Introduction to Natural Language Processing

Chapter 10

Outline

Homework 2

Review CFGs

Parsing with CFGs

- top down
- bottom up
- search
- top-down with bottom-up filtering
- problems
- introduction of dynamic programming
- Earley

Review

Linguistic Knowledge: Constituents

- groupings of words into larger units which behave similary and have a particular part of speech as their head
- phrase: NP headed by NOUN, VP ...

Formal Linguistic Representation: CFGs

- but, declarative formalisms only define the legal strings of a language
- parsing algorithms are needed to specify how to recognize these strings and assign structure to them

Analyzing Language in Terms of these Representations: Syntactic Parsing

- identify component parts and how related
- to see if a sentence is grammatical
- to assign a semantic structure

Review

Context Free Grammars

- sets of terminals (either lexical items or parts of speech)
- sets of non-terminals (the constituents of the language)
- sets of rules of the form A $\rightarrow \alpha$, where A is a non-terminal and α is a string of zero or more terminals and non-terminals

Examples

Constituents

- S $\rightarrow$ NP VP
- NP $\rightarrow$ Det Noun
- VP $\rightarrow$ V
- ...

Parts of Speech

- Det $\rightarrow$ a
- Noun $\rightarrow$ flight
- V $\rightarrow$ left
- ...

Parsing

Parsing

- recognizing and assigning structure (previous chapters)

Syntactic parsing

- recognizing a grammatical sentence and assigning a syntactic structure

Parsing with a CFG

- assigning a correct tree (or derivation) to a sentence given some grammar
- Chapter 9: declarative CFG formalism
- Chapter 10: algorithms for using the CFG formalism to compute parse trees for a sentence

CFG Parsing

Parsing with a CFG means assigning a correct tree (or trees) to a sentence given some CFG grammar.

The leaves of the tree cover all and only the input and the tree corresponds to a valid derivation according to the grammar.

Note that *correct* here means that the tree is consistent with the input and the grammar. It doesn't mean that it's the right tree or the proper way to represent English in any more global sense of correct.

Who Cares?

Grammar Checking

Semantic Analysis

Question-Answering / Information Extraction

Speech Recognition Language Models

Parsing as Search

As with finite-state recognition and transduction, parsing can be viewed as a search. The search space corresponds to the space of trees generated by the grammar. The search is guided by the structure of the space and by the input.

- regular expression parsing
  - search space of all possible paths through the FSA
  - search space defined by the FSA
- CFG syntactic parsing
  - search space of all possible parse trees
  - search space defined by the CFG

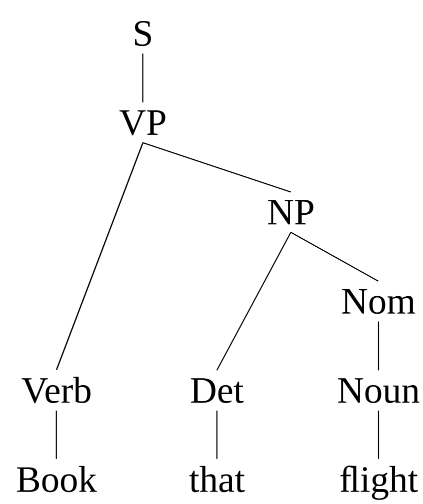
We'll start with the basic methods of parsing, see what's wrong with them, and then move on to a better method.

Parsing

A mini grammar and lexicon

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

The parse of the sentence *Book that flight* according to the mini grammar



Parsing (continued)

What kind of constraints can be used to connect the grammar and the sentence when searching for the parse tree?

- top-down (goal-directed) strategy
  - tree should have one root (grammar constraint)
- bottom up (data-driven) strategy
  - tree should have three leaves (input sentence constraint)

A Note on the Input

For right now, we'll assume the following:

- the input is not tagged
- this input consists of unanalyzed word tokens
- all the words in the input are known
- all the word in the input are available simultaneously (i.e., they're buffered)

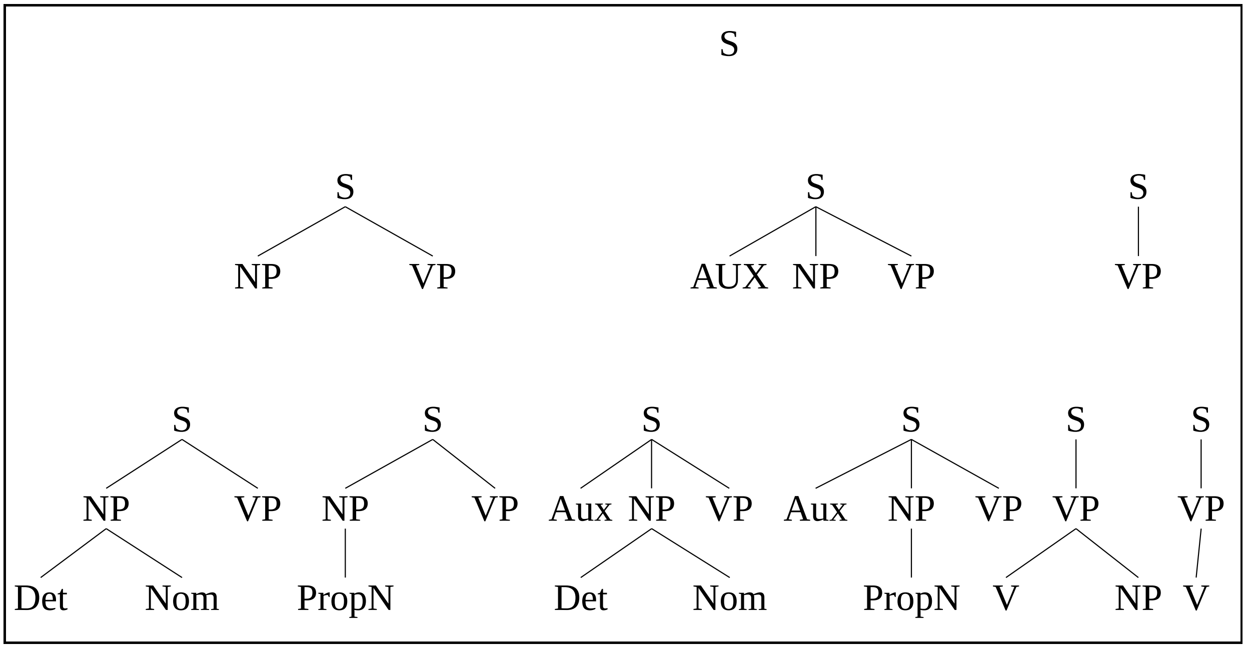
Top-Down Parsing

When the search is primarily goal or expectation-drive (by the structure of the grammar), then we're doing some kind of top-down search.

The primary goal is that we can start out by trying to find a tree rooted as S, since we're trying to parse sentences.

Trees are then built from the root node S to the leaves.

A Partial Top-Down Search Space



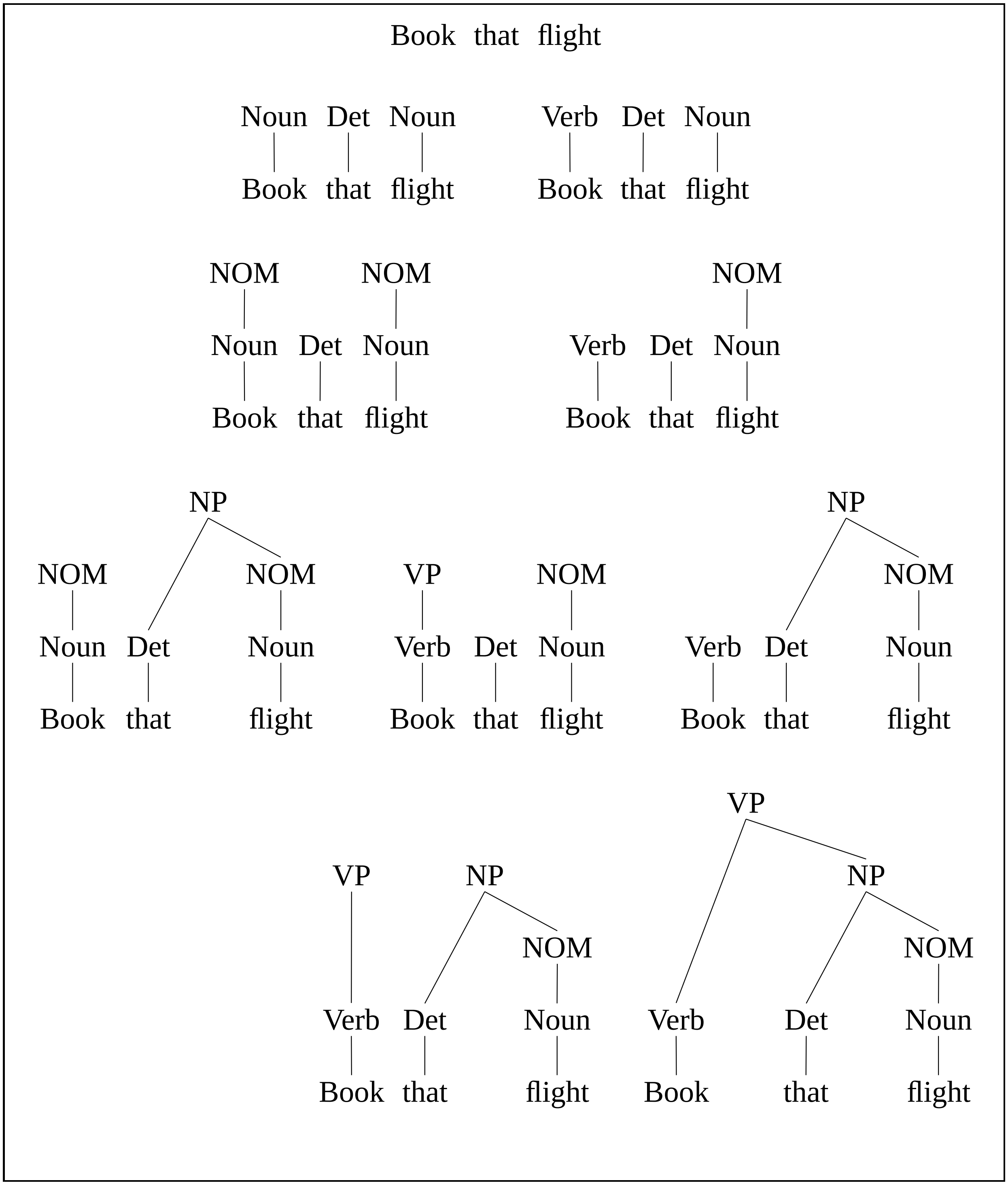
- assuming parallelism

Bottom-Up Parsing

When the search is primarily data-driven (by the input words), then we're doing some kind of bottom-up search.

The primary early consideration here is that the lowest subtrees of the final tree must hook up with the start symbol.

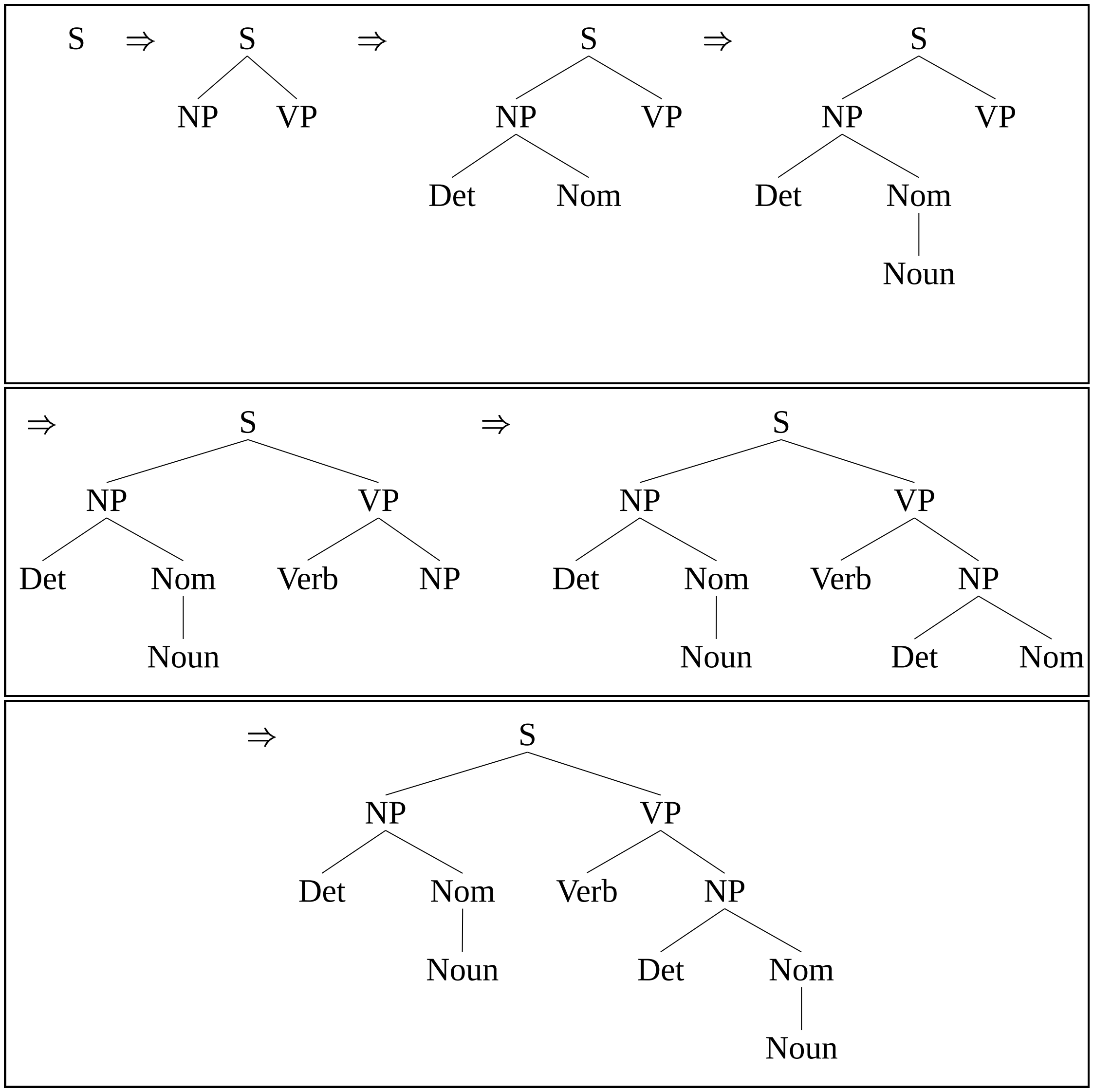
A Partial Bottom-Up Space



Search Control Issues

This discussion left out a few issues. Such as . . .

Search Control Issues (cont.)



- non-parallel strategies (e.g., depth-first)
- which leaf node to expand next (e.g., leftmost)
- which of the applicable grammar rules to try (e.g., order in the grammar)

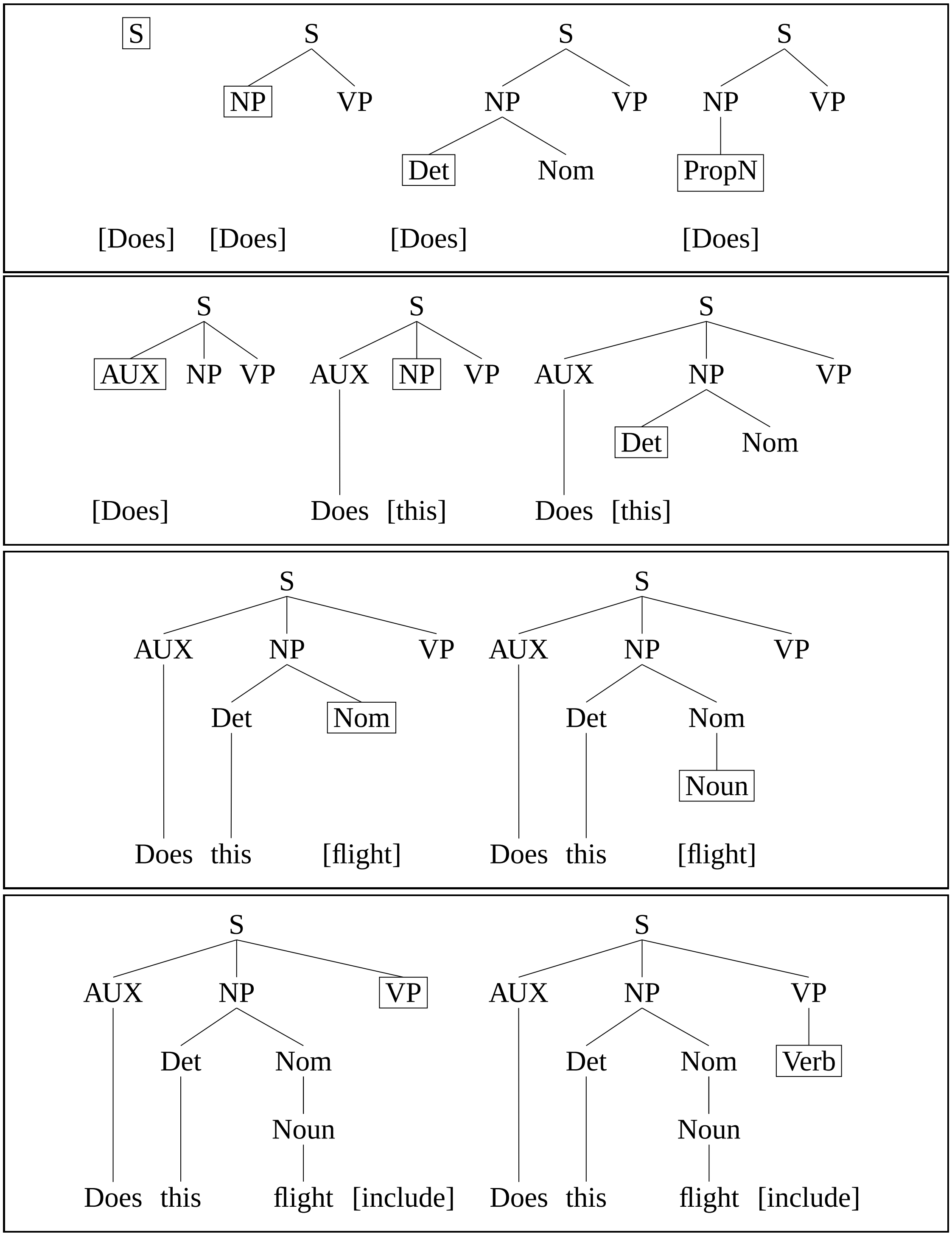
Top-Down Depth-First Left-to-Right

Initialize agenda with "S" tree and pointer to first word and make this current search state (cur)

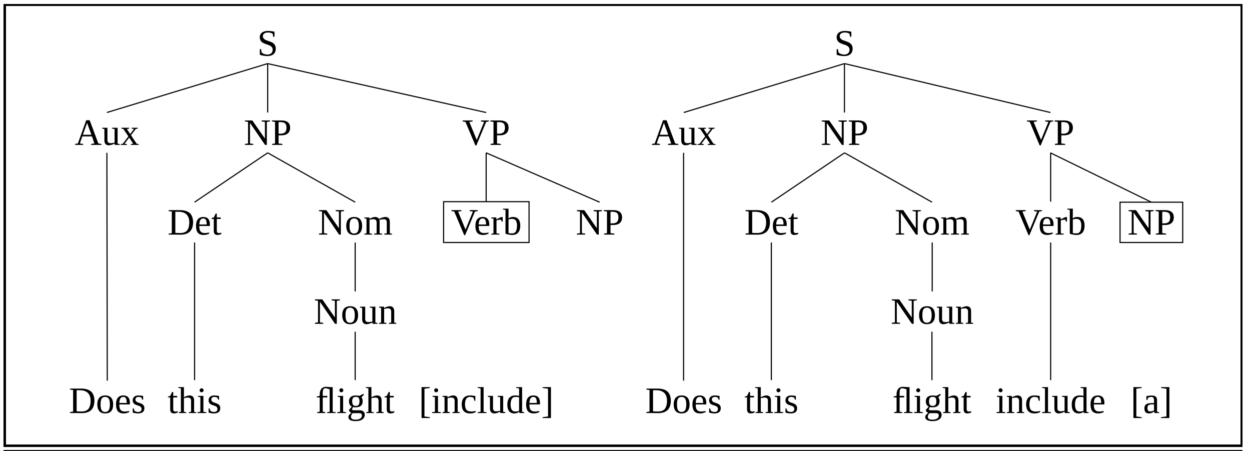
Loop until successful parse or empty agenda

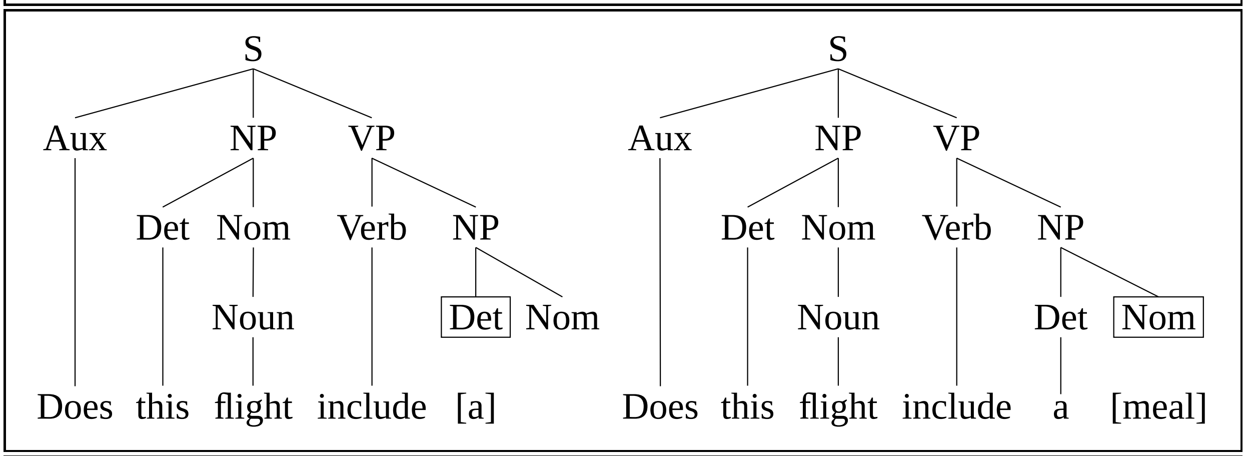
- apply all applicable grammar rules to leftmost unexpanded node of cur
  - if this node is a POS and matches input, push onto agenda
  - otherwise push new trees onto agenda
- pop new cur from agenda

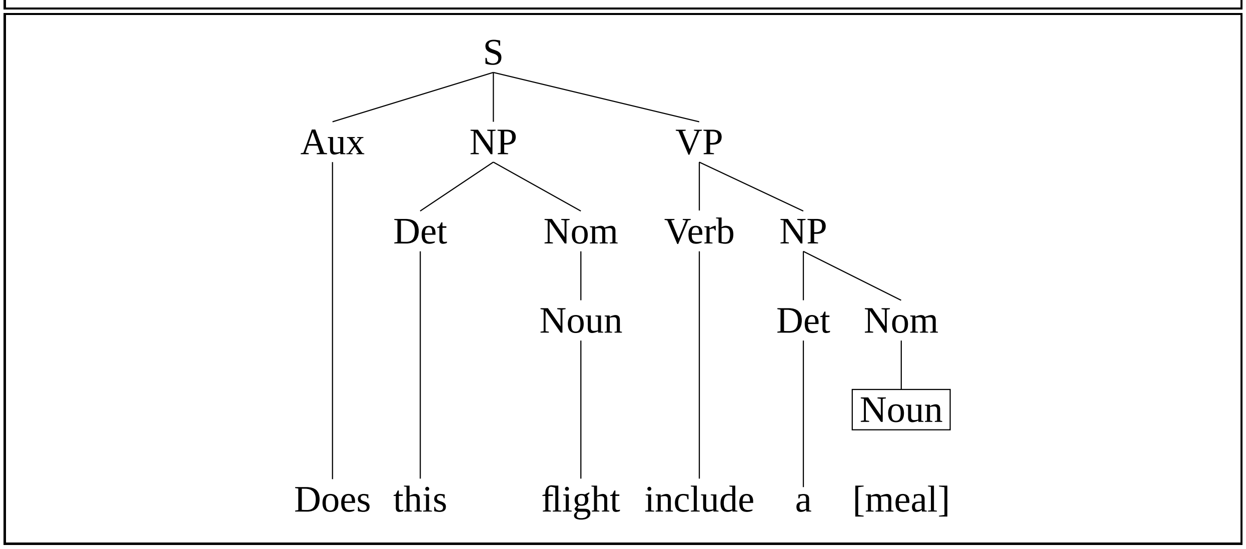
Top-Down Depth-First Left-to-Right



The Search Continued







T-D Depth-First L-R 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
```

ERRATA: Assume that line numbering starts with the assignment of agenda as line 1.

- The “CAT” function on line 7 returns the grammatical category of a node in a parse tree. So this line is checking to see if the grammatical category of the current node to be expanded in this search state is a part-of-speech category.

- The “if” on line 8 should be similar to the one on line 7, ie. CAT(NODE-TO-EXPAND(current-search-state))
- The “subset” symbol on line 9 should be an “in” instead.
- Remove the “else return reject” on lines 12 and 13; it causes a premature termination of the search.
- The call to NEXT on line 19 should be a POP since this algorithm is explicitly depth-first.

Compare with ND-Recognize!

function ND-RECOGNIZE(*tape, machine*) **returns** accept or reject

agenda ← {(Initial state of machine, beginning of tape)}
current-search-state ← NEXT(*agenda*)
loop
  if ACCEPT-STATE?(*current-search-state*) returns true **then**
    return accept
  else
    agenda ← *agenda* ∪ GENERATE-NEW-STATES(*current-search-state*)
  if *agenda* is empty **then**
    return reject
  else
    current-search-state ← NEXT(*agenda*)
end

function GENERATE-NEW-STATES(*current-state*) **returns** a set of search-states

current-node ← the node the current search-state is in
index ← the point on the tape the current search-state is looking at
return a list of search states from transition table as follows:
  (*transition-table[current-node,ε], index*)
  ∪
  (*transition-table[current-node, tape[index]], index + 1*)

function ACCEPT-STATE?(*search-state*) **returns** true or false

current-node ← the node search-state is in
index ← the point on the tape search-state is looking at
if *index* is at the end of the tape **and** *current-node* is an accept state of machine **then**
  return true
else
  return false

Top-Down vs. Bottom-Up

There are advantages and disadvantages to both.

Top-Down

- only searches in the space of reasonable answers
- suggests hypotheses that are not consistent with the data
- has problems with left-recursion (infinite spaces)

Bottom-Up

- only forms hypotheses consistent with the data
- suggests hypotheses that make no sense globally
- also has problems with infinite spaces

A Hybrid Approach

Neither top-down nor bottom-up adequately exploit all the constraints.

There are many way to combine top-down expectations with bottom-up data to get a more efficient search.

The most popular methods use one method as the basic search control strategy to generate trees.

They then use constraints from the other method to dynamically filter out "bad" structures.

We'll explore top-down parsing with bottom-up filtering.

Adding Bottom-Up Filtering

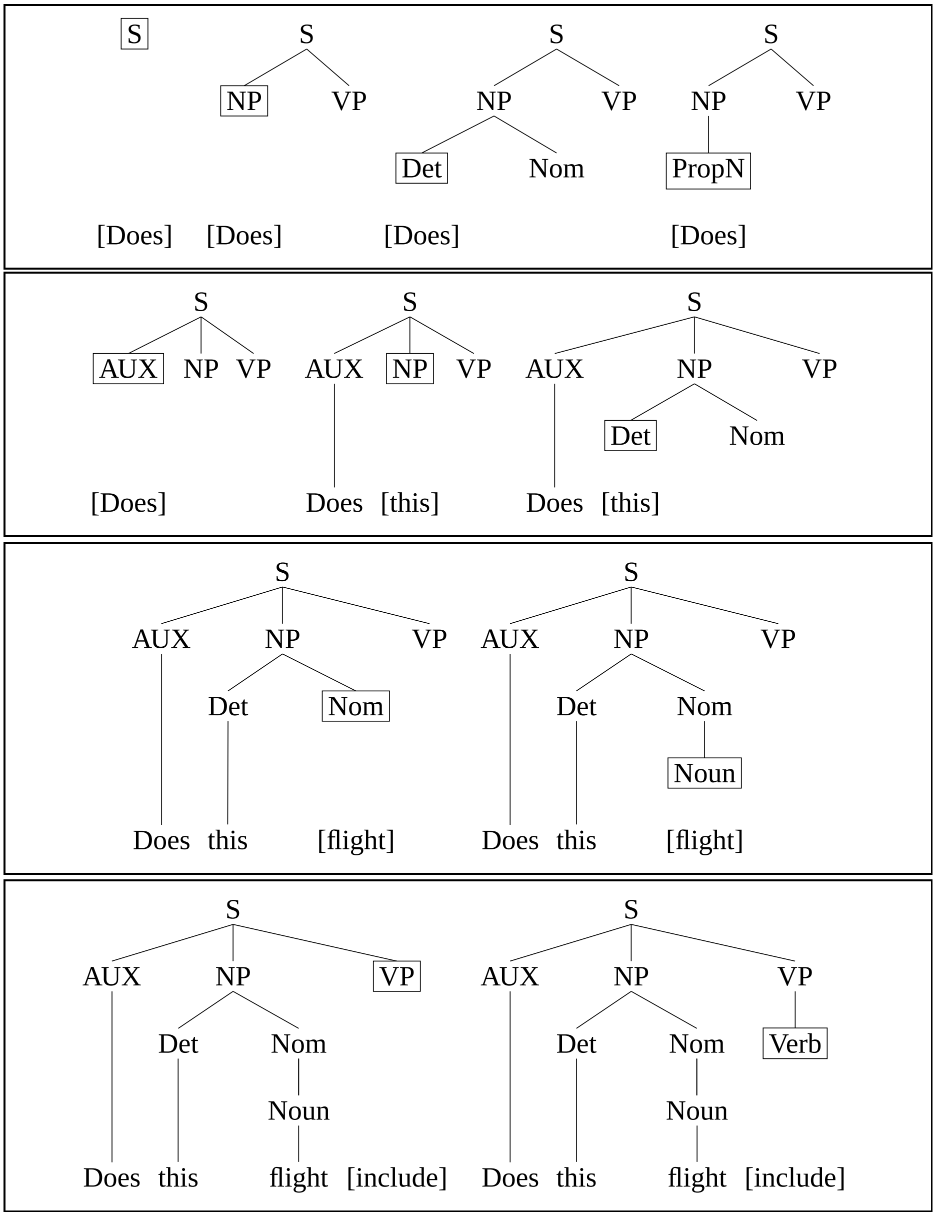
Top-Down, Depth First, L2R parsing

- expands non-terminals along the tree's left edge down to leftmost leaf
- moves on to expand down to next leftmost leaf
- when succesful, current input word is the first word in the derivation

So, lookahed to left corner of the tree

- B is a left corner of A if A $=^*=>$ B
- build table with left-corners of all non-terminals in grammar
- consult table before applying rule

Adding Bottom-Up Filtering



Left-Corner Observation: in a successful parse, the current input word is first in the derivation of the unexpanded node

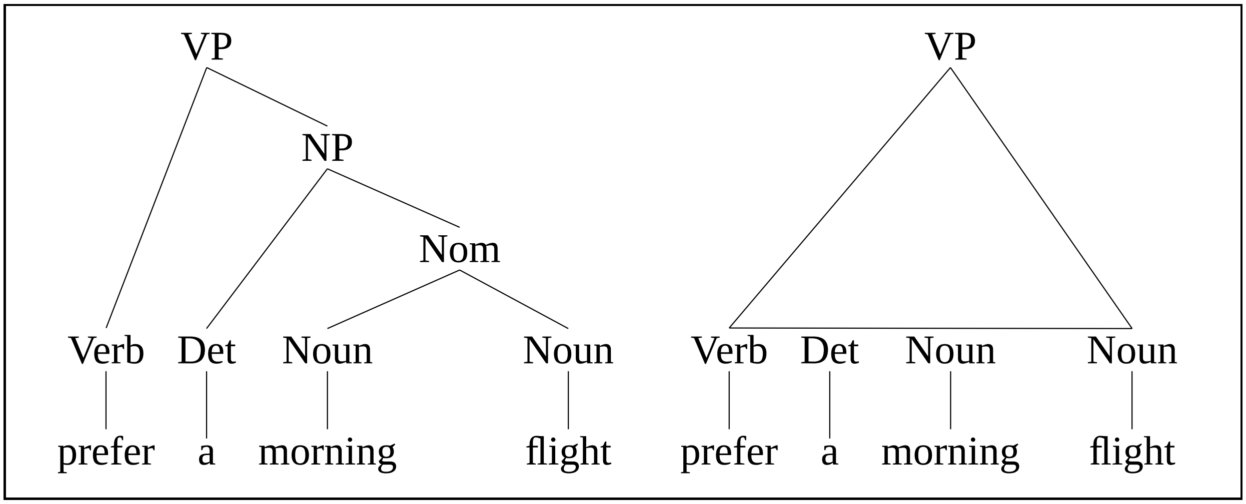
Bottom-Up Left-Corner Filtering

Don't consider any expansion where the current input cannot serve as the left-corner of that expansion.

<u>Left-Corner</u> of a tree:

- the first word along the left edge of a derivation
- B is a left corner of A if A derives Bα

Left-Corners



Verb and *prefer* are both left-corners of VP

Example

Consider a top-down parser parsing the following input:

- *Does this flight include a meal?*

Recall that the grammar contains the following rules:

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

Using the left corner filter:

- only the second S rule is viable
- parser with filtering avoids previous backtracking

Knowledge for Left-Corner Filtering

Given our mini grammar...

S → *NP VP*	*Det* → *that* | *this* | *a*
S → *Aux NP VP*	*Noun* → *book* | *flight* | *meal* | *money*
S → *VP*	*Verb* → *book* | *include* | *prefer*
NP → *Det Nominal*	*Aux* → *does*
Nominal → *Noun*	
Nominal → *Noun Nominal*	*Prep* → *from* | *to* | *on*
NP → *Proper-Noun*	*Proper-Noun* → *Houston* | *TWA*
VP → *Verb*	
VP → *Verb NP*	*Nominal* → *Nominal PP*

Here are the left corners for each non-terminal:

- S: Det, Proper-Noun, Aux, Verb
- NP: Det, Proper-Noun
- Nominal: Noun
- VP: Verb

Summing Up

Parsing is a search problem which may be implemented with many search strategies

- top-down or bottom up each have problems
- combining them solves some

Today's Outline

Review Parsing

- top-down
- bottom-up
- top-down with bottom-up filtering
- problems

Dynamic Programming/Earley

Review

Parsing with a CFG is the task of assigning a correct tree (or derivation) to a string given some grammar.

A correct tree is consistent with the grammar and the leaves of the tree cover all and only the words in the input.

There may be a huge number of correct trees for any given input.

Top-Down Parsers and Bottom-Up Parsers both generate too many useless trees.

Top-Down with Bottom-up Lookahead (left-corner table) is more efficient.

- pre-compute all POS that can serve as the leftmost POS in the derivations of each non-terminal category

Remaining Problems

Even after filtering...

- left recursive grammars
- ambiguity
- reparsing inefficiencies due to backtracking

Left Recursion

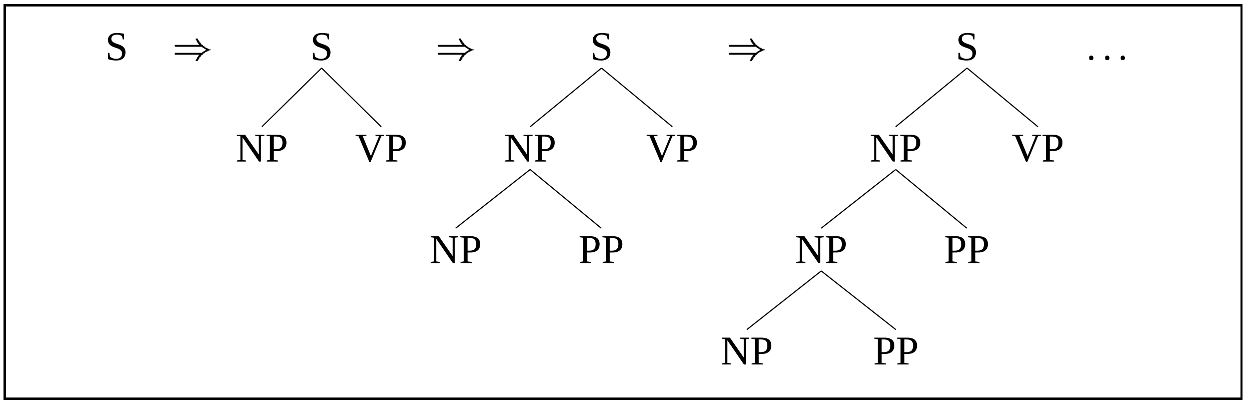
In top-down, depth-first, left-to-right parsers, a left recursive grammar can cause the search to never return due to an infinite expansion of a tree.

Example left-recursive rules:

- A derives A B (immediate)
  - NP $\rightarrow$ NP PP
  - VP $\rightarrow$ VP PP
  - S $\rightarrow$ S and S
- A derives α A B and α derives epsilon (indirect)
  - NP $\rightarrow$ Det Nom
  - Det $\rightarrow$ NP ' s

Left Recursion (continued)

NP $\rightarrow$ NP PP



Some non-solutions

- don't use recursive rules
- don't use top-down parsing
- rule ordering (doesn't work)

Poor Solutions to Left Recursion

Automatically detect and rewrite left-recursion

- might not get a correct or useful parse tree

Limit the depth of recursion in parsing to some analytically or empirically set limit

- limit is arbitrary

Rule Ordering

Basic idea... non-recursive rules first

Bad

- NP $\rightarrow$ NP PP
- NP $\rightarrow$ Det Nominal

Better (but still no good)

- NP $\rightarrow$ Det Nominal
- NP $\rightarrow$ NP PP

Grammar Rewriting

It is always the case that a left-recursive grammar can be re-written into a weakly equivalent non-left-recursive one.

Can be done...

- by hand
- automatically (see the book)

May make rules unnatural

Depth Bound

Set an arbitrary bound.

Set an analytically derived bound.

Run tests and drive a reasonable bound empirically.

Ambiguity

Chapter 8 discussed POS Ambiguity.

Here we are concerned with Structural Ambiguity.

Given a grammar, Global Ambiguity potentially leads to multiple parses for the same input (if we force it to).

- *I saw a woman with a telescope.*

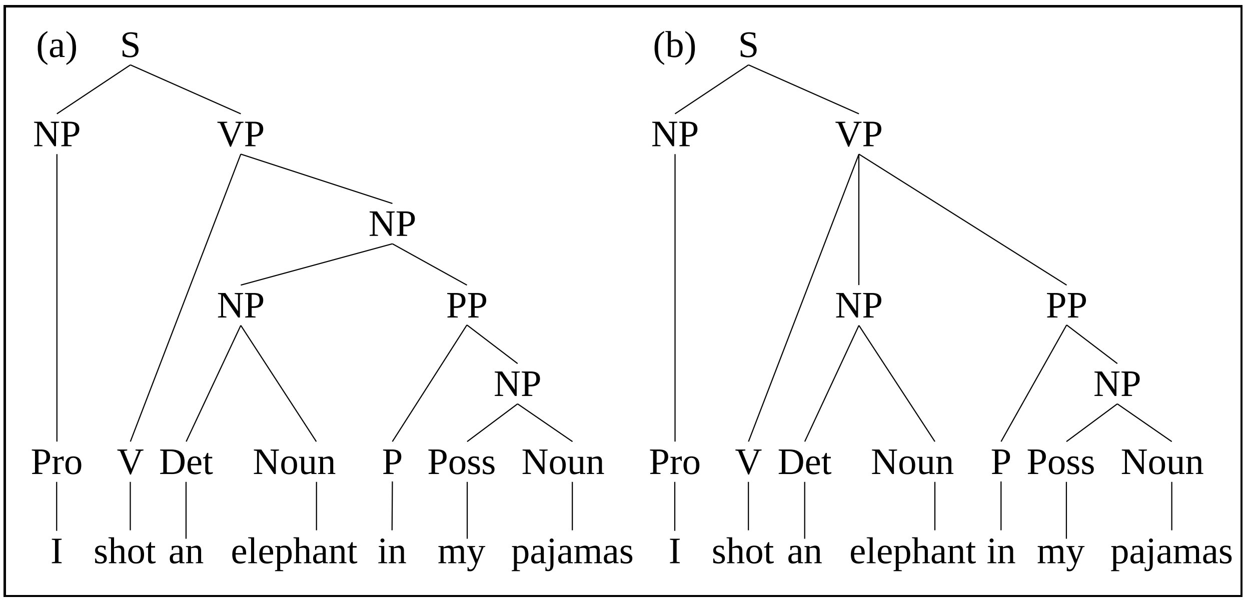
Local Ambiguity, in contrast, leads to hypotheses that are locally reasonable but eventually lead nowhere and result in inefficient backtracking. Filtering helps a little.

- *Book that flight.*

Common Structural Ambiguities

Attachment ambiguity

- a constituent can be attached to the parse tree at more than one place
- PP attachment ambiguity



- gerundive VP attachment ambiguity
  - *We saw the Eiffel Tower flying to Paris*

Coordination ambiguity

- *old (men and women)* vs. *(old men) and women*

NP bracketing (*Spanish language teachers*)

Why is Ambiguity Problematic?

There are potentially an exponential number of parses for a sentence, so returning all structurally valid parses isn't always a good idea.

Some solutions:

- exploit regularities in the search space to derive common subparts only once (e.g., use dynamic programming to increase efficiency)
- heuristic search strategies
- return all possible parses and disambiguate using "other methods"
  - rely on semantics
  - rely on probabilites
  - both

Invariants

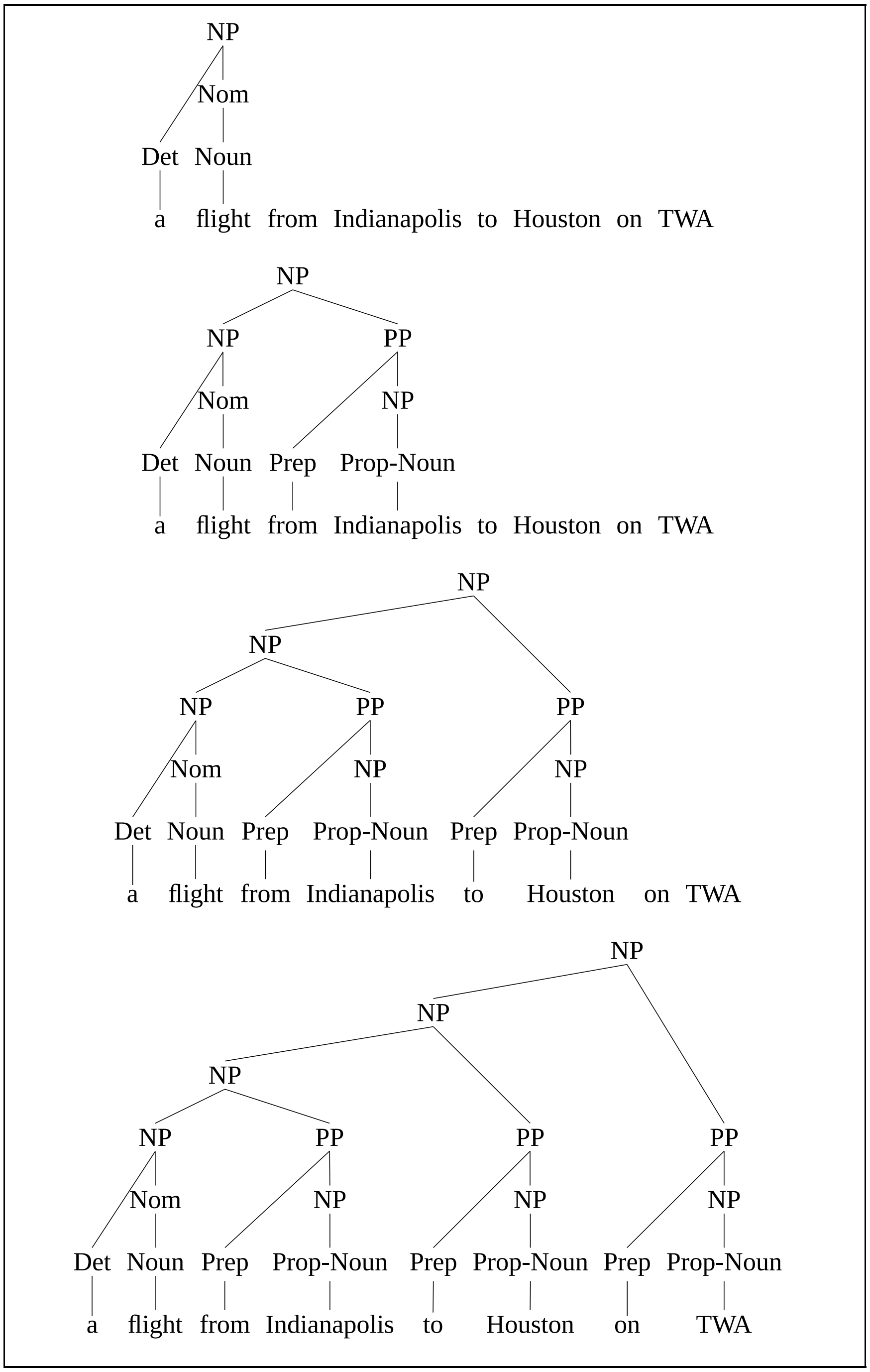
Despite all the ambiguity and backtracking there are invariants to be taken advantage of.

Consider parsing the following NP with the following rules:

- *a flight from Indianapolis to Houston on TWA*
- NP $\rightarrow$ Det Nominal
- NP $\rightarrow$ NP PP
- NP $\rightarrow$ ProperNoun

What happens with a top-down parser?

Invariants (continued)



Reuse via Dynamic Programming

Our current algorithm builds valid trees, discards them during backtracking, then rebuilds them.

- the subtree for *a flight* was derived 4 times!

Dynamic programming is one answer to problems that have sub-problems that get solved again and again.

We want an algorithm that fills a table with solutions to sub-problems that:

- does not do repeated work
- does top-down search with bottom-up filtering
- solves the left-recursion problem
- solves an exponential problem in $O(N^3)$ time

Reuse will be our solution to inefficiency

Dynamic Programming

Systematically fill in tables of solutions to subproblems.

When complete, the tables contain the solutions to all of the subproblems needed to solve the whole problem.

For parsing, the tables store subtrees for constituents.

Solves reparsing inefficiencies, as subtrees are not reparsed but looked up.

Solves ambiguity explosions, as the table can *implicitly* store all parses.

Each subtree is represented only once and shared by all that need it.

Dynamic Programming and Parsing

We'll use the Earley algorithm, which fills a table (*the chart*) in a single pass over the input.

The table will be size N+1 where N is the number of words in the input.

It is fruitful to think of the table entries as sitting between the words in the input string keeping track of states of the parse at these positions.

For each word position in the sentence, the chart contains a list of states representing the partial parse trees generated so far.

So, chart[0] contains all partial parse trees generated at the beginning of the sentence.

The table entries will represent three distinct kinds of things:

- completed constituents
- in-progress constituents
- predicted constituents

Predicted constituents will be advanced as completed constituents are found. This results in more predictions.

States

We'll call these table entries states and represent the progress made in recognizing the state's rule with what are called Dotted Rules.

The three kinds of states correspond to different dot locations.

- VP $\rightarrow$ V NP $\cdot$
- NP $\rightarrow$ Det $\cdot$ Nominal
- S $\rightarrow$ $\cdot$ VP

States (continued)

Given rules like these we need to keep track of a couple of things: where the represented constituent is in the input, and what its parts are.

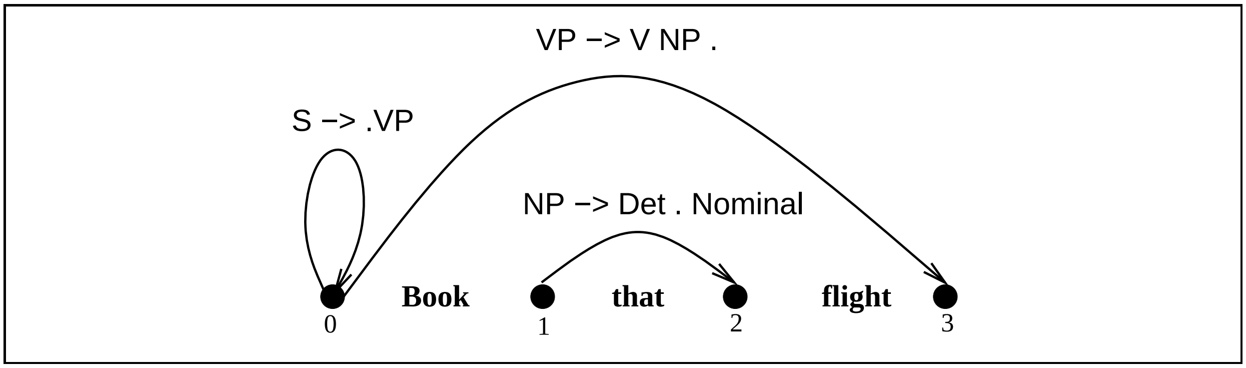
- A $\rightarrow \alpha$, [x, y]
- x indicates where the state begins
- y indicates where the dot lies

Example

Example states in parsing *Book that flight.*

- S $\rightarrow$ $\cdot$ VP, [0,0]
  - the first 0 indicates that the constituent begins at the start of the input
  - the second 0 indicates that the dot is here as well, and thus indicates a top-down prediction
- NP $\rightarrow$ Det $\cdot$ Nominal, [1,2]
  - the NP begins at position 1
  - the dot is at position 2
  - Det has thus been successfully parsed
  - Nominal is thus predicted next
- VP $\rightarrow$ V NP $\cdot$, [0,3]
  - VP is completed
  - no further predictions from this rule
  - a successful VP parse of the entire input

Graphical Representation of States



Success

The final answer is found by looking at the last entry of the table. In particular, if we find the following kind of state there we've succeeded:

S $\rightarrow \alpha\cdot$, [0,N]

But note that the chart will also contain a record of all possible parses of the input string given the grammar, not just the succesful one(s).

Parsing

So... parsing is sweeping through the table - *without* backtracking - creating the three kinds of states.

New predicted states are based on existing table entries (predicted, or in-progress) that predict a certain constituent at that spot.

New in-progress states are created by updating older states to reflect the fact that previously expected completed constituents have been located.

New complete states are created when the dot in an in-progress state moves to the end.

Note that states are never removed.

More Specifically

1. Predict all the states you can.

2. Read an input.

See what predictions you can match. Extend matched states, add new predictions. Go to next state (goto 2).

3. At the end, see if state [N+1] contains a complete S.

Earley Algorithm

The Earley algorithm has three main functions that process the states in the chart and thus do all the work.

Predictor: Adds predictions into the chart.

Completer: Moves the dot to the right when new constituents are found.

Scanner: Reads the input word and enter states representing those words into the chart.

The Three Operators

Operator I/O

- input: single state
- output: new states that are added to the chart if not already present
- note that states are never removed

Predictor and Completer

- new states are added to the chart entry being processed

Scanner

- output: new states are added to the next chart entry

Predictor

Intuition: new states represent top-down expectations.

Applied when non part-of-speech non-terminals are to the right of a dot.

- S $\rightarrow \cdot$ VP [0,0]

Generates one new state for each alternative expansion of the non-terminal in the grammar.

- VP $\rightarrow \cdot$ V [0,0]
- VP $\rightarrow \cdot$ V NP [0,0]

Same chart entry as generating state.

Completer

Intuition: parser has discovered a constituent, so must find and advance states that were looking for this grammatical category at this position in the input.

Applied when dot has reached right end of rule.

- NP $\rightarrow$ Det Nom $\cdot$ [1,3]

Find all states with dot at 1 and expecting an NP

- VP $\rightarrow$ V $\cdot$ NP [0,1]

Adds new (completed) state(s) to current chart

- VP $\rightarrow$ V NP $\cdot$ [0,3]

Thus, new states are generated by copying old state and advancing dot to the expected category.

Same chart entry as generating state.

Scanner

New states for predicted part of speech.

Applicable when part of speech is to the right of a dot.

- VP $\rightarrow \cdot$ V NP [0,0] 'Book ...'

Looks at current word in input.

If match, adds state(s) to next chart entry.

- VP $\rightarrow$ V $\cdot$ NP [0,1]

NOTE: Early parser uses top-down predictions to help disambiguate part of speech ambiguities. Only those parts of speech of a word that are predicted by some state will find their way into the chart.

Earley Algorithm

```
function EARLEY-PARSE(words, grammar) returns chart

  ENQUEUE((γ → • S, [0,0]), chart[0])
  for i ← from 0 to LENGTH(words) do
   for each state in chart[i] do
     if INCOMPLETE?(state) and
              NEXT-CAT(state) is not a part of speech then
        PREDICTOR(state)
     elseif INCOMPLETE?(state) and
              NEXT-CAT(state) is a part of speech then
         SCANNER(state)
     else
        COMPLETER(state)
   end
  end
  return(chart)

 procedure PREDICTOR((A → α • B β, [i, j]))
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
         ENQUEUE((B → • γ, [j, j]), chart[j])
    end

 procedure SCANNER((A → α • B β, [i, j]))
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
       ENQUEUE((B → word[j], [j, j+1]), chart[j+1])

 procedure COMPLETER((B → γ •, [j,k]))
    for each (A → α • B β, [i, j]) in chart[j] do
         ENQUEUE((A → α B • β, [i,k]), chart[k])
    end

 procedure ENQUEUE(state, chart-entry)
    if state is not already in chart-entry then
        PUSH(state, chart-entry)
    end
```

Our Grammar and Lexicon Again

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

Example: *Book that flight*

Chart[0]		
$\gamma \rightarrow \bullet S$	[0,0]	Dummy start state
$S \rightarrow \bullet NP\ VP$	[0,0]	Predictor
$NP \rightarrow \bullet Det\ NOMINAL$	[0,0]	Predictor
$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	Predictor
$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	Predictor
$S \rightarrow \bullet VP$	[0,0]	Predictor
$VP \rightarrow \bullet Verb$	[0,0]	Predictor
$VP \rightarrow \bullet Verb\ NP$	[0,0]	Predictor

Chart[1]		
$Verb \rightarrow book \bullet$	[0,1]	Scanner
$VP \rightarrow Verb \bullet$	[0,1]	Completer
$S \rightarrow VP \bullet$	[0,1]	Completer
$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
$NP \rightarrow \bullet Det\ NOMINAL$	[1,1]	Predictor
$NP \rightarrow \bullet Proper\text{-}Noun$	[1,1]	Predictor

Chart[2]		
$Det \rightarrow that \bullet$	[1,2]	Scanner
$NP \rightarrow Det \bullet NOMINAL$	[1,2]	Completer
$NOMINAL \rightarrow \bullet Noun$	[2,2]	Predictor
$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[2,2]	Predictor

Chart[3]		
$Noun \rightarrow flight \bullet$	[2,3]	Scanner
$NOMINAL \rightarrow Noun \bullet$	[2,3]	Completer
$NOMINAL \rightarrow Noun \bullet NOMINAL$	[2,3]	Completer
$NP \rightarrow Det\ NOMINAL \bullet$	[1,3]	Completer
$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
$S \rightarrow VP \bullet$	[0,3]	Completer
$NOMINAL \rightarrow \bullet Noun$	[3,3]	Predictor
$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[3,3]	Predictor

Errata

The order of the states in this chart doesn't reflect the order in which the algorithm would enter them. Chart[0] should look like the following:

Gamma $\rightarrow \cdot$ S

S $\rightarrow \cdot$ NP VP

S $\rightarrow \cdot$ Aux NP VP

S $\rightarrow \cdot$ VP

NP $\rightarrow \cdot$ Det NOMINAL

NP $\rightarrow \cdot$ Proper-Noun

VP $\rightarrow \cdot$ Verb

VP $\rightarrow \cdot$ Verb NP

Book that flight (Chart [0])

Seed chart with top-down predictions for S from grammar (Chart[0])

When dummy start state is processed, it's passed to Predictor, which produces states representing every possible expansion of S, and adds these and every expansion of the left corners of these trees to the bottom of Chart[0]

When VP $\rightarrow$ $\cdot$ Verb [0,0] is reached, Scanner is called, which consults first word of input and adds first state to Chart[1], VP $\rightarrow$ $Book \cdot$ [0,1]

Note, when VP $\rightarrow$ $\cdot$ Verb NP [0,0] is reached in Chart[0], Scanner does not need to add VP $\rightarrow$ $Book \cdot$ [0,1] again to Chart[1]

Book that flight (Chart [1])

When VP $\rightarrow$ Book $\cdot$ [0,1] is passed to Completer, it finds 2 states in Chart[0] whose left corner is V and adds them to Chart[1], moving dots to the right.

When VP $\rightarrow$ V $\cdot$ [0,1] is passed to Completer, S $\rightarrow$ VP $\cdot$ [0,1] is added to Chart[1] since VP is a left corner of S

The last 2 rules in Chart[1] are added by the Predictor when VP $\rightarrow$ V $\cdot$ NP is processed.

Etc.

Returning the Parse

Augment Completer to add pointers

Chart[0]

S0	$\gamma \rightarrow \bullet S$	[0,0]	[]	Dummy start state
S1	$S \rightarrow \bullet$ *NP VP*	[0,0]	[]	Predictor
S2	$NP \rightarrow \bullet$ *Det NOMINAL*	[0,0]	[]	Predictor
S3	$NP \rightarrow \bullet$ *Proper-Noun*	[0,0]	[]	Predictor
S4	$S \rightarrow \bullet$ *Aux NP VP*	[0,0]	[]	Predictor
S5	$S \rightarrow \bullet$ *VP*	[0,0]	[]	Predictor
S6	$VP \rightarrow \bullet$ *Verb*	[0,0]	[]	Predictor
S7	$VP \rightarrow \bullet$ *Verb NP*	[0,0]	[]	Predictor

Chart[1]

S8	*Verb* $\rightarrow$ *book* $\bullet$	[0,1]	[]	Scanner
S9	*VP* $\rightarrow$ *Verb*$\bullet$	[0,1]	[S8]	Completer
S10	*S* $\rightarrow$ *VP*$\bullet$	[0,1]	[S9]	Completer
S11	*VP* $\rightarrow$ *Verb* $\bullet$ *NP*	[0,1]	[S8]	Completer
S12	*NP* $\rightarrow \bullet$ *Det NOMINAL*	[1,1]	[]	Predictor
S13	*NP* $\rightarrow \bullet$ *Proper-Noun*	[1,1]	[]	Predictor

Chart[2]

S14	*Det* $\rightarrow$ *that*$\bullet$	[1,2]	[]	Scanner
S15	*NP* $\rightarrow$ *Det*$\bullet$*NOMINAL*	[1,2]	[S14]	Completer
S16	*NOMINAL* $\rightarrow \bullet$ *Noun*	[2,2]	[]	Predictor
S17	*NOMINAL* $\rightarrow \bullet$ *Noun NOMINAL*	[2,2]	[]	Predictor

Chart[3]

S18	*Noun* $\rightarrow$ *flight*$\bullet$	[2,3]	[]	Scanner
S19	*NOMINAL* $\rightarrow$ *Noun*$\bullet$	[2,3]	[S18]	Completer
S20	*NOMINAL* $\rightarrow$ *Noun*$\bullet$ *NOMINAL*	[2,3]	[S18]	Completer
S21	*NP* $\rightarrow$ *Det NOMINAL* $\bullet$	[1,3]	[S14,S19]	Completer
S22	*VP* $\rightarrow$ *Verb NP* $\bullet$	[0,3]	[S8,S21]	Completer
S23	*S* $\rightarrow$ *VP*$\bullet$	[0,3]	[S22]	Completer
S24	*NOMINAL* $\rightarrow \bullet$ *Noun*	[3,3]	[]	Predictor
S25	*NOMINAL* $\rightarrow \bullet$ *Noun NOMINAL*	[3,3]	[]	Predictor

Useful Properties

Error handling

Alternative control strategies

Error Handling

What happens when we look at the contents of the last table column and don't find a S $\rightarrow \alpha\cdot$ rule?

Is it a total loss? Is there some way to proceed?

Yes. The chart contains every constituent and combination of constituents that could have been found for that input given the grammar.

Also useful for partial parsing or shallow parsing as in information extraction.

Alternative Control Strategies

It's trivial to take the Earley top-down strategy and change it to bottom-up or ...

... to change it to a best first strategy based on the probabilities of the constituents. Simply compute and store the probabilities of constituents in the chart as you are parsing. Then instead of expanding the columns/states in a fixed order you allow the probabilities to control the order of expansion.

Summing Up

Ambiguity, left-recursion, and repeated re-parsing of sub-trees present major problems for parsers

Solution: Use Dynamic Programming, e.g., the Earley algorithm

For Next Time

Discuss project assignment

Start reading 2 Q-A papers

Chapter 11

Start studying for midterm