Parsing: Top-down Vs. Bottom-up Structural Ambiguities

LU XIAO
LXIA004@SYR.EDU
213 HINDS HALL

ADOPTED SOME MATERIALS DEVELOPED IN PREVIOUS COURSES BY NANCY MCCRACKEN, LIZ LIDDY AND OTHERS; AND SOME INSTRUCTOR RESOURCES FOR THE BOOK "SPEECH AND LANGUAGE PROCESSING" BY DANIEL JURAFSKY AND JAMES H. MARTIN

OFFICE HOURS IN MARCH BREAK

Thursday, March 14

11 am. – 12 pm. Hinds Hall 213

Parsing

- Parsing with CFGs refers to the task of assigning proper trees to input strings
- Proper here means a tree that covers all and only the elements of the input and has an S at the top
- It doesn't actually mean that the system can select the correct tree from among all the possible trees

Parsing Algorithms Defined:

- The process of finding a derivation (i. e. sequence of productions) leading from the START symbol to the TERMINAL symbols
  - Shows how a particular sentence ***could be*** generated by the rules of the grammar
- If sentence is structurally ambiguous, more than one possible derivation is produced
- Can solve both the recognition and analysis problems
  - Is this sentence derived from this grammar?
  - Give the derivation(s) that can derive this sentence.
- Parsing algorithms give a strategy for finding a derivation by making choices among the derivation rules and deciding when the derivation is complete or not.

Assumptions (For Now)

- We have all the words already in some buffer
- The input isn't POS tagged
- We won't worry about morphological analysis
- All the words are known

- These are all problematic in various ways, and would have to be addressed in real applications.

Top-down Parser

- Goal-driven
- At each stage, the parser looks at a non-terminal symbol (starting with S) and then sees which rules can be applied
  - Typically progresses from top-to-bottom, left-to-right
  - Non-deterministic (can be rewritten in more than one way)
- When rules derive lexical elements (words), check with the input to see if the right sentence is being derived
- An algorithm may include a backtracking mechanism
  - When it is determined that the wrong rule has been used, it backs up and tries another rule

Example Grammar

- The flight grammar from the text has multiple rules for S:

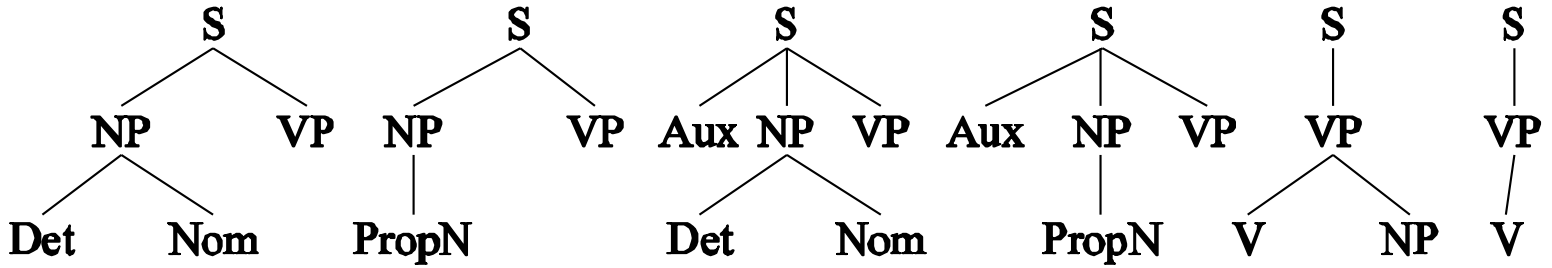
$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$
$S \rightarrow VP$
$NP \rightarrow Pronoun$
$NP \rightarrow Proper\text{-}Noun$
$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$
$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$
$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$
$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$

$Det \rightarrow that \mid this \mid a$
$Noun \rightarrow book \mid flight \mid meal \mid money$
$Verb \rightarrow book \mid include \mid prefer$
$Pronoun \rightarrow I \mid she \mid me$
$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$Aux \rightarrow does$
$Preposition \rightarrow from \mid to \mid on \mid near \mid through$

Example Derivation

- Derivation for "Book that flight" (from the text)
  - The Start symbol **S**
  - Can derive 3 rules as follows:

  - Each non-terminal can derive additional rules



  - Only the last two trees can derive the word "book" as first in the input

Top-down Parsing Demo

- NLTK parsing demos
  - Top-down parsing using a recursive descent algorithm
    - Top down parsing with back-tracking
    - Must not have left-recursion in the grammar rules
      ~~expr -> expr + term~~

    nltk.app.rdparser()

- Described in NLTK book, Chapter 8, Analyzing Sentence Structure

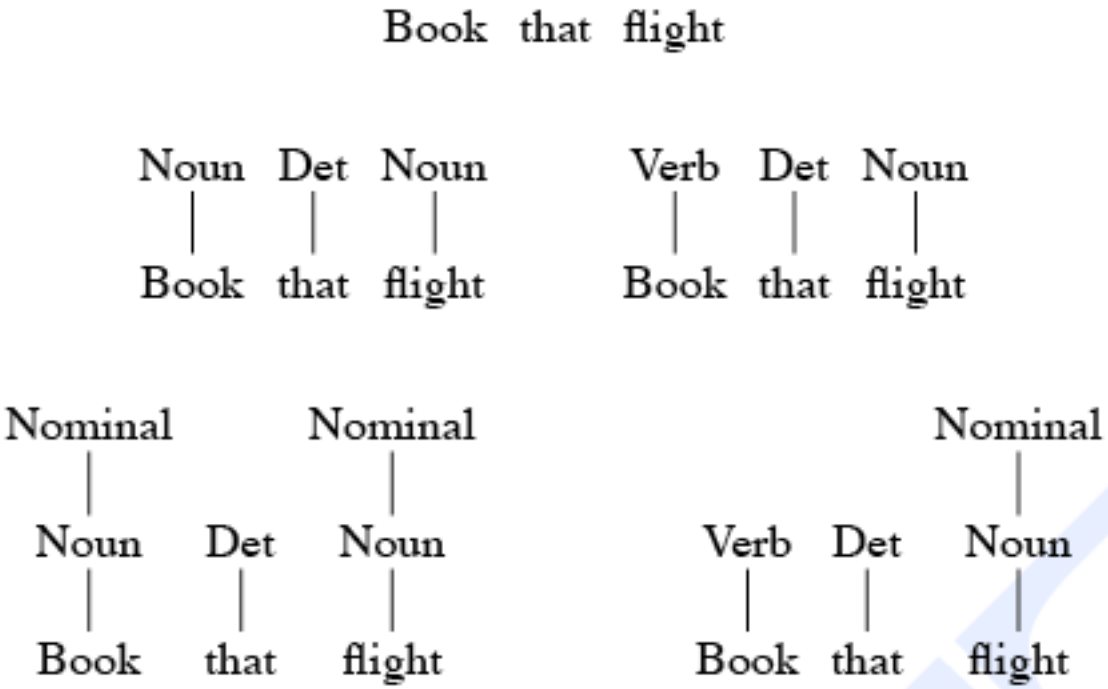
Bottom-up Parser

- Data-driven
- Looks at words in input string first, checks / assigns their category(ies), and tries to combine them into acceptable structures in the grammar
- Involves scanning the derivation so far for sub-strings which match the right-hand-side of grammar / production rules and using the rule that would show their derivation from the non-terminal symbol of that rule

Bottom-up Derivation

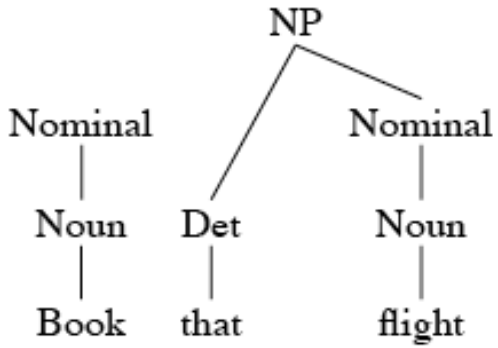
- Starts with input text

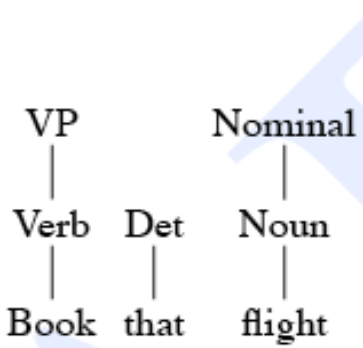
- derive the text from rules, in this case, two possible lexical rules

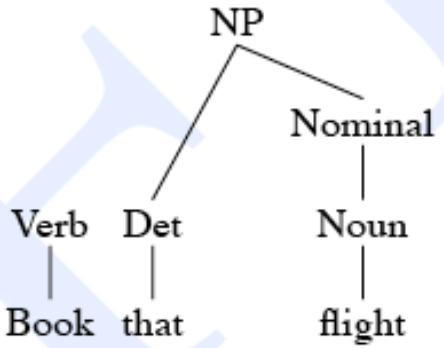


Bottom-up Derivation

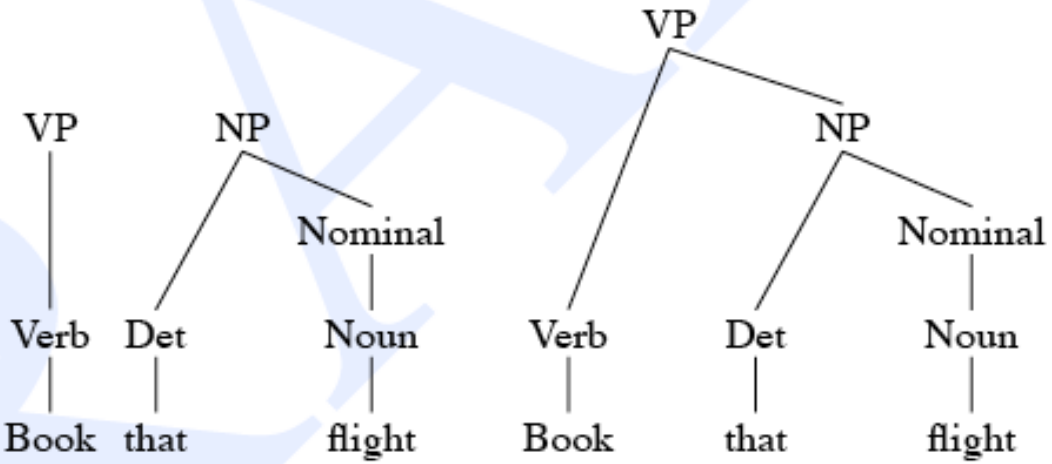
- Each of those can be derived from nonterminals. Only the rightmost tree can continue the derivation here:







- And only one succeeds: S -> VP



Bottom-up Parsing

- Algorithm called shift/reduce parsing
  - Scans the input from left to right and keeps a "stack" of the partial parse tree so far
  - Chooses shift or reduce operations
    - The shift operation looks at the next input and shifts it onto the stack
    - The reduce operation looks at N symbols on the stack and if they match the RHS of a grammar rule, reduces the stack by replacing those symbols with the nonterminal
- Also must either incorporate back-tracking or must keep multiple possible parses

Bottom-up Parsing Demo

- NLTK parsing demos
  - Bottom-up parsing using a shift-reduce algorithm
    - Instead of back-tracking or multiple parses, this NLTK implementation requires outside intervention to apply the correct rule when there is a choice

```
nltk.app.srparser()
```

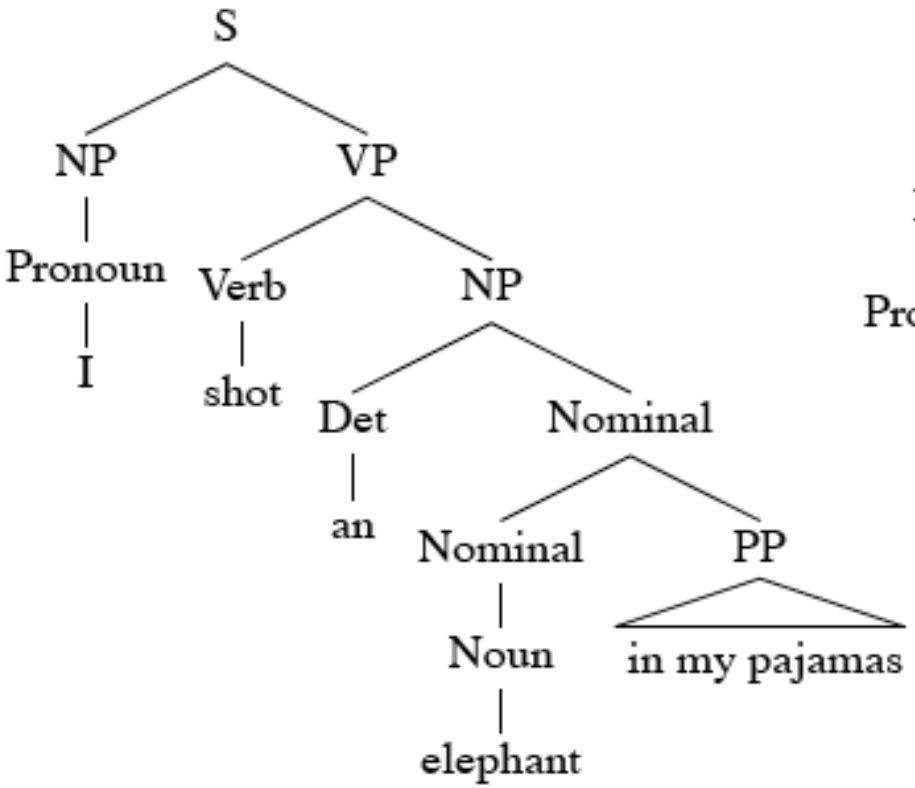
- Described in NLTK book, Chapter 8, Analyzing Sentence Structure

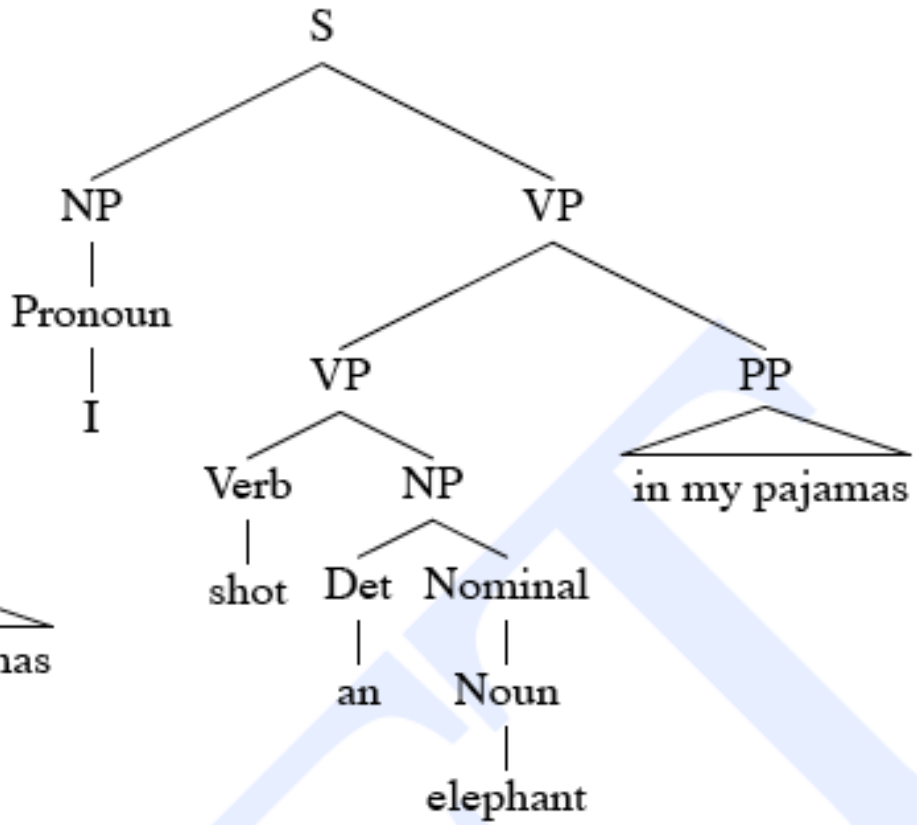
Parsing Issues

- Top-down
  - Only searches for trees that can be answers (i.e. S's)
  - But also suggests trees that are not consistent with any of the words
- Bottom-up
  - Only forms trees consistent with the words
  - But suggest trees that make no sense globally
- Note that in the "book that flight" example, there was local ambiguity between "book" being a verb or a noun that was resolved at the end of the parse
- But examples with structural ambiguity will not be resolved, resulting in more than one possible derivation

Structural Ambiguity

- *One morning I shot an elephant in my pajamas. How he got into my pajamas I don't know.* Groucho Marx, *Animal Crackers*, 1930.





Parsing Algorithms: Charts

Solutions To Parsing Problems

- Modern parsing algorithms have three key ideas:
  - Solve the problem of performance with chart parsers (running time in parsing)
  - Solve the problems of pre-defining CFG or other grammars by using Treebanks and statistical parsing
  - Partially solve the problems of correctly choosing the best parse trees by using lexicalization (information about words from the Treebank)

Parsing Algorithms

- The simple parsers that we have seen are exponential in time (recursive descent with back-tracking) and (shift reduce with back-tracking) with respect to the length of the input
- Avoid back-tracking and re-doing subtrees
  - Recall that the backtracking recursive descent expanded some subtrees multiple times
  - Use charts to record subtrees to avoid redundant computation
- Use forms of dynamic programming to search for good parse trees
  - Attempt to perform exponential process in polynomial time

Binarization Reduces Exponential Process

- Where binarization means only reducing rules with 2 right hand side (RHS) symbols
  - Allows 2 dimensional charts
- All CFG grammars have a Chomsky Normal Form where every rule has no more than 2 symbols on the RHS
  - Example grammar rule with 3 RHS symbols:

    VP -> Verb NP PP
  - Transformed to equivalent grammar with only 2 RHS symbols:

    VP -> Verb NPtemp

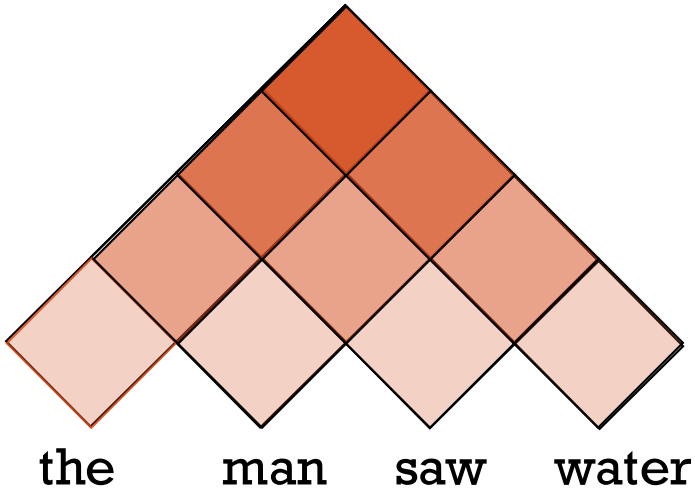
    Nptemp -> NP PP

Chart Parsers

- CKY (Cocke-Kasami-Younger) algorithm is an example
  - Bottom-up parser (but can also have top down chart parsers)
  - Requires grammar to be in Chomsky Normal Form, with only two symbols on the right-hand-side of each production
  - Fills in a data structure called a chart or a parse triangle
    - Other parsers, such as Earley's algorithm, use similar chart ideas to work on two subtrees at a time
- Key idea in parser development from 1970 - 1990

CKY Parsing

- For input of length n, fills a parse table triangle of size (n, n) , where each element has the non-terminal production representing the span of text from position i to j.
  - Cells in first (bottom) layer describe trees of single words
  - Cells in second layer describes how rewrite rules can be used to combine trees in first layer for trees with two words
  - Etc.



Parsing: Treebanks Statistical Parsing

23

Grammar Engineering

- Before you can parse you need a grammar.
- So where do grammars come from? Grammar Engineering
  - Hand-crafted decades-long efforts by humans to write grammars (typically in some particular grammar formalism of interest to the linguists developing the grammar)
- Problems with Grammar Engineering
  - Typical grammars built before the 1990's would overgeneralize; Real broad-coverage language grammar could give rise to millions of parses on a single sentence
  - but if structuring grammar to restrict parses, it would leave up to 30% of the sentence without parses

The Rise Of Annotated Data

- TreeBanks
  - Semi-automatically generated sets of parse trees for the sentences in some corpus (manually corrected by human annotators).
- Building a treebank seems a lot slower and less useful than building a grammar, but a treebank offers several advantages:
  - Reusability of the labor: many parsers, POS taggers, etc.; valuable resource for linguistics
  - Broad coverage
  - Frequencies and distributional information
  - A way to evaluate systems on the same text

Penn Treebank

- Example of manually annotated sentence structure

```
( (S
    (NP-SBJ (DT The) (NN move))
    (VP (VBD followed)
      (NP
        (NP (DT a) (NN round))
        (PP (IN of)
          (NP
            (NP (JJ similar) (NNS increases))
            (PP (IN by)
              (NP (JJ other) (NNS lenders)))
            (PP (IN against)
              (NP (NNP Arizona) (JJ real) (NN estate) (NNS loans))))))
      (, ,)
      (S-ADV
        (NP-SBJ (-NONE- *))
        (VP (VBG reflecting)
          (NP
            (NP (DT a) (VBG continuing) (NN decline))
            (PP-LOC (IN in)
              (NP (DT that) (NN market)))))))
    (. .)))
```


[Marcus et al. 1993, *Computational Linguistics*]

Getting Grammar From A Treebank

- Given an annotated sentence,

(11.10) [NP Shearson's] [JJ easy-to-film], [JJ black-and-white] "[SBAR Where We Stand]" [NNS commercials]

- We can make a grammar rule:

```
NP → NP JJ , JJ `` SBAR '' NNS
```

- And we'll make rules for sub-trees as well

Sample Rules For Noun Phrases

```
NP → DT JJ NNS
NP → DT JJ NN NN
NP → DT JJ JJ NN
NP → DT JJ CD NNS
NP → RB DT JJ NN NN
NP → RB DT JJ JJ NNS
NP → DT JJ JJ NNP NNS
NP → DT NNP NNP NNP NNP JJ NN
NP → DT JJ NNP CC JJ JJ NN NNS
NP → RB DT JJS NN NN SBAR
NP → DT VBG JJ NNP NNP CC NNP
NP → DT JJ NNS , NNS CC NN NNS NN
NP → DT JJ JJ VBG NN NNP NNP FW NNP
NP → NP JJ , JJ `` SBAR '' NNS
```

Treebank Grammars

- We could read off the grammar from the Treebank
  - The grammar is the set of rules (local subtrees) that occur in the annotated corpus
  - They tend to avoid recursion (and elegance and parsimony)
    - i.e. they tend to the flat and redundant
  - Penn TreeBank (III) has about 17500 grammar rules under this definition.
- But the main use of the Treebank is to provide the probabilities to inform the statistical parsers, and the grammar does not actually have to be generated.
- The grammar hovers behind the Treebank; it is in the minds of the human annotators (and in the annotation manual!)

Probabilistic Context-free Grammars

- By way of introduction to statistical parsers, we first introduce the idea of associating probabilities with grammar rewrite rules.
  - Attach probabilities to grammar rules
  - The expansions for a given non-terminal sum to 1

VP -> Verb	.55
VP -> Verb NP	.40
VP -> Verb NP PP	.05

Getting The Probabilities

- From a treebank of annotated data, get the probabilities that any non-terminal symbol is rewritten with a particular rule
  - So for example, to get the probability for a particular VP rule just count all the times the rule is used and divide by the number of VPs overall.
- The parsing task is to generate the parse tree with the highest probability (or the top n parse trees)
- For a PCFG parser, the probability of a parse tree is the product of the probabilities of the rules used in the derivation

$$P(T,S) = \prod_{node \in T} P(rule(n))$$

Typical Approach To PCFG Parser

- Use CKY as the backbone of the algorithm
- Assign probabilities to constituents as they are completed and placed in the table
- Use the max probability for each constituent going up

S → NP VP	0.9
S → VP	0.1
VP → V NP	0.5
VP → V	0.1
VP → V @VP_V	0.3
VP → V PP	0.1
@VP_V → NP PP	1.0
NP → NP NP	0.1
NP → NP PP	0.2
NP → N	0.7
PP → 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

	1 fish	2 people	3 fish	4 tanks
0	N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	NP → NP NP 0.0049 VP → V NP 0.105 S → VP 0.0105	NP → NP NP 0.0000686 VP → V NP 0.00147 S → NP VP 0.000882	NP → NP NP 0.0000009604 VP → V NP 0.00002058 S → NP VP 0.00018522
1		N → people 0.5 V → people 0.1 NP → N 0.35 VP → V 0.01 S → VP 0.001	NP → NP NP 0.0049 VP → V NP 0.007 S → NP VP 0.0189	NP → NP NP 0.0000686 VP → V NP 0.000098 S → NP VP 0.01323
2			N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	NP → NP NP 0.00196 VP → V NP 0.042 S → VP 0.0042
3				N → tanks 0.2 V → tanks 0.3 NP → N 0.14 VP → V 0.03 S → VP 0.003
4				

Call buildTree(score, back) to get the best parse

Example showing filled-in CKY chart for a PCFG for sentence "fish people fish tanks" from Chris Manning

Problems With PCFG Parsing

- But this typical approach always just picks the most likely rule in the derivation
  - For example, if it is more likely that a prepositional phrases attaches to the noun phrase that it follows instead of the verb, then the probabilistic parser will always attach prepositional phrases to the closest noun
- The probability model we're using is only based on the rules in the derivation…
  - Doesn't use the words in any real way
  - Doesn't take into account where in the derivation a rule is used
    - E.g. the parent of the non-terminal of the derivation
    - Most probable parse isn't usually the right one (the one in the treebank test set).
- Collect statistics and use in a better way

Parsing:
Lexicalized Statistical Parsing
Evaluation Of Parsing
Available Parsers

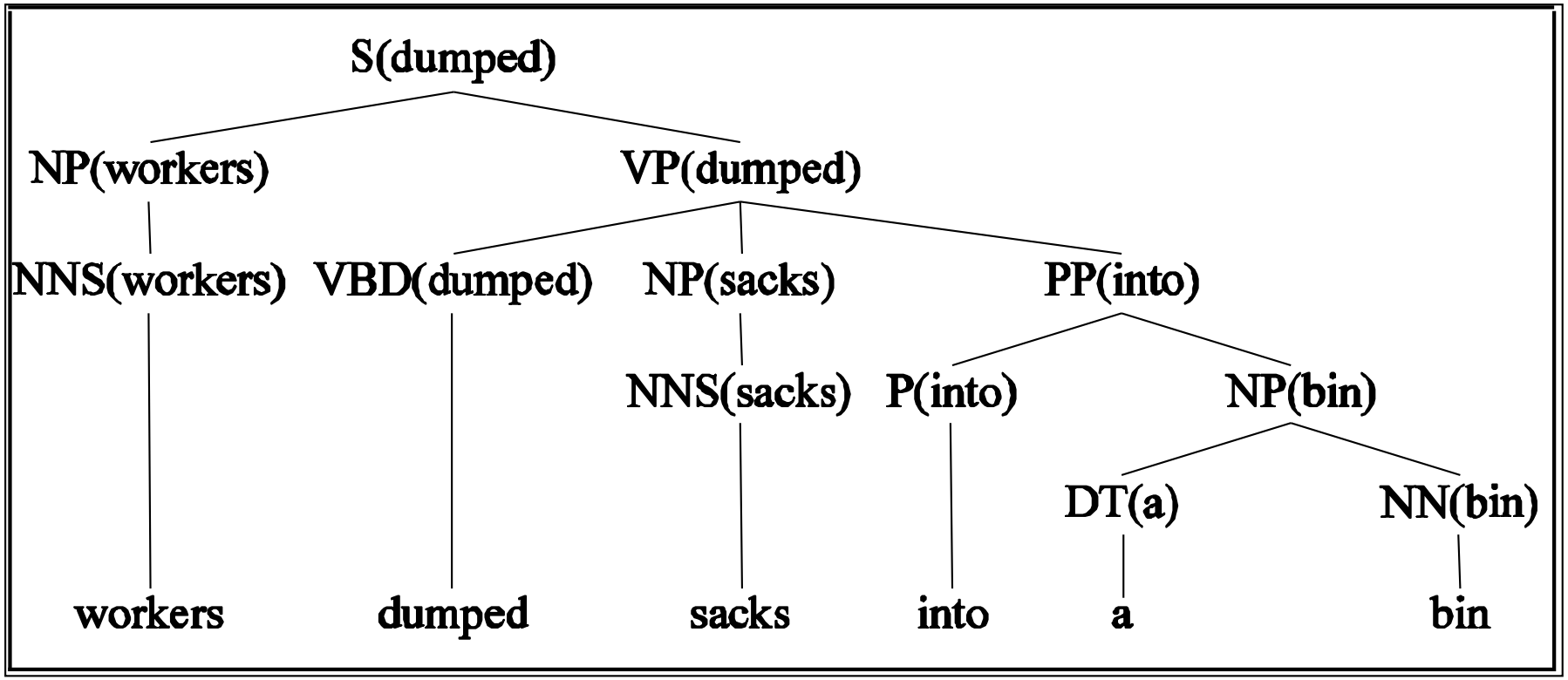
Lexicalized Statistical Parsing

- Add lexical dependencies to the scheme of probabilities
  - Integrate the preferences of particular words into the probabilities in the derivation
  - I.E. Condition the rule probabilities on the actual words
- To do that we're going to make use of the notion of the head of A phrase
  - The head of an NP is its noun
  - The head of A VP is its verb
  - The head of A PP is its preposition

  (It's really more complicated than that but this will do.)
- Main parsing breakthrough idea of the 1990's
- Expand the set of phrase types with phrase type/word
  - In practice, we learn probabilities to automatically detect head words

Example (Right)

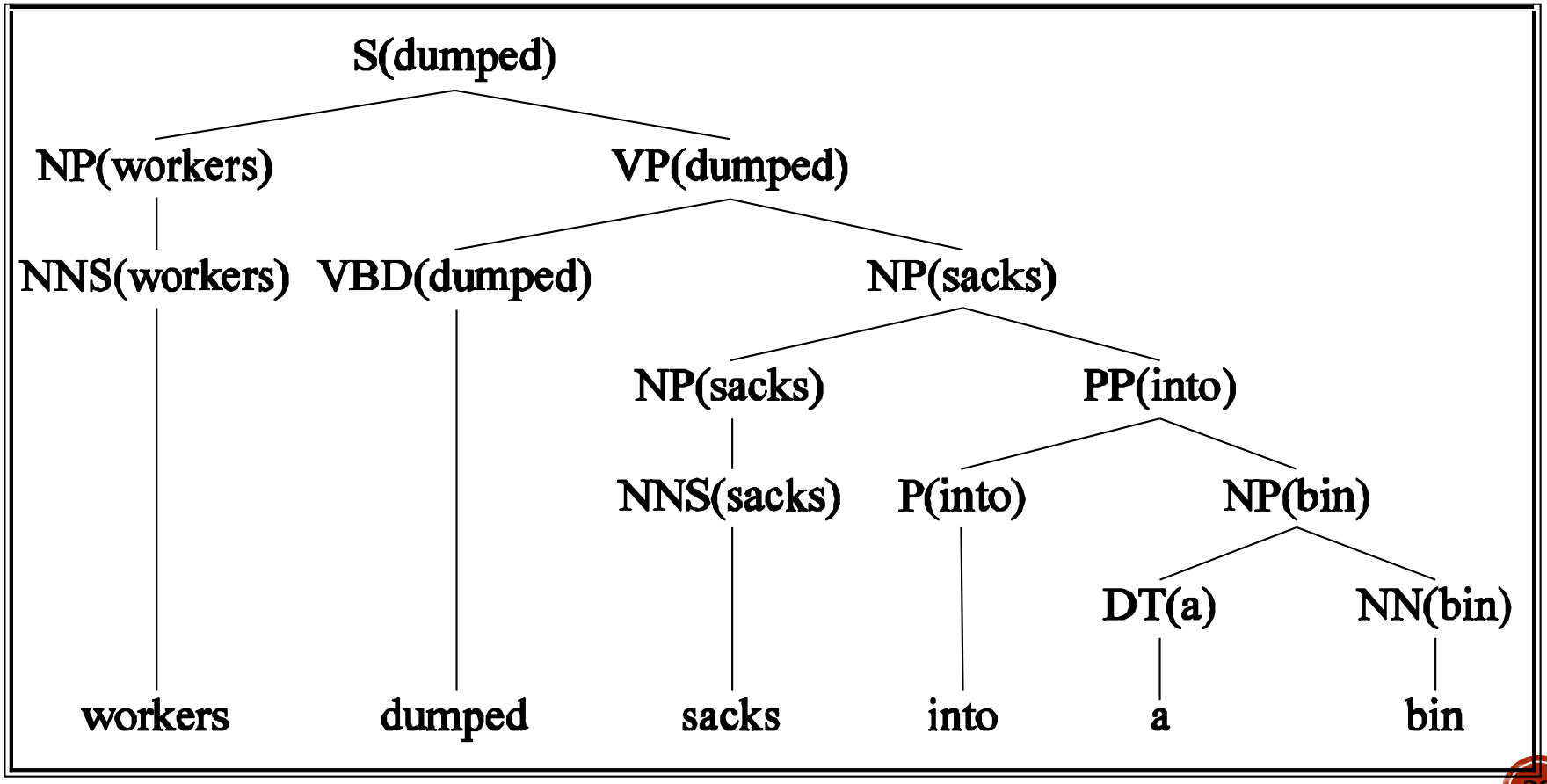
- Should we attach the prepositional phrase with head "into" to the verb "dumped"?



- In this tree, each phrase type, such as NP or VP, is also shown with its attached head word.

Example (Wrong)

- Or should we attach the prepositional phrase with head "into" to the noun "sacks"?



Preferences

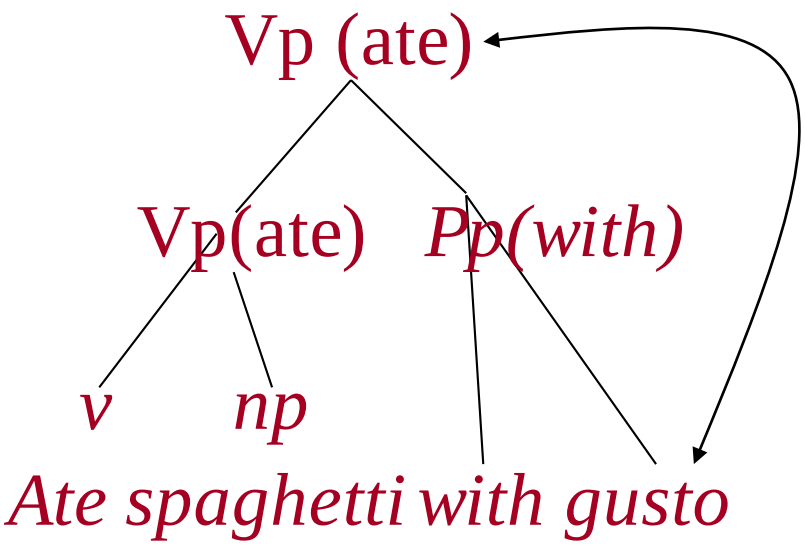
- The issue here is the attachment of the PP. So the affinities we care about are the ones between dumped and into vs. sacks and into.
  - So count the places where dumped is the head of a constituent that has a PP child with into as its head and normalize
  - Vs. the situation where sacks is a constituent with into as the head of a PP child.
- In general, collect statistics on preferences (aka affinities)
  - Use verb subcategorization
    - Particular verbs have affinities for particular VPs
  - Objects affinities for their verbs, mostly their parents and grandparents
    - Some objects fit better with some verbs than others

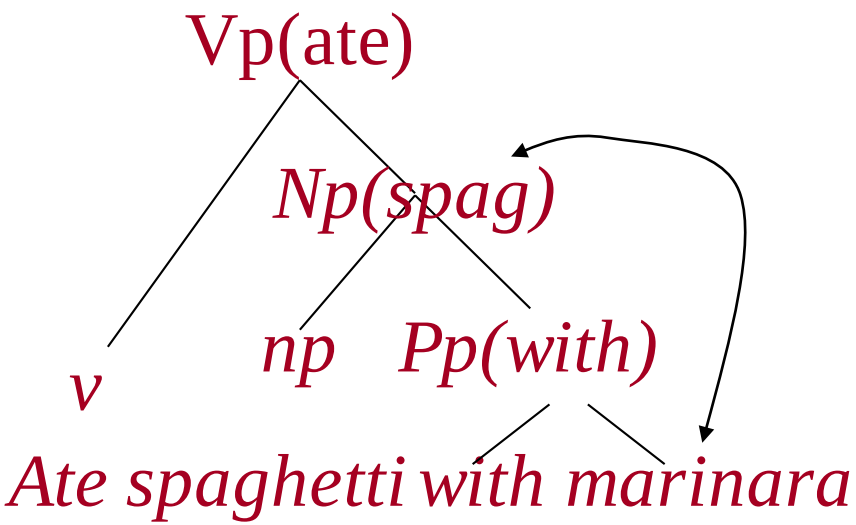
Preference Example

- Consider the VPs
  - Ate spaghetti with gusto
  - Ate spaghetti with marinara
- The affinity of gusto for eat is much larger than its affinity for spaghetti
- On the other hand, the affinity of marinara for spaghetti is much higher than its affinity for ate

Preference Example (2)

- Note the relationship here is more distant and doesn't involve a headword since gusto and marinara aren't the heads of the PPs.





Note

- Jim Martin: "In case someone hasn't pointed this out yet, this lexicalization stuff is a thinly veiled attempt to incorporate semantics into the syntactic parsing process…
  - Duhh..,.Picking the right parse requires the use of semantics."

Last Points

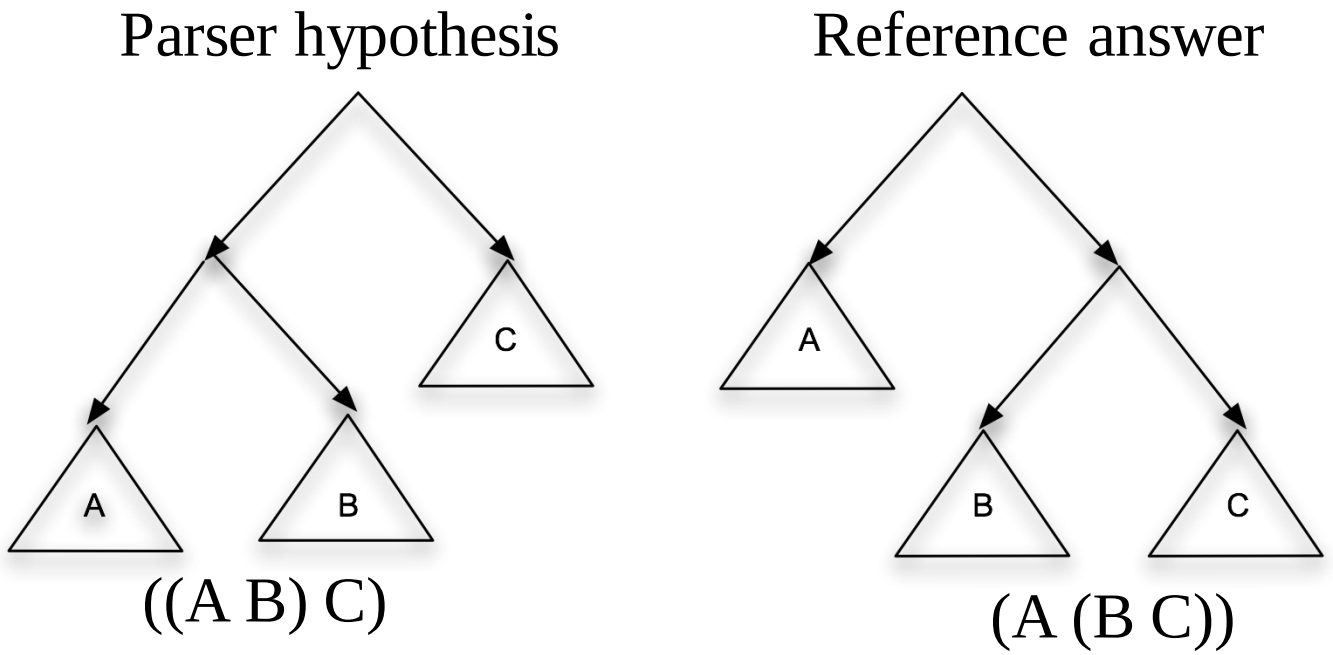
- Statistical parsers are getting quite good, but it's still quite challenging to expect them to come up with the correct parse given only statistics from syntactic and lexical information.
- But if our statistical parser comes up with the top-N parses, then it is quite likely that the correct parse is among them.
- Lots of current work on
  - Re-ranking to make the top-N list even better.
- There are also grammar-driven parsers that are competitive with the statistical parsers, notably the CCG (Combinatory Categorial Grammar) parsers

Evaluation

- Given that it is difficult/ambiguous to produce the entire correct tree, look at how much of content of the trees are correctly produced
  - Evaluation measures based on the correct number of constituents (or sub-trees) in the system compared to the reference (gold standard)
- Precision
  - What fraction of the sub-trees in our parse matched corresponding sub-trees in the reference answer
    - How much of what we're producing is right?
    - Reduce number of false positives
- Recall
  - What fraction of the sub-trees in the reference answer did we actually get?
    - How much of what we should have gotten did we get?
    - Reduce number of false negatives
- F-measure combines precision and recall to give an overall score.

Evaluation

- An additional evaluation measure that is often reported is that of Crossing Brackets errors, in which the subtrees are equal, but they are put together in a different order.



Available Parsers

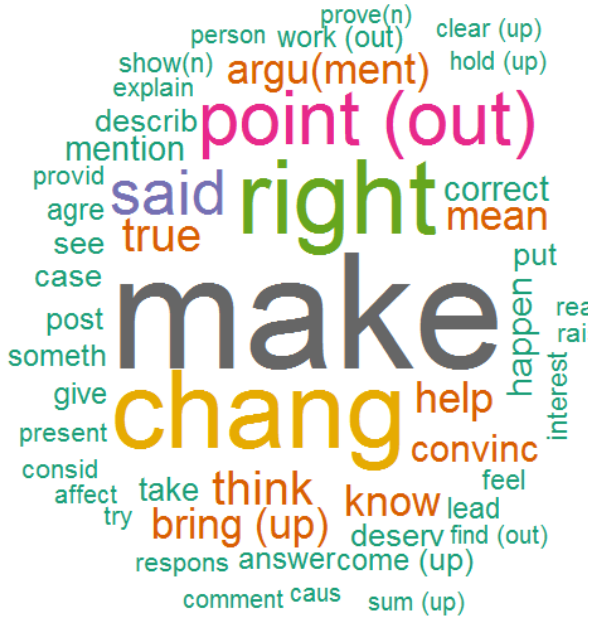
- Among the family of lexicalized statistical parsers are the original Collins parser (Michael Collins 1996, 1999) and the Charniak parser (1997)
  - both are publicly available and widely used, for non-commercial purposes.
- The Charniak series of parsers is still under development, by Eugene Charniak and his group; it produces N-best parse trees.
  - Its evaluation is on the Penn Treebank at about 91% F measure.
- Another top performing parser, originally by Dan Klein and Christopher Manning, is available from the Stanford NLP group
  - combines "separate PCFG phrase structure and lexical dependency experts". Demo at: http://nlp.stanford.edu:8080/parser/
- The CCG (Combinatory Categorial Grammar) parsers are available from their open source page
  - http://groups.inf.ed.ac.uk/ccg/software.html: OpenNLP CCG library

Dependency Parsing

- Dependency parsing has some resemblance to lexicalized statistical parsing because of the importance of the lexical entities (words) to capturing the syntactic structure
- But dependency parsing produces a simpler representation of the structure.
  - Can be easier to use in some semantic applications
- Parsing algorithms are similar to constituent parses
  - Statistics for dependency relations learned from Penn Treebank
  - Used bottom-up parser to find best parse(s)
  - Some additional mechanism used to find non-projective parses

Persuaded Users: Why were They Persuaded?

Analysis of Rationales (Reddit "Change My View"



(a) The tag cloud representation of the governors extracted from explanations.

(b) The tag cloud representation of the objects extracted from explanations.