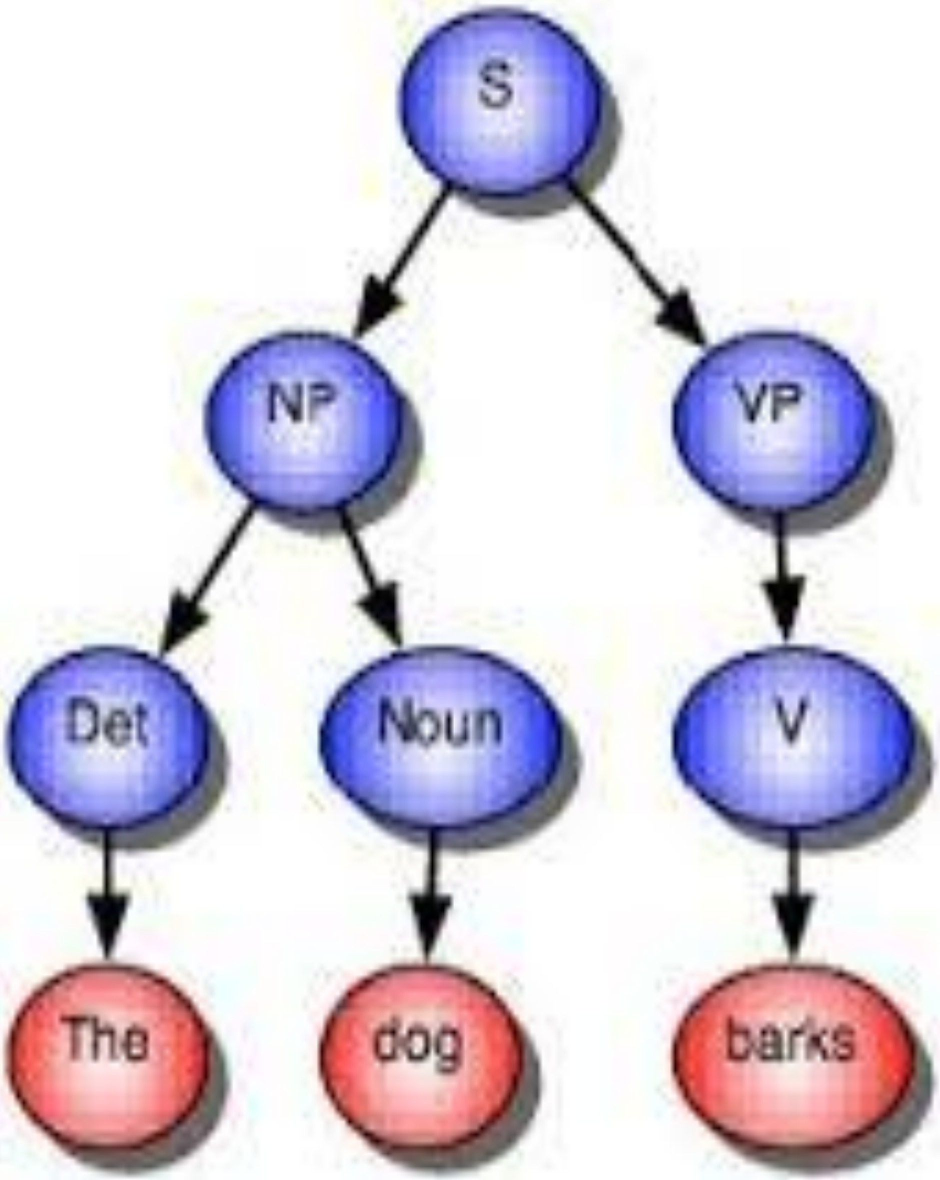
Parsing



CFG and Parsing

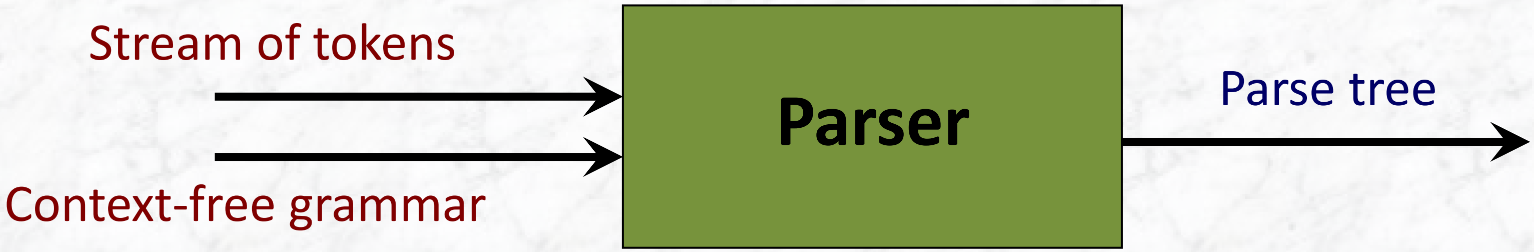
Instructor : Dr. Hanaa Bayomi Ali
Mail : h.mobarz @ fci-cu.edu.eg

Introduction

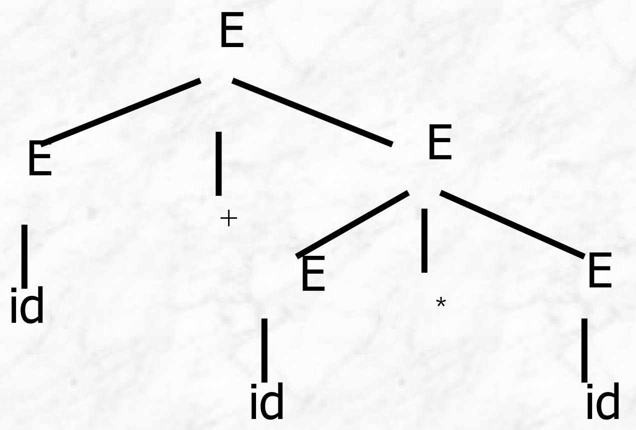
- Parsing is a process that constructs a syntactic structure (i.e. parse tree) from the stream of tokens.

- We already learned how to describe the syntactic structure of a language using (context-free) grammar.

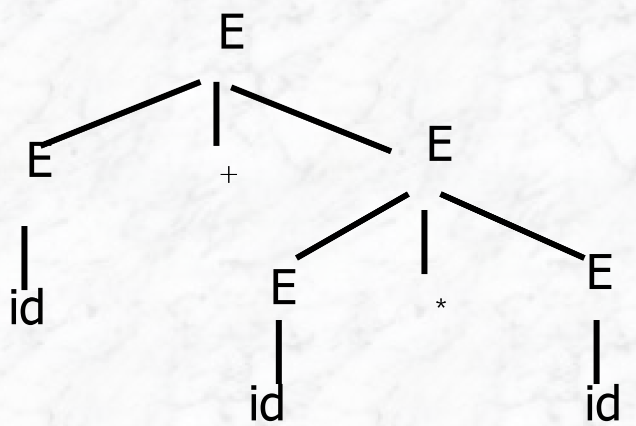
- So, a parser only needs to do this?



Top-down and Bottom-up parser



Top-down parsing



Bottom-up parsing

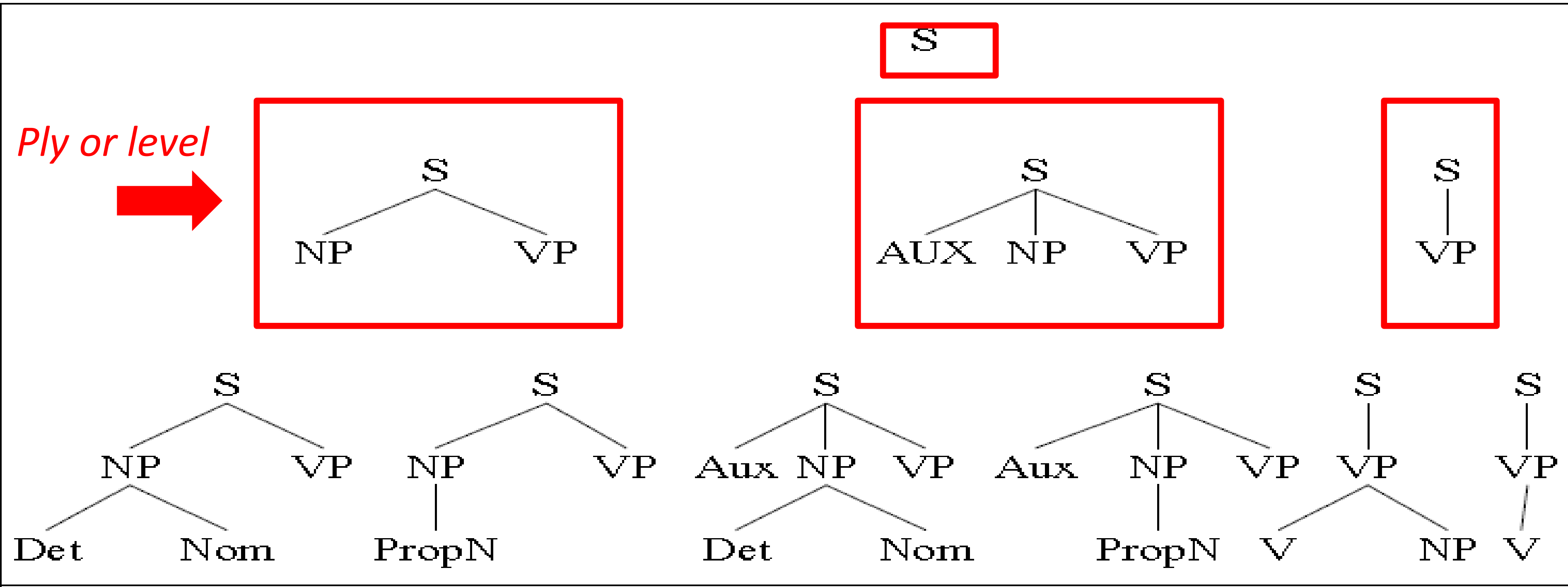
$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow id + E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

$$
\begin{aligned}
& E + E \\
&\Rightarrow E + E * E \\
&\Rightarrow E + E * id \\
&\Rightarrow E + id * id \\
&\Rightarrow id + id * id
\end{aligned}
$$

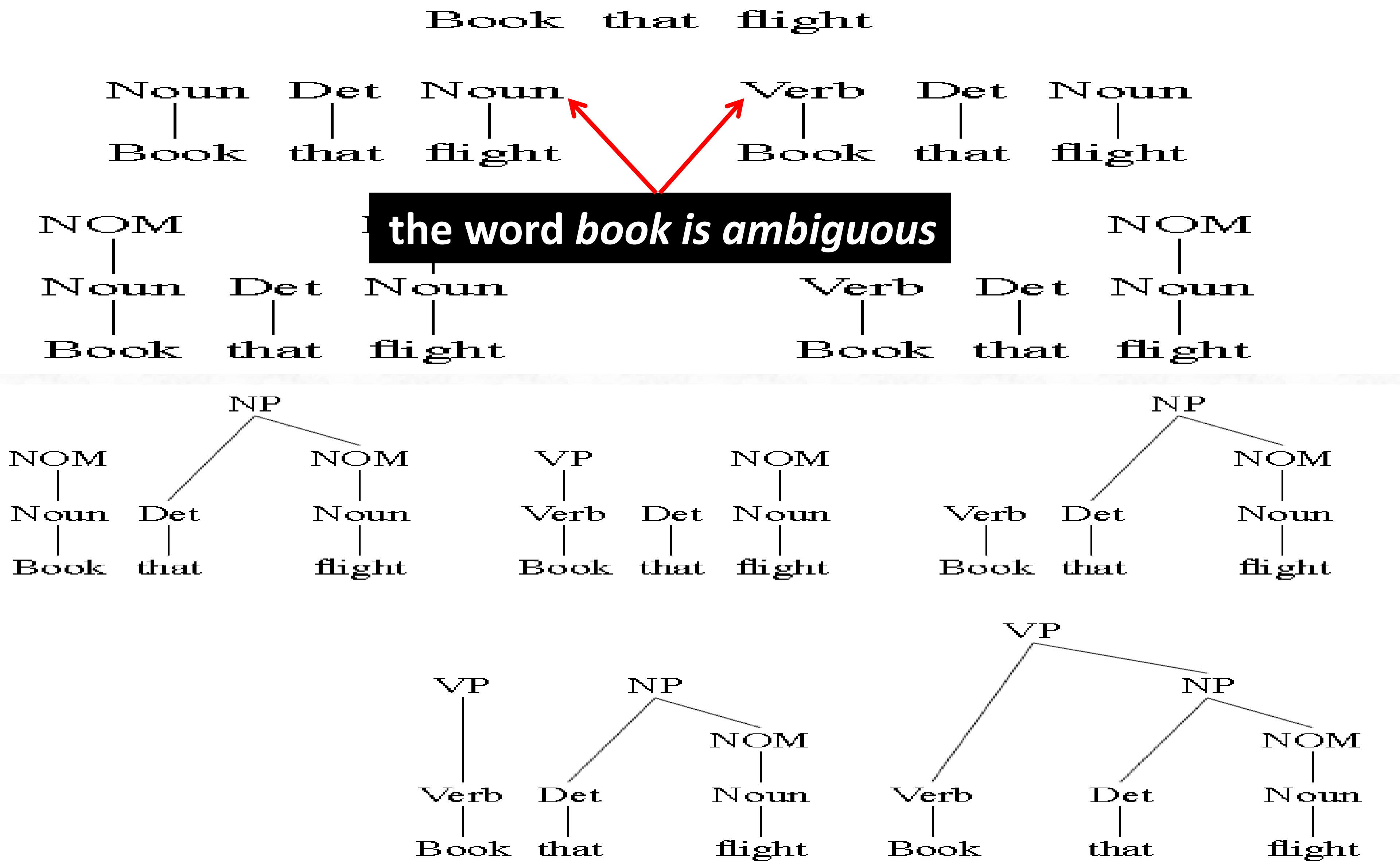
Top-down Parser

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	$Prep \rightarrow from \mid to \mid on$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	$Nominal \rightarrow Nominal\ PP$

Input :
Book that flight



Bottom-up parser



Top–Down VS Bottom–Up Parsing

Top – Down (goal-directed search)	Bottom-Up (data-directed search)
A parse tree is created from root to leaves	A parse tree is created from leaves to root
Tracing leftmost derivation	Tracing rightmost derivation
EXPECTATION-DRIVEN parsing	DATA-DRIVEN parsing
- Only search among grammatical answers - BUT: suggests hypotheses that may not be consistent with data - Problem: left-recursion	- Only forms hypotheses consistent with data - BUT: may suggest hypotheses that make no sense globally - More powerful than top-down parsing
Backtracking: Try different structures and backtrack if it does not matched the input (depth first)	**Predictive:** Guess the structure of the parse tree from the next input

Top –Down parsing

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until lower fringe of the parse tree matches the input string

1 *At a node labeled A, select a production with A on its LHS and, for each symbol on its RHS, construct the appropriate child*

2 *When a terminal symbol is added to the fringe and it doesn’t match the fringe, backtrack*

3 *Find the next node to be expanded*

The key is picking the right production in step 1

– *That choice should be guided by the input string*

Example

CFR for “the classic expression grammar”

1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	$\rightarrow$	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	$\rightarrow$	<u>number</u>
9		|	<u>id</u>

Consider the input string `x – 2 * y`

Example (cont)

Prod'n	Sentential form	Input
–	⟨goal⟩	↑x – 2 ∗ y
1	⟨expr⟩	↑x – 2 ∗ y
2	⟨expr⟩ + ⟨term⟩	↑x – 2 ∗ y
4	⟨term⟩ + ⟨term⟩	↑x – 2 ∗ y
7	⟨factor⟩ + ⟨term⟩	↑x – 2 ∗ y
9	id + ⟨term⟩	↑x – 2 ∗ y
–	id + ⟨term⟩	x ↑– 2 ∗ y
–	⟨expr⟩	↑x – 2 ∗ y
3	⟨expr⟩ – ⟨term⟩	↑x – 2 ∗ y
4	⟨term⟩ – ⟨term⟩	↑x – 2 ∗ y
7	⟨factor⟩ – ⟨term⟩	↑x – 2 ∗ y
9	id – ⟨term⟩	↑x – 2 ∗ y
–	id – ⟨term⟩	x ↑– 2 ∗ y
–	id – ⟨term⟩	x – ↑2 ∗ y
7	id – ⟨factor⟩	x – ↑2 ∗ y
8	id – num	x – ↑2 ∗ y
–	id – num	x – 2 ↑∗ y
–	id – ⟨term⟩	x – ↑2 ∗ y
5	id – ⟨term⟩ ∗ ⟨factor⟩	x – ↑2 ∗ y
7	id – ⟨factor⟩ ∗ ⟨factor⟩	x – ↑2 ∗ y
8	id – num ∗ ⟨factor⟩	x – ↑2 ∗ y
–	id – num ∗ ⟨factor⟩	x – 2 ↑∗ y
–	id – num ∗ ⟨factor⟩	x – 2 ∗ ↑y
9	id – num ∗ id	x – 2 ∗ ↑y
–	id – num ∗ id	x – 2 ∗ y ↑

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	<u>number</u>
9		|	<u>id</u>

Left-Recursion -> Non-termination

Another possible parse for `x - 2 * y`

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\text{x} - 2 * \text{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\text{x} - 2 * \text{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow\text{x} - 2 * \text{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow\text{x} - 2 * \text{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\text{x} - 2 * \text{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\text{x} - 2 * \text{y}$
2	$\cdots$	$\uparrow\text{x} - 2 * \text{y}$

*If the parser makes **the wrong choices**, expansion doesn´t terminate!*

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

<foo> -> <foo> α | β

→

<foo>	-> β <bar>	
<bar>	-> α <bar>	| ε

where *bar* is a new non-terminal

The new grammar defines the same language as the old grammar, **using only right recursion.**

Example

The expression grammar contains two cases of left recursion

<foo> <foo> α β

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / *Factor*
	|	*Term*		|	*Factor*

Applying the transformation yields

B <bar> α

Expr	→	*Term* *Expr'*	*Term*	→	*Factor* *Term'*
Expr'	→	+ *Term* *Expr'*	*Term'*	→	* *Factor* *Term'*
	|	- *Term* *Expr'*		|	/ *Factor* *Term'*
	|	ε		|	ε

These fragments use only right recursion

With this grammar, a top-down parser will

- ***terminate***
- ***backtrack on some inputs***

Shift-Reduce Parsing

- A simple kind of bottom-up parser

- In common with all bottom-up parsers, a shift-reduce parser tries to find sequences of words and phrases that correspond to the right hand side of a grammar production, and replace them with the left-hand side, until the whole sentence is reduced to an **S**

- A common remedy to handle left-recursive rules is to run them with a bottom up search strategy

Shift-Reduce Parsing Cont'

- As input, it uses two arguments: the ***list*** of words to parse and a symbol **S,** representing the parsing goal

- The algorithm consists of a two-step loop:(Shift-Reduce)

  1. **Shift** a word from the sentence to parse onto a stack
  2. Apply a sequence of grammar rules to **reduce** elements of the stack

- This loop is repeated until:

  - There are no more words in the list
  - The stack is reduced to the parsing goal

Shift-Reduce Parsing Trace- example

Simple Grammar

S → NP VP

NP → Det N

VP → V NP

Det → ‘the’ | ‘a’

N → ‘waiter’ | ‘meal’

V → ‘brought’

Shift-Reduce Parsing Trace- example

It.	Stack	S/R	Word list
0			[the, waiter, brought, the, meal]
1	[the]	Shift	[waiter, brought, the, meal]
2	[det]	Reduce	[waiter, brought, the, meal]
3	[det, waiter]	Shift	[brought, the, meal]
4	[det, noun]	Reduce	[brought, the, meal]
5	[np]	Reduce	[brought, the, meal]
6	[np, brought]	Shift	[the, meal]
7	[np, v]	Reduce	[the, meal]
8	[np, v, the]	Shift	[meal]
9	[np, v, det]	Reduce	[meal]
10	[np, v, det, meal]	Shift	[]
11	[np, v, det, n]	Reduce	[]
12	[np, v, np]	Reduce	[]
13	[np, vp]	Reduce	[]
14	[s]	Reduce	[]

S → NP VP
NP → Det N
VP → V NP
Det → 'the' | 'a'
N → 'waiter' | 'meal'
V → 'brought'

CFGs and PCFGs (Probabilistic) Context-Free Grammars

CFGs

$S \rightarrow NP\ VP$
$VP \rightarrow V\ NP$
$VP \rightarrow V\ NP\ PP$
$NP \rightarrow NP\ NP$
$NP \rightarrow NP\ PP$
$NP \rightarrow N$
$NP \rightarrow e$
$PP \rightarrow P\ NP$

people fish tanks
people fish with rods

$N \rightarrow$ people
$N \rightarrow$ fish
$N \rightarrow$ tanks
$N \rightarrow$ rods
$V \rightarrow$ people
$V \rightarrow$ fish
$V \rightarrow$ tanks
$P \rightarrow$ with

Phrase structure grammars
= context-free grammars (CFGs)

G = (T, N, S, R)

T is a set of terminal symbols

N is a set of nonterminal symbols

S is the start symbol ($S \in N$)

R is a set of rules/productions of the form $X \to \gamma$

$X \in N$ and $\gamma \in (N \cup T)^*$

A grammar G generates a language L.

Probabilistic – or stochastic – context-free grammars (PCFGs)

G = (T, N, S, R, P)

- T is a set of terminal symbols
- N is a set of nonterminal symbols
- S is the start symbol (S ∈ N)
- R is a set of rules/productions of the form $X \rightarrow \gamma$
- P is a probability function
  - P: R $\rightarrow$ [0,1]
  - $\forall X \in N, \sum_{X \rightarrow \gamma \in R} P(X \rightarrow \gamma) = 1$

A grammar G generates a language model L.

$$\sum_{\gamma \in T^*} P(\gamma) = 1$$

A PCFG

$S \rightarrow NP\ VP$	1.0
$VP \rightarrow V\ NP$	0.6
$VP \rightarrow V\ NP\ PP$	0.4
$NP \rightarrow NP\ NP$	0.1
$NP \rightarrow NP\ PP$	0.2
$NP \rightarrow N$	0.7
$PP \rightarrow P\ NP$	1.0

N → *people*	0.5
N → *fish*	0.2
N → *tanks*	0.2
N → *rods*	0.1
V → *people*	0.1
V → *fish*	0.6
V → *tanks*	0.3
P → *with*	1.0

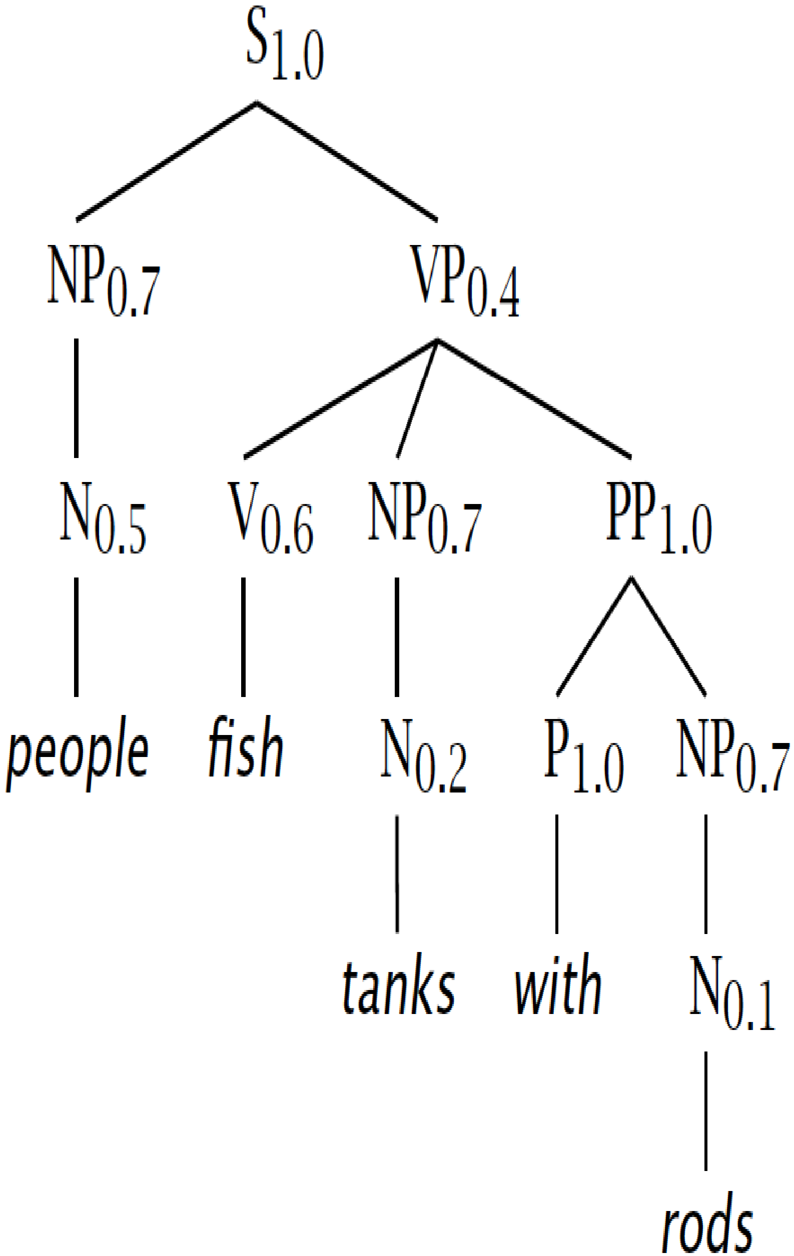
The probability of trees and strings

P(*t*) – The probability of a tree *t* is the product of the probabilities of the rules used to generate it.

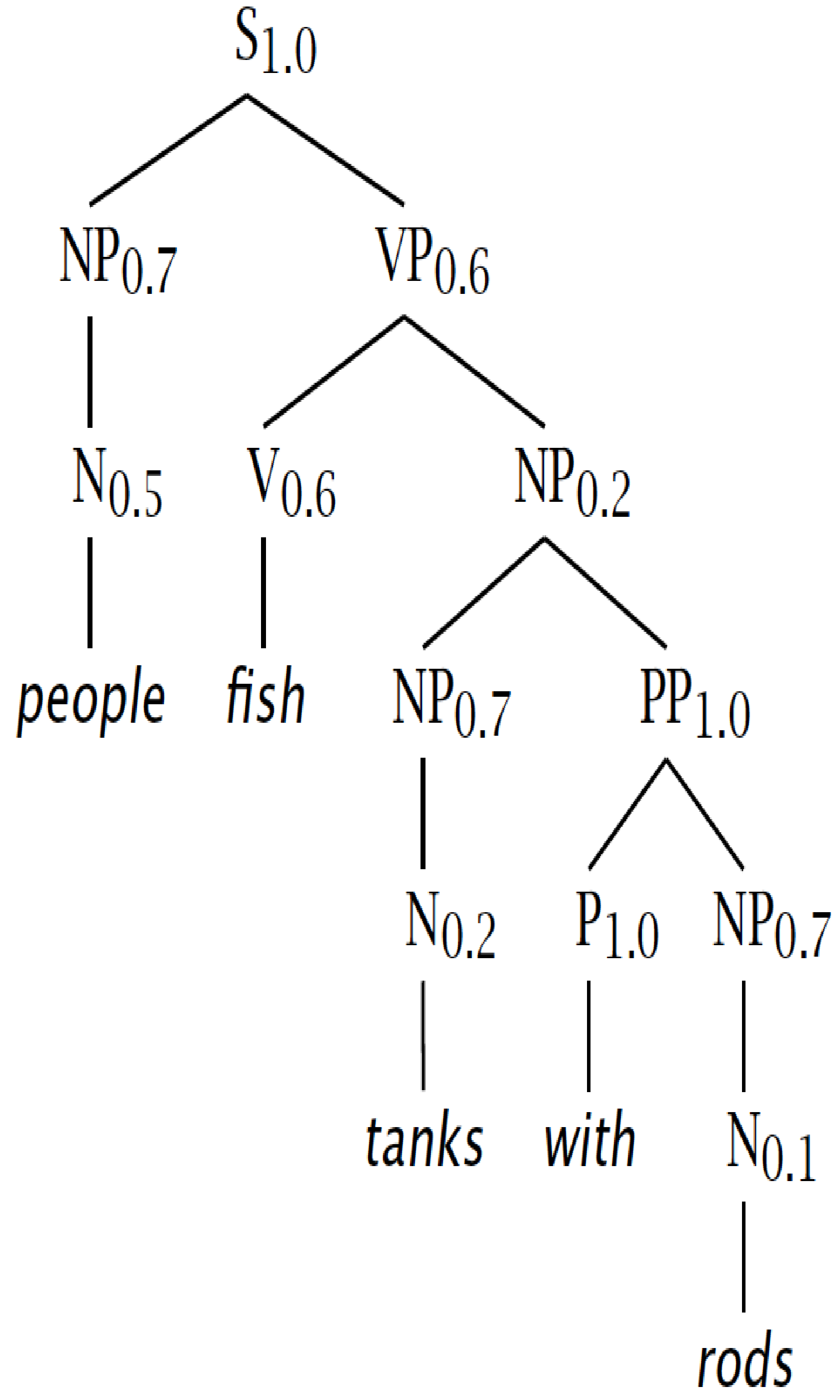
P(*s*) – The probability of the string *s* is the sum of the probabilities of the trees which have that string as their yield

$$P(s) = \Sigma_j P(s, t) \quad \text{where } t \text{ is a parse of } s$$
$$= \Sigma_j P(t)$$

t_1:


S1.0
NP0.7
VP0.4
N0.5
V0.6
NP0.7
PP1.0
people
fish
N0.2
P1.0
NP0.7
tanks
with
N0.1
rods

t_2:


S1.0
NP0.7
VP0.6
N0.5
V0.6
NP0.2
people
fish
NP0.7
PP1.0
N0.2
P1.0
NP0.7
tanks
with
N0.1
rods

Tree and String Probabilities

s = *people fish tanks with rods*

$P(t_1) = 1.0 \times 0.7 \times 0.4 \times 0.5 \times 0.6 \times 0.7$
$\times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1$
$= 0.0008232$

Verb attach

$P(t_2) = 1.0 \times 0.7 \times 0.6 \times 0.5 \times 0.6 \times 0.2$
$\times 0.7 \times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1$
$= 0.00024696$ [more depth → small number]

Noun attach

$P(s) = P(t_1) + P(t_2)$
$= 0.0008232 + 0.00024696$
$= 0.00107016$

Useful PCFG Tasks

Observation likelihood: To classify and order sentences.

determining how likely a string is to be produced by a PCFG.

$P(s) = \Sigma_j P(s, t)$ where t is a parse of s

Most likely derivation: To determine the most likely parse tree for a sentence.

picking the parse with the highest probability is the correct way to do disambiguation. (Max)