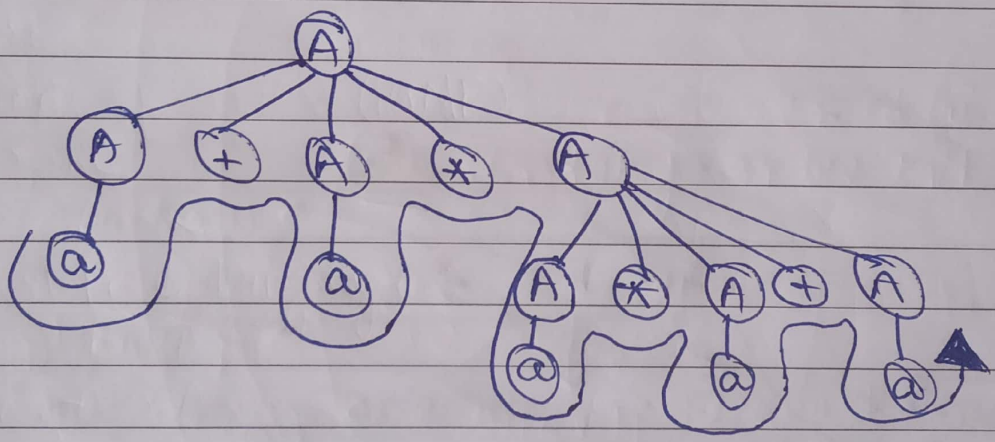
Q) Find out leftmost derivative tree :-

$A \rightarrow A+A*A$

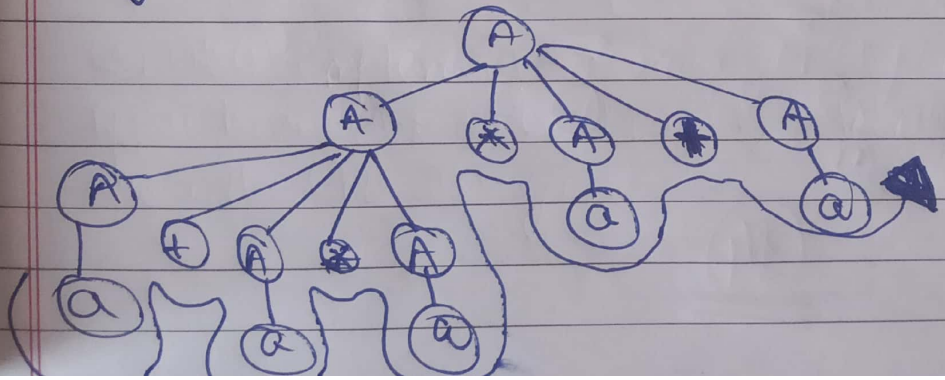
$A \rightarrow A*A+A$

$A \rightarrow a$.

["a+a*a*a+a."]



Rightmost derivative tree :-



Ambiguous	Unambiguous
(1) More than one left and right derivation tree.	only one left and right derivative tree.
(2) Different structure of derivative tree.	Same structure of derivative tree.
(3) (a) More than one left derivation tree. (b) More than one right derivation tree. (c) More than one left and right derivative tree.	

Compiler.

Phases of a compiler :-

* No. of phases = (6)
* First (3) phase called Analysis phase.
* Last (3) phase called Synthesis phase.

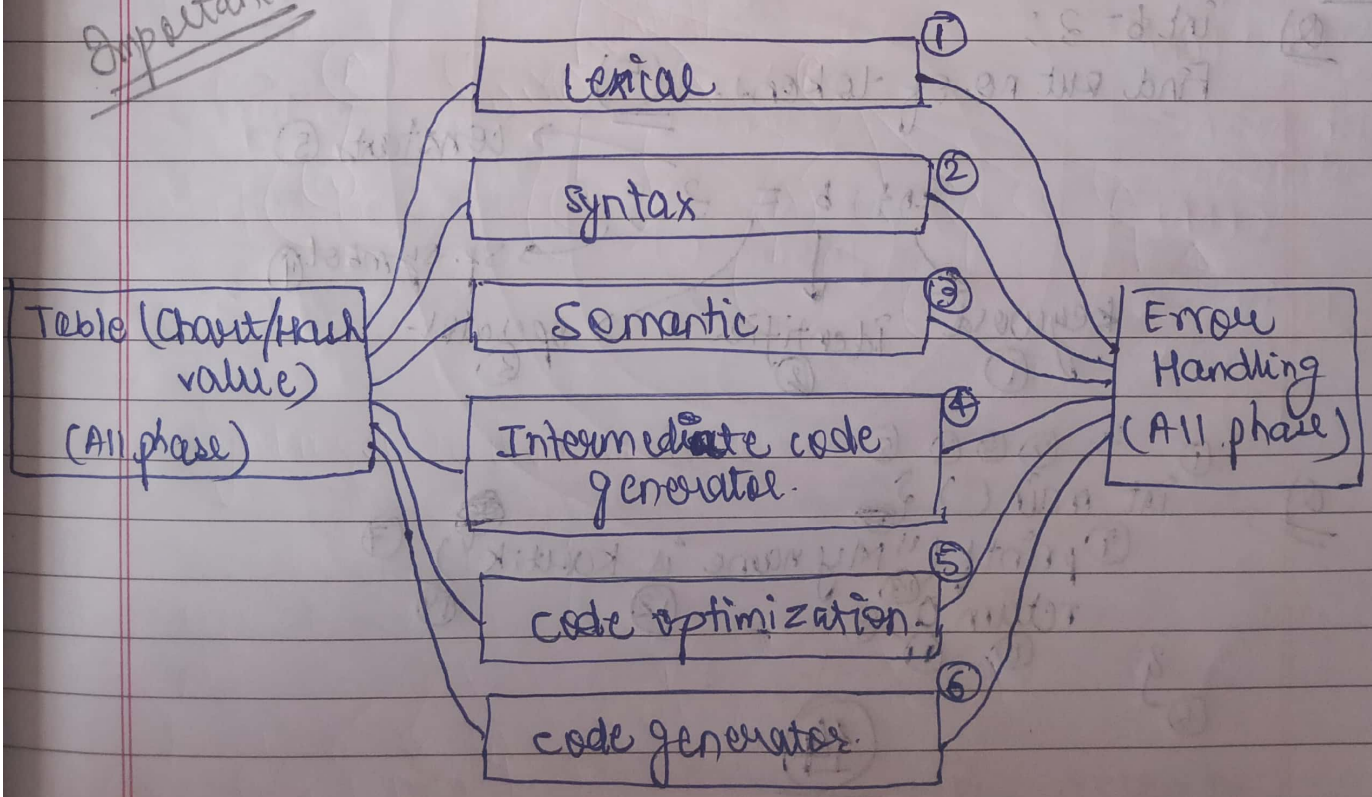
(A) Analysis Phase -

1. Lexical Phase.
2. Syntax Phase.
3. Semantic Phase.

(B) Synthesis Phase -

1. Intermediate code Generator.
2. code optimization.
3. code Generator.

Important.



Token :-

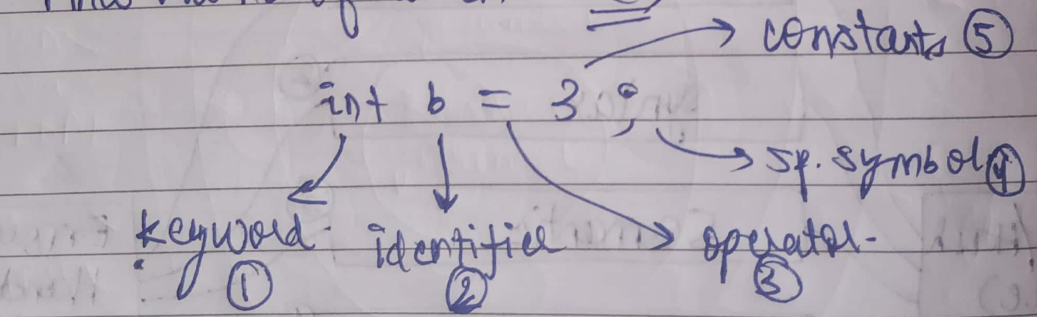
Sequence of character which represent smallest unit of information in the code.

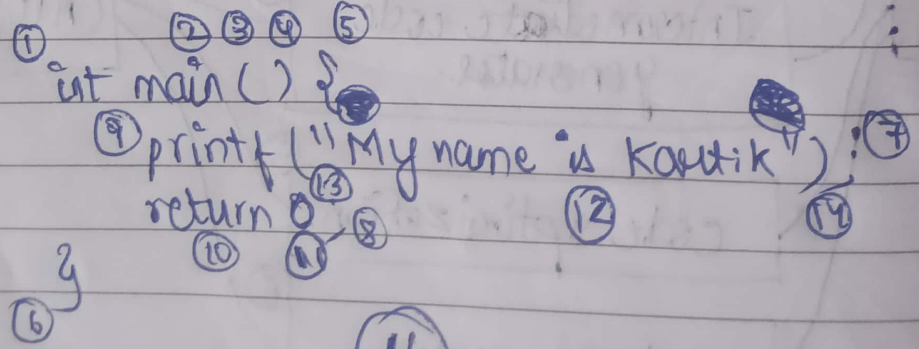
Identifier (variable/function names)
Keyword (32 in C)
constant value (5, 6, 10, 12)
Special symbols (; ,)
operator (Bitwise, Arithmetic, Logical, comparison, assignment, etc.)

Non-token :-

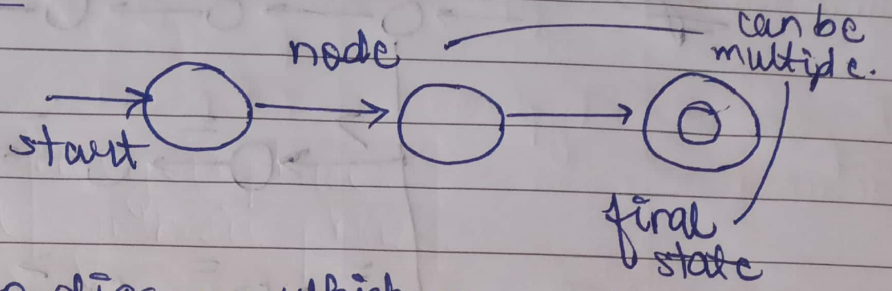
Newline character (\n)
comments
Blank space.

int b = 3;
Find out no. of tokens. (5).





Transition Diagram

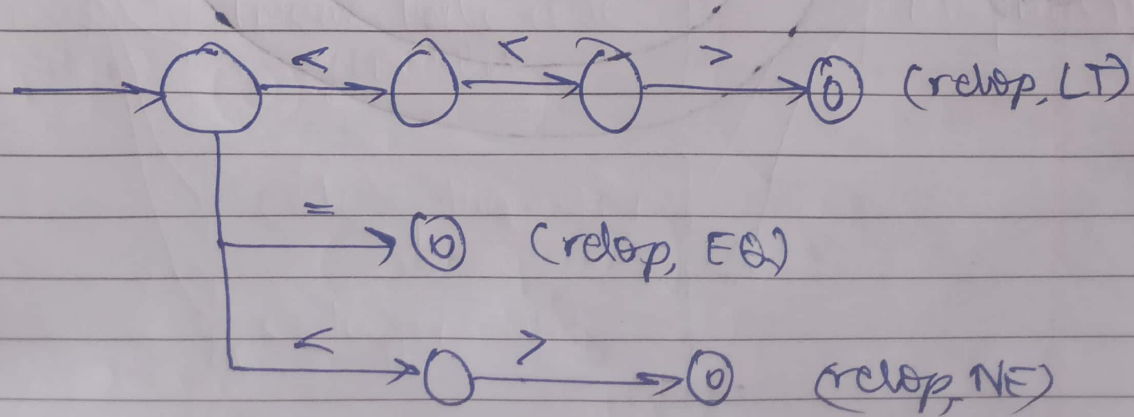


(1) States → transition diagram which represent collection of nodes/ circle

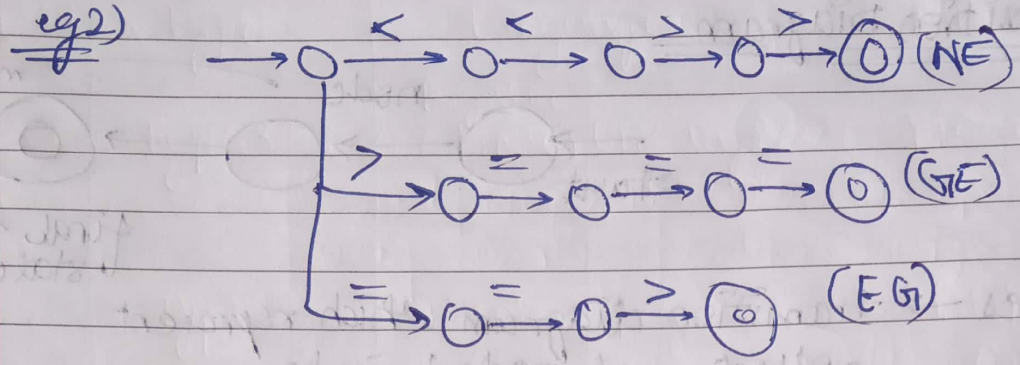
(2) Lexeme → sequence of characters in the source program which represent pattern matching for a token.

Lexeme	Token Name	Attribute.
if	if	—
then	then	—
else	else	—
any identifier	Id	—
any number	Number/ Num	—
<	Rel-op	LT
>	"	GT
=	"	EQ
<=	"	LE
>=	"	GE
<>	"	NE

eg 1)



eg2)

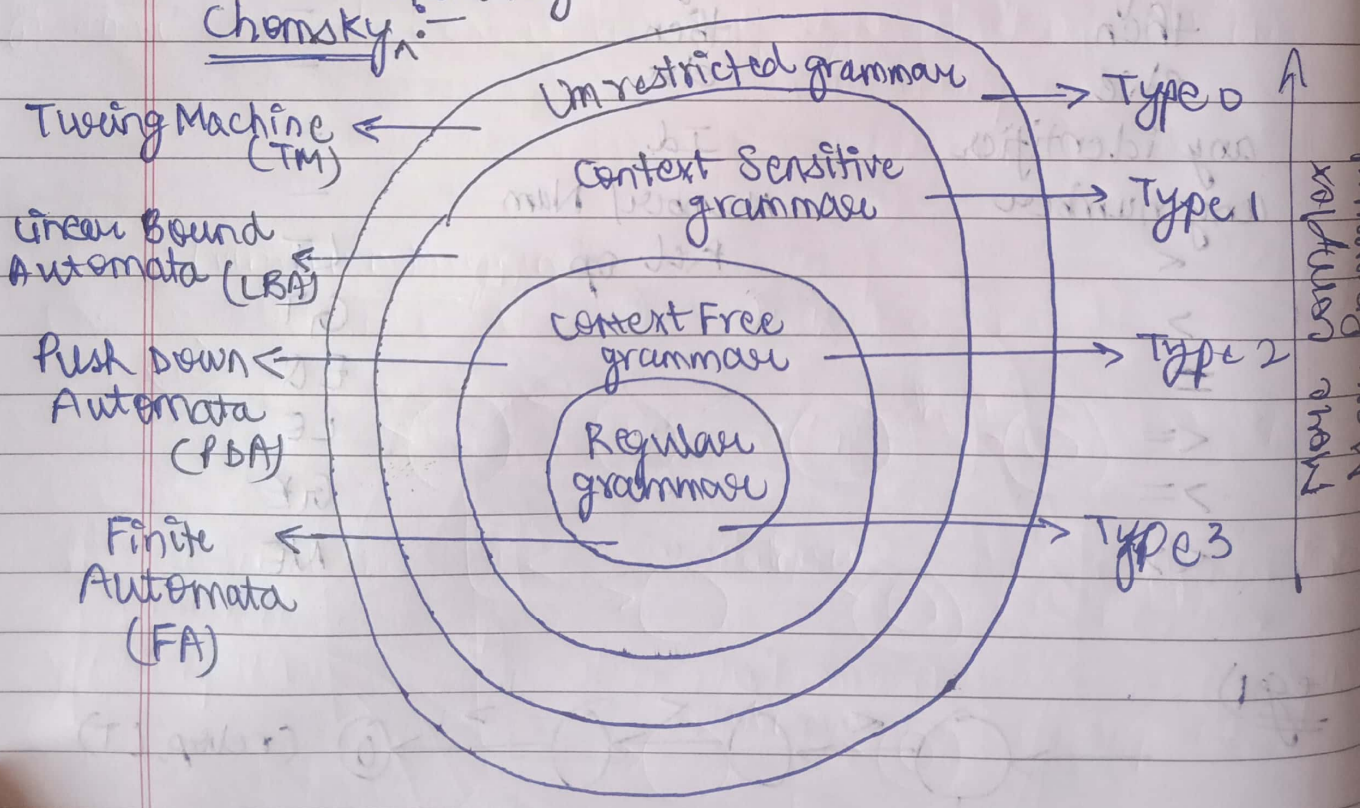


eg3) < < = LE

< < > NE

= = > EG.

Chomsky Hierarchy:



① Type 0 : Infinite Length [Turing Machine] (T.M.)

Example.

$A \rightarrow AAAA$

$AA \rightarrow A$

$AAA \rightarrow A$

$AA \rightarrow AA$.

② Type 1 : Infinite length [LBA]

Example.

$|LHS| <= |RHS|$ (restriction)

$AA \rightarrow AAA$

$A \rightarrow AA$

$AAA \rightarrow AAAAaa$.

③ Type 2 : Finite length [PDA]

Example.

$|LHS| = 1$.

$A \rightarrow AAA$

$A \rightarrow aA$

$A \rightarrow AAAAaa$. $A \rightarrow a$

④ Type 3 : Finite length [FA]

Example:

$A \rightarrow a$

$A \rightarrow aA$

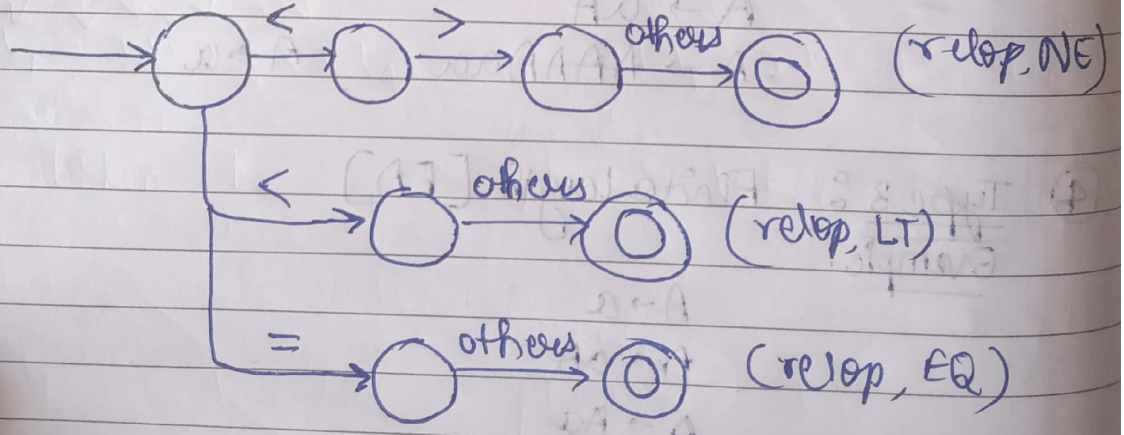
$A \rightarrow Aa$

$A \rightarrow aa$.

Grammar	Type	Language	Automata
Regular Grammar	Type 3	Regular (Finite) Language	Finite Automata
Context Free grammar	Type 2	Context Free Language	Push down Automata
Context sensitive grammar	Type 1	context sensitive language.	Linear Bounded Automata
Unrestricted grammar	Type 0	Recursive Language.	Turing Machine

Transition Diagram :-

Ex :-

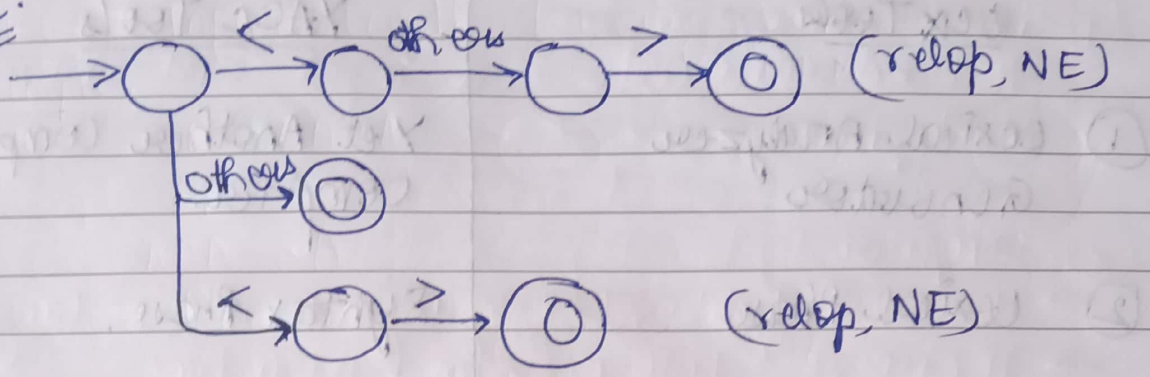


(relop, EQ)

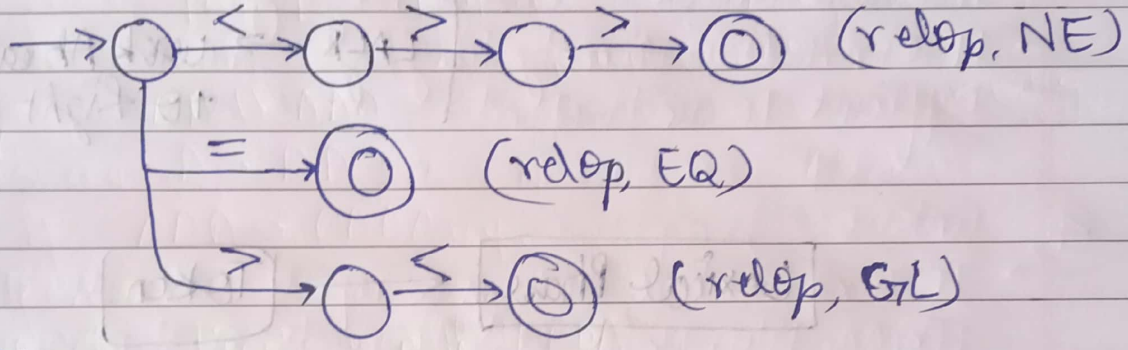
token name

attribute name.

Ex-2.

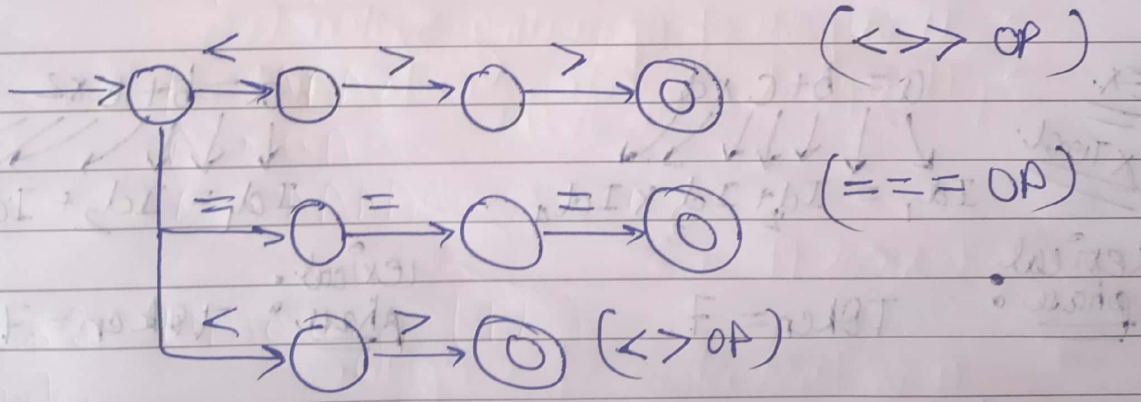


Ex-3.

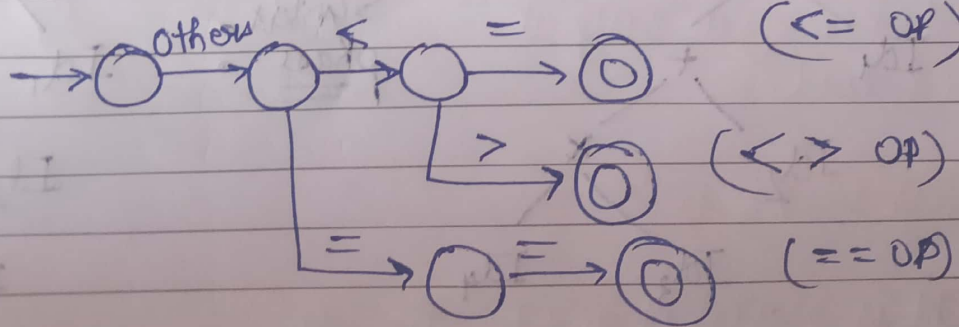


Easy Language Method.

Ex-1.



Ex-2.



Lex Tools	YACC Tools
① Lexical Analyzer Generator	Yet Another Compiler Compiler.
② Lexical Phase	Syntax Phase.
③ Support all operating system.	Unix operating system.
	LALR → Look Ahead Left to Right (Parsing)

Lexical Phase ← Token.

Syntax Phase ← Parser.

Ex. $a = b + c \times d$

Lex Tool.

$Id_1 = Id_2 + Id_3 \times Id_4$

Lexical phase: Token = 7.

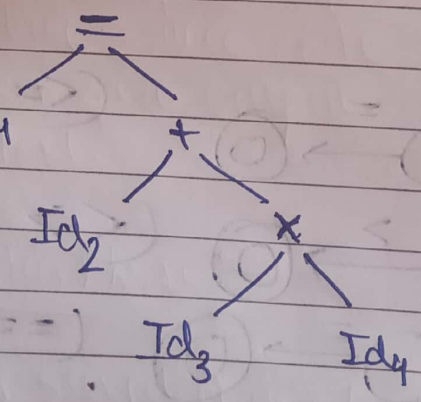
$a = b + c \times 2$

$Id_1 = Id_2 + Id_3 \times Num.$

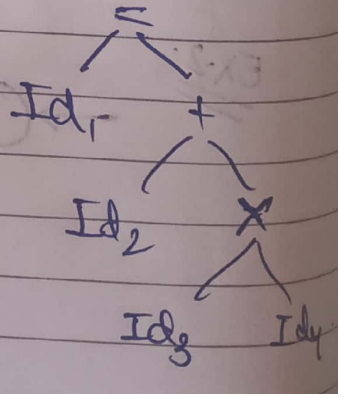
Lexical phase: Token = 7

YACC tool

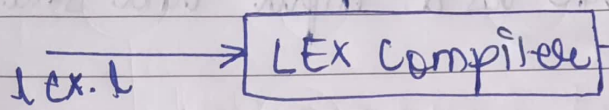
Syntax phase:

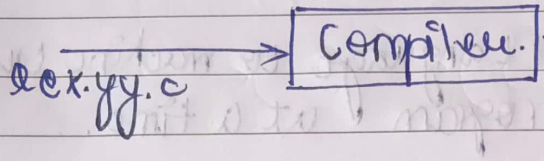


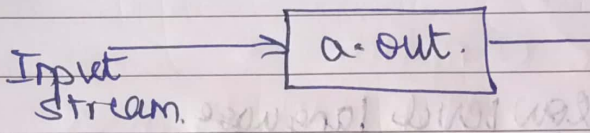
Syntax phase:



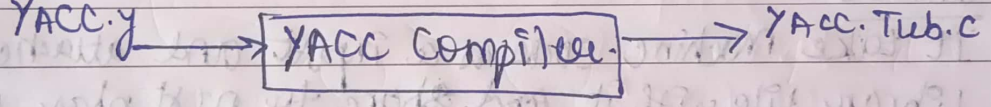
* **Lex Tool :-** Lexical Analyzer Generator.

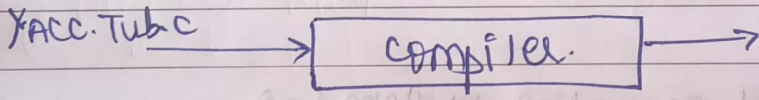


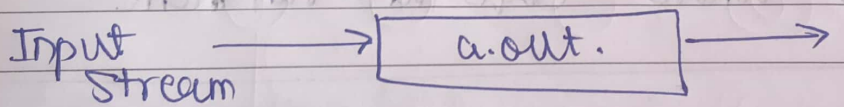




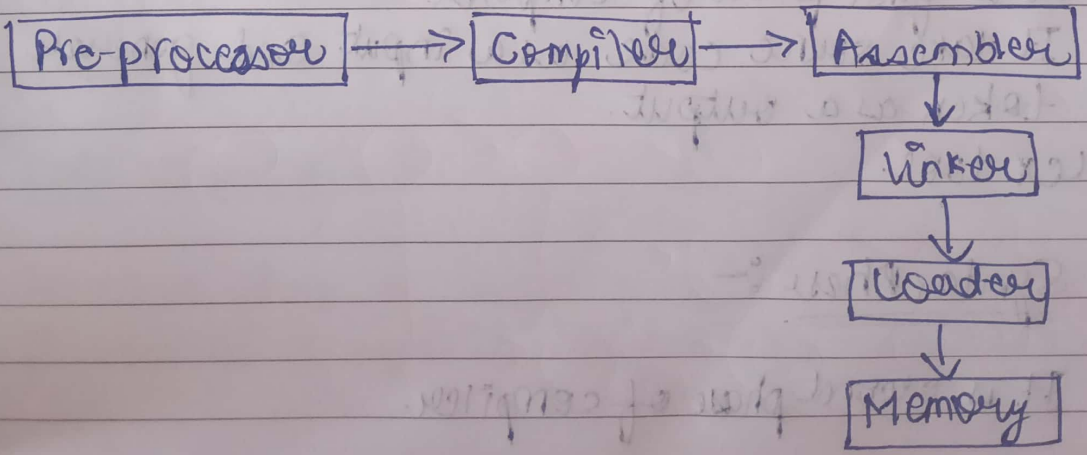
* **YACC Tool :-** Yet Another Compiler Compiler.







Relation b/w Compiler, Assembler, Linker and Loader:-



① Preprocessor
It take source code as a input and share to next phase.

② Compiler
→ It convert high level language to machine language.
→ It execute full program at a time.

③ Assembler
→ It is also called low level language.
→ It converts assembly language to machine language
→ Difficult to understand.

④ linker
→ It take machine code as a input and attached library file with it and share to next phase loader.

⑤ Loader
→ It is part of operating system (os)
→ It loaded whole code and run them.

---

1) Lexical Phase :-

It is first phase of compiler.
It take source code as a input and generate token as a output.
lex tool.

2) Syntax Phase :-

It is second phase of compiler.

It take input the tokens and parse tree as an output.
YACC tool.

3) Semantic phase :-

It is third phase of compiler.
Semantic error.
Type checking → Static Type checking (compile time)
eg. C, C++, Java.
→ Dynamic Type checking (runtime)
eg. Perl, Ruby, Python.

4) Intermediate code generator :-

It is fourth phase of compiler.
It takes machine code as a input and generate intermediate code as a output.

5) Code generator :-

It is sixth phase of compiler.
It take intermediate code as input and output code as a output.

5) Code optimization :-

It is best phase of compiler.
It reduce complexity → Space complexity
Time complexity.

Types of compiler :-

① Multipass compiler →

(i) It is also called 2 or 3 compiler.

(ii) It is step by step execution.

eg. COBOL, Fortran, Algol.

② Source to source compiler →

(i) It takes input as High level language.

(ii) It outputs high level language.

eg.

C# compiler

C++ compiler

Java compiler

COBOL compiler

Algol compiler.