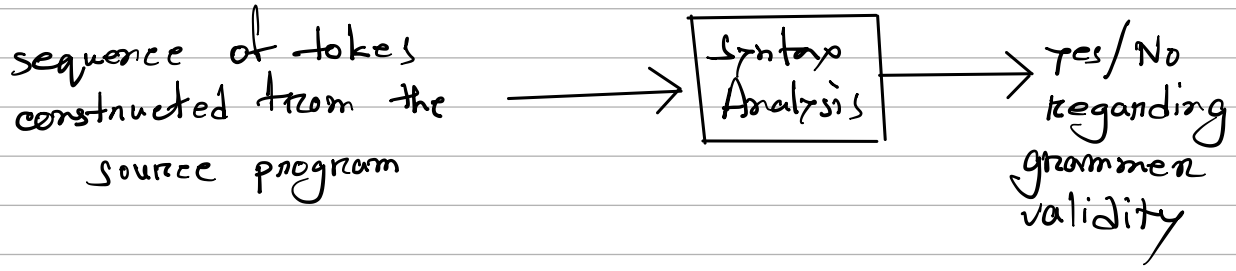
Lecture 3: Syntax Analysis

* During implementation syntax & semantic analysis are combined together.

syntax analysis ⇒ Grammer Checking



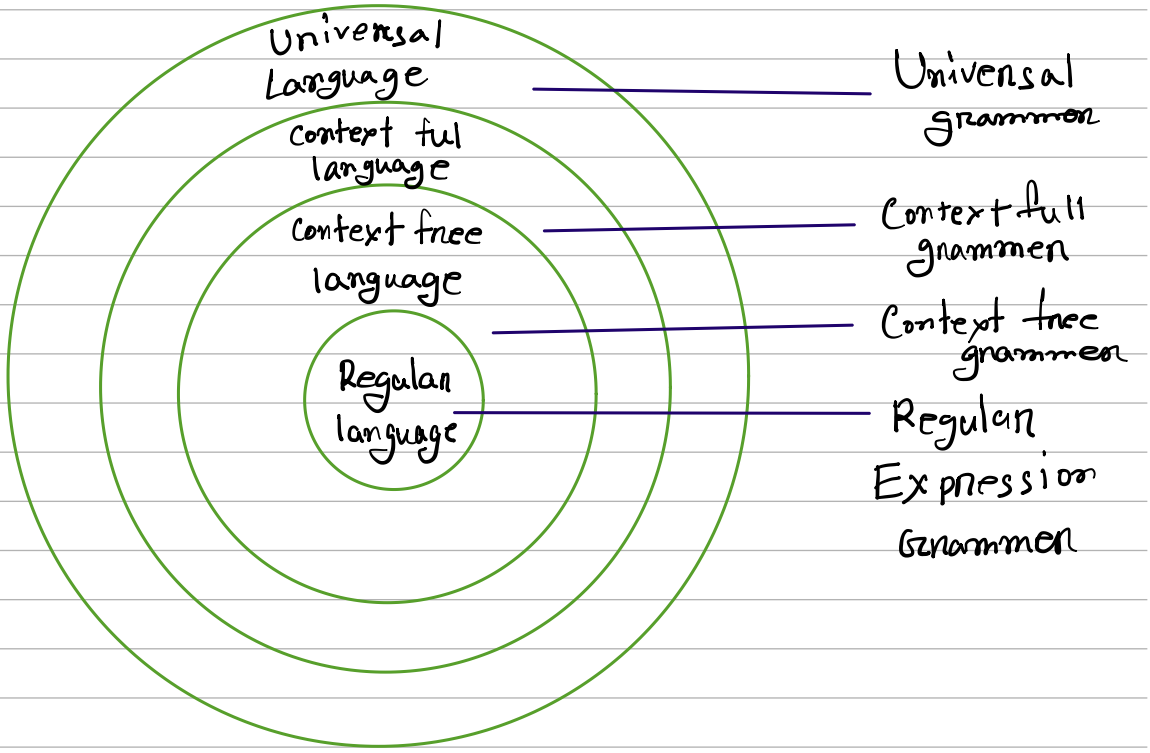
Noam Chomskey
↳ invented the hierarchy of language.

* Language is a set of strings.
Grammer of the language permits

Ate rice he → not permitted ✗
He ate rice → permitted ✓

∴ Grammer is the tool that defines what is acceptable.

Defined by



* Human / Natural language is contextful.

Selim stucked Kanim, he was crying.

↓ who?

↳ requires context → tough to get context

* All programming languages are context free. (CFG

reason to use context free grammer

- The purpose of compiler is to produce executable that exactly matches the programmer's idea. There cannot be any vagueness. To translate exactly we don't need context.
- It is hard to construct contextful compiler.

Components of CFG:

(i) Σ ← alphabet of terminal symbols.

(ii) A set of non-terminals

(iii) Production rules that convert a nonterminal into a string of terminals and/or non-terminals.

(iv) A starting non-terminal.

Production rules for arithmetic expressions:

$E \to E + E$

$E \to E - E$

$E \to E * E$

$E \to E / E$

$\vdots$

$E \to$ number

$E \to$ identifier → variable names

$E \to (E)$

→ on the left side we have a single non-terminal than it is context-free. otherwise context-ful.

$\Sigma = \{ +, -, *, /, \text{number}, \text{identifier}, (,) \}$

↳ set of token types from lexical analysis

id + number * id

$E \to E + E$

$\to E + E * E$

$\to$ id $+ E * E$

$\to$ id + number $* E$

$\to$ id + number * id

} derivation

leftmost derivation:

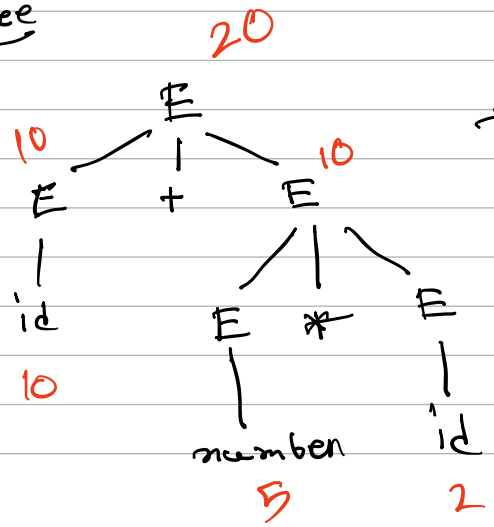
E
↓
E + E
↓
id + E
↓
id + E * E
↓
id + number * E
↓
id + number * id

always changing the leftmost non-terminal first as long as we can

rightmost:

E
↓
E + E
↓
E + E * E
↓
E + E * id
↓
E + number * id
↓
id + number * id

parse tree / syntax tree



→ is this same for all the derivations?

if x + 5 x 4

x = 10
y = 2

another leftmost derivation

E

↓

E * E

↓

E + E * E

↓

id + E * E

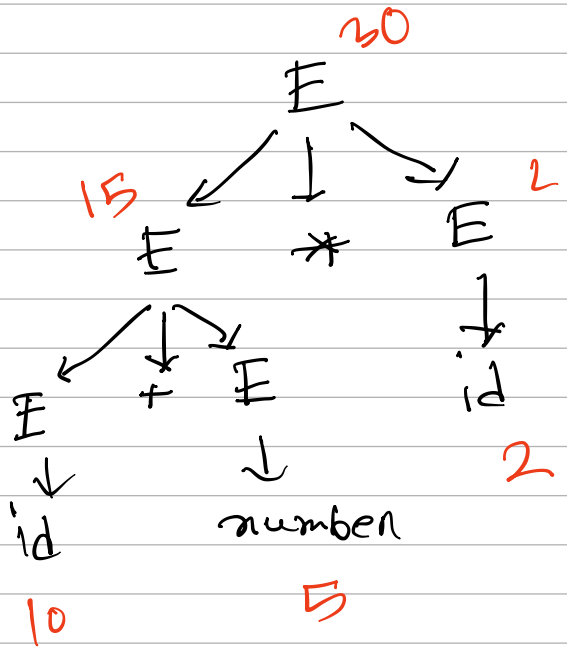
↓

id + number * E

↓

id + number * id

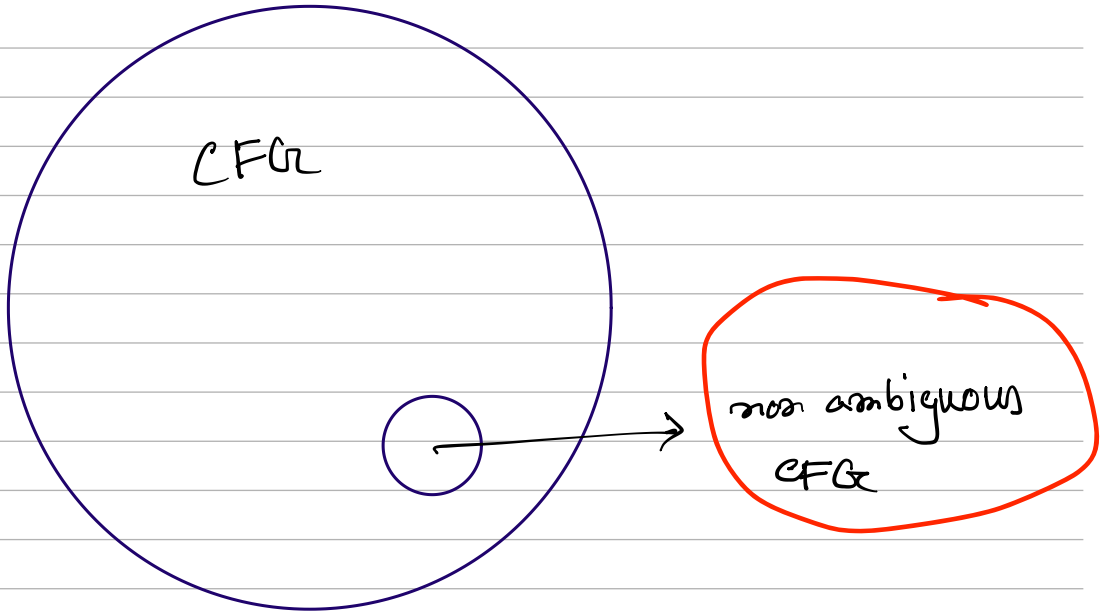
Parse tree for this:



For the same values, we get two different outputs for two different parse trees. This type of grammer is called ambiguous.

We cannot use ambiguous grammer for our programming language.

∴ All CFGs cannot be used for programming languages.



Why we have to use semicolon in C ?
" " " " " indents in Python ?

all these are to maintain the non ambiguity of grammer.

context free grammer → Parser generator.