CFG	Context-Free Language
Set of rewrite rules	{**words**: **derived** with **rules** in CFG} Regular Language + Recursion

RL	CFL
finite memory to recognize	**infinite** memory to recognize
Recognize structures without nesting • **ie:** valid variable names	Recognize structures with nesting • **ie:** nested parentheses
RE, DFA, NFA	**CFG**
Express a RL	Express a CFL

Regular Language ⇒ Context Free

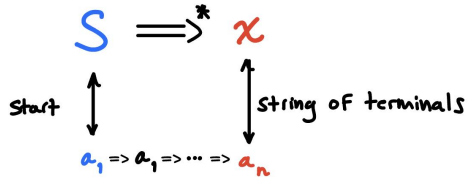
Ex: CFG for a RL	
Regular Language $\Sigma = \{a, b\}$ $L = \{a^n b^n : n \in N\}$	
N	{S}
T	{a, b}
P	$S \rightarrow \varepsilon$ $S \rightarrow aSb$

CFG (formally)		Convention	
N	non-terminal	• Lower-Case • Start of Alphabet	(a, b, c, …)
T	terminal symbols	• Lower-Case • End of Alphabet	(w, x, y, z)
P	Production Rules	• Upper-Case • Start of alphabet	(A, B, C, …)
S	Start Symbol		

T/N Sequences	V*	• Greek letters	(α, β, γ, …)

Derivations

Derives	Directly Derives
A ⇒* γ (1+ rules)	A ⇒ γ (1 rule)

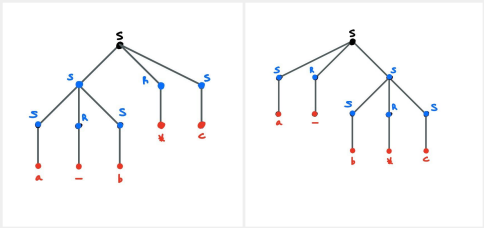
Derivation of a word


Leftmost/Rightmost Derivations

How to eliminate choice:

Leftmost Derivation	Rightmost Derivation
derive leftmost N first	derive rightmost N first
S ⇒ **B**gC ⇒ abgC ⇒ abgef	S ⇒ Bg**C** ⇒ Bgef ⇒ abgef

2 Leftmost Derivations	
S → a | b | c | SRS R → + | - | * | /	
S ⇒ SRS ⇒ SRSRS ⇒ aRSRS ⇒ a-SRS ⇒ a-bRS ⇒ a-b*S ⇒ a-b*c	S ⇒ SRS ⇒ aRS ⇒ a-S ⇒ a-SRS ⇒ a-bRS ⇒ a-b*S ⇒ a-b*c



Derivations & Parse Trees

Recognition Algorithm

2 Functions

1. **Syntax:**
    a. is the **'word(program)'** in the language(valid)
2. **Meaning:**
    a. **representation** of the structure = **meaning** to the program!
    b. **ie:** does a-b*c mean (a-b)*c or a-(b*c)?

For some grammars: 1 derivation style ⇒ unique derivation.

For others: 1 derivation style ⇒ multiple derivations

If we care only about **syntax**: any derivation proves that the word is in the language.

Ambiguous/Unambiguous Grammars

Any Leftmost Derivation ⇒ **choose** the same non-terminal to expand (the leftmost one)

Distinct Leftmost Derivation ⇒ they **chose** different expansions of some Non-Terminal

Two distinct leftmost derivations ⇒ distinct parse trees

Ambiguous	2+ parse trees	2+ Leftmost Derivations **or** 2+ Rightmost Derivation
Unambiguous	1 Parse Tree	1 Leftmost Derivation **AND** 1 Rightmost Derivation

Structure of a parse tree == structure of a program

derivation = unique parse tree

Multiple Derivations = multiple parse trees

We can eliminate this uncertainty, if we can eliminate the choice in derivation:
A word can have multiple derivations by **order** and **choice of rules**
A parse tree does not care about **order**. It only cares about **choice of rules**

derivations (by choice of rules) = # parse trees

- # leftmost derivations = # parse trees
- # rightmost derivations = # parse trees

Parse Trees & Arithmetic Expressions

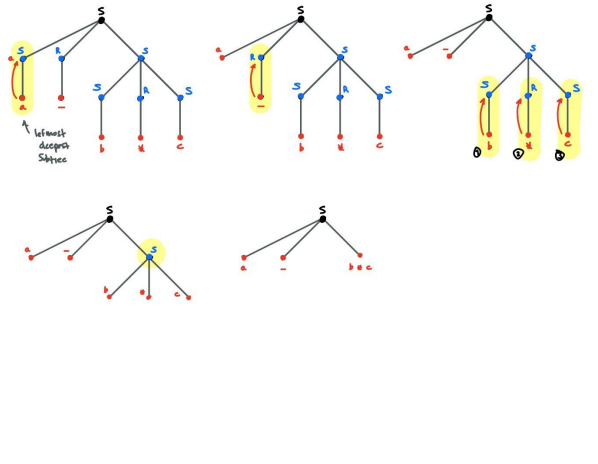
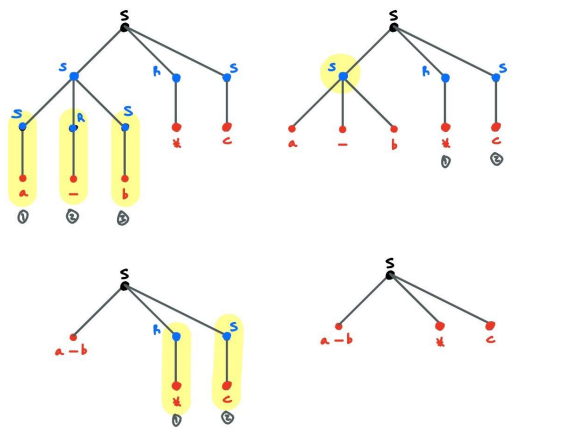
Depth-First Post-Order Traversal

A way to evaluate a parse tree (determine meaning) representing **arithmetic expressions**

Unique parse trees ⇒ Unique traversals ⇒ same word, different meanings

Steps:

1. Find LeftMost subtree
2. Evaluate (replace parent with child)
3. Repeat for all nodes of the tree

	
a-(b*c)	(a-b)*c

Ways to avoid Ambiguity:

Method 1

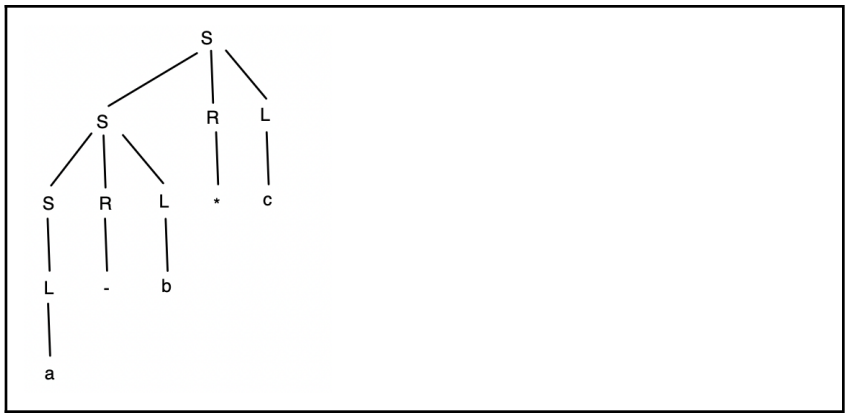
precedence heuristics

Ex	
Force the language syntax to require parentheses	
S → a | b | c | (SRS) R → + | - | * | /	
S ⇒ (SRS) ⇒ ((SRS)RS) ⇒ ((aRS)RS) ⇒ ((a-S)RS) ⇒ ((a-b)RS) ⇒ ((a-b)*S) ⇒ ((a-b)*c)	S ⇒ (SRS) ⇒ (aRS) ⇒ (a-S) ⇒ (a-(SRS)) ⇒ (a-(bRS)) ⇒ (a-(b*S)) ⇒ (a-(b*c))
✅ grammar is unambiguous. the words: (a-(b*c)) & ((a-b)*c) are different	

Method 2

Left/Right Associativity

Left Associative Grammar
S → SRL | L L → a | b | c R → + | − | ∗ | /
Only 1 Leftmost Derivation for a-b*c: S ⇒ SRL ⇒ SRLRL ⇒ aRLRL ⇒ a-LRL ⇒ a-bRL ⇒ a-b*L ⇒ a-b*c



Method 3

BEDMAS

Changes the associativity of a grammar. A grammar that follows BEDMAS rules more closely by making ∗ and / appear further down the tree. deeper parts ⇒ evaluated first ⇒ higher precedence

S → SPT | T T → TRF | F F → a | b | c |(S) P → + | − R → ∗ | /	S → S + T | T T → T ∗ F | F F → a | b | c | (S) Since (+, -) and (*, /) have the same precedence: