Natural Language Processing: Introduction to Syntactic Parsing

Barbara Plank
DISI, University of Trento
barbara.plank@disi.unitn.it

NLP+IR course, spring 2012

Note: Parts of the material in these slides are adapted version of slides by Jim H. Martin, Dan Jurasky, Christopher Manning

Today

Moving from words to bigger units

- Syntax and Grammars
- Why should you care?
- Grammars (and parsing) are key components in many NLP applications, e.g.
  - Information extraction
  - Opinion Mining
  - Machine translation
  - Question answering

Overview

- Key notions that we'll cover
  - Constituency
  - Dependency
- Approaches and Resources
  - Empirical/Data-driven parsing, Treebank
- Ambiguity / The exponential problem
- Probabilistic Context Free Grammars
  - CFG and PCFG
  - CKY algorithm, CNF
- Evaluating parser performance
- Dependency parsing

Two views of linguistic structure:
1. Constituency (phrase structure)

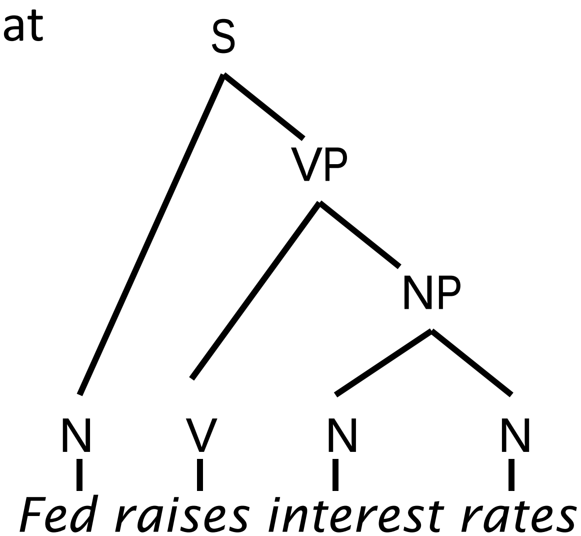
- The basic idea here is that groups of words within utterances can be shown to act as single units
- For example, it makes sense to the say that the following are all ***noun phrases*** in English...

Harry the Horse	a high-class spot such as Mindy's
the Broadway coppers	the reason he comes into the Hot Box
they	three parties from Brooklyn

- Why? One piece of evidence is that they can all precede verbs.

Two views of linguistic structure: 1. Constituency (phrase structure)

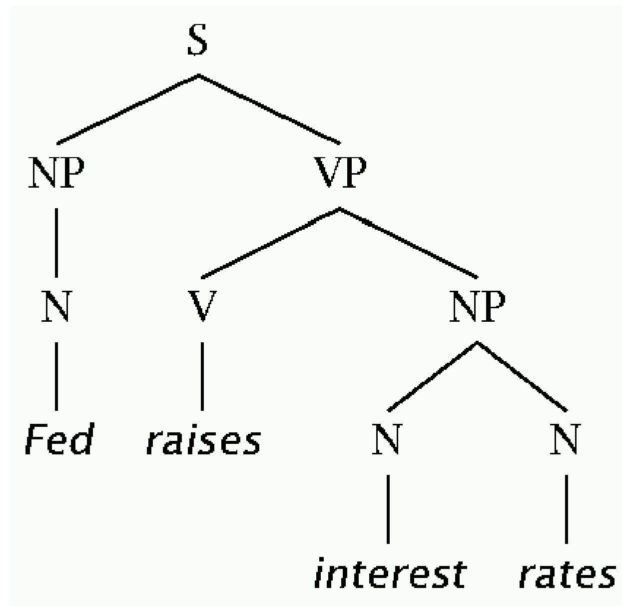
- Phrase structure organizes words into nested constituents.
- How do we know what is a constituent? (Not that linguists don't argue about some cases.)
  - Distribution: a constituent behaves as a unit that can appear in different places:
    - John talked [to the children] [about drugs].
    - John talked [about drugs] [to the children].
    - *John talked drugs to the children about
  - Substitution/expansion/pro-forms:
    - I sat [on the box/right of the box/there].



Headed phrase structure

To model constituency structure:

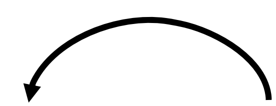
- VP $\rightarrow$ … VB* …
- NP $\rightarrow$ … NN* …
- ADJP $\rightarrow$ … JJ* …
- ADVP $\rightarrow$ … RB* …
- PP $\rightarrow$ … IN* …



- Bracket notation of a tree (Lisp S-structure):
  (S (NP (N Fed)) (VP (V raises) (NP (N interest) (N rates))))

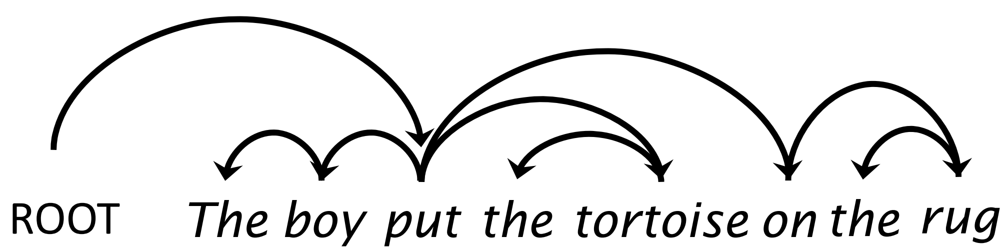
Two views of linguistic structure:
2. Dependency structure

- In CFG-style phrase-structure grammars the main focus is on *constituents.*
- But it turns out you can get a lot done with binary relations among the lexical items (words) in an utterance.
- In a dependency grammar framework, a parse is a tree where
  - the nodes stand for the words in an utterance
  - The links between the words represent dependency relations between pairs of words.
    - Relations may be typed (labeled), or not.



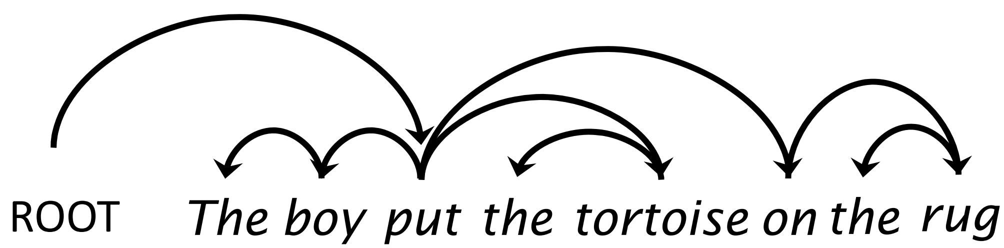
dependent head
modifier governor

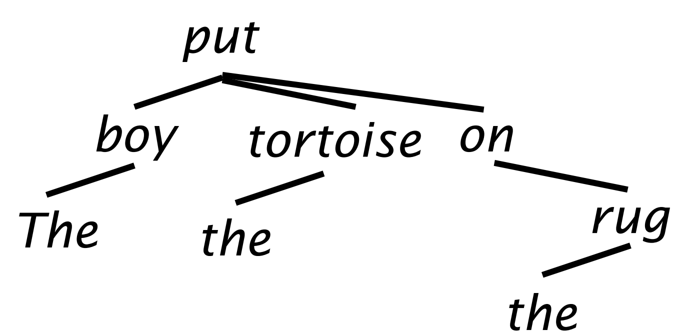
Sometimes arcs drawn in opposite direction



Two views of linguistic structure:
2. Dependency structure

- Alternative notations (e.g. rooted tree):





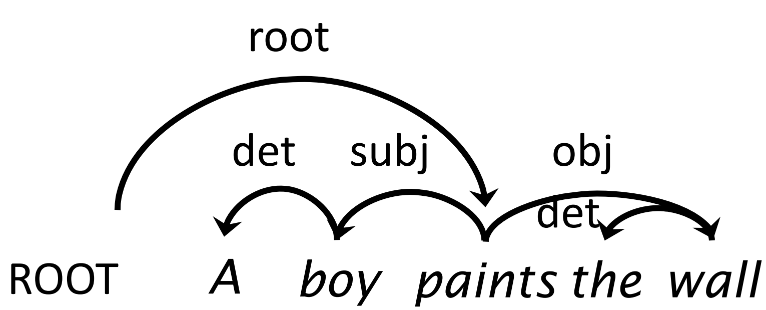
Dependency Labels

Argument dependencies:

- Subject (subj), object (obj), indirect object (iobj)…

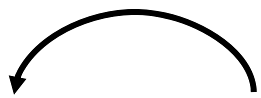
Modifier dependencies:

- Determiner (det), noun modifier (nmod), verbal modifier (vmod), etc.



Quiz question

- In the following sentence, which word is *nice* a dependent of?



There is a nice warm breeze out in the balcony.

1. warm
2. in
3. breeze
4. balcony

Comparison

- Dependency structures explicitly represent
  - head-dependent relations (directed arcs),
  - functional categories (arc labels).
- Phrase structures explicitly represent
  - phrases (nonterminal nodes),
  - structural categories (nonterminal labels),
  - possibly some functional categories (grammatical functions, e.g. PP-LOC).
- (There exist also hybrid approaches, e.g. Dutch Alpino grammar).

Statistical Natural Language Parsing

Parsing: The rise of data and statistics

The rise of data and statistics: Pre 1990 ("Classical") NLP Parsing

- Wrote symbolic grammar (CFG or often richer) and lexicon

S $\rightarrow$ NP VP	NN $\rightarrow$ *interest*
NP $\rightarrow$ (DT) NN	NNS $\rightarrow$ *rates*
NP $\rightarrow$ NN NNS	NNS $\rightarrow$ *raises*
NP $\rightarrow$ NNP	VBP $\rightarrow$ *interest*
VP $\rightarrow$ V NP	VBZ $\rightarrow$ *rates*

- Used grammar/proof systems to prove parses from words
- This scaled very badly and didn't give coverage.

Classical NLP Parsing:
The problem and its solution

- Categorical constraints can be added to grammars to limit unlikely/weird parses for sentences
  - But the attempt make the grammars not robust
    - In traditional systems, commonly 30% of sentences in even an edited text would have *no* parse.
- A less constrained grammar can parse more sentences
  - But simple sentences end up with ever more parses with no way to choose between them
- We need mechanisms that allow us to find the most likely parse(s) for a sentence
  - Statistical parsing lets us work with very loose grammars that admit millions of parses for sentences but still quickly find the best parse(s)

The rise of annotated data: The Penn Treebank

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

```
( (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)))))))
   (. .)))
```

Most well known part is the Wall Street Journal section of the Penn TreeBank.

1 M words from the 1987-1989 Wall Street Journal newspaper.

The rise of annotated data

- Starting off, building a treebank seems a lot slower and less useful than building a grammar

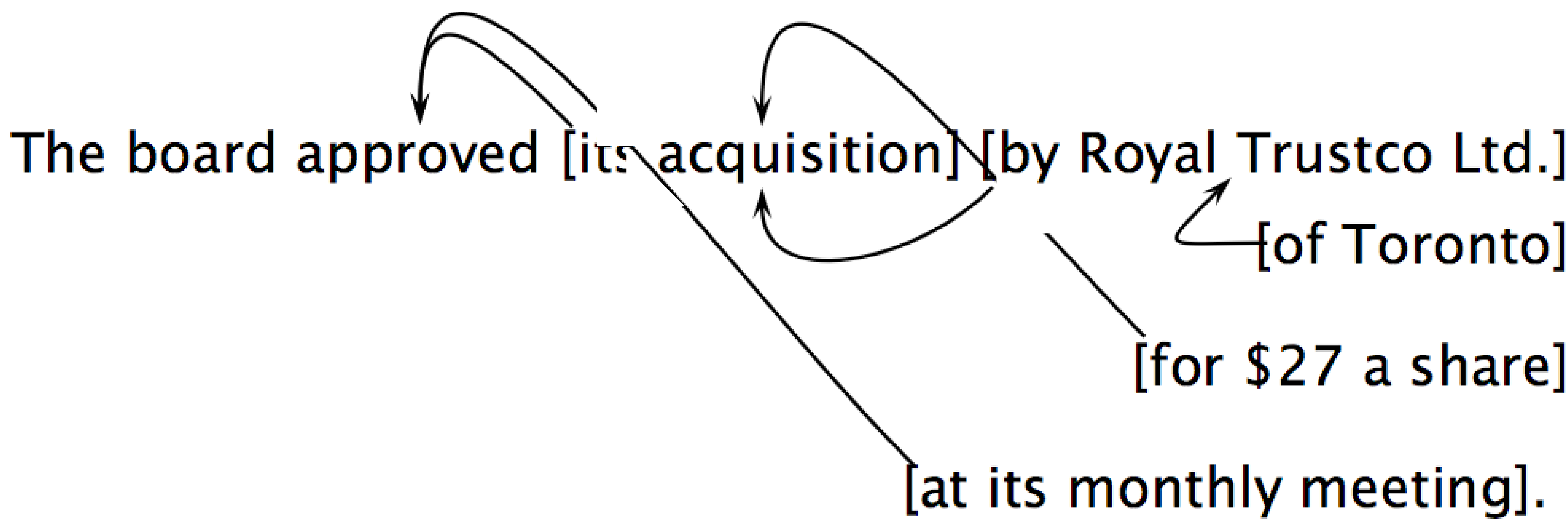
- But a treebank gives us many things
  - Reusability of the labor
    - Many parsers, POS taggers, etc.
    - Valuable resource for linguistics
  - Broad coverage
  - Statistics to build parsers
  - A way to evaluate systems

Statistical Natural Language Parsing

An exponential number of attachments

Attachment ambiguities

- A key parsing decision is how we ‘attach’ various constituents



Attachment ambiguities

- How many distinct parses does the following sentence have due to PP attachment ambiguities?

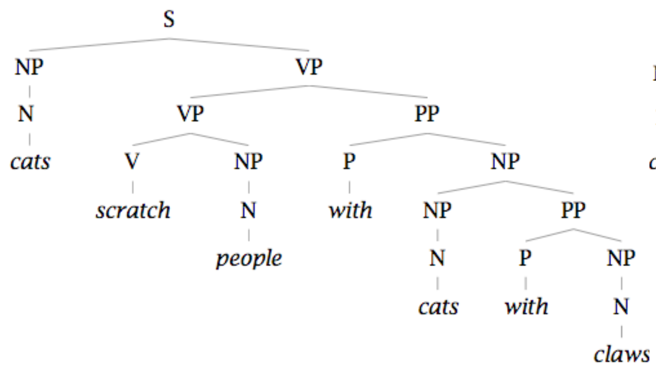
John wrote the book with a pen in the room.

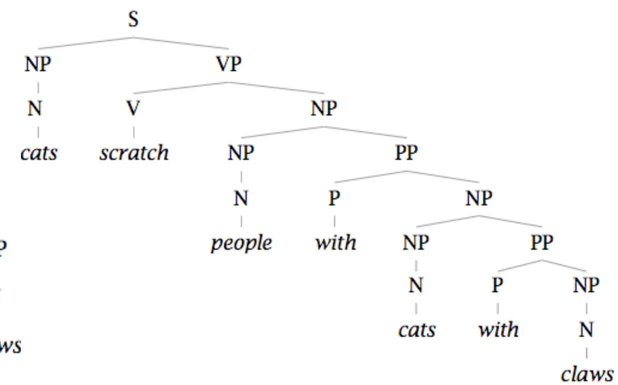
John wrote [the book] [with a pen] [in the room].
John wrote [[the book] [with a pen]] [in the room].
John wrote [the book] [[with a pen] [in the room]].
John wrote [[the book] [[with a pen] [in the room]]].
John wrote [[[the book] [with a pen]] [in the room]].

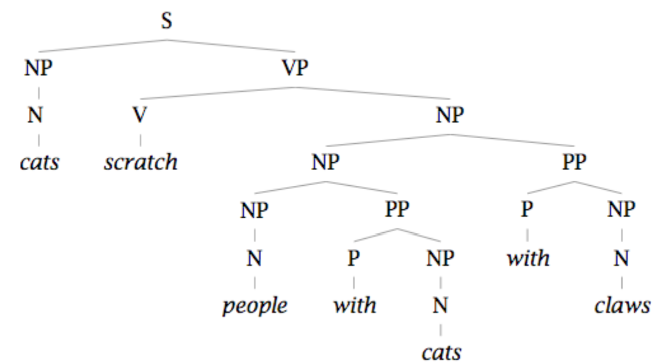
Catalan numbers: $C_n = (2n)!/[(n+1)!n!]$ - an exponentially growing series

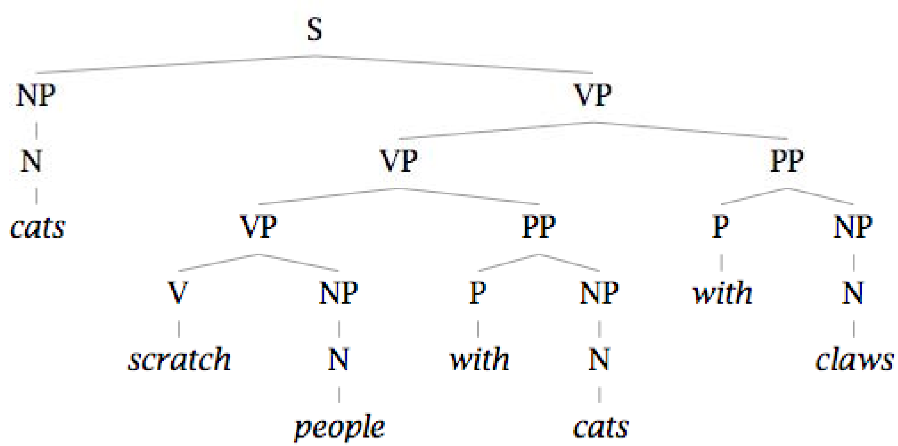
1 1
2 2
3 5
4 14
5 42
6 132
7 429
8 1430

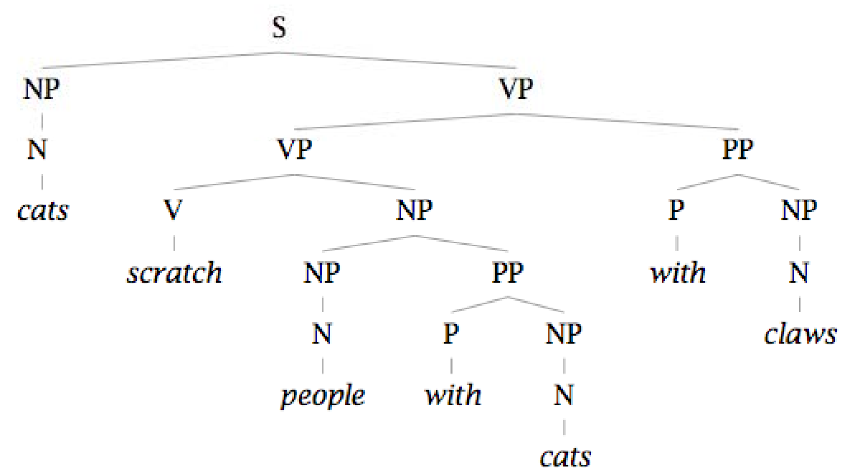
Two problems to solve:
1. Avoid repeated work…



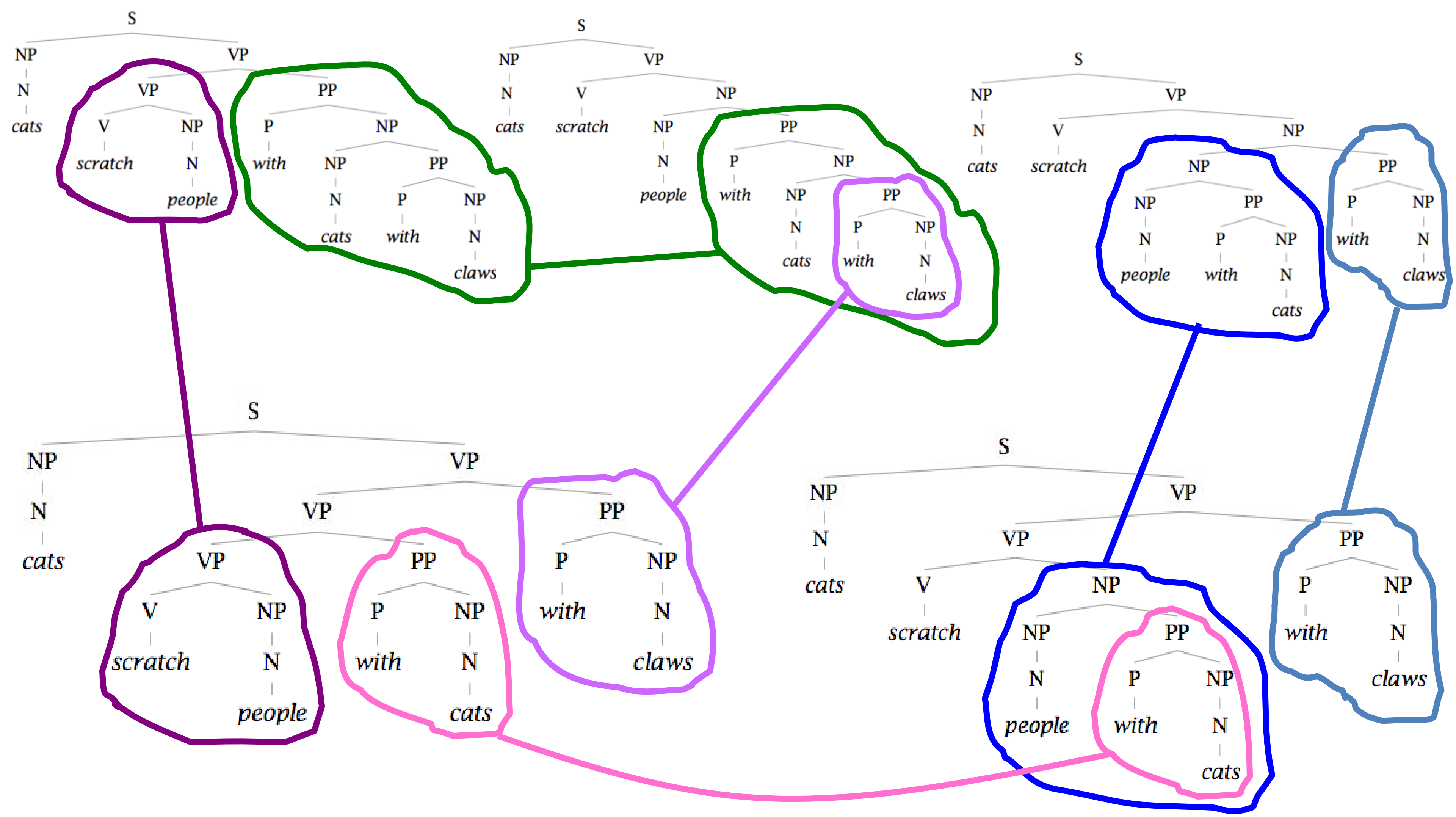






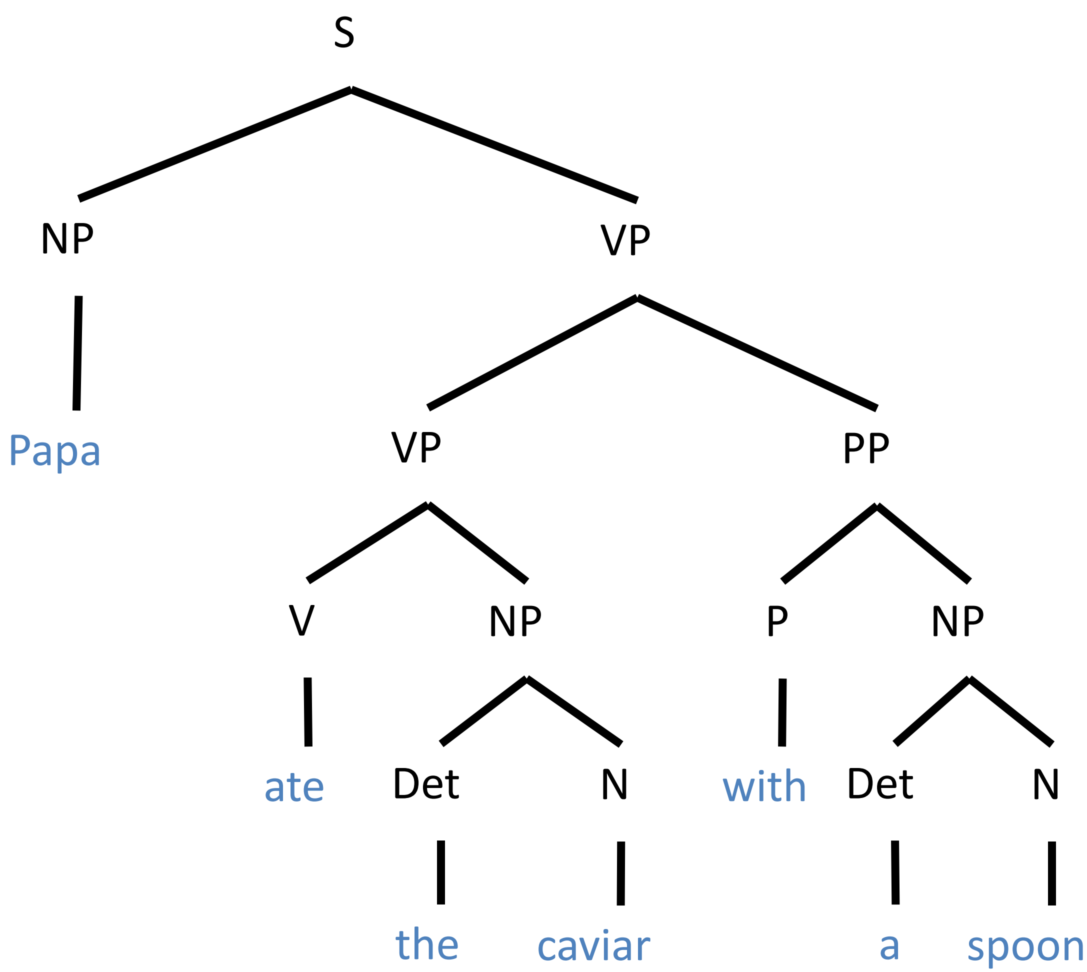


Two problems to solve:
1. Avoid repeated work…



Two problems to solve:
2. Ambiguity - Choosing the correct parse

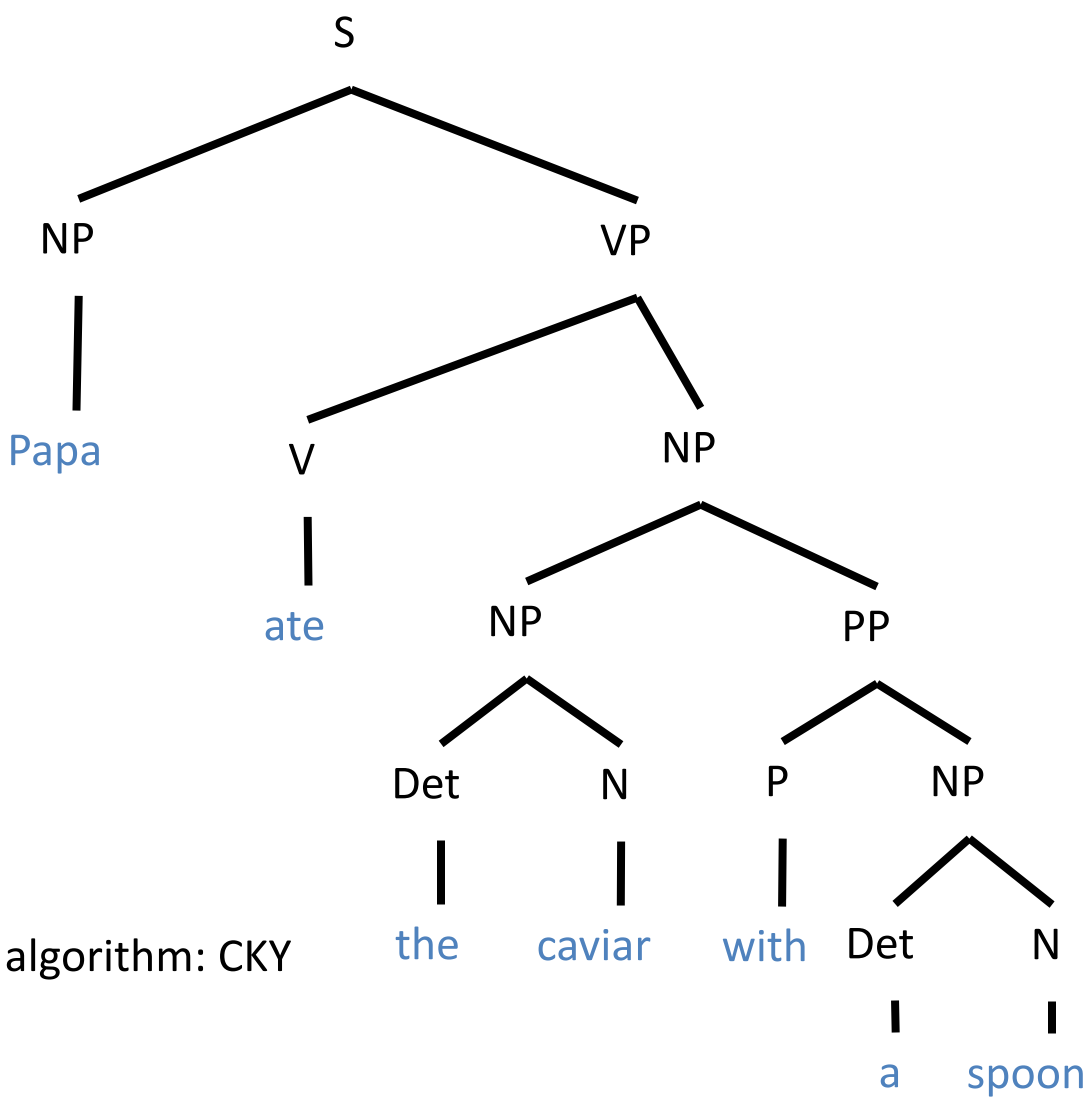
- S → NP VP
- NP → Det N
- NP → NP PP
- VP → V NP
- VP → VP PP
- PP → P NP



- NP → Papa
- N → caviar
- N → spoon
- V → spoon
- V → ate
- P → with
- Det → the
- Det → a

Two problems to solve:
2. Ambiguity - Choosing the correct parse

S → NP VP
NP → Det N
NP → NP PP
VP → V NP
VP → VP PP
PP → P NP



NP → Papa
N → caviar
N → spoon
V → spoon
V → ate
P → with
Det → the
Det → a

→ need an efficient algorithm: CKY

Syntax and Grammars

CFGs and PCFGs

A phrase structure grammar

Grammar rules

$S \rightarrow NP\ VP$

$VP \rightarrow V\ NP$

$VP \rightarrow V\ NP\ PP$ **n-ary (n=3)**

$NP \rightarrow NP\ NP$ **binary**

$NP \rightarrow NP\ PP$

$NP \rightarrow N$ **unary**

$PP \rightarrow P\ NP$

people fish tanks

people fish with rods

Lexicon

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

- A grammar G generates a language L.

Probabilistic – or stochastic – Context-free Grammars (PCFGs)

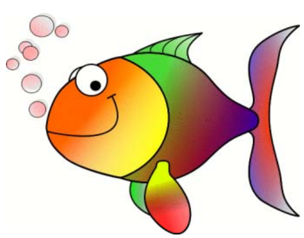
- G = (T, N, S, R, P)
  - T is a set of terminal symbols
  - N is a set of nonterminal symbols
  - S is the start symbol (S ∈ N)
  - R is a set of rules/productions of the form $X \rightarrow \gamma$
  - P is a probability function
    - P: R $\rightarrow$ [0,1]
    - $$\forall X \in N, \sum_{X \rightarrow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

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

Example PCFG

S → NP VP	1.0
VP → V NP	0.6
VP → V NP PP	0.4
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



Getting the probablities:

- Get a large collection of parsed sentences (treebank)
- Collect counts for each non-terminal rule expansion in the collection
- Normalize
- Done

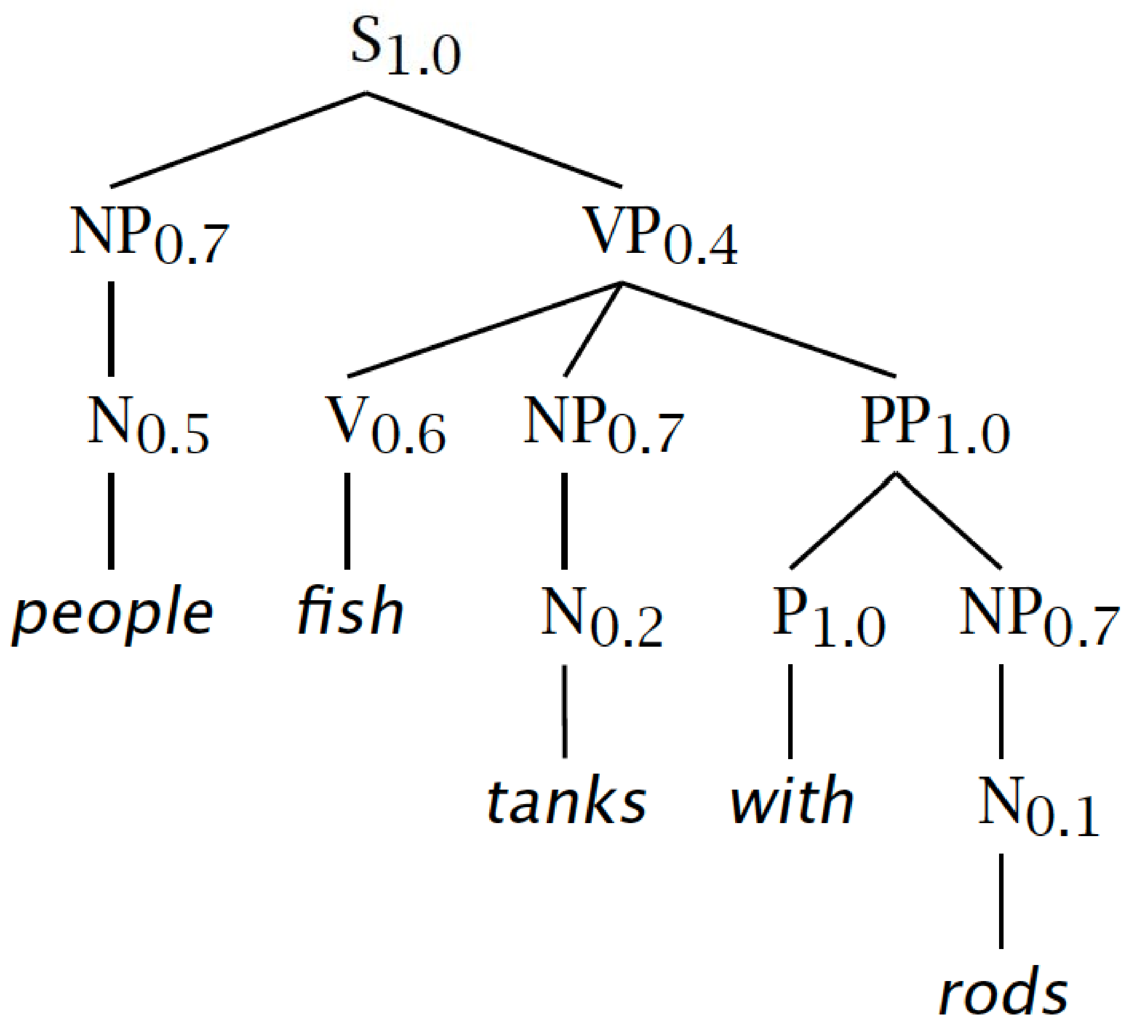
The probability of trees and strings

- 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

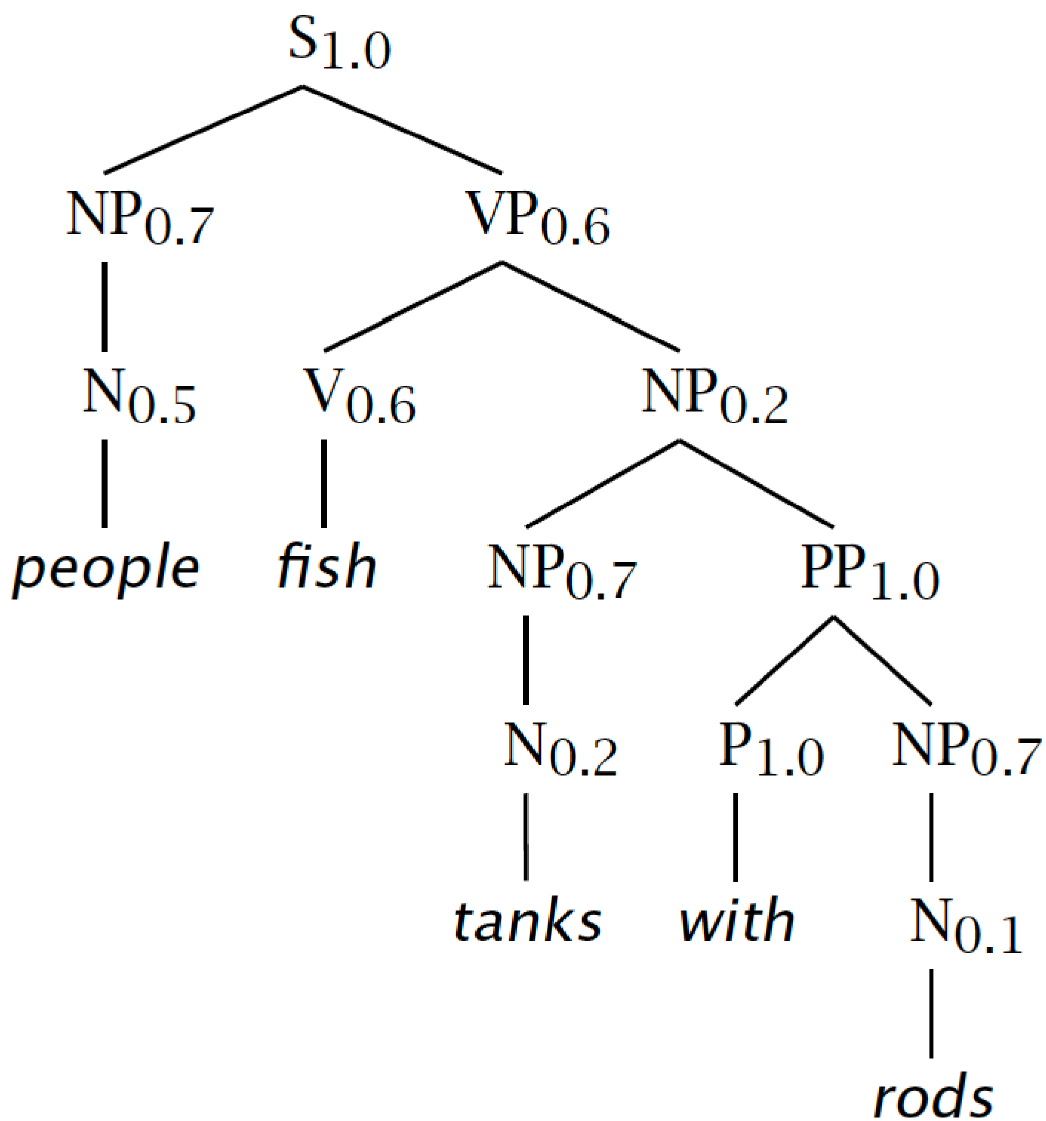
$$P(s) = \Sigma_j P(s, t) \text{ where } t \text{ is a parse of } s$$
$$= \Sigma_j P(t)$$

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
- $P(t_2) = 1.0 \times 0.7 \times 0.6 \times 0.5 \times 0.6 \times 0.2$
  $\times 0.7 \times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1$
  $= 0.00024696$

  Noun attach
- $P(s) = P(t_1) + P(t_2)$
  $= 0.0008232 + 0.00024696$
  $= 0.00107016$
- PCFG would choose t1

Grammar Transforms

Restricting the grammar form for efficient parsing

Chomsky Normal Form

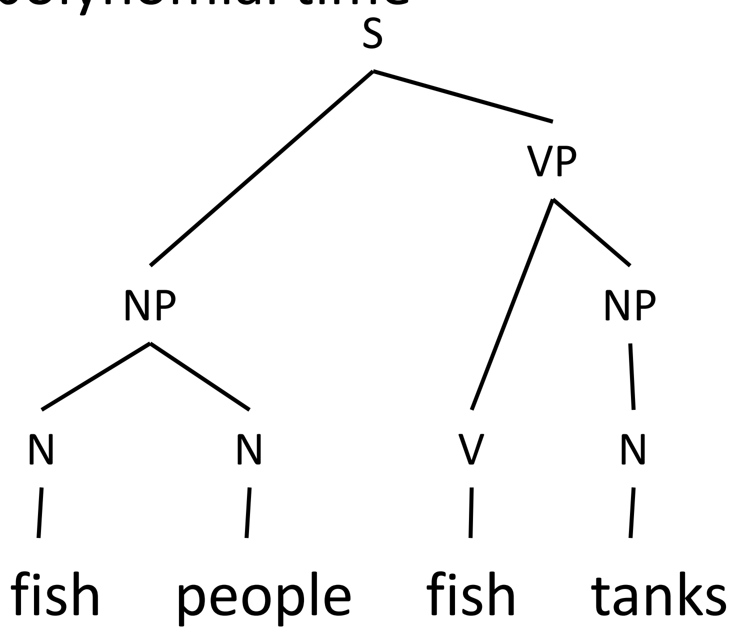
- All rules are of the form $X \rightarrow Y\ Z$ or $X \rightarrow w$
  - $X, Y, Z \in N$ and $w \in T$
- A transformation to this form doesn't change the weak generative capacity of a CFG
  - That is, it recognizes the same language
    - But maybe with different trees
- Empties and unaries are removed recursively
  NP $\rightarrow$ *e* emtpy rule *(imperative w/ empty subject: fish!)*
  NP $\rightarrow$ N unary rule
- n-ary rules (for n>2) are divided by introducing new nonterminals: A -> B C D A -> B @C @C -> C D

CKY Parsing

Polynomial time parsing of
(P)CFGs

Dynamic Programming

- We need a method that fills a table with partial results that
  - Does not do (avoidable) repeated work
  - Solves an exponential problem in (approximately) polynomial time

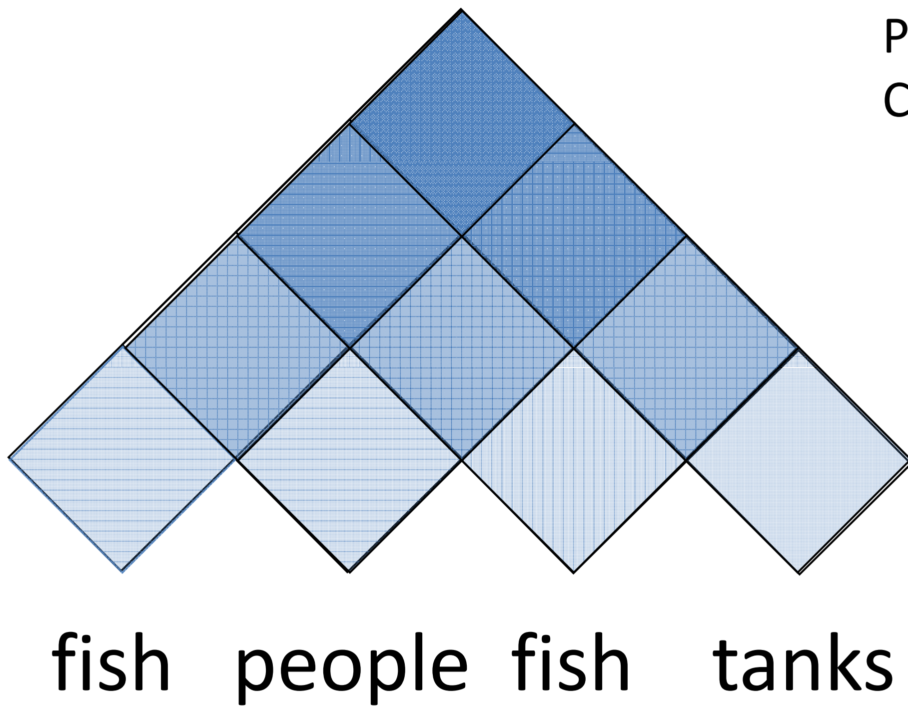


PCFG

Rule	**Prob θ_i**
S $\rightarrow$ NP VP	θ_0
NP $\rightarrow$ NP NP	θ_1
…	
N $\rightarrow$ fish	θ_{42}
N $\rightarrow$ people	θ_{43}
V $\rightarrow$ fish	θ_{44}

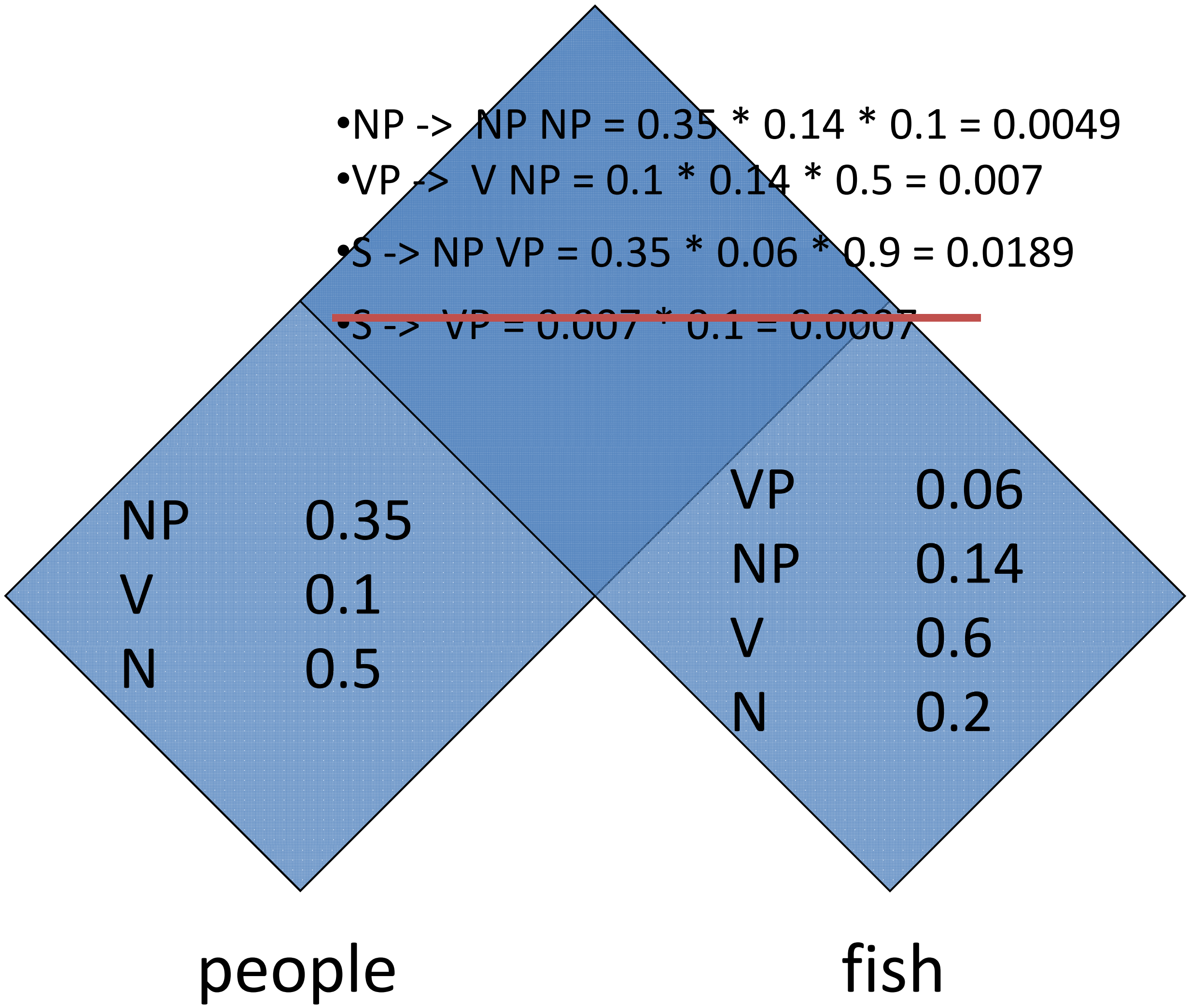
…

Cocke-Kasami-Younger (CKY) Constituency Parsing



Parsing chart
Cells over spans of words

Viterbi (Max) Scores



Just store best way of making S

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

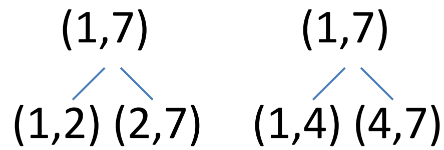
Extended CKY parsing

- Original CKY only for CNF
  - Unaries can be incorporated into the algorithm easily
- Binarization is *vital*
  - Without binarization, you don’t get parsing cubic in the length of the sentence and in the number of nonterminals in the grammar

The CKY algorithm (1960/1965) … extended to unaries

```
function CKY(words, grammar) returns [most_probable_parse,prob]
  score = new double[#(words)+1][#(words)+1][#(nonterms)]
  back = new Pair[#(words)+1][#(words)+1][#nonterms]]
  for i=0; i<#(words); i++
    for A in nonterms
      if A -> words[i] in grammar
        score[i][i+1][A] = P(A -> words[i])
    //handle unaries
    boolean added = true
    while added
      added = false
      for A, B in nonterms
        if score[i][i+1][B] > 0 && A->B in grammar
          prob = P(A->B)*score[i][i+1][B]
          if prob > score[i][i+1][A]
            score[i][i+1][A] = prob
            back[i][i+1][A] = B
            added = true
```

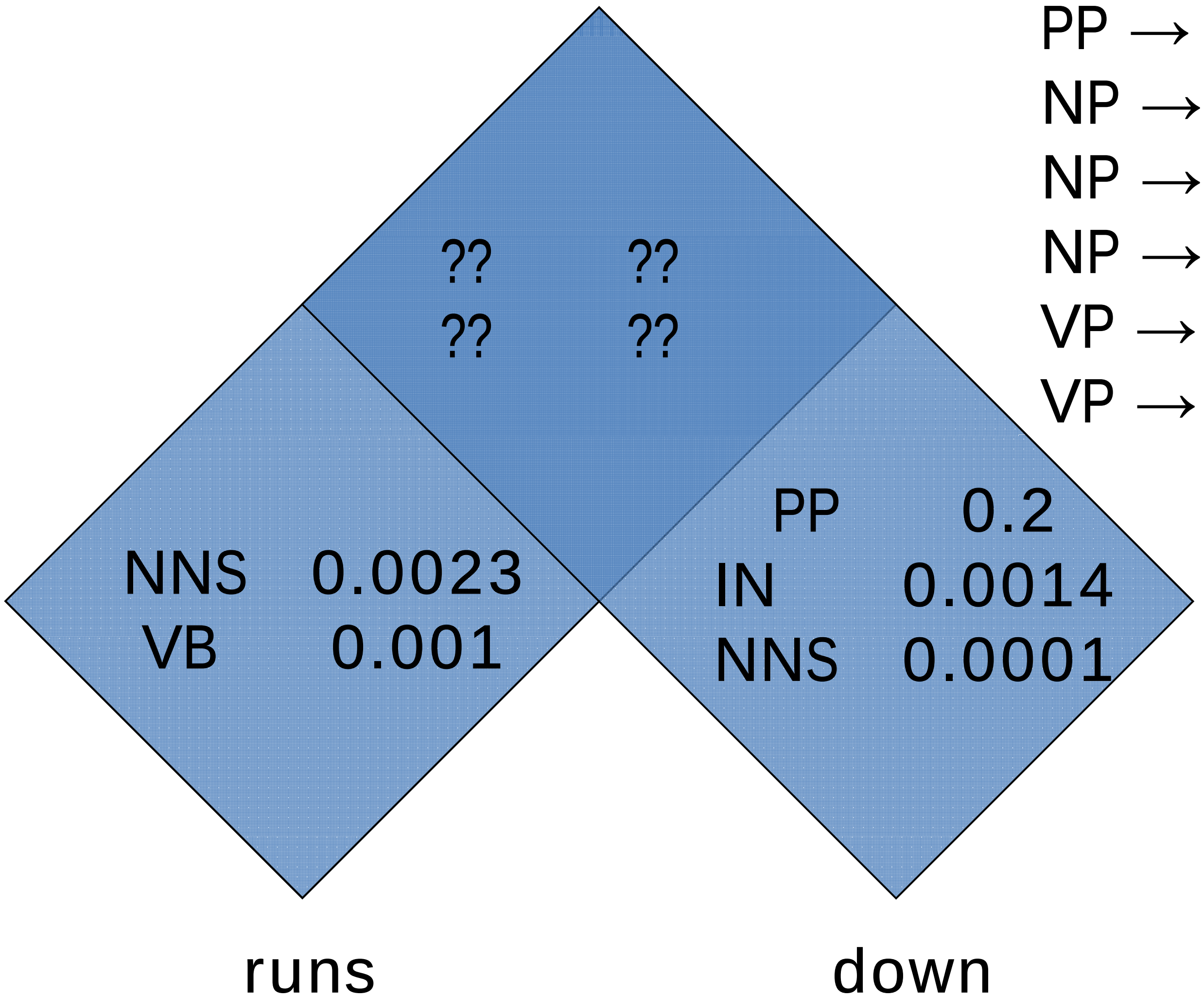
The CKY algorithm (1960/1965) … extended to unaries



O(n^3)
cubic

```
for span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
      for A,B,C in nonterms
        prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
        if prob > score[begin][end][A]
          score[begin]end][A] = prob
          back[begin][end][A] = new Triple(split,B,C)
    //handle unaries
    boolean added = true
    while added
      added = false
      for A, B in nonterms
        prob = P(A->B)*score[begin][end][B];
        if prob > score[begin][end][A]
          score[begin][end][A] = prob
          back[begin][end][A] = B
          added = true
return buildTree(score, back)
```

Quiz Question!



PP → IN	0.002
NP → NNS NNS	0.01
NP → NNS NP	0.005
NP → NNS PP	0.01
VP → VB PP	0.045
VP → VB NP	0.015

What constituents (with what probability can you make?

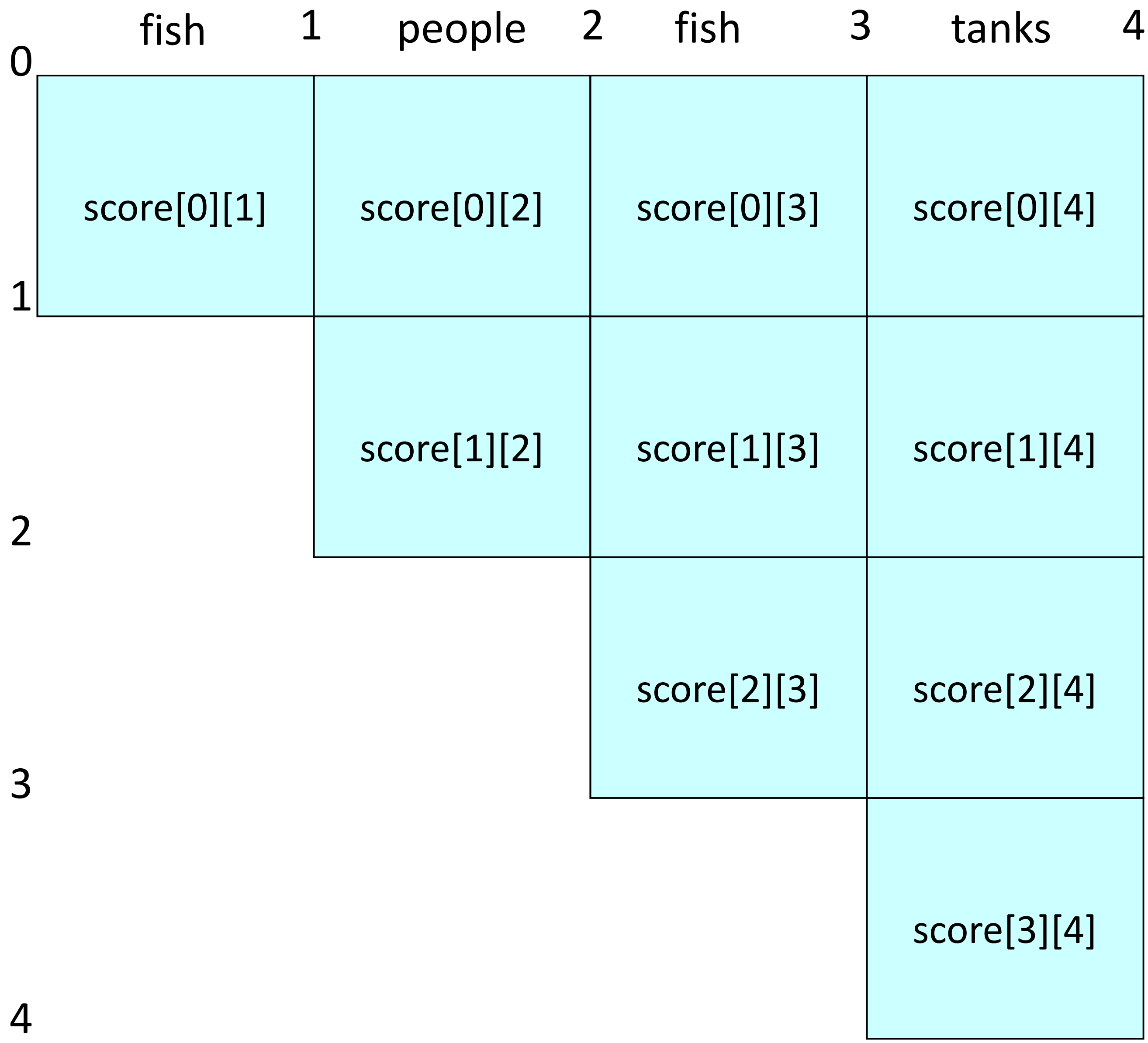
CKY Parsing

A worked example

The grammar

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


fish
people
fish
tanks
0
1
2
3
4
score[0][1]
score[0][2]
score[0][3]
score[0][4]
score[1][2]
score[1][3]
score[1][4]
score[2][3]
score[2][4]
score[3][4]

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


```
for i=0; i<#(words); i++
   for A in nonterms
     if A -> words[i] in grammar
       score[i][i+1][A] = P(A -> words[i]);
```

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

	0 fish 1	1 people 2	2 fish 3	3 tanks 4
0	N → fish 0.2 V → fish 0.6			
1		N → people 0.5 V → people 0.1		
2			N → fish 0.2 V → fish 0.6	
3				N → tanks 0.2 V → tanks 0.1

```
// handle unaries
boolean added = true
  while added
    added = false
    for A, B in nonterms
      if score[i][i+1][B] > 0 && A->B in grammar
        prob = P(A->B)*score[i][i+1][B]
        if(prob > score[i][i+1][A])
          score[i][i+1][A] = prob
          back[i][i+1][A] = B
          added = true
```

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

	fish (0–1)	people (1–2)	fish (2–3)	tanks (3–4)
0	N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006			
1		N → people 0.5 V → people 0.1 NP → N 0.35 VP → V 0.01 S → VP 0.001		
2			N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	
3				N → tanks 0.2 V → tanks 0.1 NP → N 0.14 VP → V 0.03 S → VP 0.003

```
prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
if (prob > score[begin][end][A])
    score[begin]end][A] = prob
    back[begin][end][A] = new Triple(split,B,C)
```

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

	fish 1	people 2	fish 3	tanks 4
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 → NP VP 0.00126		
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	
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 → NP VP 0.00378
3				N → tanks 0.2 V → tanks 0.1 NP → N 0.14 VP → V 0.03 S → VP 0.003
4				

```
//handle unaries
boolean added = true
while added
  added = false
  for A, B in nonterms
    prob = P(A->B)*score[begin][end][B];
    if prob > score[begin][end][A]
      score[begin][end][A] = prob
      back[begin][end][A] = B
      added = true
```

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

	fish (0–1)	people (1–2)	fish (2–3)	tanks (3–4)
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		
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	
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 0.1 NP → N 0.14 VP → V 0.03 S → VP 0.003
4				

```
for split = begin+1 to end-1
  for A,B,C in nonterms
    prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
    if prob > score[begin][end][A]
      score[begin]end][A] = prob
      back[begin][end][A] = new Triple(split,B,C)
```

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

	fish 1	people 2	fish 3	tanks 4
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	
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	
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 0.1 NP → N 0.14 VP → V 0.03 S → VP 0.003
4				

```
for split = begin+1 to end-1
  for A,B,C in nonterms
    prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
    if prob > score[begin][end][A]
      score[begin]end][A] = prob
      back[begin][end][A] = new Triple(split,B,C)
```

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

0 fish 1 people 2 fish 3 tanks 4

	1	2	3	4
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	
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 0.1 NP → N 0.14 VP → V 0.03 S → VP 0.003
4				

```
for split = begin+1 to end-1
  for A,B,C in nonterms
    prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
    if prob > score[begin][end][A]
      score[begin]end][A] = prob
      back[begin][end][A] = new Triple(split,B,C)
```

Rule	Prob
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

Rule	Prob
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

0 fish 1 people 2 fish 3 tanks 4

	1	2	3	4
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	N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	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 0.1 NP → N 0.14 VP → V 0.03 S → VP 0.003

3 split points
Same as before

At the end backtrace
to get highest prob parse

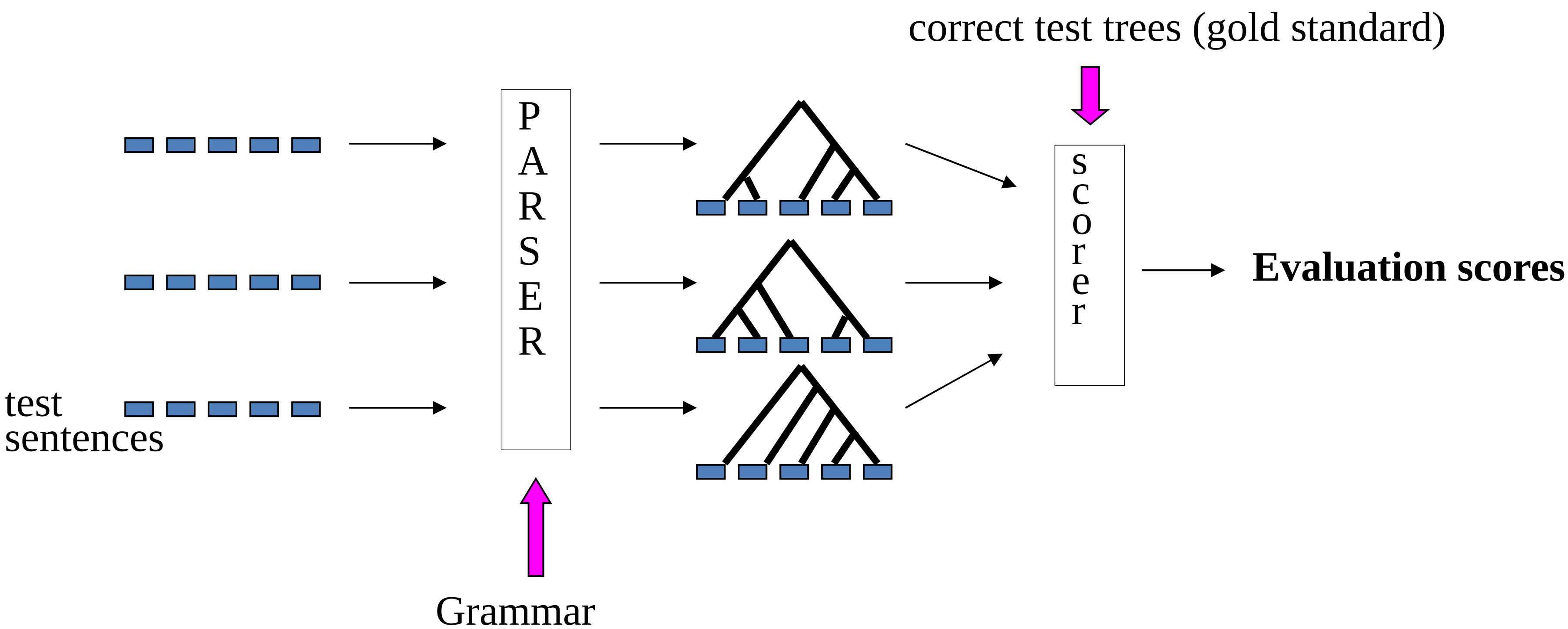
Actually store spans
S(0,4) -> NP(0,2) VP(2,4)

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

Parser Evaluation

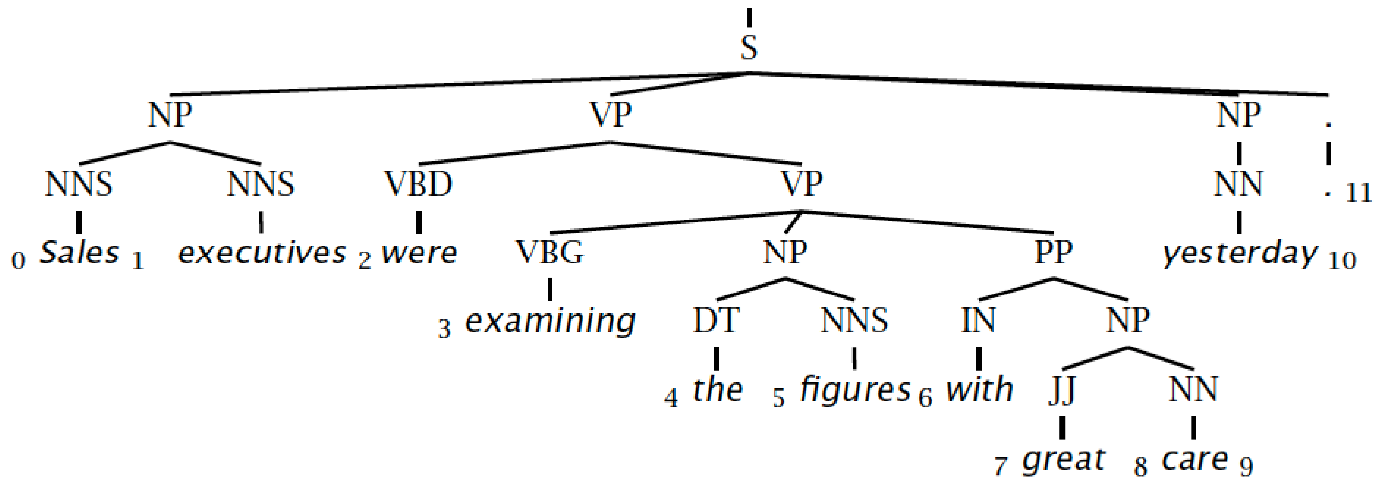
Measures to evaluate constituency and dependency parsing

Evaluating Parser Performance

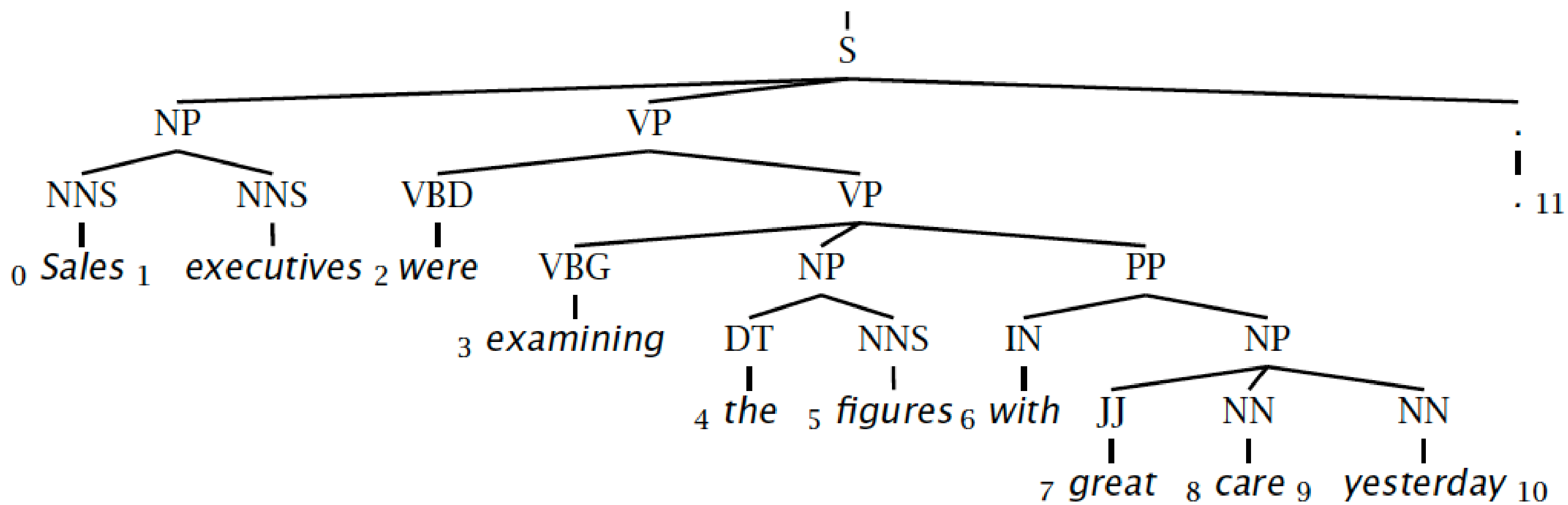


Evaluation of Constituency Parsing: bracketed P/R/F-score

Gold standard brackets: **S-(0:11), NP-(0:2)**, VP-(2:9), VP-(3:9), **NP-(4:6)**, PP-(6-9), NP-(7,9), NP-(9:10)



Candidate brackets: **S-(0:11), NP-(0:2)**, VP-(2:10), VP-(3:10), **NP-(4:6)**, PP-(6-10), NP-(7,10)



Evaluation of Constituency Parsing: bracketed P/R/F-score

Gold standard brackets:

S-(0:11), NP-(0:2), VP-(2:9), VP-(3:9), **NP-(4:6)**, PP-(6-9), NP-(7,9), NP-(9:10)

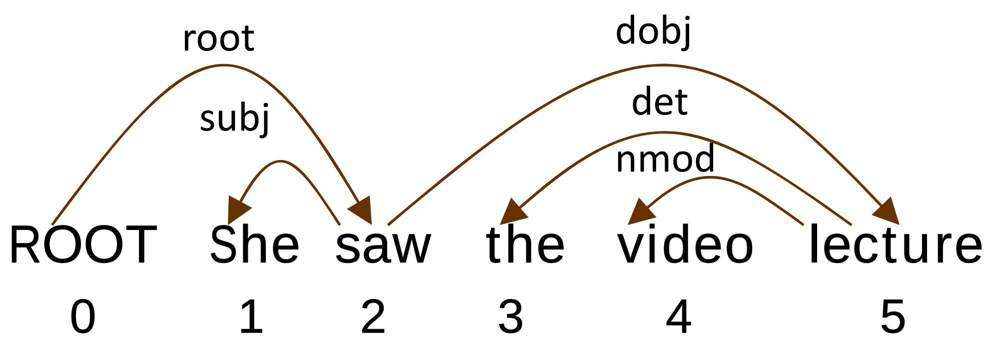
Candidate brackets:

S-(0:11), **NP-(0:2)**, VP-(2:10), VP-(3:10), **NP-(4:6)**, PP-(6-10), NP-(7,10)

Labeled Precision	3/7 = 42.9%
Labeled Recall	3/8 = 37.5%
F1	40.0%

(Parseval measures)

Evaluation of Dependency Parsing: (labeled) dependency accuracy



Unlabeled Attachment Score (UAS)
Labeled Attachment Score (LAS)
Label Accuracy (LA)

UAS = 4 / 5 = 80%
LAS = 2 / 5 = 40%
LA = 3 / 5 = 60%

Gold

1	She	2	subj
2	saw	0	root
3	the	5	det
4	video	5	nmod
5	lecture	2	dobj

Parsed

1	She	2	subj
2	saw	0	root
3	the	4	det
4	video	5	vmod
5	lecture	2	iobj

How good are PCFGs?

- Simple PCFG on Penn WSJ: about 73% F1
- Strong independence assumption
  - S -> VP NP   (e.g. independent of words)
- Potential issues:
  - Agreement
  - Subcategorization

Agreement

- This dog
- Those dogs

- This dog eats
- Those dogs eat

- *This dogs
- *Those dog

- *This dog eat
- *Those dogs eats

For example, in English, determiners and the head nouns in NPs have to agree in their number.

- Our earlier NP rules are clearly deficient since they don't capture this constraint
  - *NP → DT N*
    - Accepts, and assigns correct structures, to grammatical examples (*this flight*)
    - But its also happy with incorrect examples (*these flight)
  - Such a rule is said to *overgenerate.*

Subcategorization

- Sneeze: John sneezed
- Find: Please find [a flight to NY]$_{NP}$
- Give: Give [me]$_{NP}$[a cheaper fare]$_{NP}$
- Help: Can you help [me]$_{NP}$[with a flight]$_{PP}$
- Prefer: I prefer [to leave earlier]$_{TO\text{-}VP}$
- Told: I was told [United has a flight]$_{S}$
- …

- *John sneezed the book
- *I prefer United has a flight
- *Give with a flight

- Subcat expresses the constraints that a predicate (verb for now) places on the number and type of the argument it wants to take

Possible CFG Solution

- Possible solution for agreement.
- Can use the same trick for all the verb/VP classes.

- SgS -> SgNP SgVP
- PlS -> PlNp PlVP
- SgNP -> SgDet SgNom
- PlNP -> PlDet PlNom
- PlVP -> PlV NP
- SgVP ->SgV Np
- …

CFG Solution for Agreement

- It works and stays within the power of CFGs
- But its ugly
- And it doesn’t scale all that well because of the interaction among the various constraints explodes the number of rules in our grammar.

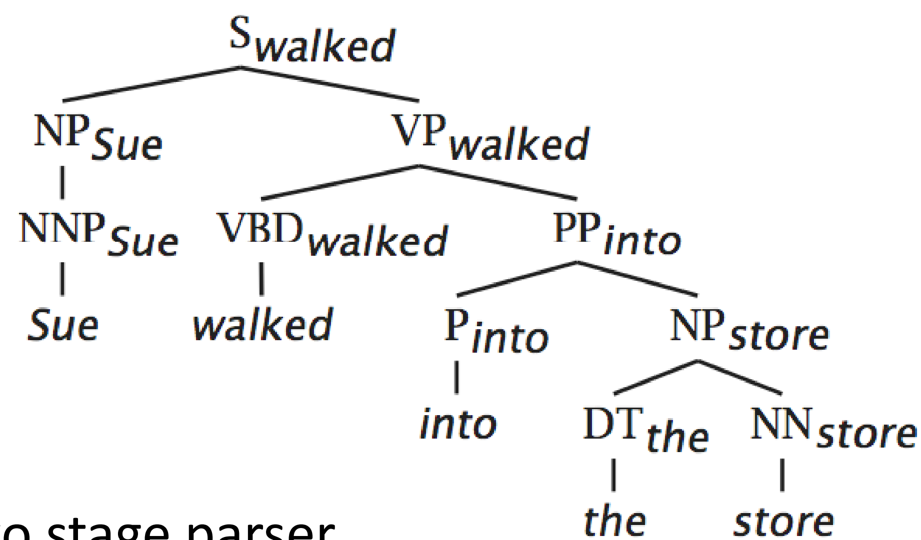
- Alternatives: head-lexicalized PCFG, parent annotation, more expressive grammar formalism (HPSG, TAG, …)

  → lexicalized PCFGs reach ~88% Fscore (on PT WSJ)

(Head) Lexicalization of PCFGs

[Magerman 1995, Collins 1997; Charniak 1997]

- The head word of a phrase gives a good representation of the phrase's structure and meaning
- Puts the properties of words back into a PCFG



- Charniak Parser: two stage parser
  1. lexicalized PCFG (generative model) generates n-best parses
  2. disambiguator (discriminative MaxEnt model) to choose parse

Dependency Parsing

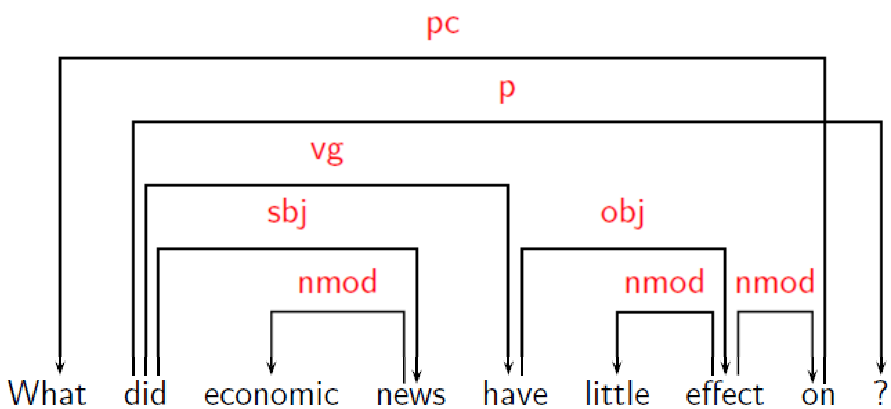
A brief overview

Dependency Parsing

- A dependency structure can be defined as a **directed graph G**, consisting of:
  - a set V of nodes,
  - a set E of (labeled) arcs (edges)
- A graph G should be: connected (For every node i there is a node j such that i → j or j → i), acyclic (no cycles) and single-head constraint (have one parent, except root token).
- The dependency approach has a number of advantages over full phrase-structure parsing.
  - Better suited for free word order languages
  - Dependency structure often captures the syntactic relations needed by later applications
    - CFG-based approaches often extract this same information from trees anyway

Dependency Parsing

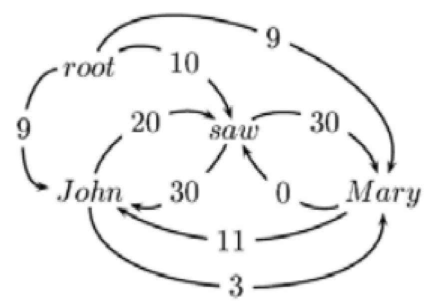
- Modern dependency parsers can produce either projective or non-projective dependency structures



- Non-projective structures have crossing edges
  - long-distance dependencies
  - free word order languages, e.g. Dutch
    vs. English: only specific adverbials before VPs:
    - Hij heeft waarschijnlijk een boek gelezen He probably read a book.
    - Hij heeft gisteren een boek gelezen *He yesterday read a book.

Dependency Parsing

- There are two main approaches to dependency parsing
  - Dynamic Programming:
    Optimization-based approaches that search a space of trees for the tree that *best* matches some criteria
    - Treat dependencies as constituents, algorithm similar to CKY plus improved version by Eisner (1996).
    - Score of a tree = sum of scores of edges
      find best tree: Maximum spanning tree algorithms
    - Examples: MST (Ryan McDonald), Bohnet parser



  - Deterministic parsing:
    Shift-reduce approaches that greedily take actions based on the current word and state (abstract machine, use classifier to predict next parsing step)
    - Example: Malt parser (Joakim Nivre)

Tools

- Charniak Parser (constituent parser with discriminative reranker)
- Stanford Parser (provides constituent and dependency trees)
- Berkeley Parser (constituent parser with latent variables)
- MST parser (dependency parser, needs POS tagged input)
- Bohnet's parser (dependency parser, needs POS tagged input)
- Malt parser (dependency parser, needs POS tagged input)

Summary

- Context-free grammars can be used to model various facts about the syntax of a language.
- When paired with parsers, such grammars constitute a critical component in many applications.
- Constituency is a key phenomena easily captured with CFG rules.
  - But agreement and subcategorization do pose significant problems
- Treebanks pair sentences in corpus with their corresponding trees.
- CKY is an efficient algorithm for CFG parsing
- Alternative formalism: Dependency structure

Reference & credits

- Jurafsky & Manning (2nd edition) chp 12, 13 & 14
- Thanks to Jim H. Martin, Dan Jurafsky, Christopher Manning, Jason Eisner, Rada Mihalcea for making their slides available
  - http://www.cs.colorado.edu/~martin/csci5832/lectures_and_readings.html
  - http://www.nlp-class.org (coursera.org)
  - http://www.cse.unt.edu/~rada/CSCE5290/
  - http://www.cs.jhu.edu/~jason/465/