Overview

- Background – Parsing
- Parsing as Search
  - Top-down Parsing
  - Bottom-up Parsing
- A Basic Top-Down Parser
  - Adding bottom-up filtering
- Problems: Basic Top-Down Parser
  - Left-recursion
  - Ambiguity
  - Repeated Parsing Subtrees
- Dynamic Programming

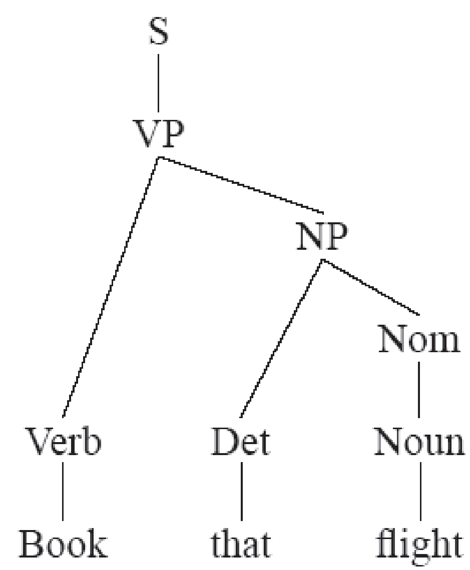
Background

- Syntactic parsing
  - The task of recognizing a sentence and assigning a syntactic structure to it
- Since CFGs are a declarative formalism – do not specify how the parse tree for a given sentence should be computed
- Parse trees are useful in applications such as :
  - Grammar checking
  - Semantic analysis
  - Machine translation
  - Question answering
  - Information extraction

Parsing as Search

- The parser can be viewed as searching through the space of all possible parse trees to find the correct parse tree for the sentence.
- ***How can we use the grammar to assign the parse tree?***

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



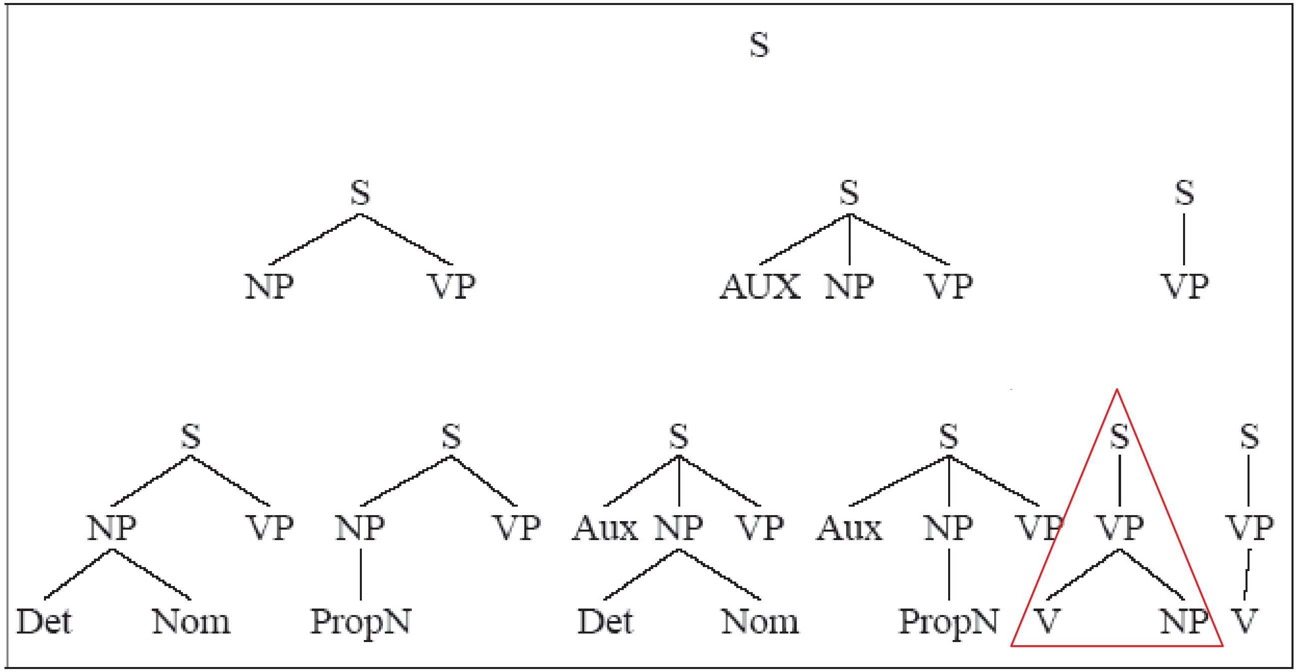
Parsing as Search

- The goal of a parsing search is to find all trees whose root is the start symbol S, which cover exactly the words in input.
- Two kinds of constraints:
  - Data – input words as leaves (*book, that, flight*)
  - Grammar – must have one root, which must be the start symbol S
- Leads to the two search strategies:
  - *Top-down* or goal-directed search
  - *Bottom-up* or data-directed search

Top-down Parser

- Trying to build from the root node S down to the leaves
- Start by start symbol S.
- Find the grammar rules with S on the left-hand side and expand the constituents in new trees
- At each level or **ply**, use the right-hand sides of the rules to provide new sets of expectations
- Tree grows until reaches the part-of-speech categories at the bottom
- Trees whose leaves fail to match all the words in the input can be rejected

Top-down Parser

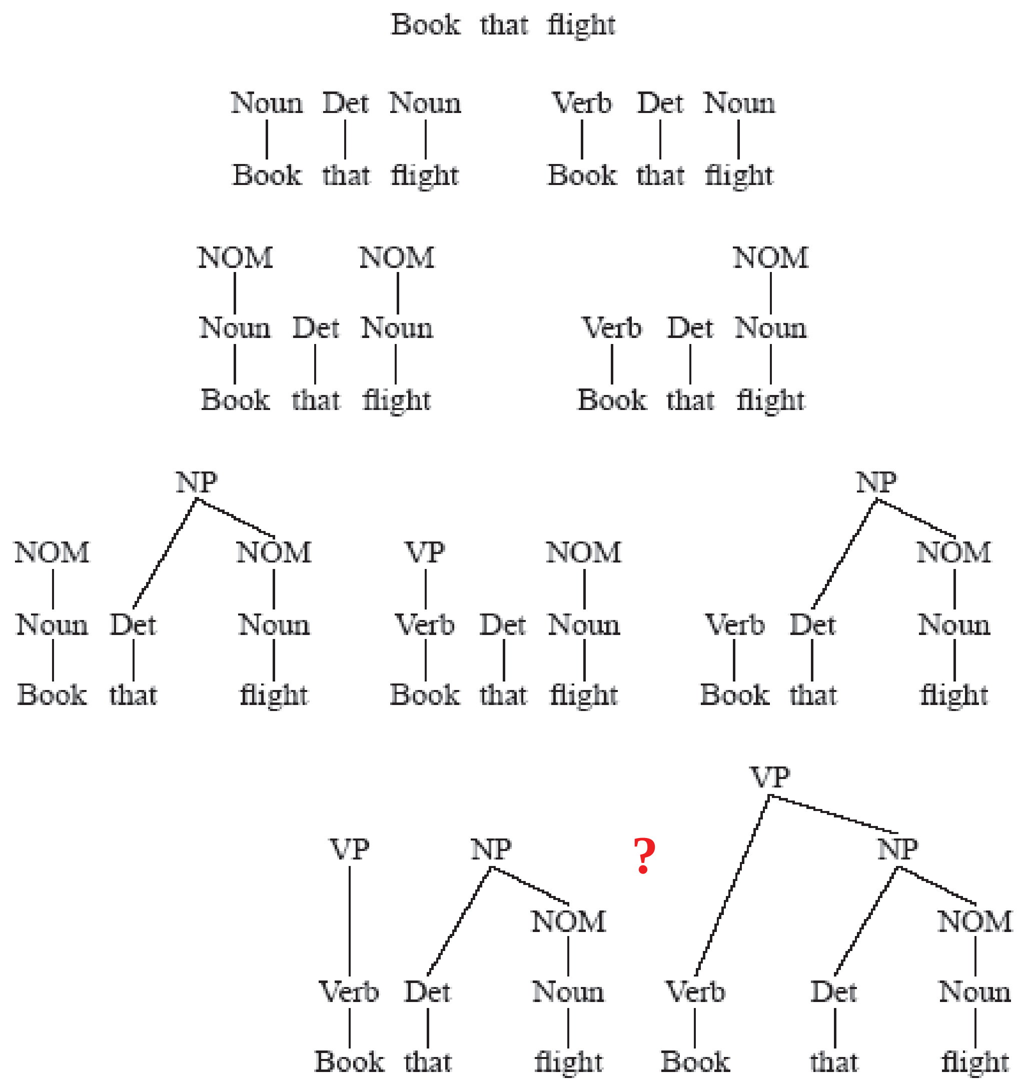


$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal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

Bottom-up Parsing

- Starts with the words of the input and tries to build trees from the words up, by applying rules from the grammar
- Initially *for each input word* build partial trees with the part-of-speech
- For each ply, find the right-hand side of the rule that match the sequence of non-terminals
- Parse is success, if parser succeeds in building a tree rooted in the start symbol S

Bottom-up Parsing



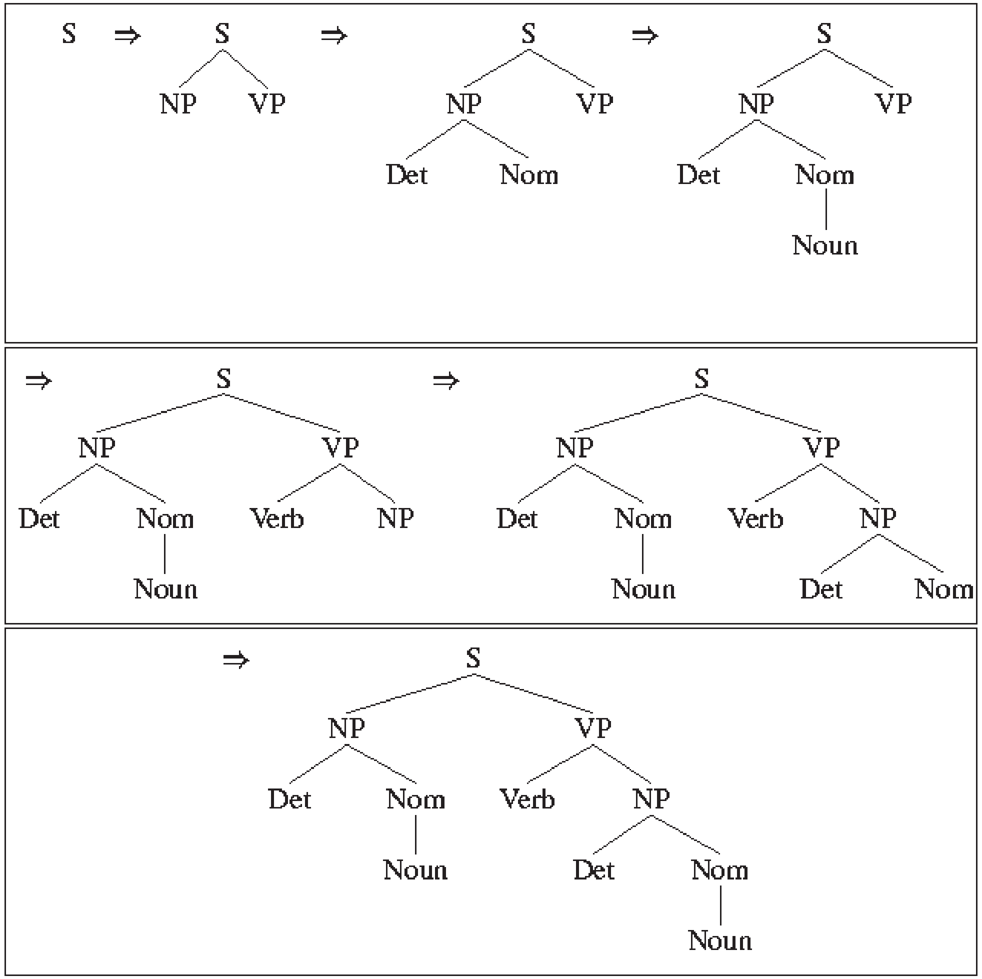
Top-down Vs Bottom-up

- ***Comparisons***
- The top-down strategy never wastes time exploring trees that cannot result in an *S*.
  - Spend considerable effort on *S* trees that are not consistent with the input.
  - Generate trees before ever examining the input.
- The bottom-up strategy never suggest trees that are *not at least locally grounded* in the actual input
  - Trees that have no hope to leading to an *S* are generated with wild abandon.

A Basic Top-Down Parser

- Solution: Incorporate features of both the top-down and bottom-up approaches
- In top-down depth-first approach:
  - Left-most unexpanded leaf node is expanded first
  - Grammar rules are applied according to their textual order

A Basic Top-Down Parser



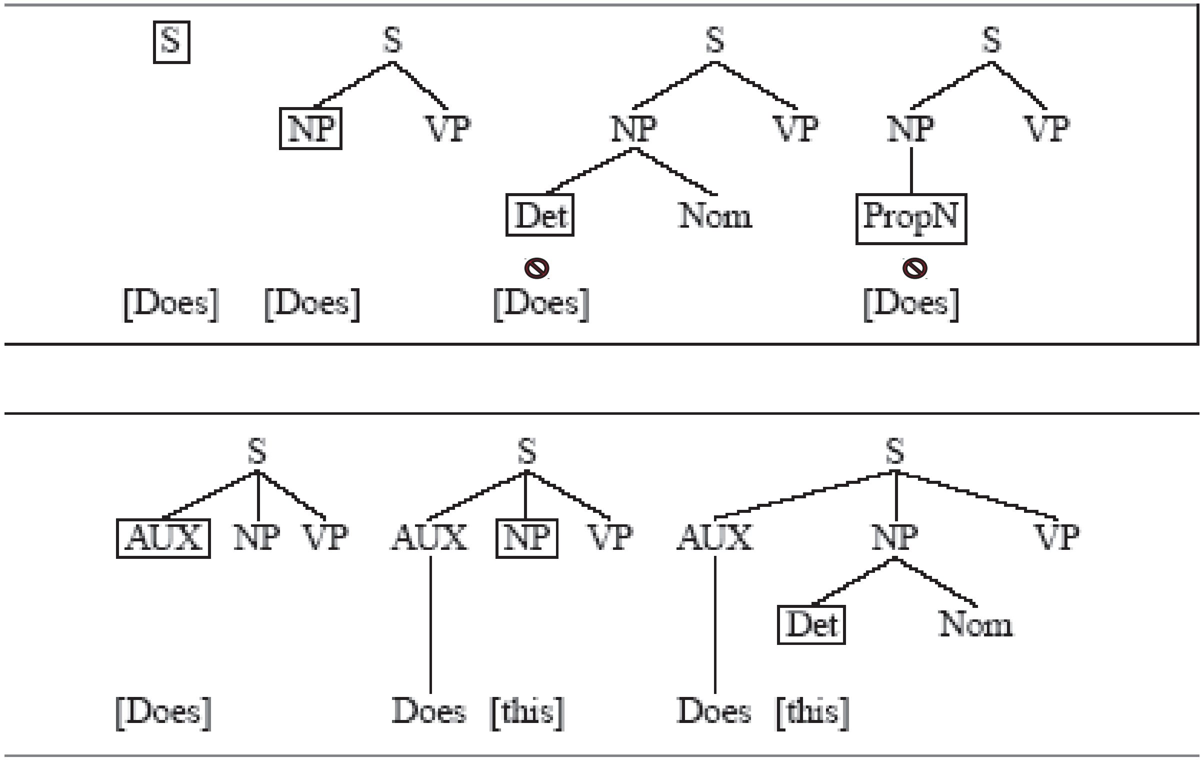
A top-down depth-first derivation

A Basic Top-Down Parser

- Top-down, depth-first, left-to-right approach: expand the left-most unexpanded node in the tree
  - The node currently being expanded is shown in a box
  - The current input word is bracketed

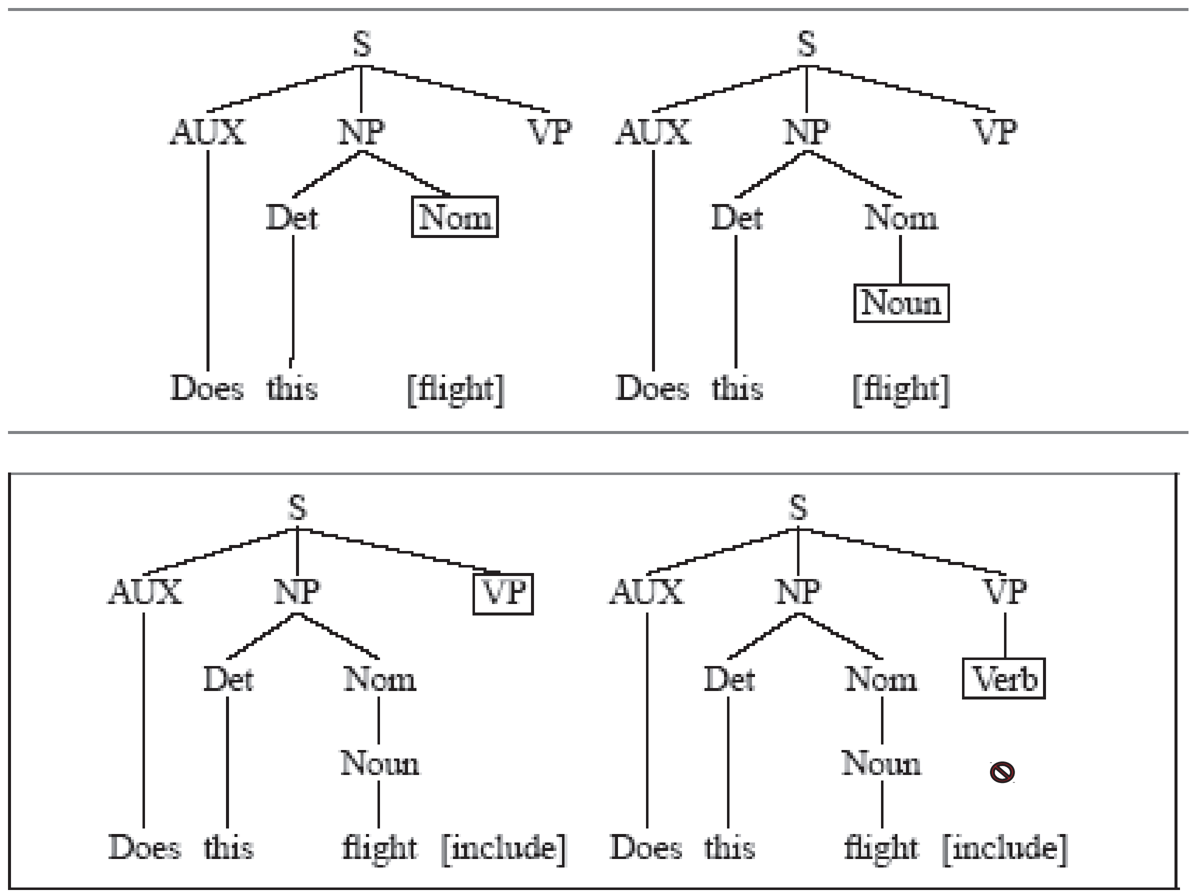
A Basic Top-Down Parser

Does this flight include a meal?



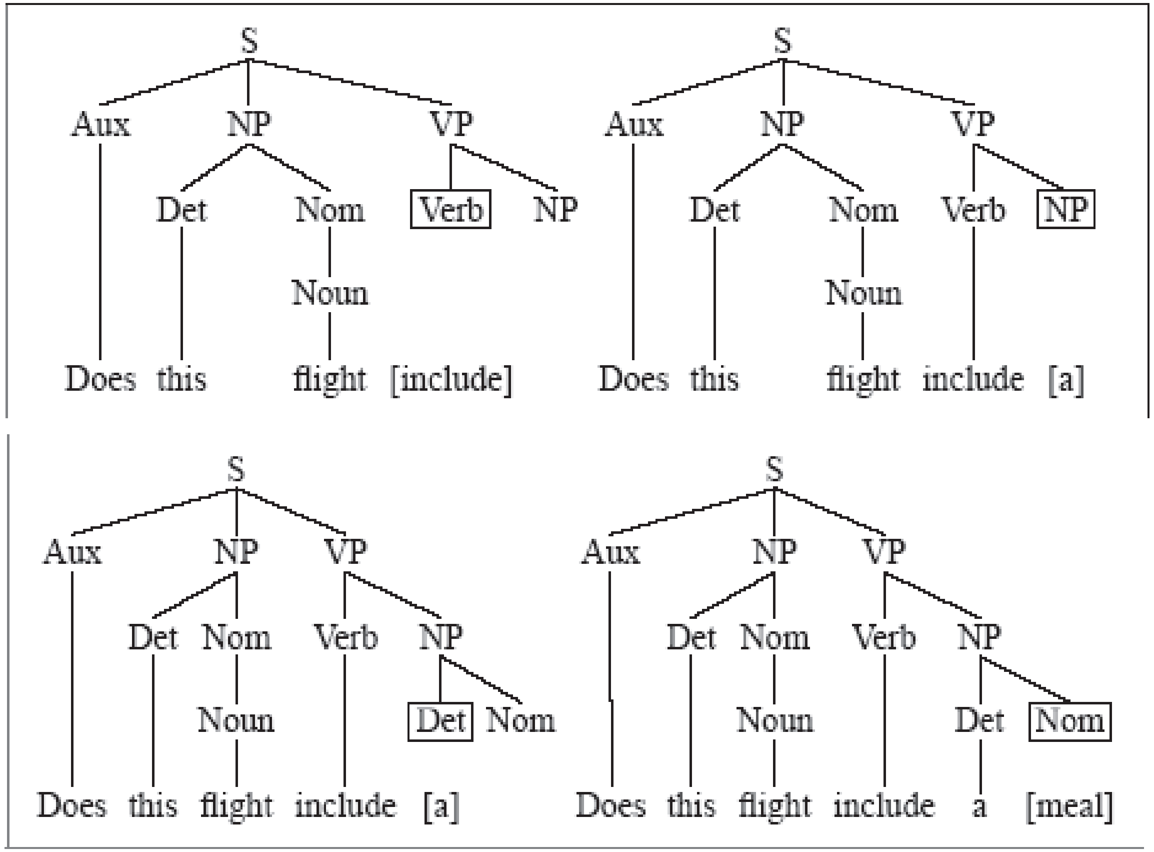
A top-down, depth-first, left-right derivation

A Basic Top-Down Parser



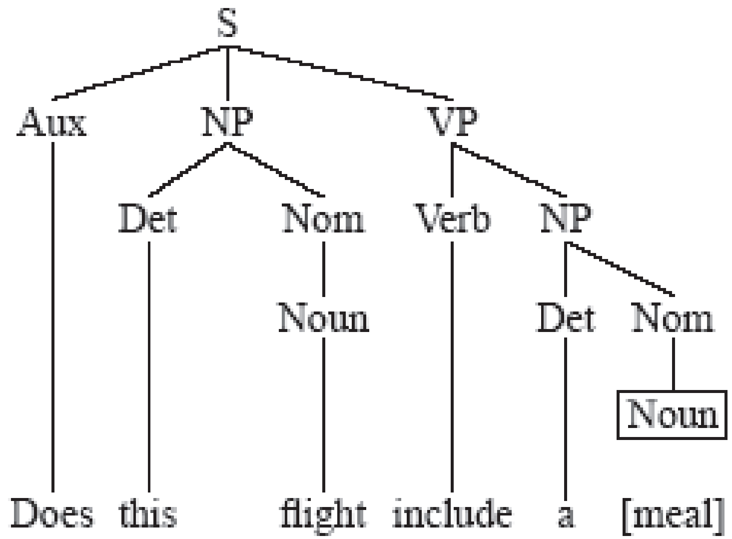
A top-down, depth-first, left-right derivation continued

A Basic Top-Down Parser



A top-down, depth-first, left-right derivation continued

A Basic Top-Down Parser



A top-down, depth-first, left-right derivation continued

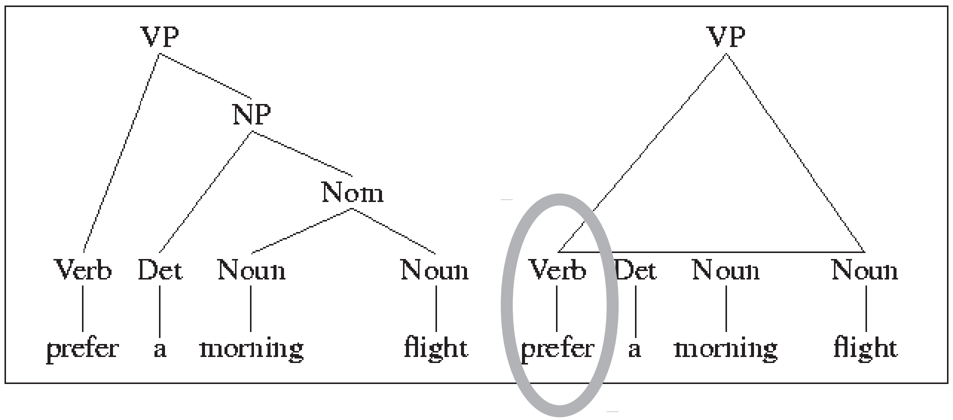
A Basic Top-Down Parser

```
function TOP-DOWN-PARSE(input, grammar) returns a parse tree

 agenda ← (Initial S tree, Beginning of input)
 current-search-state ← POP(agenda)
 loop
   if SUCCESSFUL-PARSE?(current-search-state) then
     return TREE(current-search-state)
   else
     if CAT(NODE-TO-EXPAND(current-search-state)) is a POS then
       if CAT(node-to-expand)
           ⊂
           POS(CURRENT-INPUT(current-search-state)) then
         PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
       else
         return reject
     else
       PUSH(APPLY-RULES(current-search-state, grammar), agenda)
   if agenda is empty then
      return reject
   else
      current-search-state ← NEXT(agenda)
 end
```

A Basic Top-Down Parser

- ***Adding bottom-up filtering***
- The parser should not consider any grammar rule if the current input cannot serve as the *first word along the left edge of some derivation* from this rule
- The first word along the left edge of a derivation is called as the **left-corner** of the tree



A Basic Top-Down Parser

- ***Adding bottom-up filtering***
- Left-corner notion
  - For non-terminals A and B, B is a left-corner of A if : $A \stackrel{*}{\Rightarrow} B\alpha$
- Three rules to expand S :

$$S \rightarrow NP\ VP$$
$$S \rightarrow Aux\ NP\ VP$$
$$S \rightarrow VP$$

Does this flight include a meal ?

- Using the left-corner notion, it is easy to see that only the $S \rightarrow Aux\ NP\ VP$ rule is a viable candidate
- Since the word *Does* can not serve as the left-corner of other two *S*-rules

A Basic Top-Down Parser

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

Category	**Left Corners**
S	Det, Proper-Noun, Aux, Verb
NP	Det, Proper-Noun
Nominal	Noun
VP	Verb

Left-corner table for the above Grammar

Problems: Basic Top-Down Parser

- Problems with the top-down parser
  - Left-recursion
  - Ambiguity
  - Inefficient reparsing of subtrees
- Solution: Earley algorithm

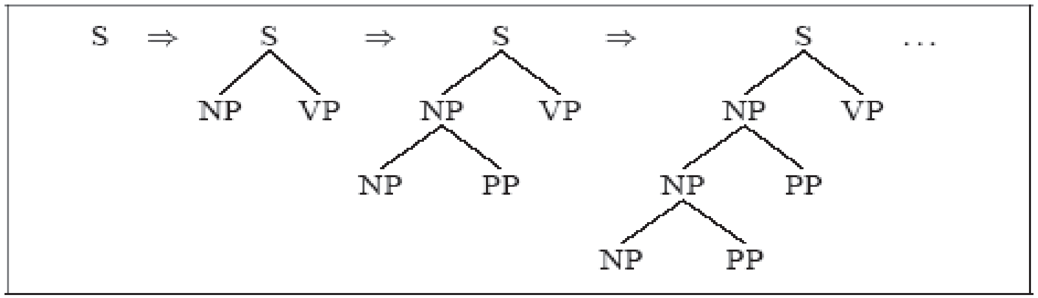
Problem: Left-recursion

- Exploring infinite search space, when **left-recursive grammars** are used
- A grammar is left-recursive if it contains at least one non-terminal *A*, s.t. $A \overset{*}{\Rightarrow} \alpha A \beta$, for some α and β and $\alpha \overset{*}{\Rightarrow} \varepsilon$.

$$NP \rightarrow NP\ PP$$
$$VP \rightarrow VP\ PP$$
$$S \rightarrow S \text{ and } S$$

Left-recursive rules

$NP \rightarrow NP\ PP$



Problem: Left-recursion

- Two reasonable methods for dealing with left-recursion in a backtracking top-down parser:
  - Rewriting the grammar
  - Explicitly managing the depth of the search during parsing
- Rewrite each rule of left-recursion

$$A \rightarrow A\beta \mid \alpha \quad \Rightarrow \quad \begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta A' \mid \varepsilon \end{aligned}$$

left-recursive *Weakly equivalent non-left-recursive*

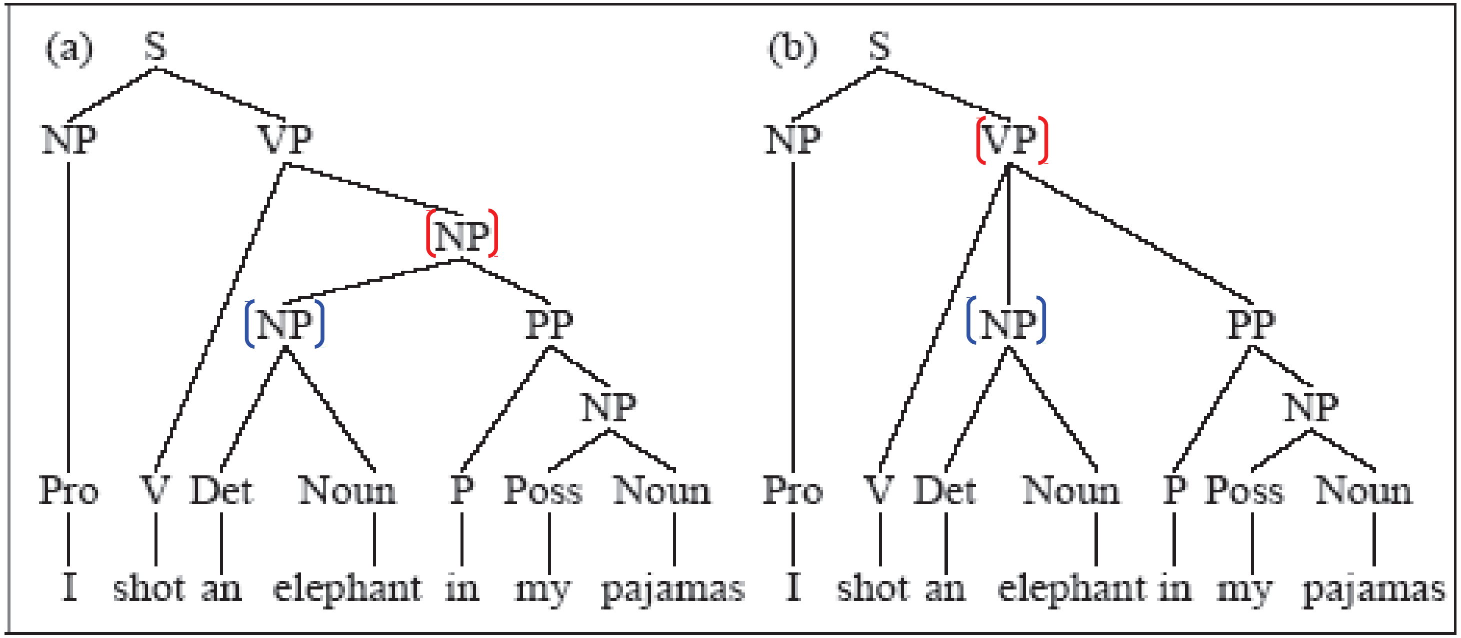
- Rewriting may make semantic interpretation quite difficult

Problem: Ambiguity

- Lexical category ambiguity – word has more than one pos
- Disambiguation – choosing the correct pos for a word
- Structural ambiguity – grammar assigns *more than one possible parse* to a sentence
- Three kinds of structural ambiguity:
  - attachment ambiguity
  - coordination ambiguity
  - noun-phrase bracketing ambiguity

Problem: Ambiguity

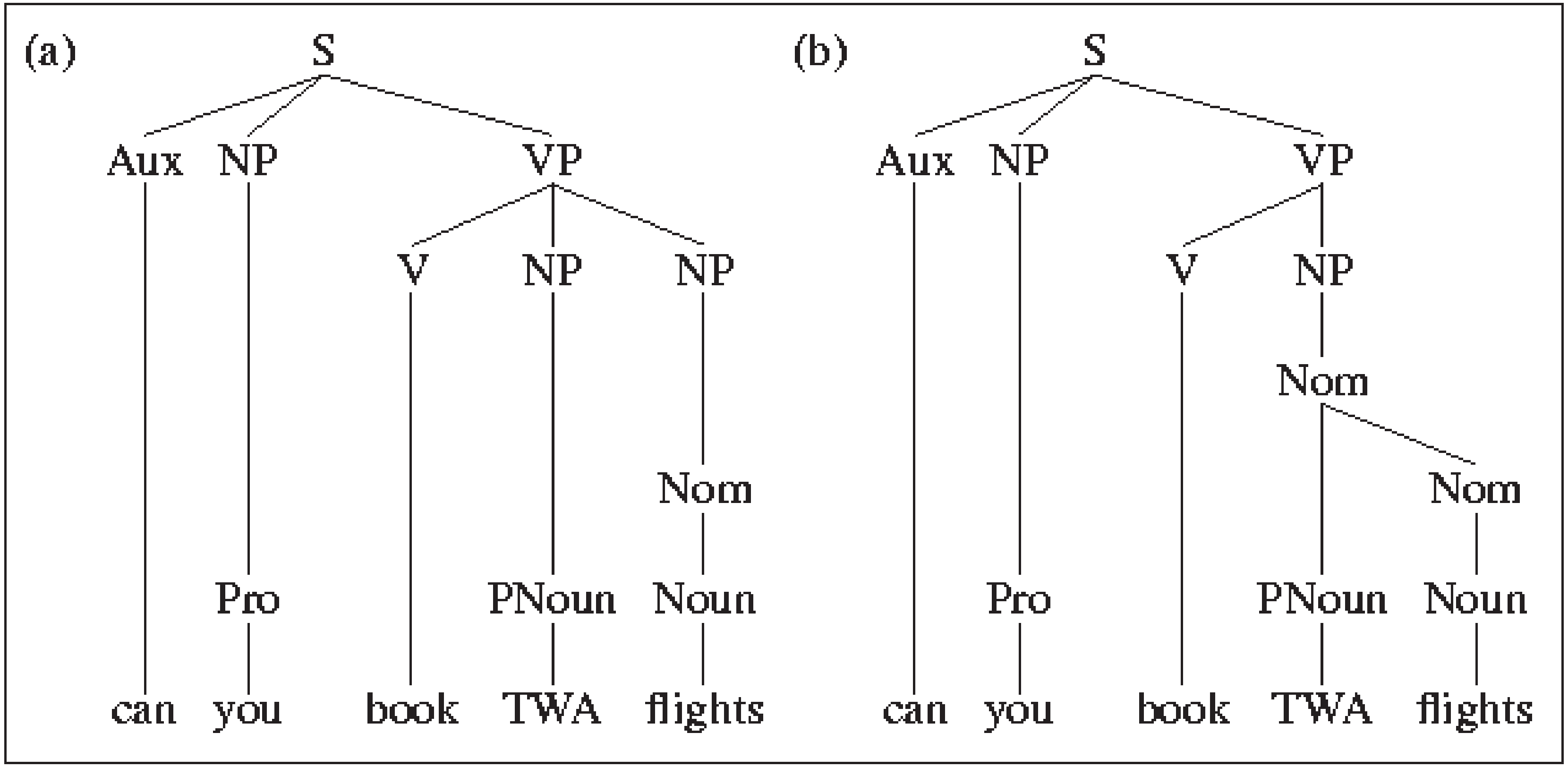
- Attachment ambiguity
  - A particular constituent can be attached to the parse tree at *more than one place*



a) elephant is in the pajamas *b) Captain did the shooting in his pajamas*

Problem: Ambiguity

- The sentence "Can you book TWA flights" is ambiguous:



a) Can you book flights on behalf of TWA

b) Can you book flights run by TWA

Problem: Ambiguity

- *Coordination ambiguity* – different sets of phrases can be conjoined by conjunction like *and*

  old men and women

  [old [men and women]] ⇒ *old men and old women*

  [old men] and [women] ⇒ *only the men who are old*

- Parsing sentence thus requires disambiguation:
  - Choosing the correct parse from a multitude of possible parser
  - Requiring both statistical and semantic knowledge

Problem: Ambiguity

- Parsers which do not incorporate disambiguators may simply return all the possible parse trees for a given input.
- Potentially exponential number of parses that are possible for certain inputs
  - Show me the meal on Flight UA 386 from San Francisco to Denver.
  - The three PP’s at the end of this sentence yield a total of 14 parse trees for this sentence.
  - Solution: use dynamic programming

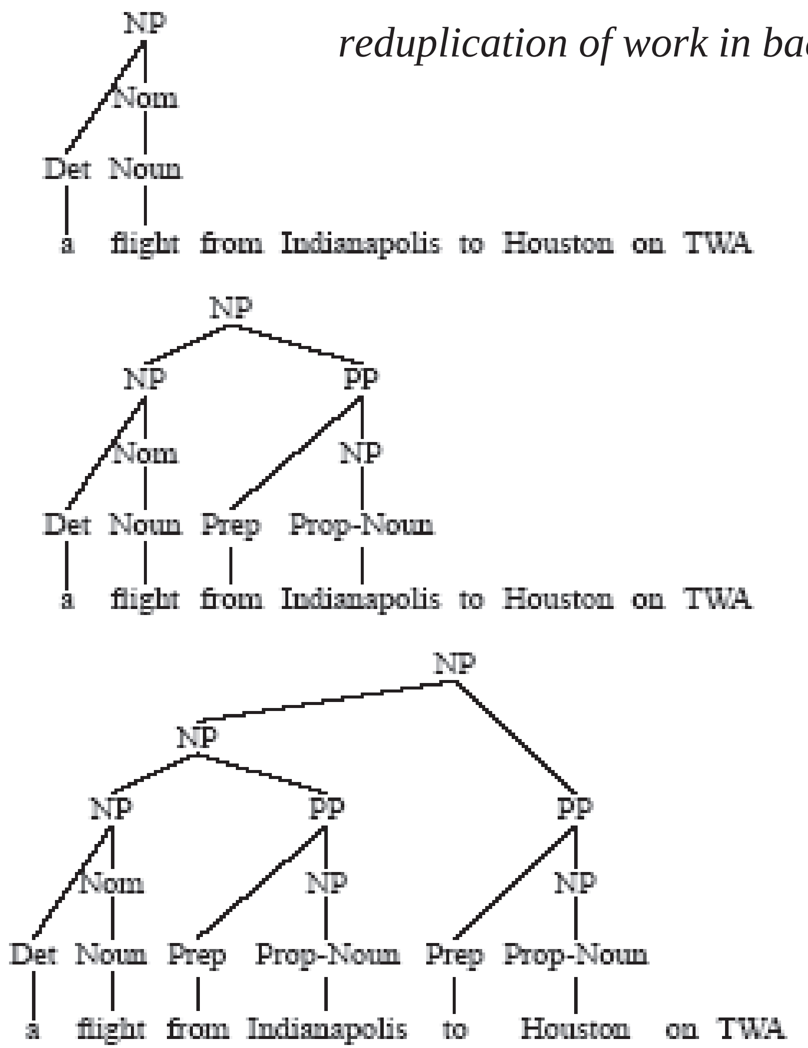
Problem: Repeated Parsing Subtrees

- The parser often builds valid trees for portions of the input, *then discards* them during backtracking, only *to find that it has to rebuild them again.*

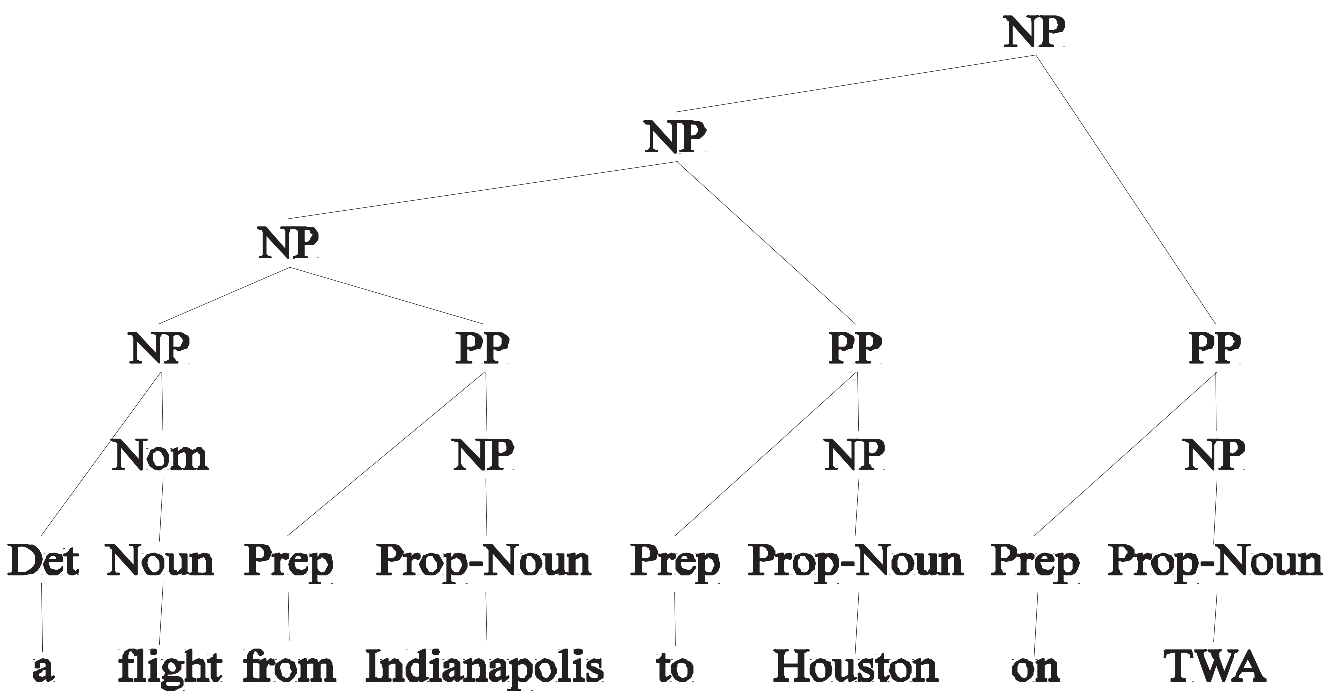
a flight	4
From Indianapolis	3
To Houston	2
On TWA	1
A flight from Indianapolis	3
A flight from Indianapolis to Houston	2
A flight from Indianapolis to Houston on TWA	1

Problem: Repeated Parsing Subtrees

reduplication of work in backtracking approach



Problem: Repeated Parsing Subtrees



reduplication of work in backtracking approach

Dynamic Programming

- Dynamic programming provides framework for solving the three kinds of problems afflicting top-down or bottom-up parsers
- Dynamic programming approaches systematically fill in tables of solutions to sub-problems
- When complete, the tables contain the solution to all the sub-problems needed to solve the problem
- Using tables to store the sub-trees for each constituents in the input solves reparsing and the ambiguity problem
- Also solves left-recursion problem

Dynamic Programming

- Three well-known dynamic parsers:
  - Cocke-Younger-Kasami (CYK) algorithm
  - Graham-Harrison-Ruzzo (GHR) algorithm
  - Earley algorithm

References

- Speech and Language Processing, *Jurafsky and H.Martin*
  [Chapter 10. Parsing with Context-Free Grammars]

Thank You