NLP

Grammar:

It is generally represented as (V, T, P, S), where

V = Vertices, T = Terminals, P = Production, S = Start symbol.

Example:

$S \rightarrow aSB$
$S \rightarrow aB$
$B \rightarrow b$

} Grammar, then; $V = \{S, B\}$ → left hand side symbols of production

$T = \{a, b\}$

$P = \{S \rightarrow aSB,\ S \rightarrow aB,\ B \rightarrow b\}$

$S = \{S\}$ → start symbol

Derivation:-

Getting the string from the Grammar is "Derivation".

Ex:- deriving "aabb"

$S \Rightarrow aSB$ → In "aSB", if you replace 'S' first as part of derivation, then it is "left most derivation". else, if you replace 'B' first, then it is "right most derivation"

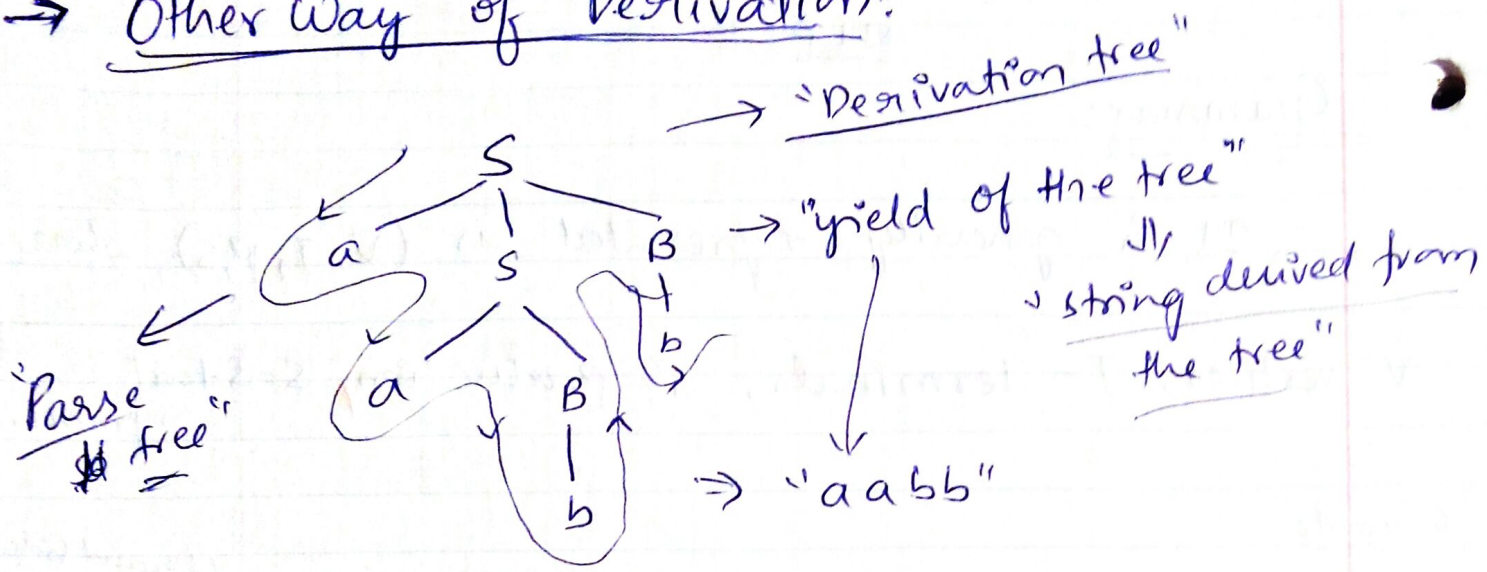
$\Rightarrow aaBB$ (left most derivation)

$\Rightarrow aabB$

$\Rightarrow aabb$

"Intermediate forms are called sentential forms (or) sequential forms"

→ Other Way of Derivation:



⊙ How to Construct Grammars?

⊙ Given a Grammar, how to identify that Grammar is representing?

Problem:-

Given, language $L = \{aa, ab, ba, bb\}$ → of strings, length, 2. Construct Grammar, which accepts set of all strings of length, '2'.

Sol:-

$$S \rightarrow aa|ab|ba|bb$$

- Steps:
  - First construct the building blocks, then extend it

Ex: $(a+b)(a+b)$ is bldg blk of $(aa|ab|ba|bb)$

$A \quad A$

$\Rightarrow \quad S \rightarrow AA$

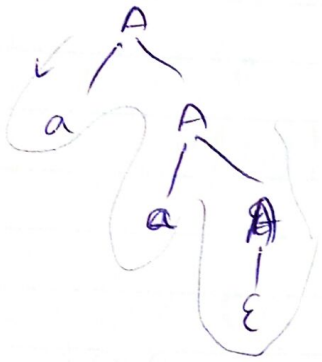
$A \rightarrow a|b$

Ex 2:

$a^n \mid n \geq 0$

a, aa, aaa, aaaa, ...

$A \rightarrow aA \mid \varepsilon$



$\Rightarrow aa$

Ex 3:

$(a+b)^*$

$S \rightarrow aS \mid bS \mid \varepsilon$

"abab"

Ex 4: Set of all strings of length, atleast, 2

① $(a+b)(a+b)(a+b)^*$

$S \rightarrow AAB$

$A \rightarrow a \mid b$

$B \rightarrow aB \mid bB \mid \varepsilon$

② "atmost 2"

$(a+b+\varepsilon)(a+b+\varepsilon)$

A A

$S \rightarrow AA$

$A \rightarrow a \mid b \mid \varepsilon$

③ "starting with 'a' and ending with 'b'"

$S \rightarrow aAb$

$A \rightarrow aA \mid bA \mid \varepsilon$

④ starting and ending with diff symbols

$a(a+b)^*b + b(a+b)^*a$

$S \rightarrow aAb \mid bAa$

$A \rightarrow aA \mid bA \mid \varepsilon$

⑤ starting & ending with same symbol.

$S \rightarrow aAa \mid bAb \mid a \mid b \mid \varepsilon$

$A \rightarrow aA \mid bA \mid \varepsilon$

⑥ Construct grammar such that,

$a^m b^n / n \geq 1$

$\Rightarrow \quad S \rightarrow aSb / ab$

⑦ Generate set of palindromes.

→ $ww^R \cup waw^R \cup wbw^R$

Sol:- $S \rightarrow aSa / bSb / a / b / \varepsilon$

CLASSIFICATION OF GRAMMARS

- According to chomsky,
  - Type 3
  - Type 2
  - Type 1
  - Type 0.

Type 3:- / Regular Grammar / "machine Finite Automata"

If all the productions are of the form,

$A \rightarrow \alpha B / \beta$

$A, B \in V$

$\alpha, \beta \in T^*$

→ right linear grammar as 'B' is towards right side

$A \to B\alpha / \beta$

$A, B \in V$

$\alpha, \beta \in T^*$

} left-linear grammar (LLG)

Ex:-

$A \to aB / a$
$B \to aB / bB / a / b$ } RLG

$A \to Ba / a$
$B \to Ba / Bb / a / b$ } LLG

Ex:-

$A \to Ba / a$ → LLG
$B \to aB / a$ → RLG

X Type 3. It cannot be combination of LLG, RLG.

Type 2 :- / Context-Free-Grammar.

If all the productions are of the form,

$A \to \alpha$, where $A \in V$
$\alpha \in (V \cup T)^*$

Ex:-

$A \to aAb / ab$

$A \Rightarrow aAb$
$\Rightarrow aabb$

} we call it context-free, because we can replace 'A' without looking at the context.

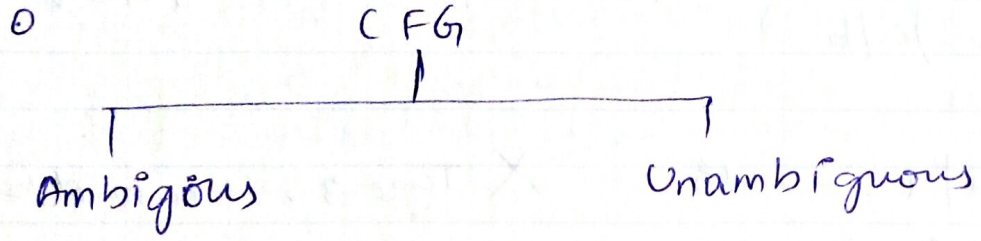
→ But in context-sensitive Grammar (Type-1), we will have $aA \to ab / aAb$

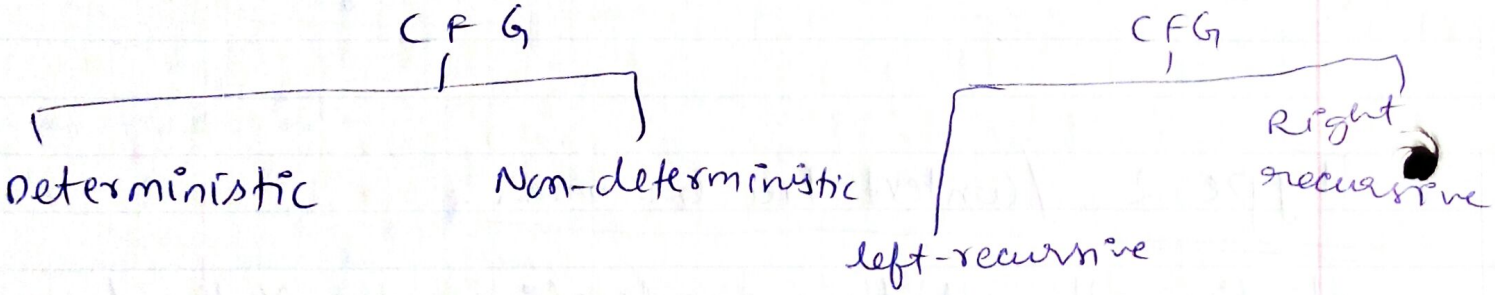
we have to look, whether before 'A', 'a' is present (or) not.

∴ Language generated is "Context - free language"

∴ machine is "Push Down Automata" (PDA)

CFG - Dichotomy :-





⊙ If we want to find whether a string, w, exists in the language generated by CFG. i.e, $w \in L(G)$ (or) not.

Sol:- we have to increase the length of the string, w to by using the grammar to $|w|$.

but, if we have $A \to \varepsilon$ → epsilon productions

(or)

$A \to B$ → unit productions,

then again the length of the string reduces

and we may not be able to check it correctly,

So, we have to make sure that our grammar (or) production doesn't contain "Epsilon" (or) "Unit productions". then, we can increase the length by '1' at each time, and check whether string is present in grammar (or) not

⊙ One such algorithm to find the membership of a string in grammar is 'CYK algorithm'

Elimination of ε- Productions:-

- If the language contains 'ε', we will not able to eliminate (or) remove 'ε' productions.
- To eliminate ε-productions, you should remove the "nullable" variables.
- A variable is nullable, if it generates 'ε', (or) immediately (or) after some sentential forms

Ex:- $A \rightarrow \varepsilon$

$S \rightarrow aAb \Rightarrow S \rightarrow a\varepsilon b$ → so, 'S' is not nullable. as it doesn't generate 'null'.

Ex:-

To eliminate 'ε' productions in below ex

$S \rightarrow aSb \mid aAb$

$A \rightarrow \varepsilon$

Sol: ① $A \rightarrow \varepsilon$ is a nullable variable

② Next, write productions with & without nullable variable

$S \rightarrow aSb \mid aAb \mid ab$

$A \rightarrow \varepsilon$

③ Finally eliminate, nullable variable productions, and productions which contain nullable variable.

Ex:-

$S \rightarrow AB \; ; \; A \rightarrow aAA/\varepsilon \; ; \; B \rightarrow bBB/\varepsilon$

1. Find nullable variables.

$A \rightarrow \varepsilon \; ; \; B \rightarrow \varepsilon \Rightarrow \{A, B, S\}$

2. ~~Write Productions with ε~~

- when 'S' generates 'ε', then 'ε' belongs to the language, so, we'll not be able to remove all "null" productions from the Grammar.

3. Write with and without Nullable Variables

$S \to AB \mid B \mid A \mid \varepsilon$

$A \to aAA \mid aA \mid a$

$B \to bBB \mid bB \mid b$

} removed all 'ε' productions except $S \to \varepsilon$, as 'ε' contains in the language

Ex.

$S \to AbaC$

$A \to BC$

$B \to b \mid \varepsilon$

$C \to D \mid \varepsilon$

$D \to d$

⇒ Nullable Variables = {B, C, A}

⇒ $S \to Aba \mid baC \mid Ab \mid ba$

$A \to BC \mid B \mid C$

$B \to b$

$C \to D$

$D \to d$

BUT, removing 'ε' productions, adding UNIT - productions

- but they are also useless. For example,

$A \to B$; $B \to C$; $C \to D$; $D \to d$

- To generate 'd', $A \Rightarrow B \Rightarrow C \Rightarrow D \Rightarrow d$.

} neither the length nor terminals increasing, therefore we don't use unit prod in CFG

Elimination of Unit Productions:-

$S \rightarrow Aa \mid B$

$B \rightarrow A \mid bb$

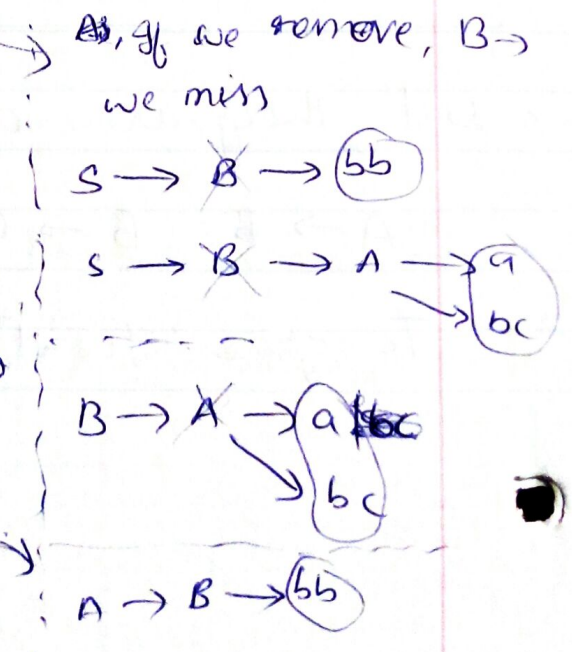
$A \rightarrow a \mid bc \mid B$

- we should make sure that ~~even~~ while eliminating unit productions, it should not affect the grammar of the language.
- So, you write all the productions after removing unit productions. and then add productions for each variable, which we'll be missing by removing unit productions.

$S \rightarrow Aa \mid bb \mid a \mid bc$

$B \rightarrow bb \mid a \mid bc$

$A \rightarrow a \mid bc \mid bb$



Ex 2:

$S \to AB$

$A \to a$

$B \to \in | b$

$C \to D$

$D \to E$

$E \to a$

Remove unit prod

$S \to AB$

$A \to a$

$B \to b | a$

$E \to a$

$D \to a$

$C \to a$

$S \to A$, $S \to B$, $S \to C$, $S \to D$, $S \to E$, $S \to a$

$C \to D \to E \to a$

$D \to E \to a$

From 'S', you can reach 'A' & 'B', but not 'C', 'D', 'E'. So, these are useless symbols. You can delete those useless symbols.

Useful Symbol:

- A symbol is said to be useful, if it is able to derive a part of the string (or) a terminal.
- It should be reachable from "start" symbol.

Steps:

Example:

$S \to AB | a$

$A \to BC | b$: $C \to aC | B$

$B \to aB | C$

1) Add all terminals to the set of Useful Symbols, U

2.

$U = \{a, b\}$

2. Add "symbols", which are generating ~~useful~~ symbols some terminals

→ $U = \{a, b, S, A\}$

3. ~~Add symbols which~~

Add production symbols, whose right hand side of production is made up of only useful symbols

$B \rightarrow aB | C$ (X) ⇒ 'a' is there but it has 'B' with it, so it cannot be included

- 'B' & 'C' are not useful as it'll not generate string.

Ex: $B \rightarrow aB | C$ ⇒ $B \Rightarrow aB$
$C \rightarrow aC | B$ $\Rightarrow aaB$
$\Rightarrow aaaB$
$\Rightarrow aaaaB$ → we'll not be able to get rid of 'B'
$\Rightarrow aaaa$ at any time
↳ (or even 'C')

- Now delete 'B' & 'C' productions, and in the productions, where 'B' & 'C' are present.

$S \rightarrow AB | a$

⇒ So, $S \rightarrow Aa$ | a

$A \rightarrow b$

- Now, see, if every symbol is reachable from 'S' (or) not.

Final Answer

$S \rightarrow a$

Chomsky Normal Form:

A context-free grammar, G is said to be in CNF, if all its production rules are of the form

$A \rightarrow BC$ (or)

$A \rightarrow a$ (or)

$A \rightarrow \varepsilon$

where A, B, C → Non-terminal symbols

a → terminal symbol

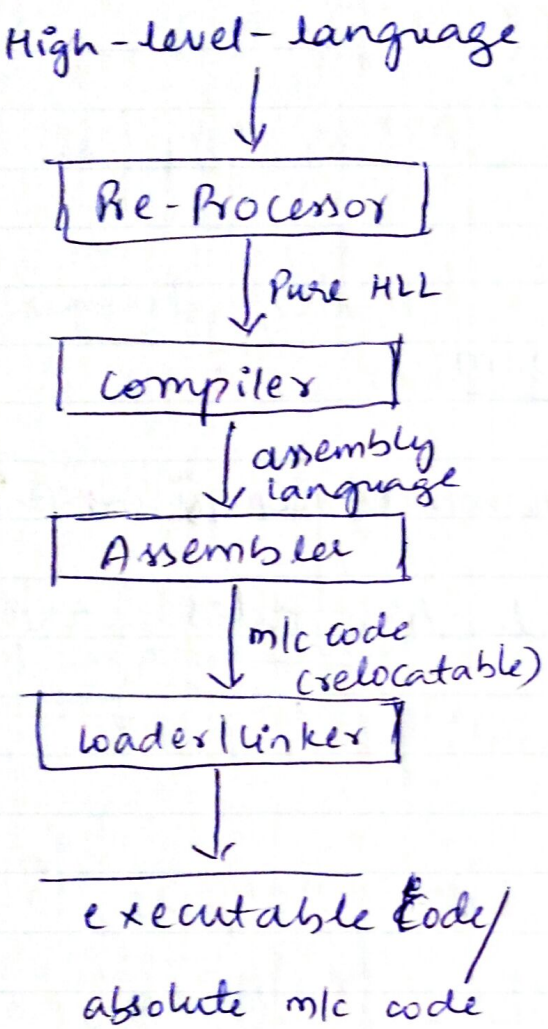
S → Start Symbol

ε → empty string

↙

It can appear only if 'ε' is in L(G).

Compiler Design :-



HLL → Pre-Processor :- generates **Pure HLL :-**

- Example in 'c' pgm, it replaces all

File inclusion ← # include < > → includes original file, for ex ⇒ conio.h

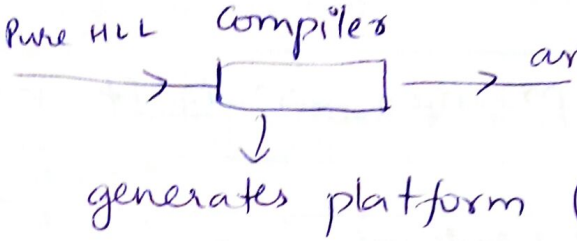
define ⇒ replaces the defined constant in all locations (or) references in code.

↓

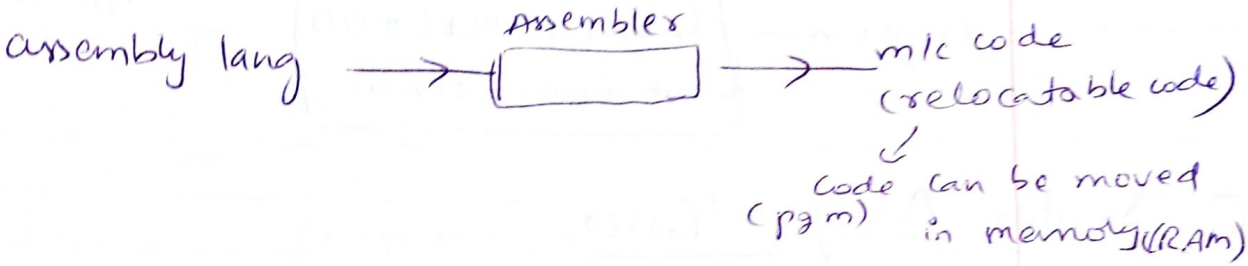
macro expansion.

HLL ——→ Pure HLL

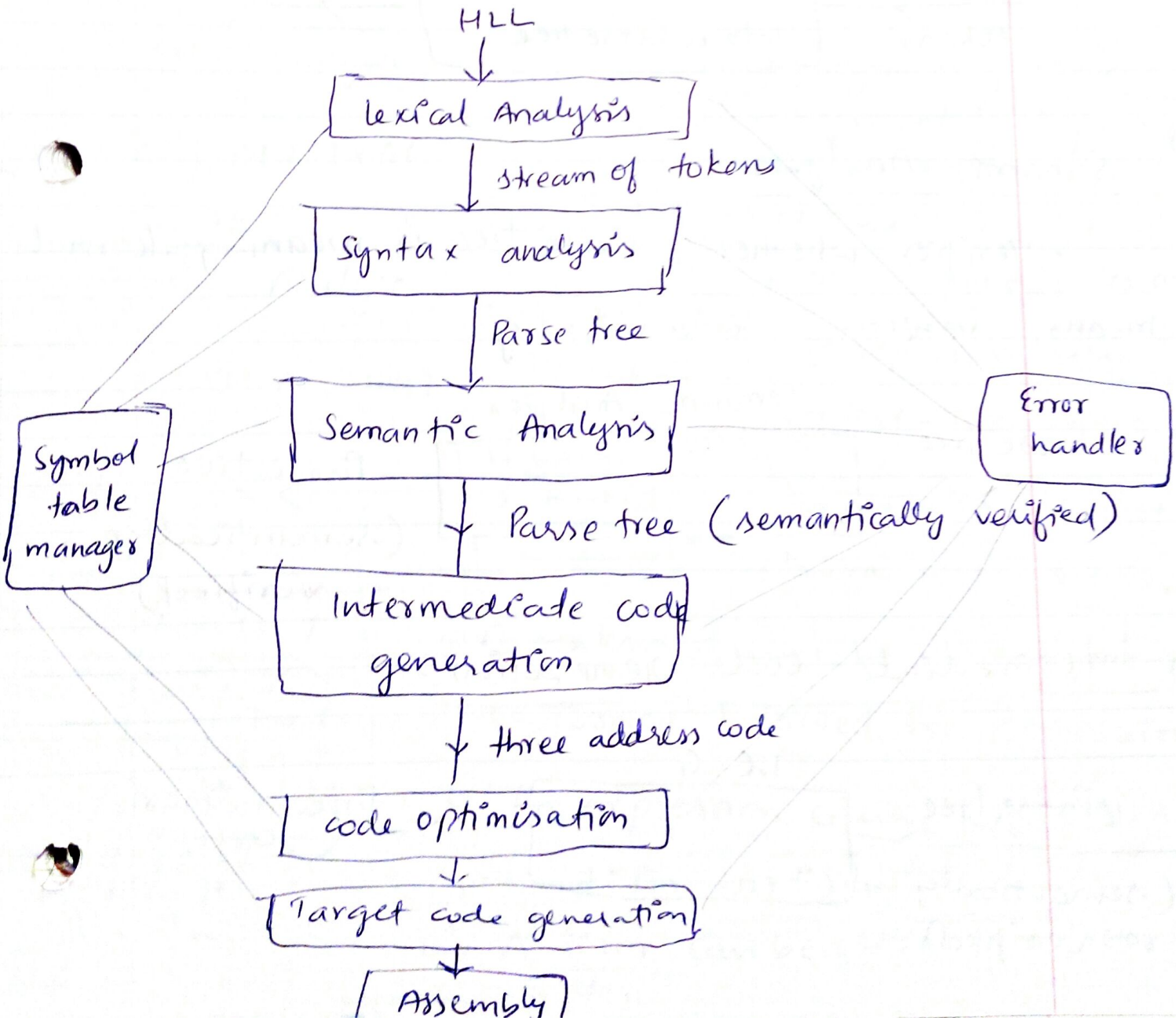
(2)



(3)

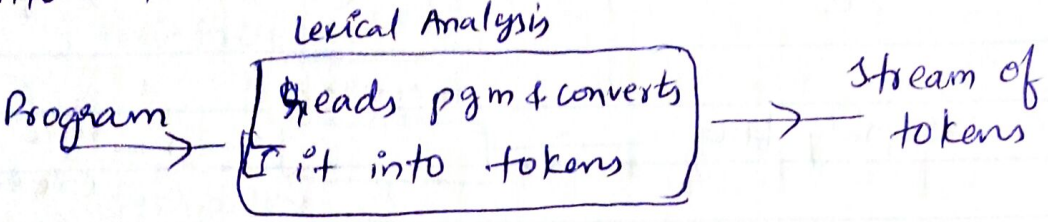


Compiler Phases:

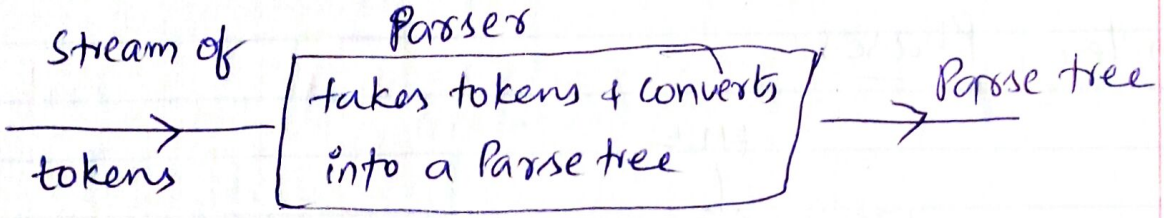


① Lexical Analysis:

- It takes the program, reads it and converts it into tokens.

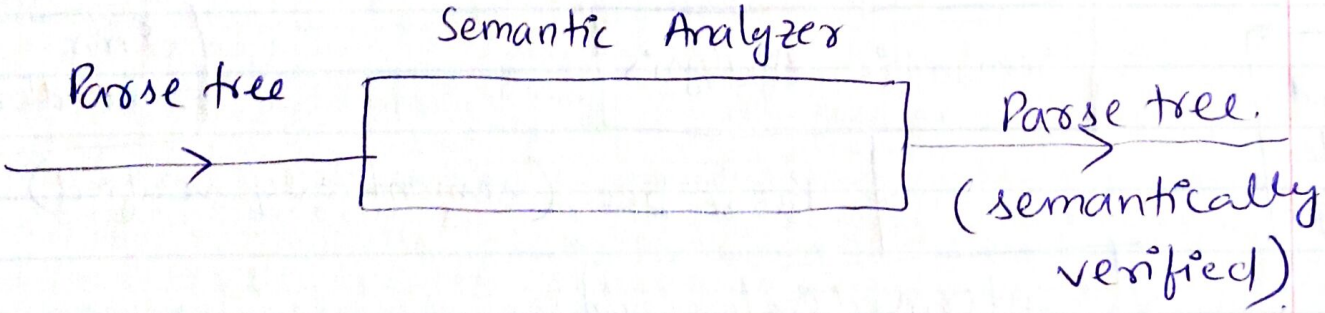


② Syntax Analyzer / Parser:

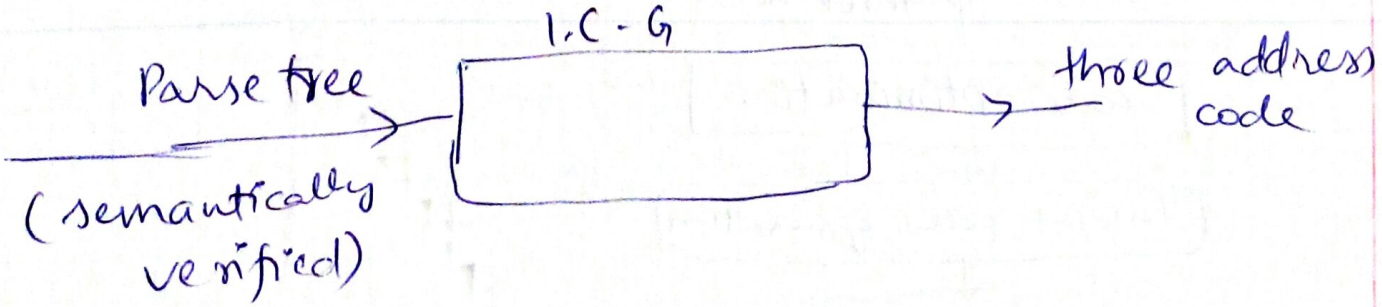


③ Semantic Analyzer

- Verifies whether, Parse tree is meaningful (or) not. means, verifies, semantically.



④ Intermediate Code - Generation:-



⑤ Code Optimisation:

- Code optimization like reducing # of address codes (or) lines is done.

Example:-

x = a + b * c /* comment */

↓ lexeme

↓ converts stream of lexemes → stream of tokens

Lexical analyser

↓

id = id + id * id

↓

Syntax Analyser

1. A programming language rules can be represented using number of production rules
2. ~~Syntax Analyser checks~~

① S → id = E;
② E → E + T / T
③ T → T * F / F
④ F → id

2. These set of rules are grammar, mostly context-free-grammar.

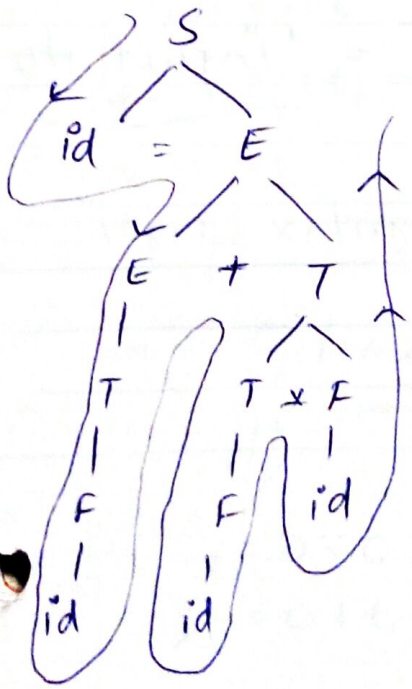
3. For example, the first statement, represents,

① A statement can be, identifier = Expression followed by semi-colon.

② An expression can be expression plus term (or) term

③ A term can be term into factor (or) factor

↓ (Parse tree)



④. A factor can be simply a identifier

- A syntax Analyzer, checks, whether we have given input, according to the format we needed (i.e, CFG, (set of prod. rules of a programming lang)

- while constructing parse tree, it ① starts with 'S', start symbol, ② statement is Identifier := Expression

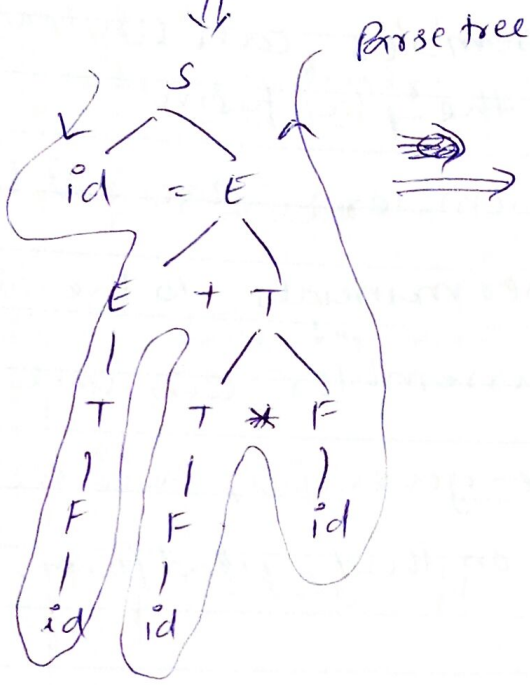
③ An expression = expression + term

④ A term is term * factor

⑤ A factor can be identifier.

- We can say the input is in required format only if, "yield of parse tree" = "input to Syntax Analyzer". else → "Syntax Error"

Syntax Analyser



Semantic Analyzer

(Parse tree semantically verified)

- Type checkings
- Variable declarations etc

Intermediate C.G → generates address codes

- Popular one = 3-address code
- 3-add-codes per line atmax

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

→ Till-here :- ~~Compi~~ Platform - independent.

Code-Optimizer

reduce - # of lines

$t_1 = b * c;$

$x = a + t_1;$

⇓

Target - Code - Generator

⇓

```
mul R1 R2
add R0 R0
mov R2 x
```

a → R0
b → R1
c → R2

- Write - code which assembler can understand on the platform
- Assemblers, are like user manuals to the platform, if assembler can understand code-generated, then we can run on that platform.

→

- Syntax - Analyzer, heart - of - the - compiler
- mostly, front-end phases → (starting part of the compiler) are implemented by → "Syntax Analyzer / Parser"

lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code G

} Syntax Analyzer / Parser "Front End"

- Later stages ⇒ Backend { 1. Code optimiser
  2. Target-code generator

Lexical Analyser:

- It reads program by "character by character" and "line by line" and generates tokens.
- It removes "comments", "whitespaces" & if any errors, it shows up.

Ex:

```
int max(x,y)
    int x,y;
{ /* find max of x and y */
  return (x>y?x:y);
}
```

→ not counted as tokens.

→ Totally, there are '25' tokens.

Ex2:

```
printf("y.d Hai",++x);
```

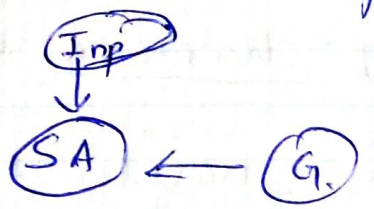
1 2 3 4 5 6 7 8 →

→ entire string literal is one-token

→ so, "8" tokens in total.

Syntax Analyzer:

- For syntax Analyzer, we need Grammar to verify whether input is correct or not



- $G = \{V, T, P, S\}$

$P = \{E \to E+E \mid E*E \mid id\}$, $V = \{E\}$, $T = \{+, *, id\}$

Ex:- id + id * id

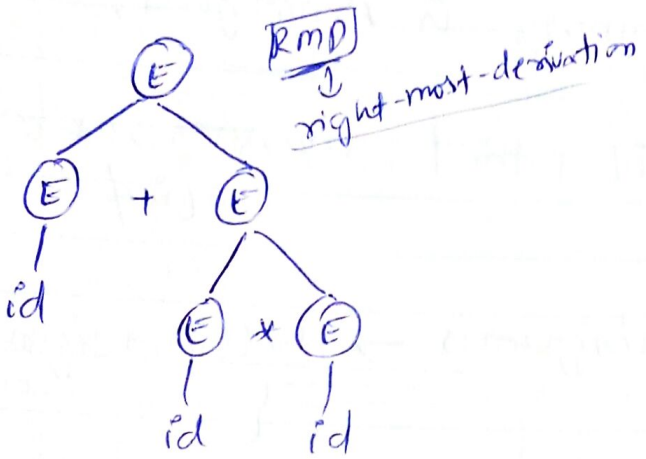
$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

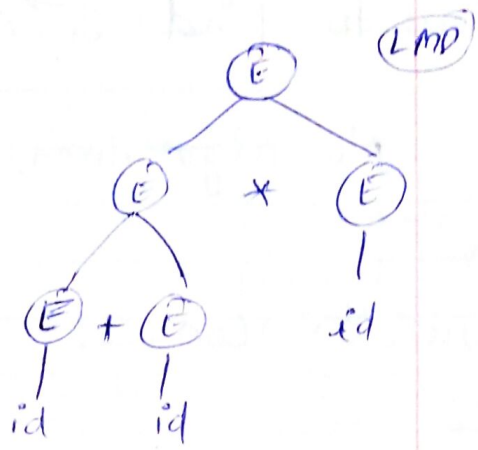
→ Left-most derivation

$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E * id$
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

→ Right-most derivation

Parse-tree



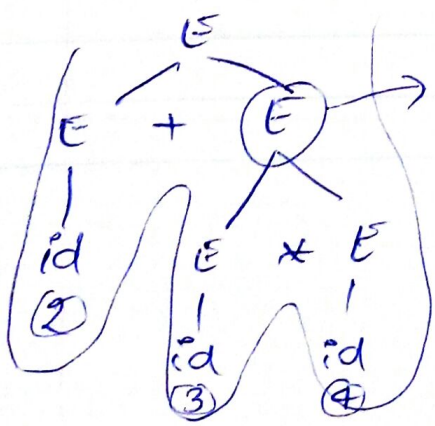


* For a given Grammar, for a given String, if we have more than one "Parse tree", then (or) more than one LMD (or) RMD

Our grammar is "Ambigous"

- If we have '2' parse trees, then parser will get confused, like which one to "generate"
- Even though, o/p looks similar, but there will be difference.

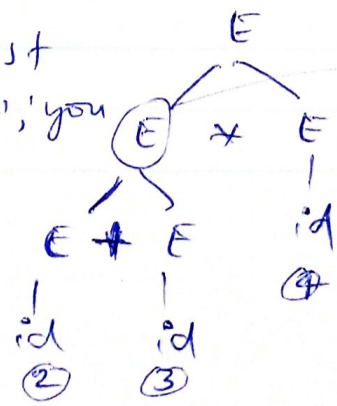
Example = 2+3*4



It'll evaluate first as to evaluate '+', you need it.

⇒ 2+3*4

= 14



but, in this case, this will get evaluated first

⇒ 2*3*4

= 20

- wrong answer

Questions

① How to find whether a Grammar is Ambiguous?

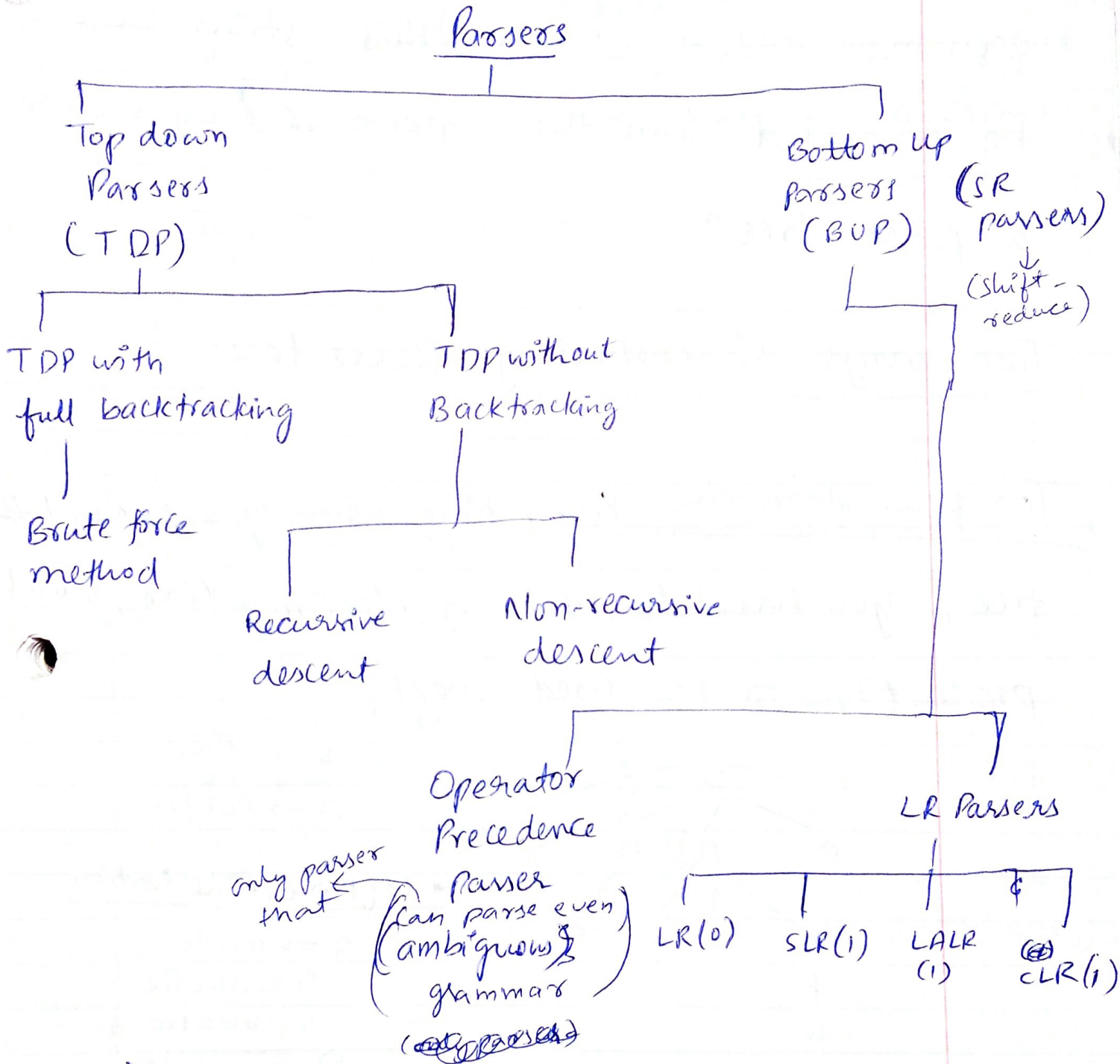
No algorithms, hit & trial manner, we have to find

② How to convert Ambiguous → Unambiguous grammar?

No algorithms, hit & try

Parsers:

Parsers :-

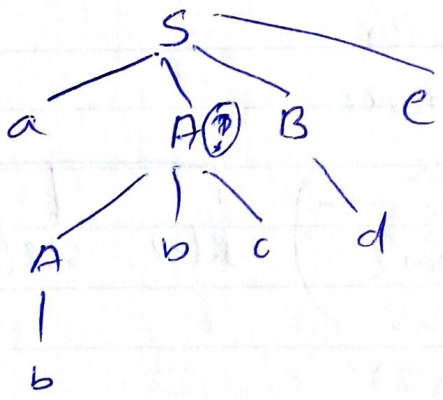

Parsers
Top down Parsers (TDP)
Bottom up parsers (BUP)
(SR parsers)
(shift-reduce)
TDP with full backtracking
TDP without Backtracking
Brute force method
Recursive descent
Non-recursive descent
Operator Precedence Parser
only parser that
(can parse even ambiguous grammar)
LR Parsers
LR(0)
SLR(1)
LALR (1)
CLR(1)

Role/Duty of a Parser:- Given a string, given a grammar, check whether this string can be generated from the Grammar? by using a parse tree?

Two ways of constructing Parse tree

Top-Down Approach: Every time when you expand the tree, you have to make a decision like, which production to be used next?

Ex:



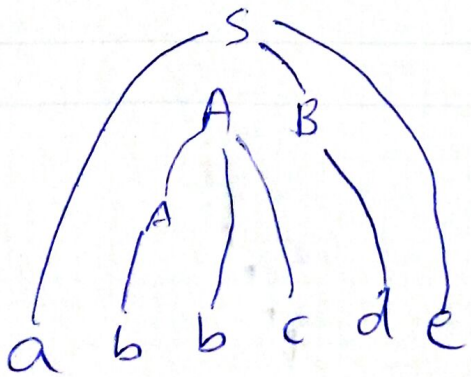
$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

- **Left most derivation**

$S \Rightarrow aABe$
$\Rightarrow aAbcBe$
$\Rightarrow abbcBe$
$\Rightarrow abbcde$

Bottom-Up:- Take the terminals & go up



tasks:

- when to reduce?
- Reverse of "Right most derivation

$S \Rightarrow aABe$
$\Rightarrow aAde$
$\Rightarrow aAbcde$
$\Rightarrow abbcde$

LL(1) Parser :-

- left to right
- LMD
- no. of look ahead's (how many symbols you are going to see, when you make a decision)

Operations:

- First()
- Follow()

→ **First():-**

- It is the first terminal of the production. when you derive all the strings

Ex:-

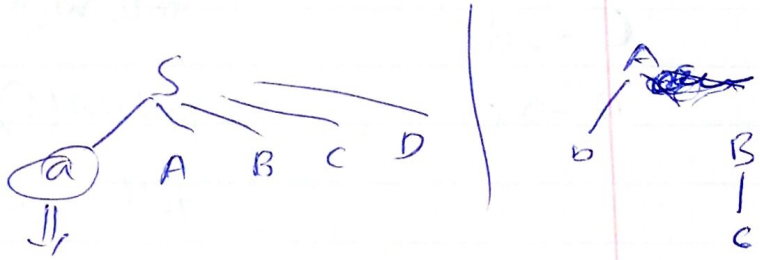
$S \rightarrow aABCD$

$A \rightarrow b$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$



First(S) = a ; First(A) = b, First(B) = c

① If $S \rightarrow ABCD$
$A \rightarrow b/\varepsilon$
$B \rightarrow c$

⇒ First(S) can be 'b' (or) 'c'.

→ **follow():**

- what is the terminal which could follow a variable in the process of derivation

- S $ → 'S' followed by '$', as always input ends with "$"

- S → ABCD

  ⇒ ABCD$ ⇒ follow(D) = $

  C → d

  ⇒ ABdD$ ⇒ follow(B) = d

⟶

- S → ABCD — follow(S) = {$}
  A → b | ε — follow(A) = first(BCD) = {c}
  B → C — follow(B) = first(CD) = {d}
  C → d — follow(C) = first(D) = {e}
  D → e — follow(D) = {$}