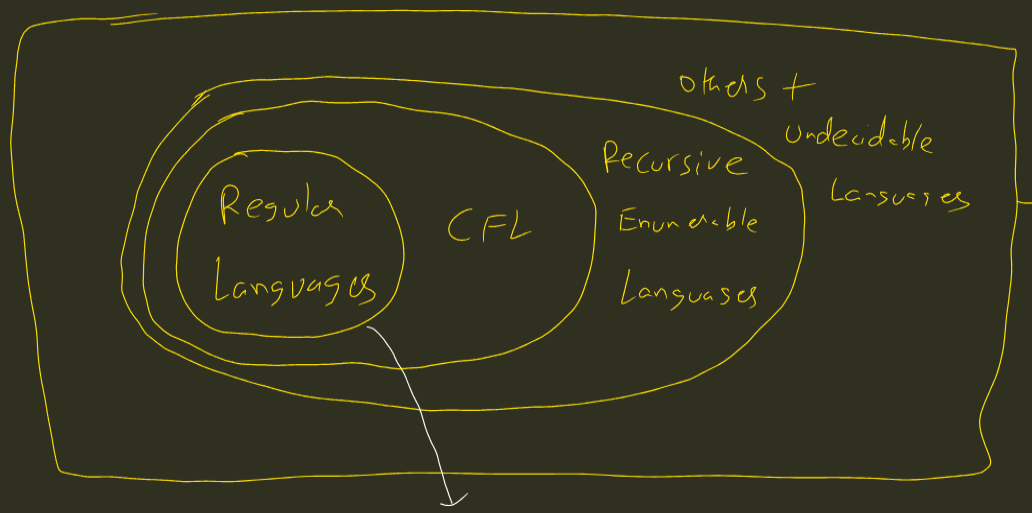
Context free grammars (CFGs) , CFL → Context-Free Languages

→ CFGs are a way of representing the Languages in the CFL class of Languages in the Chomsky hierarchy of Languages.

→ We saw the class of Regular Languages.

→ A Language is Regular iff
↳ if and only if
there exists a DFA/NFA capable of representing that Language.

→ CFL is a class of Languages which is completely a superset of Regular languages.



Class of Regular Languages ⊂ Class of Context free Languages.
(⊂ : subset of)

Every regular language is also a Context free language and we can represent those regular languages using CFGs. but not every Context free language is Regular.

Examples of Some non-Regular Languages :-

example-1 :-

$\Sigma = \{a, b\}$

$L_1 = \{a^n b^n \mid n \geq 0\}$ → set of all strings

$\{\epsilon, ab, aabb, aaabbb, ----\}$

example-2 :-

L_2 = set of all palindrome strings on any Σ.

Strings that are same even if we reverse them.

ex :- "aba", "abba", "malyalam".

example-3 :-

$\Sigma = \{(,)\}$

L_3 = set of all strings with balanced parenthesis.

all strings of the form

(), (()(), ((())).

Every open bracket '(' must be matched with some closing bracket ')' and vice versa.

)(, ())

strings of these types are not allowed in L_3.

→ Regular Languages are language sets which can be represented using DFA/NFA.

→ We call a Language Non-Regular if there exists no DFA/NFA capable of representing the given Language.

→ There is a method called "Pumping lemma", using which we can mathematically prove no DFA construction is possible for these examples.

→ We will not discuss this proof, just accept these Languages are non-regular.

These are some examples of non-Regular Language.

→ There is no DFA capable of representing them.

35%

Definition of CFG :-

Similar to DFA, NFA, CFG is represented by following elements → (V, T, P, S) where.

V: set of non-terminals (similar to a variable)

T: set of Terminals (similar to constant pieces of strings)

P: set of Production rules (or) grammar rules.

Where Every rule is of the form,

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \text{----}$$

where $X \in V$ is a non-Terminal, $\alpha_1, \alpha_2, \text{----}$ are strings made of V,T symbols joined in a string by Concatenation operator.

→ it means that X can be derived using any of α_1 (or) α_2 (or) ---

S: $S \in V$, one of the Non-terminal which we call as starting non-terminal, using which we derive every string in L.

→ CFG is a way of representing the grammar rules of a CFL. Using these recursive grammar rules we can generate all possible strings in your CFL set.

let's look at example CFGs for some languages

↓

CFG examples

(1) Consider the Language set of all palindrome strings.

$L_1 = \{$ Binary palindrome strings with $\Sigma = \{0,1\}\}$

CFG of L_1 :

$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$$

$G: (\{S\}, \{0,1,\epsilon\}, P, S)$

where P: $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$

→ So this means that S can be either made using

0.S.0 (or) 1.S.1 (or) 0 (or) 1 (or) ϵ (empty string).

→ like that, S (or) any Non-terminal represents a set of strings they can produce using their simple Production rules like $S \rightarrow 0$ (or) $S \rightarrow 1$ (or) $S \rightarrow \epsilon$

(or) recursive rules like $S \rightarrow 0S0$ where if you have generated a string 'x' with S's rules, then you can say S can also generate "0x0" strings too.

→ Like that, every non-terminal is like a Variable which represents a set of strings it can generate.

ex:-

$$A \rightarrow aA \mid \epsilon$$

→ Here A can generate strings of

$\{\epsilon, a, aa, aaa, \text{------}\}$

→ At the end we only consider the set of strings generated by your starting Non-Terminal, which is generally represented by 'S'.

⇒ In our Language example you can generate strings

$\{\epsilon, 0, 1, 00, 11, 010, 000, 111, 101, \text{-----}\}$

→ Any string that you can derive using the Production rules from your starting Non-terminal will belong to your language.

② $L_2 = \{a^n b^n \mid n \geq 0\}$

CFG of L_2

$S \rightarrow aSb \mid \epsilon$

→ Again L_2 is set of all strings 'S' can generate.

$L_2 = \{\epsilon, ab, aabb, aaabbb, \text{-----}\}$

→ S will firstly generate ϵ using $S \rightarrow \epsilon$ rule,

From there S can generate $a \epsilon b$ which is 'ab', from there S can generate a, ab, b (or) aabb and all strings of the form $a^n b^n$ where $n \geq 0$.

③ $L_3 = \{a^n \mid n \geq 0\}$

CFG for L_3

$S \rightarrow aS \mid \epsilon$

④ $L_4 = \{a^n \mid n \geq 1\}$

CFG for L_4

$S \rightarrow aS \mid a$

$L_{4.5} = \{a^n b \mid n \geq 1\}$

$S \rightarrow Ab$

$A \rightarrow aA \mid a$

⑤ $L_5 = \{\text{"abab"}\}, \Sigma = \{a, b\}$

CFG for L_5

$S \rightarrow abab$

⑥ $L_6 = \{x_1, x_2, \text{---}, x_k\}, \Sigma = \{a, b\}$, k is a finite natural number,

and $x_1, x_2, \text{---}, x_k \in \Sigma^*$

CFG for L_6

$S \rightarrow x_1 \mid x_2 \mid \text{-----} \mid x_k$

34%

(7) $L_7 = \{ a^i b^j c^k \mid i \geq 0, j \geq 0, k \geq 0 \}$

$S \longrightarrow A.B.C$

$A \longrightarrow aA \mid \epsilon$

$B \longrightarrow bB \mid \epsilon$

$C \longrightarrow cC \mid \epsilon$

(8) $L_8 = \{ b^i c^k a^j \mid i \geq \alpha, k \geq \beta, j \geq \gamma \}$

$S \longrightarrow B.C.A$

$B \longrightarrow bB \mid b^{\alpha}$

$C \longrightarrow cC \mid c^{\beta}$

$A \longrightarrow aA \mid a^{\gamma}$

(9) $L_9 = \{ a^m b^n \mid m \geq n \}$

$S \longrightarrow aSb \mid A$

$A \longrightarrow aA \mid \epsilon$

→ Notice the difference between L_9 and L_2.

in L_2 we had exactly same number of a's, b's but here we need more a's.

→ so instead of 'ϵ' in L_2, here we start inner most part with A which gives us more a's. Then use $S \rightarrow aSb$ to put as many a's as b's at start and end.

(10) $L_{10} = \{ a^m b^n \mid m \leq n \}$

$S \rightarrow aSb \mid B$

$B \longrightarrow bB \mid \epsilon$

→ Here instead of excess a's, we supply excess b's using B..

(11) $L_{11} = \{ a^m b^n \mid m > n \}$

$S \rightarrow aSb \mid A$

$A \rightarrow aA \mid a$

→ Notice the difference in L_9, L_{11}

$m \geq n$ ↙ ↘ $m > n$.

→ So here we need strictly more supply of a's so A will generate a^k, $k \geq 1$ here.

→ in L_9 A generates a^k, $\underline{k \geq 0}$.

L_9, L_{10}, L_{11} has extra conditions on m, n.

→ if there are no conditions it will become L_8

These are all simple examples just to get familiar with basic CFG construction for simple examples.

(12) $L_{12} = \Sigma^*$, $\Sigma = \{0,1\}$

↳ alphabet

↳ Σ^* (or) U (universal set)

↳ set of all possible strings made using Σ.

CFG for L_{12}

$S \rightarrow 0S \mid 1S \mid \epsilon$

→ Try to understand why this CFG works for this language.

→ Consider the string "0101100"

0"10100"

it can be derived using 0S where

↳ 1"0100"

↳ 1S

↳ 0"100"

↳ 0S ------> 0S

"0" ϵ

finally use $S \rightarrow \epsilon$

(13) $L_{13} = \Sigma^*$, $\Sigma = \{a_1, a_2, \dots a_k\}$ where k is finite.

Similar to L_{12}

CFG of L_{13}

$S \rightarrow a_1S \mid a_2S \mid \dots \mid a_kS \mid \epsilon$

(14) $L_{14} = \Sigma^* - \{\epsilon\}$ (or) Σ^+, $\Sigma = \{0,1\}$

set of all possible strings made with Σ but length ≥ 1.

CFG for L_{14}

$S \rightarrow 0S \mid 1S \mid 0 \mid 1$ → notice difference with L_{12}.

since we need strings with length ≥ 1

we use 0,1 as base case instead of ϵ.

40%

More non-trivial examples :-

L_1 = Set of all strings with balanced Parenthesis.

$$\Sigma = \{ (,) \}$$

→ Recall this from the examples of non-regular Languages.

CFG of L_1

$$S \longrightarrow (S) \mid SS \mid \epsilon$$

these will cover all cases of possible strings

Exercise :-

Verify if you can generate the following strings using the above production rules.

(i) "( )"

(ii) "( )( )"

(iii) "(( ))( )"

(iv) "((( )( ))( ))"

(v) "(( )(( )))( )"

L_2 = set of all

generate

(i) 2 variabl

(ii) all positiv

(iii) operations

CFG for L_2

P:

EXPR ⟶

VAR ⟶

NUM ⟶ 0

$G = (V, T, P, S)$

$V = \{$ EXPR,

S

Example-2 :-

L_2 = set of all possible simple arithematic expressions one can generate using

(i) 2 variables a, b allowed.

(ii) all positive integer numbers allowed (only binary representations)

(iii) operations allowed : +, *, ( ).

CFG for L_2

P:

EXPR ⟶ EXPR + EXPR | EXPR * EXPR | (EXPR) | VAR | NUM

VAR ⟶ a | b

NUM ⟶ 0 NUM | 1 NUM | 0 | 1

$G = (V, T, P, S)$

$V = \{ EXPR, VAR, NUM \}$. $T = \{ +, *, (,), a, b, 0, 1 \}$

$S =$ EXPR.

→ In this example EXPR is your start non-Terminal from which all strings in your Language are derived.

→ VAR, NUM are supporting Non-terminals.

↓ Provides variables a (or) b

↓ Provides all binary representations of numbers. In this case by giving all strings made using {0,1} with length ≥ 1.

→ This example shows us how we can Interpret simple arithematic expressions.

→ we use similar way to write grammar for if-else blocks, switch-case blocks, for, while loop blocks, ----.

That is how your programming languages are interpreted piece-by-piece writing grammar rules for every possible block.

Now CFG can represent the Language L_2.

Verify for following strings (Assume operator precedence here as $() > * > +$)

→ This precedence is not represented by our grammar but assume we have this precedence.

(i) $a + b^*(01)$

$a + b^*(01)$

EXPR → EXPR (a) + EXPR ($b^*(01)$)

a: EXPR → VAR, VAR → a ✓

$b^*(01)$: EXPR → EXPR (b) * EXPR ((01))

b: EXPR → VAR, VAR → b

(01): EXPR → (EXPR)

01: EXPR → NUM

01: NUM → 0.NUM (1)

NUM → 1.

Now we will look at "Parse trees", better way of representing How any string is derived using the grammar rules.

→ Similarly try to derive strings $b^*a^*(01+10)$, $a^*(11+a)^*b$, yourself.

Parse trees :- a parse tree of a string represents the production rules you used in order to generate the string in order.

→ Every non leaf node is a Non-Terminal Symbol.

→ Every leaf node is a Terminal Symbol.

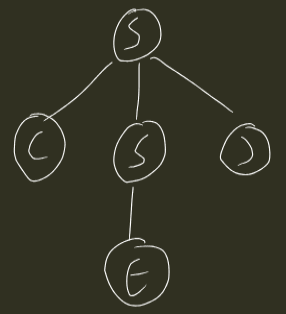
→ Every [parent → Concatenation of child nodes inorder] is one of the production rules.

→ Root node non-Terminal represents your string and from there draw its children based on which production rule you'll use. And subtree for every non-Terminal children is a parse tree for deriving those non-Terminals again.

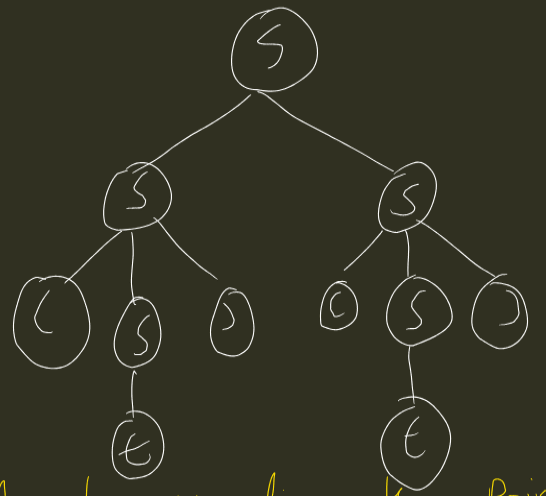
Ex : Consider the grammar

$S \rightarrow (S) \mid SS \mid \epsilon$

① Parse tree for "()"



② Parse tree for "()()"



→ Try to understand by reading the Points on left side.

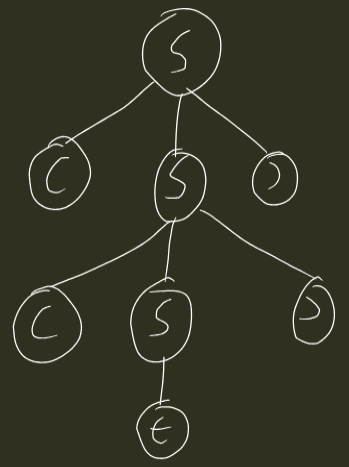
→ Every non-leaf node is a non-Terminal Symbol, whose subtree is also a parse tree for deriving its part.

→ only the root of the tree will always be your start non-Terminal from which whole string is derived

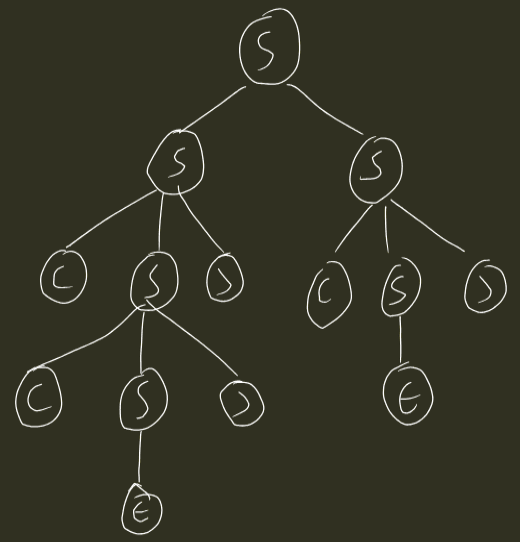
Consider the same grammar

$$S \rightarrow (S) \mid SS \mid \epsilon$$

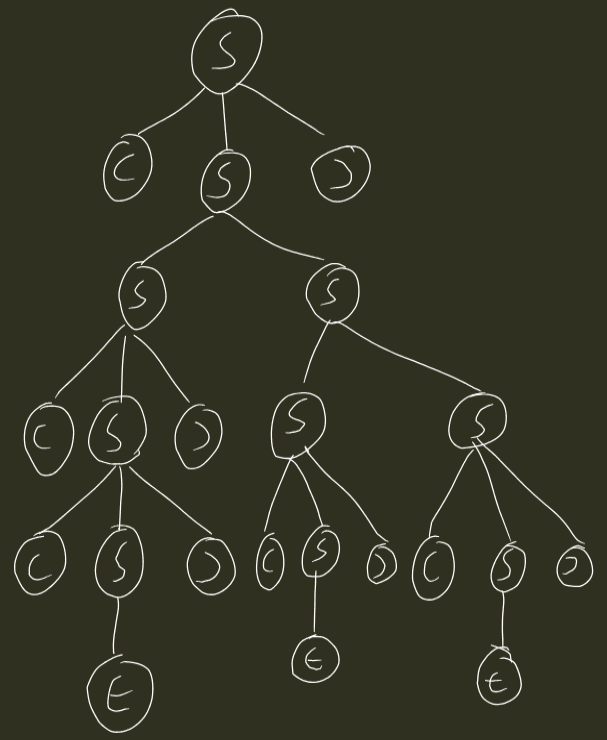
Parse tree for "(())"



Parse tree for "(())()"



Parse tree for "((()))()()"



→ Parse tree tells you exactly which production rule derived what parts in which order in your string.

→ When you are asked to draw a parse tree for a string, first break up the string acc. to the production rules, build the individual nonterminals one by one.

Again Consider the Language (L_2) of all simple arithematic expressions where Variables (a,b), all binary numbers, operators ($+, *, ()$) are allowed.

CFG for L_2

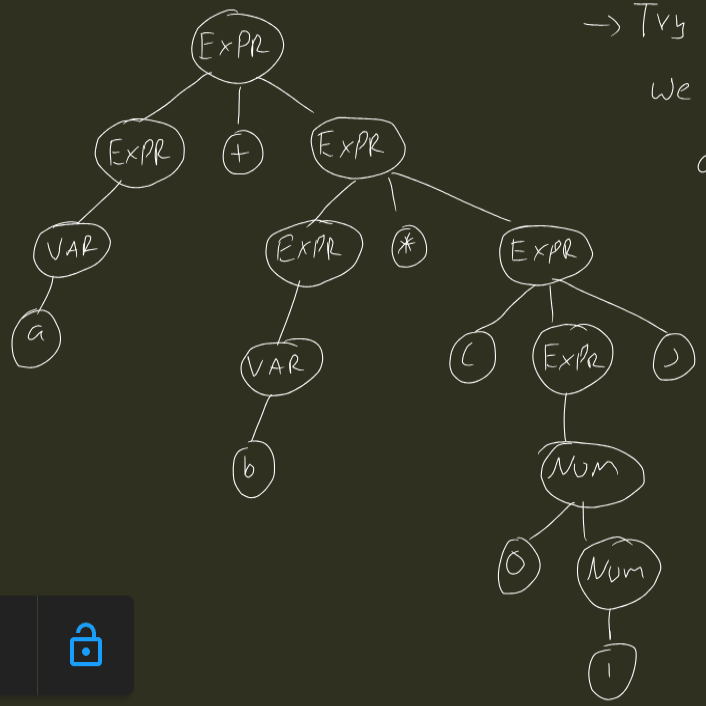
→ EXPR ⟶ EXPR + EXPR | EXPR * EXPR | (EXPR) | VAR | NUM

VAR ⟶ a | b

NUM ⟶ 0 NUM | 1 NUM | 0 | 1

→ The grammar doesnt specify any operator precedence here but assume that $() > * > +$ priority like BODMAS

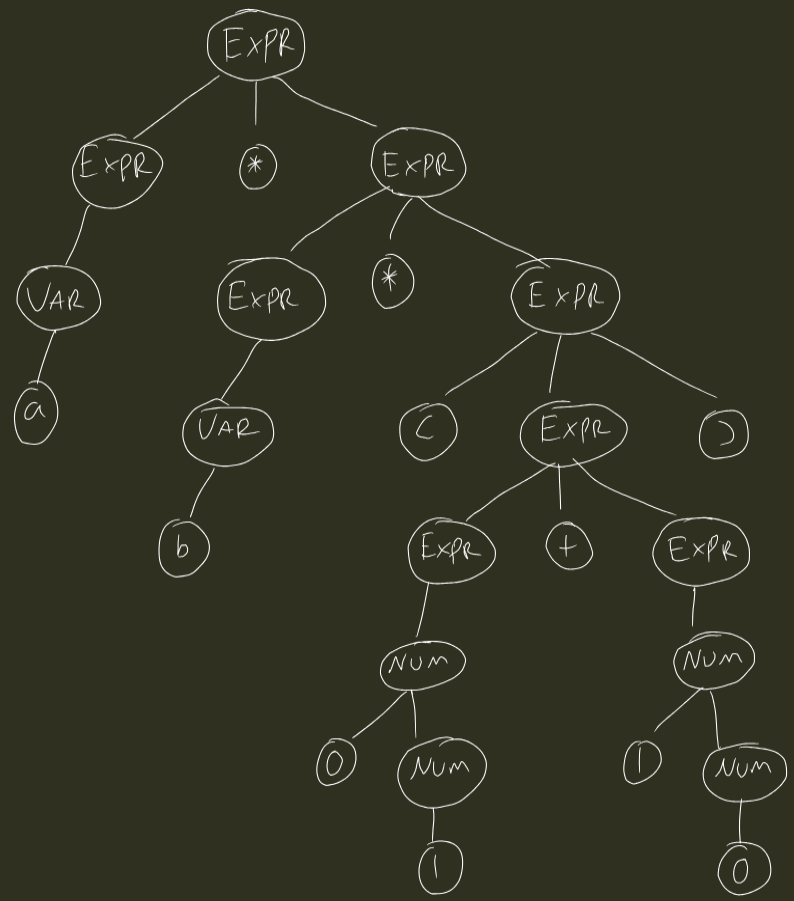
Parse tree for $a + b^*(01)$



→ Try to observe that we are strictly draving according to our production rules and nothing more.

→ every parent to all children combined is just a Production rule.

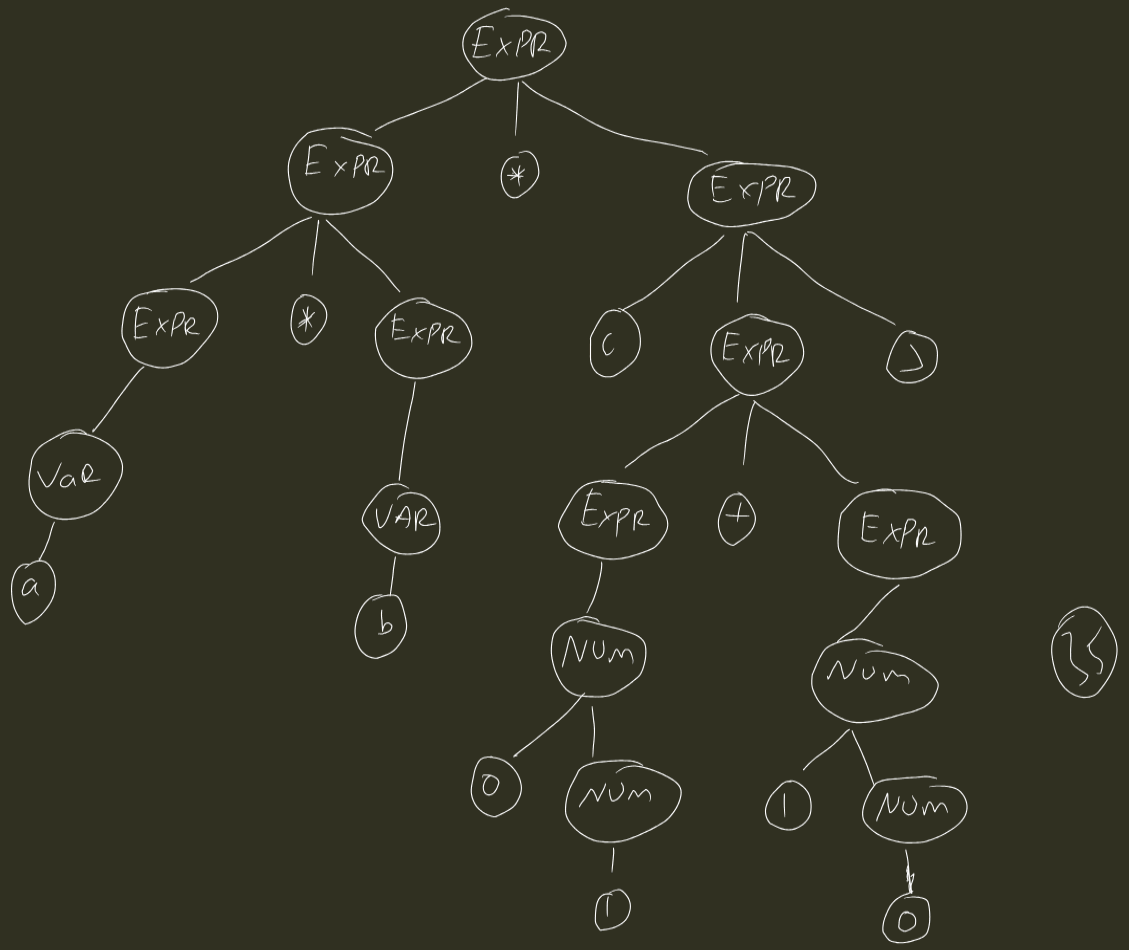
Similarly, Parse tree for $a^* b^* (01 + 10)$



→ We can also split at 2nd * here, which will give us another derivation, another tree. will see....

35%

In our previous example of Parse tree for a*b*(01+10) we can also construct a Parse tree like this below



Having multiple Parse trees for a string is a problem of the grammar.

Ambiguity in Context free grammars :-

Ambiguous grammar :- If a given grammar gives more than one Parse tree for atleast one string in your language ⇒ its an ambiguous grammar.

ex :- our ExPR grammar example we just saw is ambiguous on strings like a+b+01 (or) a*b*(01+10)...

→ But having an ambiguous grammar is not means the language is ambiguous itself. There can be other grammars for same language that are non-ambiguous.

Inherently Ambiguous Languages :-

A Language is inherently Ambiguous if there is no non-ambiguous grammar possible that can represent our Language.

examples :-

$L_1 = \{a^n b^n \mid n \geq 0\}$

G :- $S \rightarrow aSb \mid \epsilon$

→ This is a non ambiguous grammar. For all $a^n b^n$ strings the Parse tree is unique.

L_2 = Set of all balanced Parenthesis strings, $\Sigma = \{(,)\}$.

G :- $S \rightarrow SS \mid \epsilon$

→ This is an ambiguous grammar

→ for every string we can do

$S \rightarrow \underline{S}\,S$ (↓ your string, → ϵ) ∞ times so.

L_3 = our Previous simple arithmatic expressions language.

→ Ambiguous grammar

→ But languages L_2, L_3 are not inherently ambiguous, we can change their current grammars and show a new non-ambiguous grammar for them.

→ We will later see some techniques to remove ambiguity but for now understand that it is a problem.