复旦大学大数据学院 School of Data Science, Fudan University 魏忠钰

Statistical Natural Language Parsing

November 29th, 2017

Adapted from Stanford CS124U

NLP Startups

森亿智能 SYNYI·AI

核心技术 团队 应用场景 案例 合作

核心技术

Technology

坚持技术创新创造终极价值

森亿以自然语言处理、机器学习、深度学习等技术为核心，对“杂乱无章”的医疗健康大数据进行深度治理和挖掘，释放数据价值，帮助客户发掘新机遇和新洞察，助力医疗革新。

NLP Startups

森亿智能完成5500万元A轮融资，深度挖掘医疗大数据 原创

2017-11-27 07:25 投资界 Isabel 阅读：769 ☆收藏

涉及机构

红杉

华岩资本

真格基金

投资界APP

> 据其官网资料显示，森亿成立于2016年4月，以自然语言处理、机器学习、深度学习等技术为核心，对医疗健康大数据，尤其是临床病历方面，进行深度治理和挖掘，释放数据价值，帮助客户发掘新机遇和新洞察，助力医疗革新。

投资界（微信ID: pedaily2012）11月27日消息，森亿智能对外宣布，公司已获得5500万元的A轮融资，由红杉资本中国基金领投，中电健康基金战略投资。此前，公司曾于今年初宣布获得真格基金领投、华岩资本和树兰医疗跟投的近千万元融资。

据了解，本轮融资资金将主要用于加强在医学自然语言处理、医疗数据治理、机器学习领域的技术优势，进行团队和业务扩张，并在产品研发与参与行业标准规范等方面进行持续投入。

据其官网资料显示，森亿成立于2016年4月，以自然语言处理、机器学习、深度学习等技术为核心，对医疗健康大数据，尤其是临床病历方面，进行深度治理和挖掘，释放数据价值，帮助客户发掘新机遇和新洞察，助力医疗革新。

团队由国内外著名院校、知名企业的精英构成，集合了人工智能、医学、医学信息学、医疗信息化等领域专家，核心团队技术成果已申报多项国家、市级项目，累计在国际顶级期刊发表20余篇论文，并多次在国际顶级医学会议上发言。

【原创】吴恩达团队用深度学习预测死亡概率，改善临终关怀

2017-11-28 16:00 发布于 原创 转自：论智

一项调查表明，如果可以的话，大约有80%的美国人想在家度过人生中的最后时光，但只有20%的人这样做了。事实上，高达60%的死亡都发生在急诊室里，许多病人在弥留之际仍接受了类似手术等侵入性治疗。虽说配有临终关怀的医院数量从2008年的53%上升到67%，但能够得到真正关怀的病人非常少。不仅是由于专业人员的短缺造成了这种差距，技术因素也是阻碍病人获得临终关怀的重要原因。

Improving Palliative Care with Deep Learning

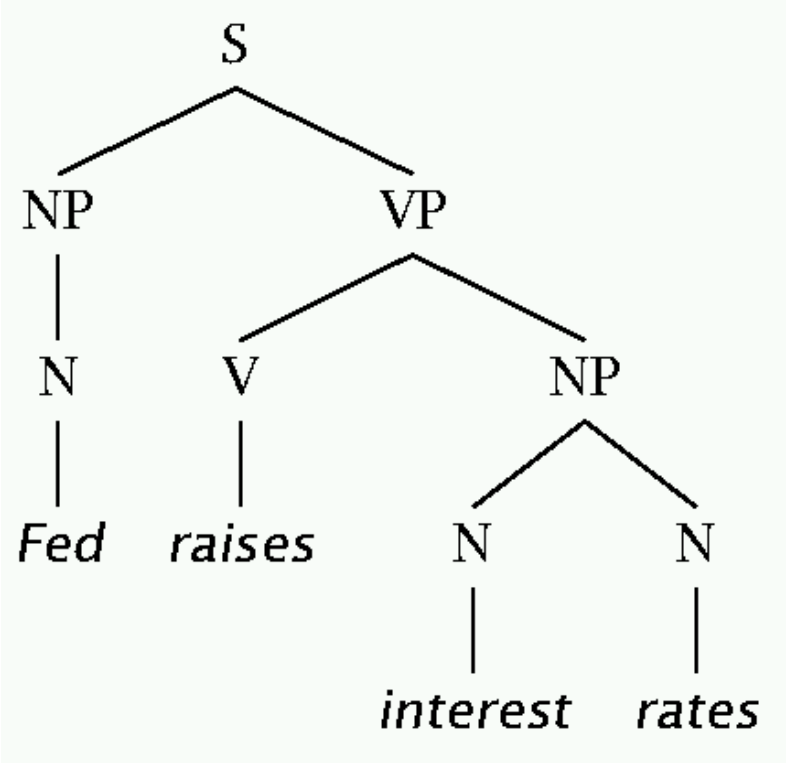
Anand Avati*, Kenneth Jung†, Stephanie Harman‡, Lance Downing†, Andrew Ng* and Nigam H. Shah†
*Dept of Computer Science, Stanford University
Email: {avati,ang}@cs.stanford.edu
†Center for Biomedical Informatics Research, Stanford University
Email: {kjung,ldowning,nigam}@stanford.edu
‡Dept of Medicine, Stanford University School of Medicine
Email: {smharman}@stanford.edu

研究人员站在临终关怀团队的角度，收集病人的疾病类型、阶段、严重程度（ICU或非ICU）、年龄等信息，输入电子病历和日期，系统就能根据之前一年的病人医疗数据预测未来一年的死亡概率。研究人员将其看作是一个二分类问题，构建了一个监督式的深度学习模型。

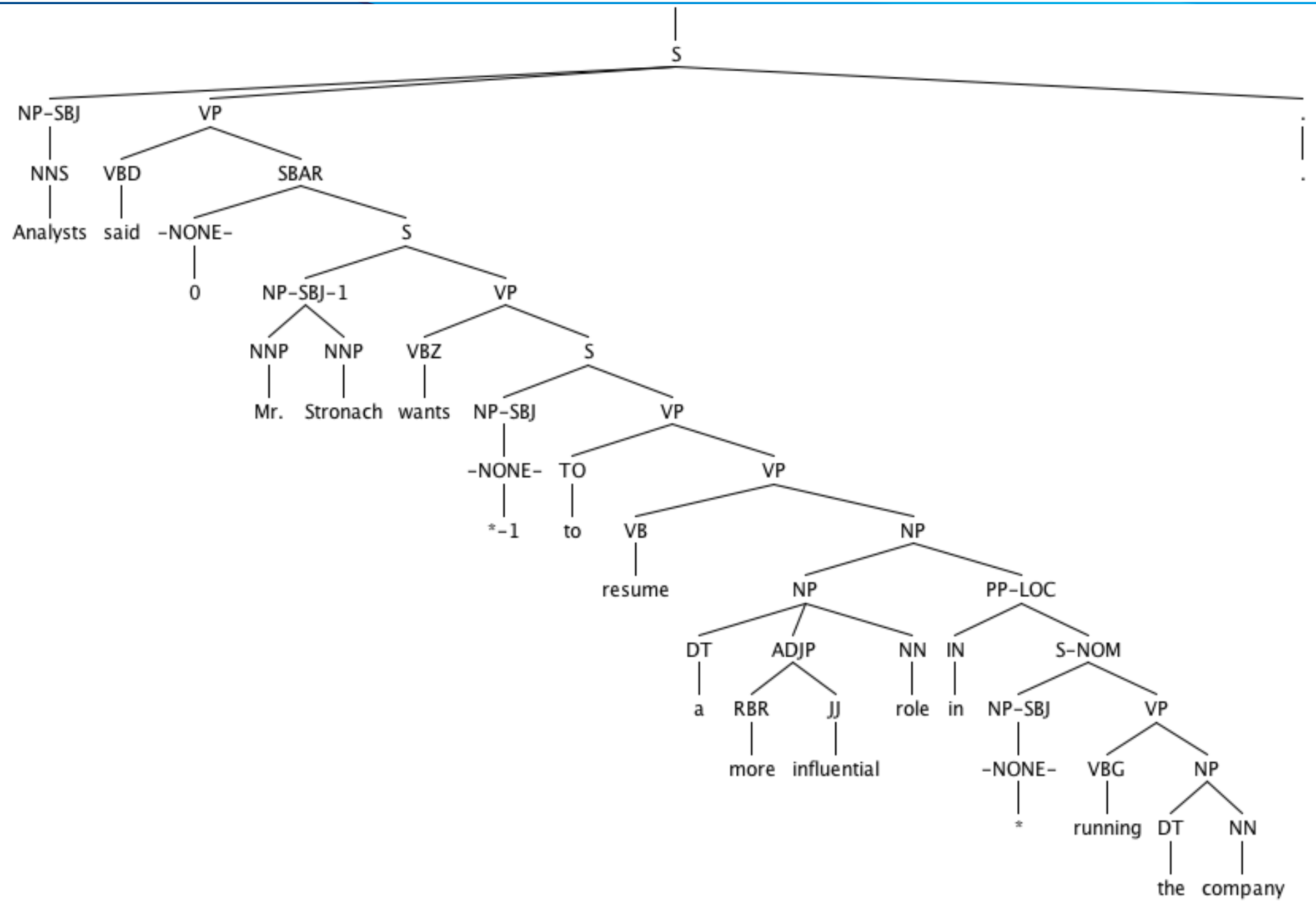
研究人员收集了斯坦福大学数据整合转化研究（STRIDE）1995年至2014年大约200万名成人和儿童患者的数据，包括诊断结论、治疗过程、用药情况和遇到的问题。然后数据被转化为一个有13654个维度的特征向量。经过训练后的模型在交叉验证时的ROC曲线下面积为0.93，平均精度得分为0.69。

Constituency (phrase structure)

- Phrase structure organizes words into nested constituents.

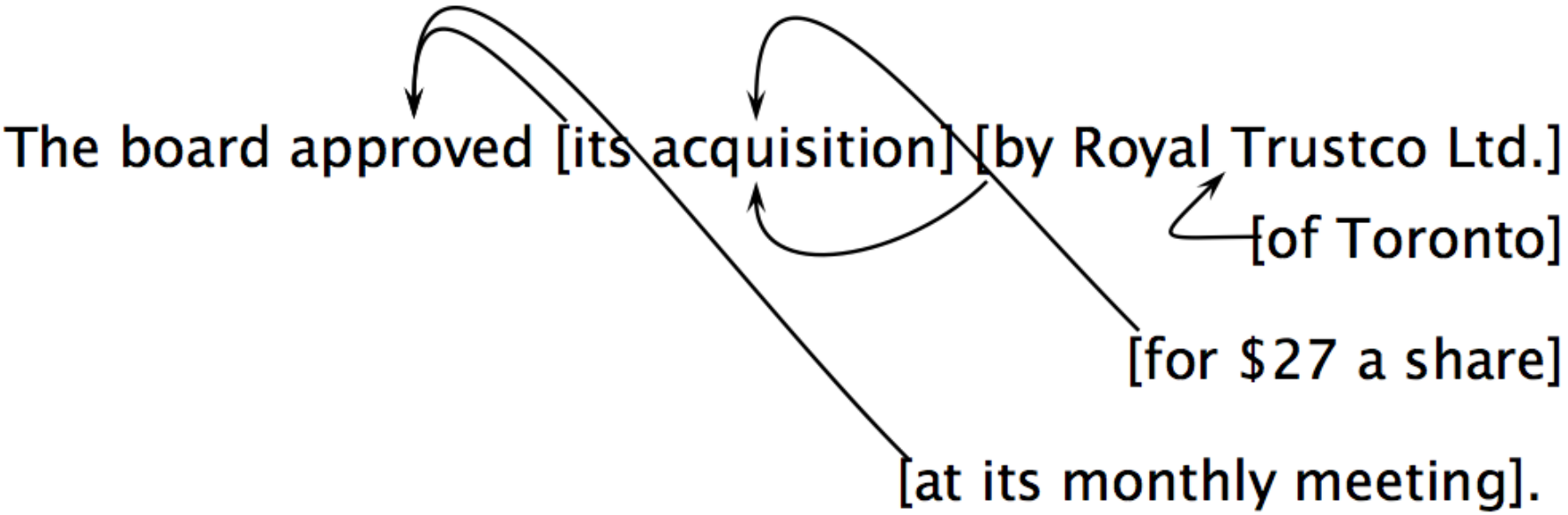


Parsing Tree Example



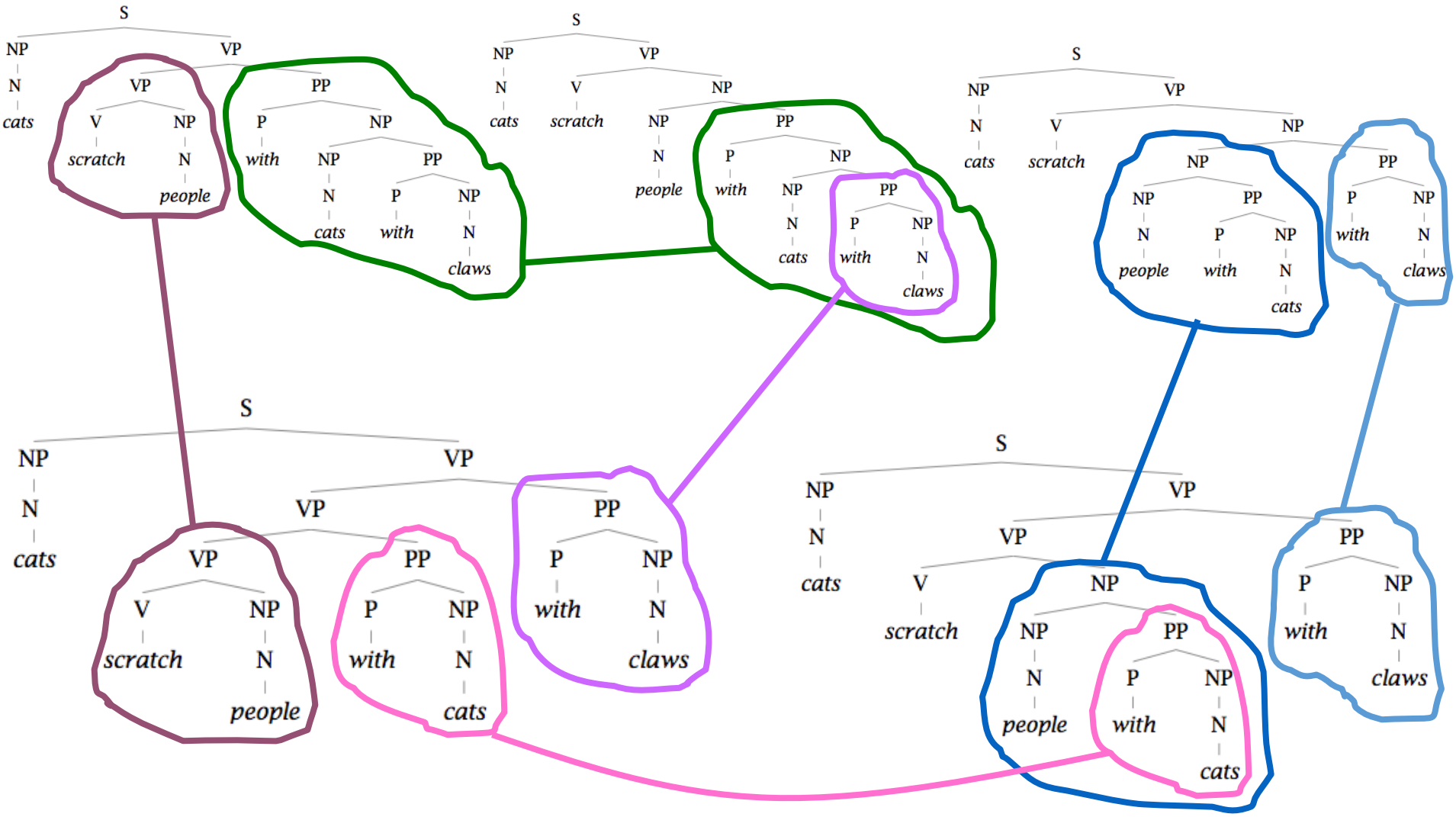
Attachment ambiguities

- A key parsing decision is how we 'attach' various constituents
  - PPs, adverbial or participial phrases, infinitives, coordinations, etc.



- Catalan numbers: $C_n = (2n)!/[(n+1)!n!]$
- An exponentially growing series, which arises in many tree-like contexts

Two problems to solve: 1. Repeated work…



Two problems to solve: 2. Choosing the correct parse

- Words are good predictors of attachment
  - Moscow sent more than 100,000 soldiers into Afghanistan …
  - Sydney Water breached an agreement with NSW Health …
- Our statistical parsers will try to exploit such statistics.

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.

A phrase structure grammar

$S \rightarrow NP\ VP$

$VP \rightarrow V\ NP$

$VP \rightarrow V\ NP\ PP$

$NP \rightarrow NP\ NP$

$NP \rightarrow NP\ PP$

$NP \rightarrow N$

$NP \rightarrow e$

$PP \rightarrow P\ NP$

$N \rightarrow$ *people*

$N \rightarrow$ *fish*

$N \rightarrow$ *tanks*

$N \rightarrow$ *rods*

$V \rightarrow$ *people*

$V \rightarrow$ *fish*

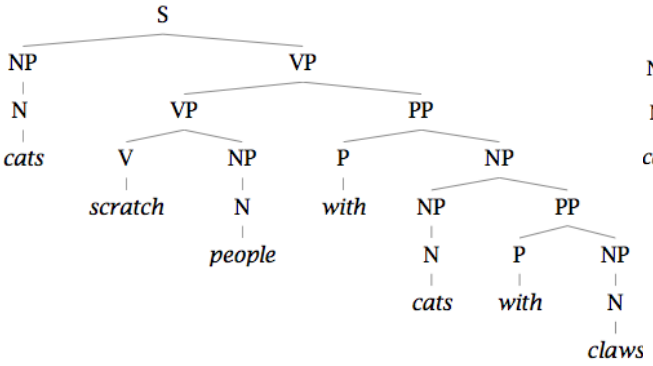
$V \rightarrow$ *tanks*

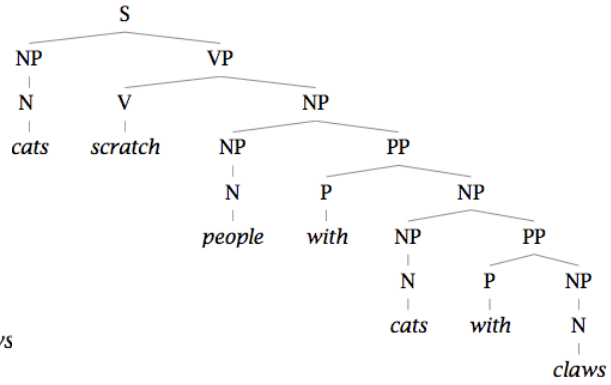
$P \rightarrow$ *with*

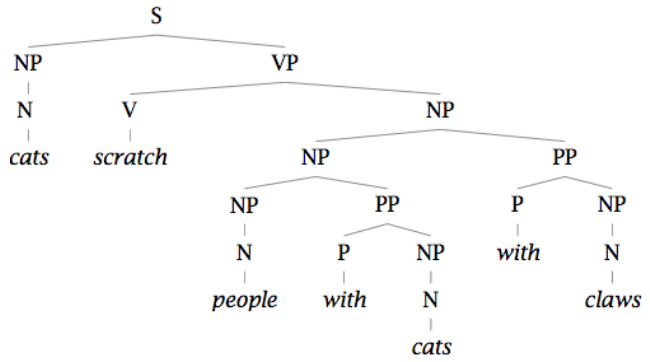
people fish tanks

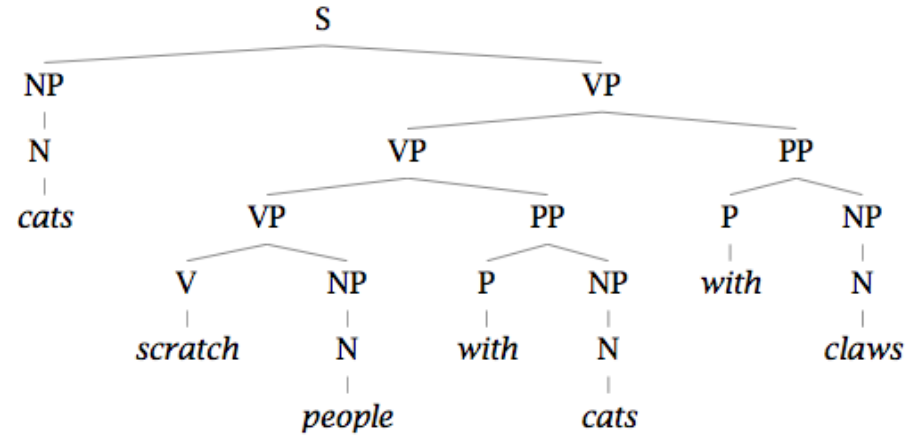
people fish with rods

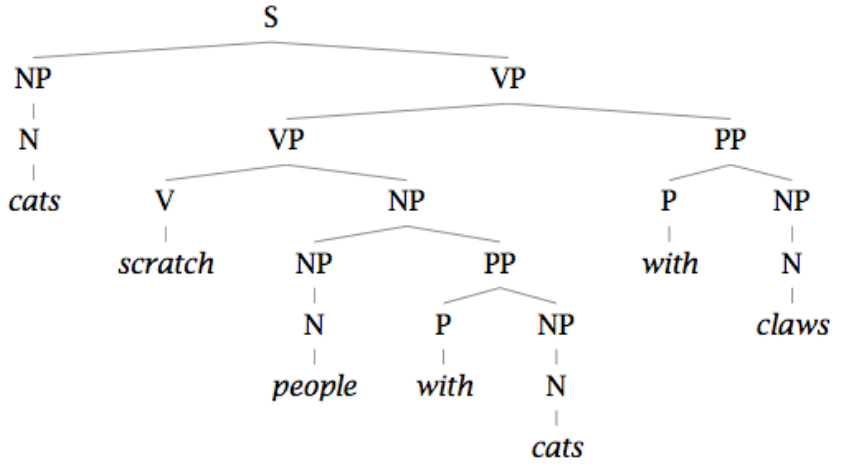
Many parses


S
NP
VP
N
VP
PP
cats
V
NP
P
NP
scratch
N
with
NP
PP
people
N
P
NP
cats
with
N
claws


S
NP
VP
N
V
NP
cats
scratch
NP
PP
N
P
NP
people
with
NP
PP
N
P
NP
cats
with
N
claws


S
NP
VP
N
V
NP
cats
scratch
NP
PP
NP
PP
P
NP
N
P
NP
with
N
people
with
N
claws
cats


S
NP
VP
N
VP
PP
cats
VP
PP
P
NP
V
NP
P
NP
with
N
scratch
N
with
N
claws
people
cats


S
NP
VP
N
VP
PP
cats
V
NP
P
NP
scratch
NP
PP
with
N
N
P
NP
people
with
N
claws
cats

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

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

A PCFG

S → NP VP	1.0	N → *people*	0.5
VP → V NP	0.6	N → *fish*	0.2
VP → V NP PP	0.4	N → *tanks*	0.2
NP → NP NP	0.1	N → *rods*	0.1
NP → NP PP	0.2	V → *people*	0.1
NP → N	0.7	V → *fish*	0.6
PP → P NP	1.0	V → *tanks*	0.3
		P → *with*	1.0

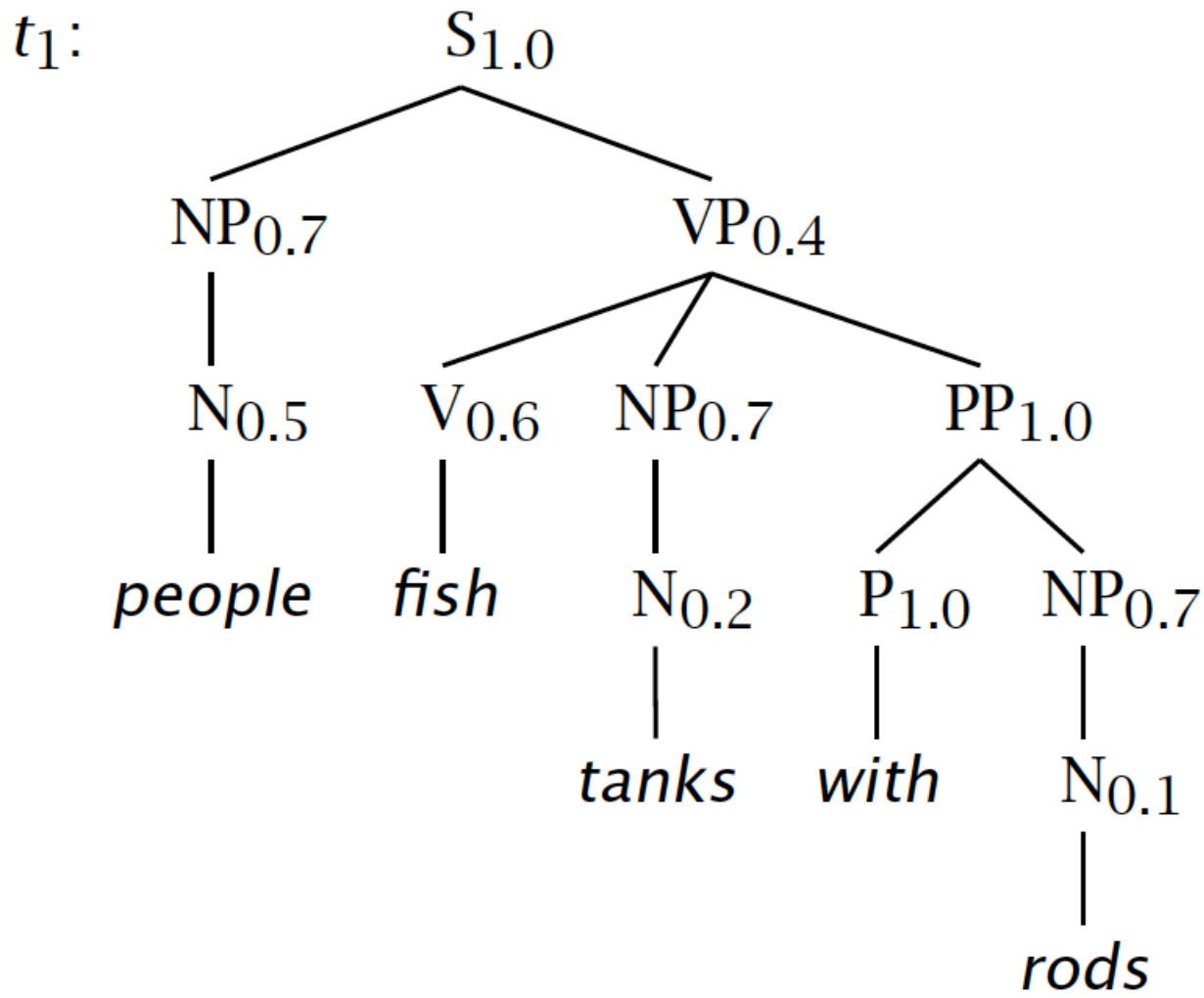
The rise of annotated data: The Penn Treebank

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

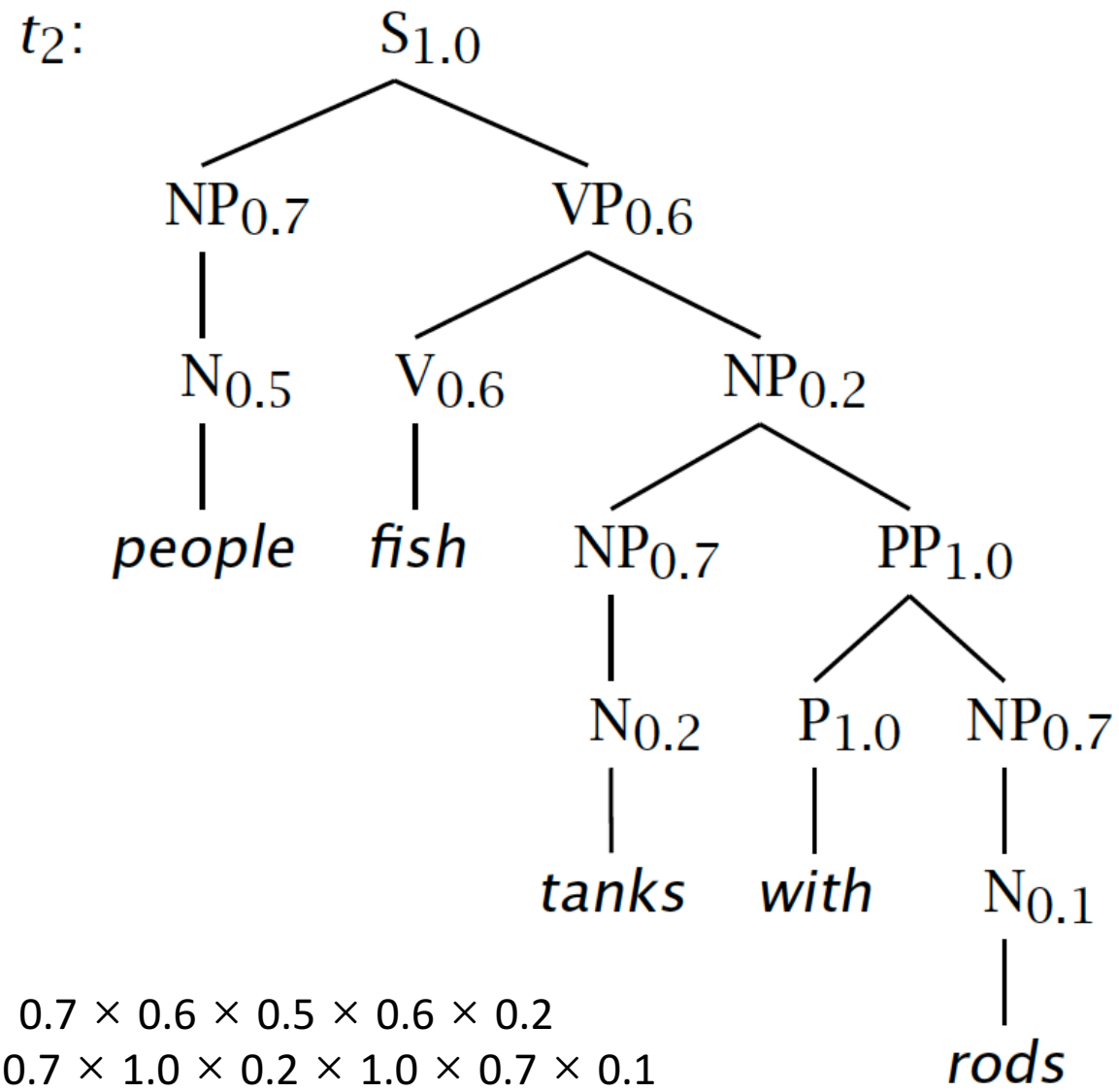
The probability of trees and strings

- 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

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



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



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

= 0.00024696

Tree and String Probabilities

- *s* = *people fish tanks with rods*
- $P(t_1) = 1.0 \times 0.7 \times 0.4 \times 0.5 \times 0.6 \times 0.7 \times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1$

  $= 0.0008232$

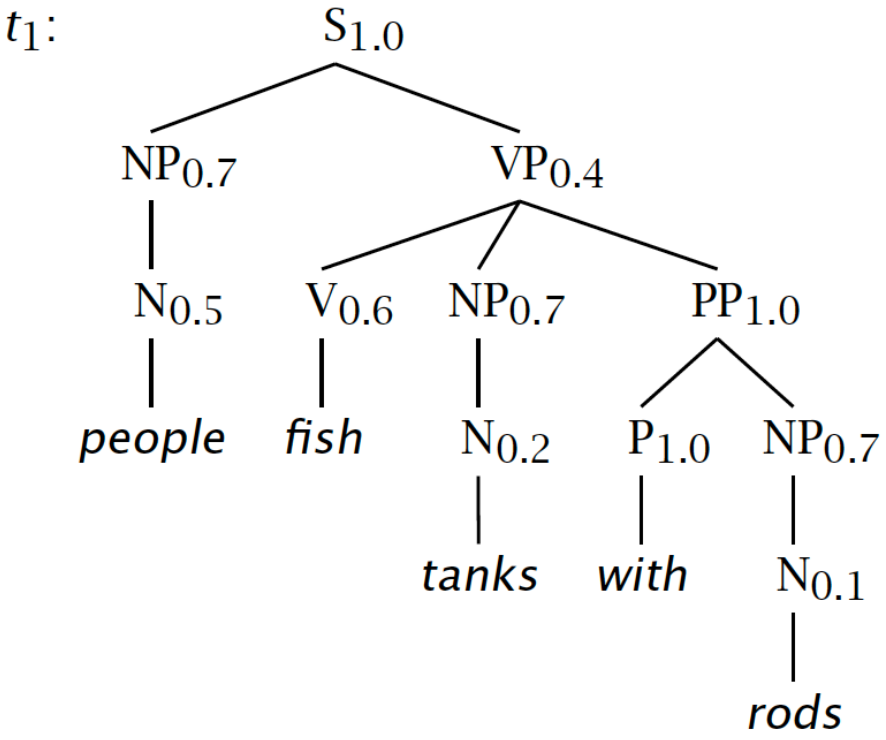
  Verb attach
- $P(t_2) = 1.0 \times 0.7 \times 0.6 \times 0.5 \times 0.6 \times 0.2 \times 0.7 \times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1$

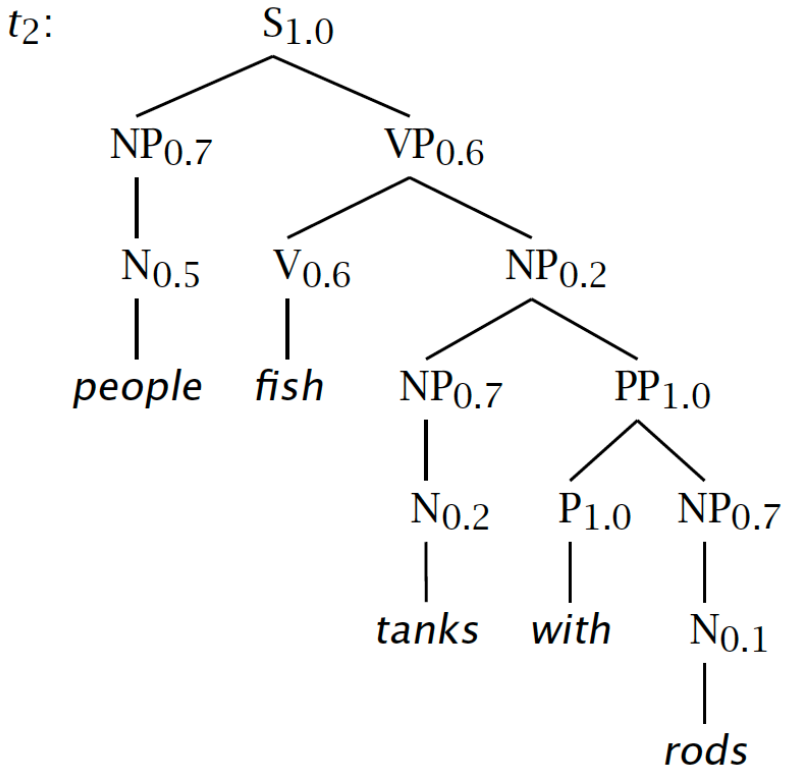
  $= 0.00024696$

  Noun attach
- $P(s) = P(t_1) + P(t_2)$

  $= 0.0008232 + 0.00024696$

  $= 0.00107016$





- A PP is more likely attached to a VP compared to NP.

Outline

- 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 generative capacity of a CFG
  - That is, it recognizes the same language
    - But maybe with different trees
- Empties and un-aries are removed recursively
- n-ary rules are divided by introducing new non-terminals ($n > 2$)

A phrase structure grammar

$S \rightarrow NP\ VP$

$VP \rightarrow V\ NP$

$VP \rightarrow V\ NP\ PP$

$NP \rightarrow NP\ NP$

$NP \rightarrow NP\ PP$

$NP \rightarrow N$

$\boxed{NP \rightarrow e}$

$PP \rightarrow P\ NP$

$N \rightarrow \textit{people}$

$N \rightarrow \textit{fish}$

$N \rightarrow \textit{tanks}$

$N \rightarrow \textit{rods}$

$V \rightarrow \textit{people}$

$V \rightarrow \textit{fish}$

$V \rightarrow \textit{tanks}$

$P \rightarrow \textit{with}$

$S \rightarrow VP$

$VP \rightarrow V$

Chomsky Normal Form steps

$S \rightarrow NP\ VP$

$S \rightarrow VP$

$VP \rightarrow V\ NP$

$VP \rightarrow V$

$VP \rightarrow V\ NP\ PP$

$VP \rightarrow V\ PP$

$NP \rightarrow NP\ NP$

$NP \rightarrow NP$

$NP \rightarrow NP\ PP$

$NP \rightarrow PP$

$NP \rightarrow N$

$PP \rightarrow P\ NP$

$PP \rightarrow P$

$N \rightarrow$ *people*

$N \rightarrow$ *fish*

$N \rightarrow$ *tanks*

$N \rightarrow$ *rods*

$V \rightarrow$ *people*

$V \rightarrow$ *fish*

$V \rightarrow$ *tanks*

$P \rightarrow$ *with*

$S \rightarrow V\ NP$

$S \rightarrow V$

Chomsky Normal Form steps

S → NP VP

VP → V NP

S → V NP

VP → V

S → V

VP → V NP PP

S → V NP PP

VP → V PP

S → V PP

NP → NP NP

NP → NP

NP → NP PP

NP → PP

NP → N

PP → P NP

PP → P

N → *people*

N → *fish*

N → *tanks*

N → *rods*

V → *people*

V → *fish*

V → *tanks*

P → *with*

Chomsky Normal Form steps

S → NP VP

VP → V NP

S → V NP

VP → V

VP → V NP PP

S → V NP PP

VP → V PP

S → V PP

NP → NP NP

NP → NP

NP → NP PP

NP → PP

NP → N

PP → P NP

PP → P

N → *people*

N → *fish*

N → *tanks*

N → *rods*

V → *people*

S → *people*

V → *fish*

S → *fish*

V → *tanks*

S → *tanks*

P → *with*

Chomsky Normal Form steps

S → NP VP

VP → V NP

S → V NP

VP → V NP PP

S → V NP PP

VP → V PP

S → V PP

NP → NP NP

NP → NP

NP → NP PP

NP → PP

NP → N

PP → P NP

PP → P

N → *people*

N → *fish*

N → *tanks*

N → *rods*

V → *people*

S → *people*

VP → *people*

V → *fish*

S → *fish*

VP → *fish*

V → *tanks*

S → *tanks*

VP → *tanks*

P → *with*

Chomsky Normal Form steps

S → NP VP

VP → V NP

S → V NP

VP → V NP PP

S → V NP PP

VP → V PP

S → V PP

NP → NP NP

NP → NP PP

NP → P NP

PP → P NP

VP → V X

X → NP PP

X = @VP_V

NP → *people*

NP → *fish*

NP → *tanks*

NP → *rods*

V → *people*

S → *people*

VP → *people*

V → *fish*

S → *fish*

VP → *fish*

V → *tanks*

S → *tanks*

VP → *tanks*

P → *with*

PP → *with*

Chomsky Normal Form steps

S → NP VP

VP → V NP

S → V NP

VP → V @VP_V

@VP_V → NP PP

S → V @S_V

@S_V → NP PP

VP → V PP

S → V PP

NP → NP NP

NP → NP PP

NP → P NP

PP → P NP

NP → *people*

NP → *fish*

NP → *tanks*

NP → *rods*

V → *people*

S → *people*

VP → *people*

V → *fish*

S → *fish*

VP → *fish*

V → *tanks*

S → *tanks*

VP → *tanks*

P → *with*

PP → *with*

A phrase structure grammar

S → NP VP

VP → V NP

VP → V NP PP

NP → NP NP

NP → NP PP

NP → N

NP → *e*

PP → P NP

N → *people*

N → *fish*

N → *tanks*

N → *rods*

V → *people*

V → *fish*

V → *tanks*

P → *with*

Chomsky Normal Form steps

S → NP VP

VP → V NP

S → V NP

VP → V @VP_V

@VP_V → NP PP

S → V @S_V

@S_V → NP PP

VP → V PP

S → V PP

NP → NP NP

NP → NP PP

NP → P NP

PP → P NP

NP → *people*

NP → *fish*

NP → *tanks*

NP → *rods*

V → *people*

S → *people*

VP → *people*

V → *fish*

S → *fish*

VP → *fish*

V → *tanks*

S → *tanks*

VP → *tanks*

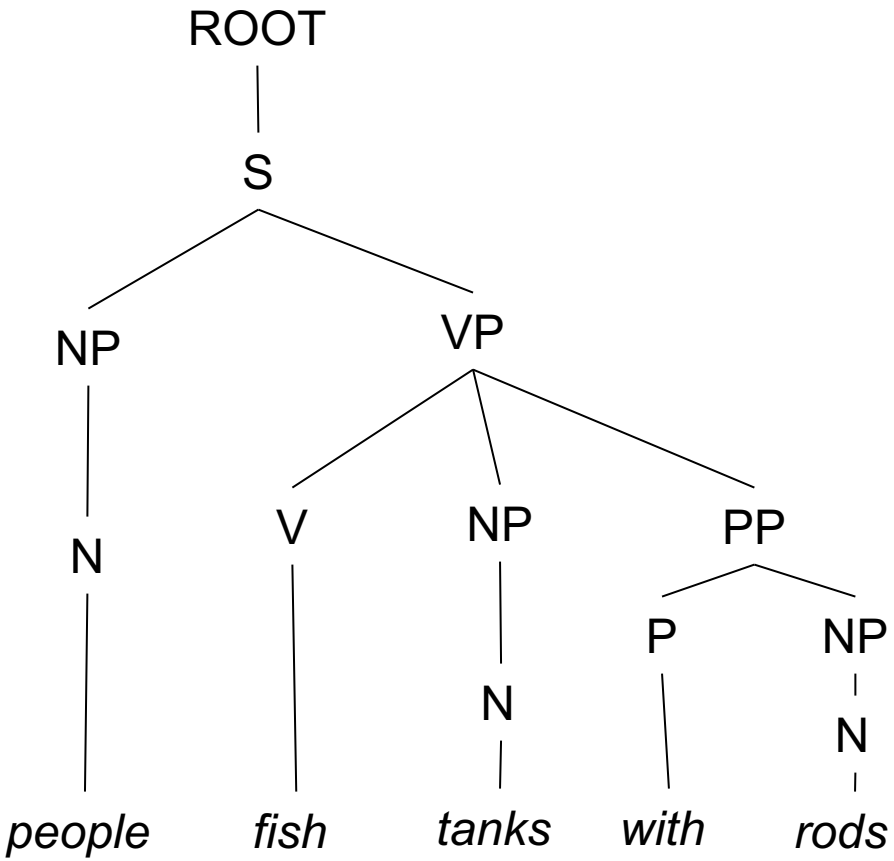
P → *with*

PP → *with*

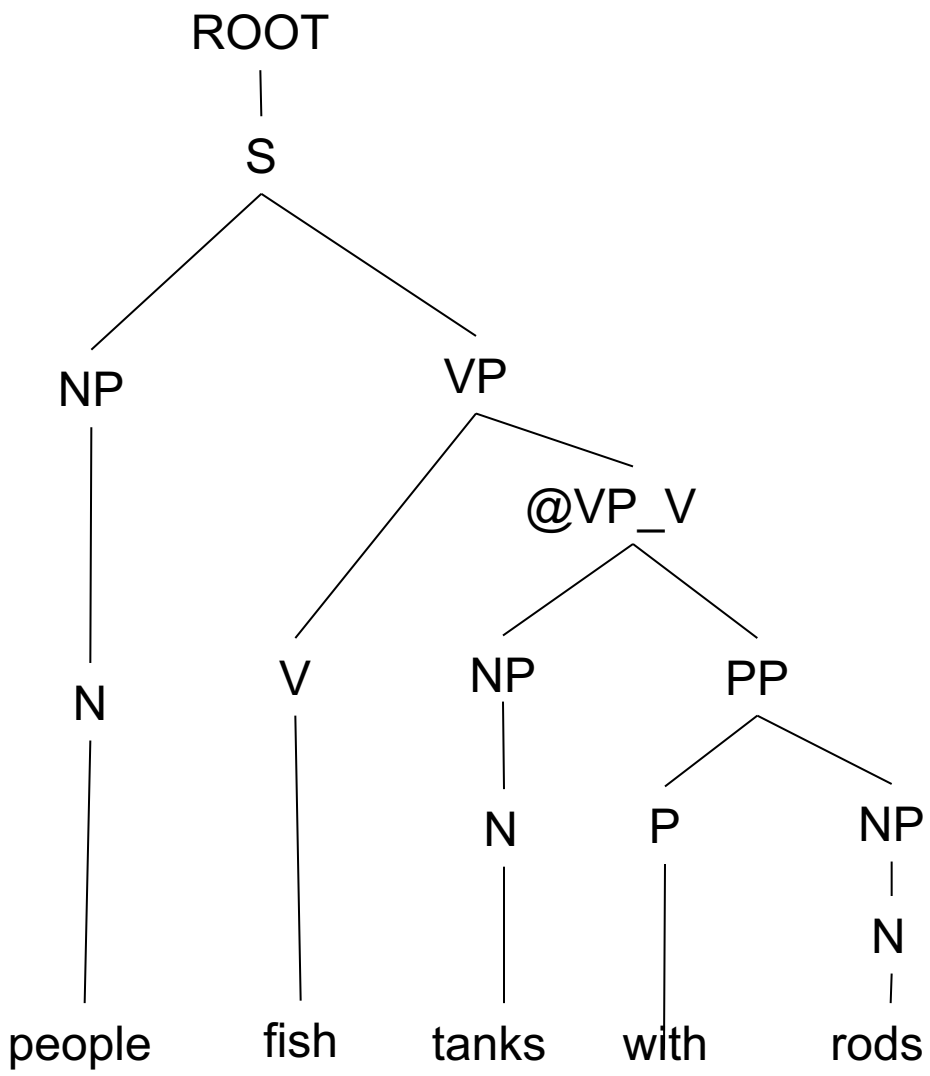
Chomsky Normal Form

- With some extra book-keeping in symbol names, you can even reconstruct the same trees with a de-transform
- You should think of this as a transformation for efficient parsing
- In practice full Chomsky Normal Form is a pain
  - Reconstructing n-aries is easy
  - Reconstructing unaries/empties is trickier
- **Binarization** is crucial for cubic time CFG parsing

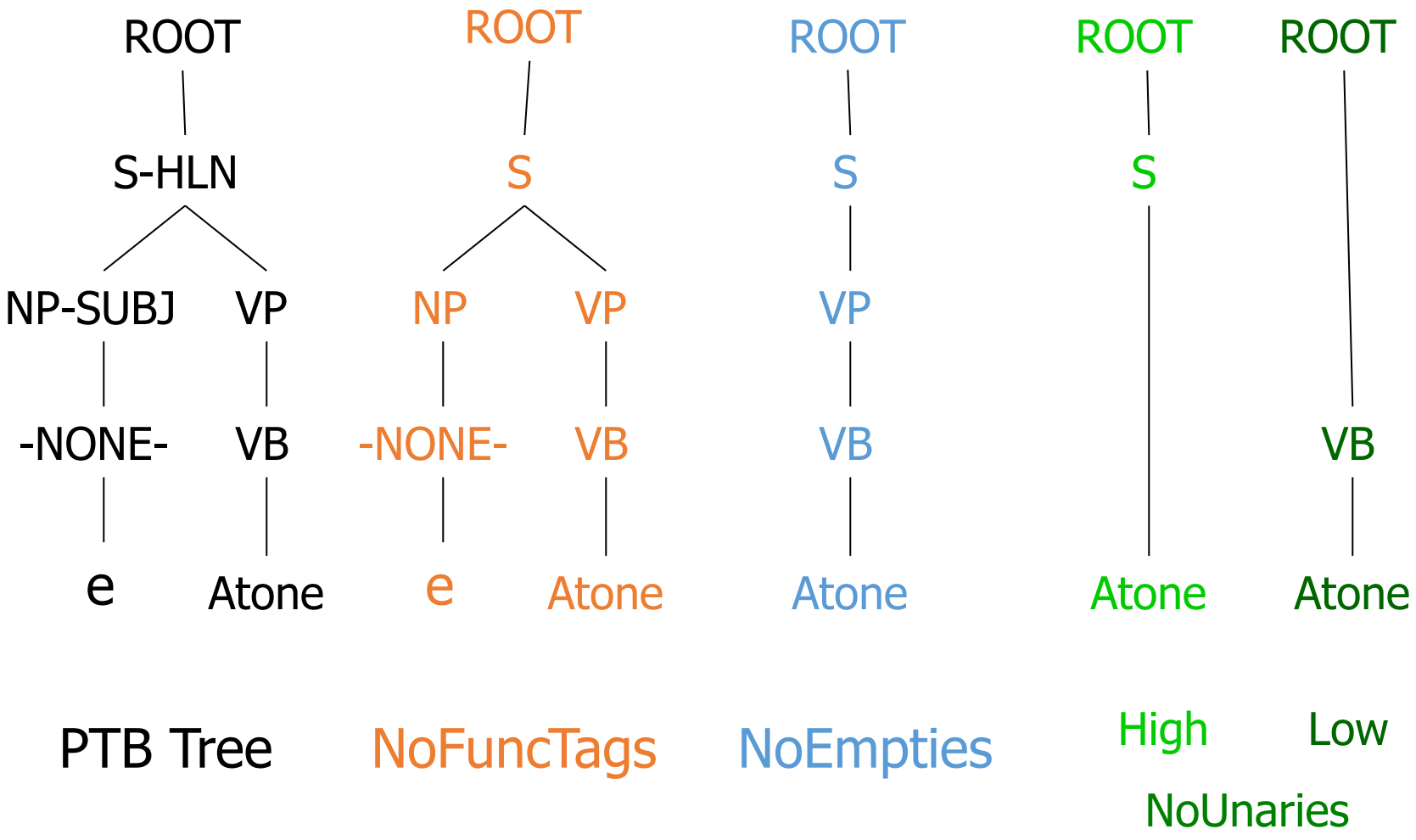
An example: before binarization…



After binarization…



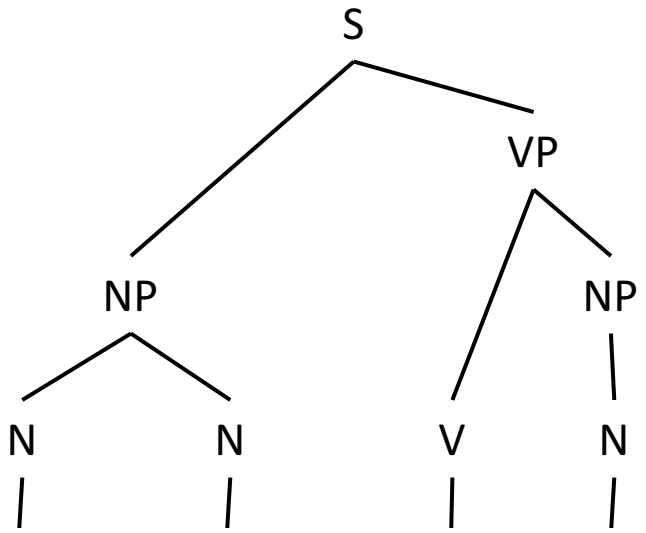
Treebank: empties and unaries



Outline

- Restricting the grammar form for efficient parsing
- Exact polynomial time parsing of (P)CFGs

Constituency Parsing

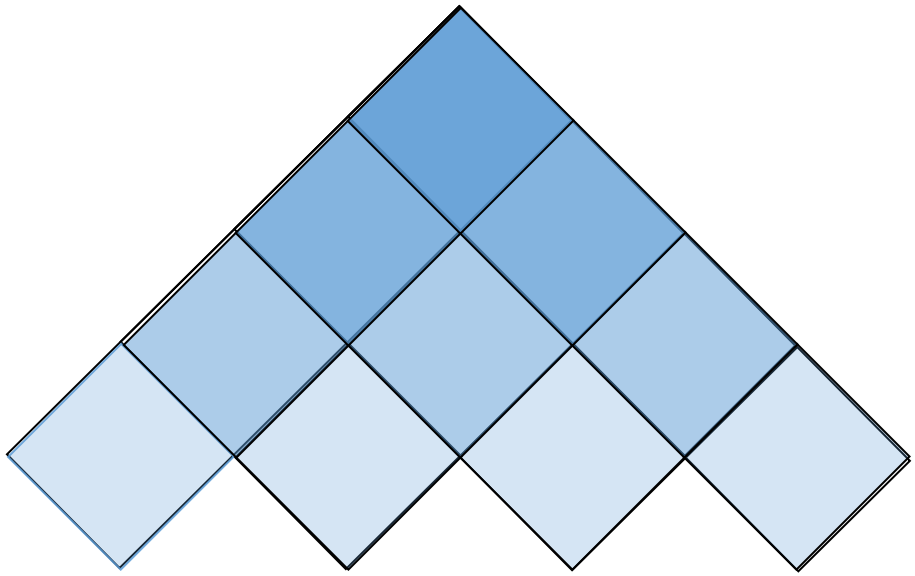


fish people fish tanks

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}
…	

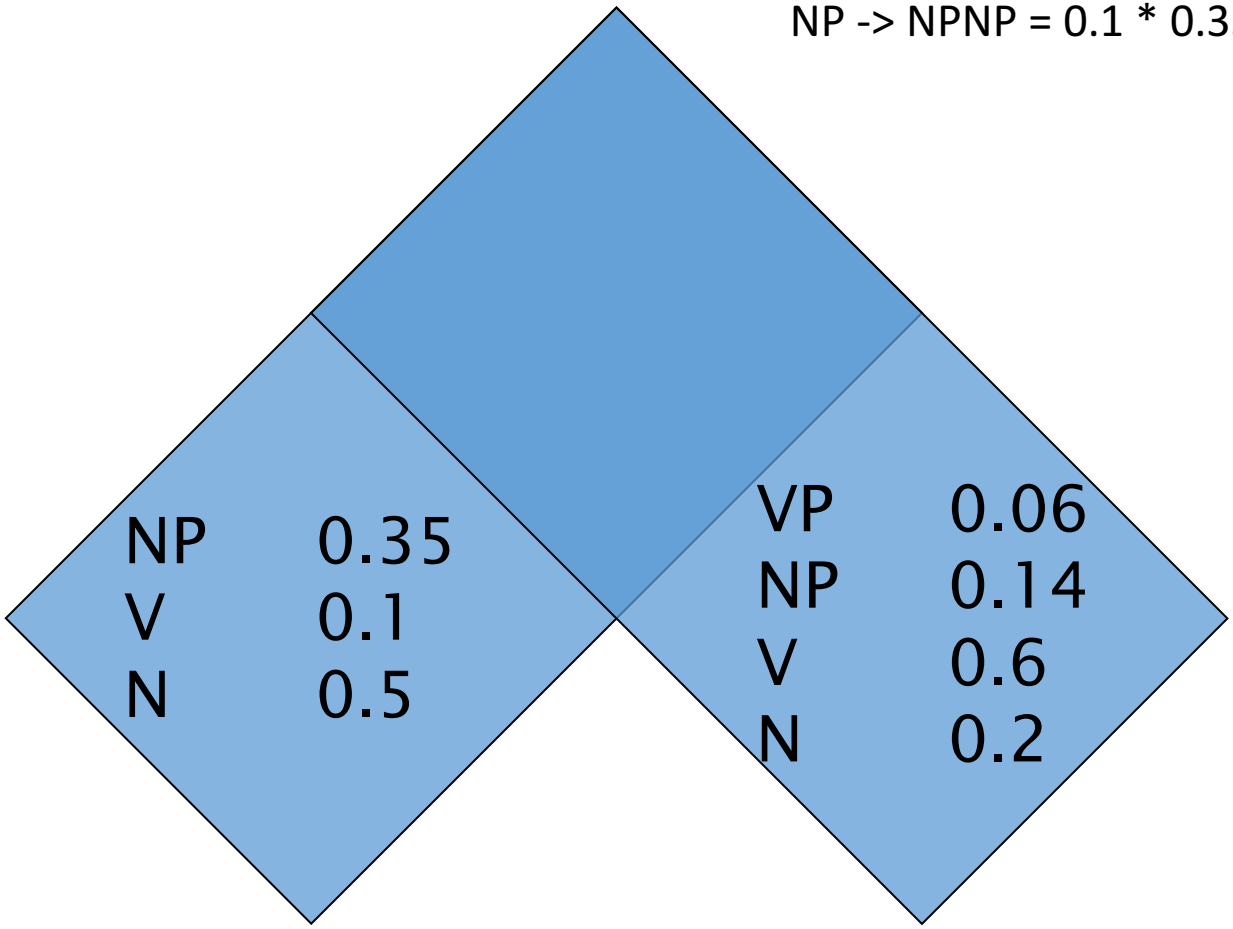
Cocke-Kasami-Younger (CKY) Constituency Parsing



fish people fish tanks

Viterbi (Max) Scores

S -> NP VP = 0.35*0.06*0.9
VP -> V NP = 0.1 * 0.5 * 0.14
NP -> NPNP = 0.1 * 0.35 * 0.14



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

- Unaries can be incorporated into the algorithm
  - Messy, but doesn't increase algorithmic complexity
- Empties can be incorporated
  - Doesn't increase complexity; essentially like u-naries
- Binarization is *vital*
  - Without binarization, you don't get parsing cubic in the length of the sentence and in the number of non-terminals 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
```

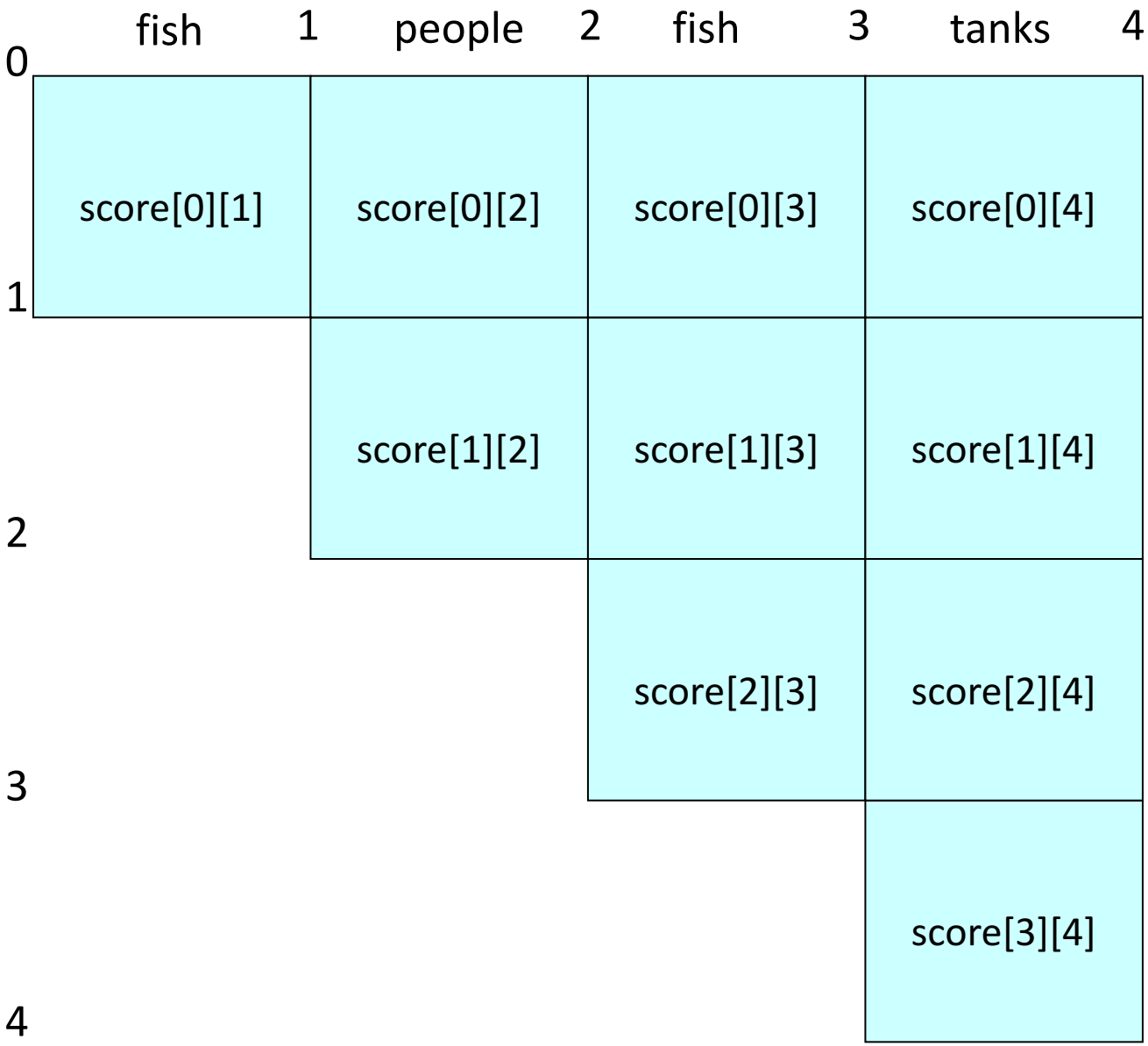
The CKY algorithm (1960/1965) extended to unaries

```
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)
```

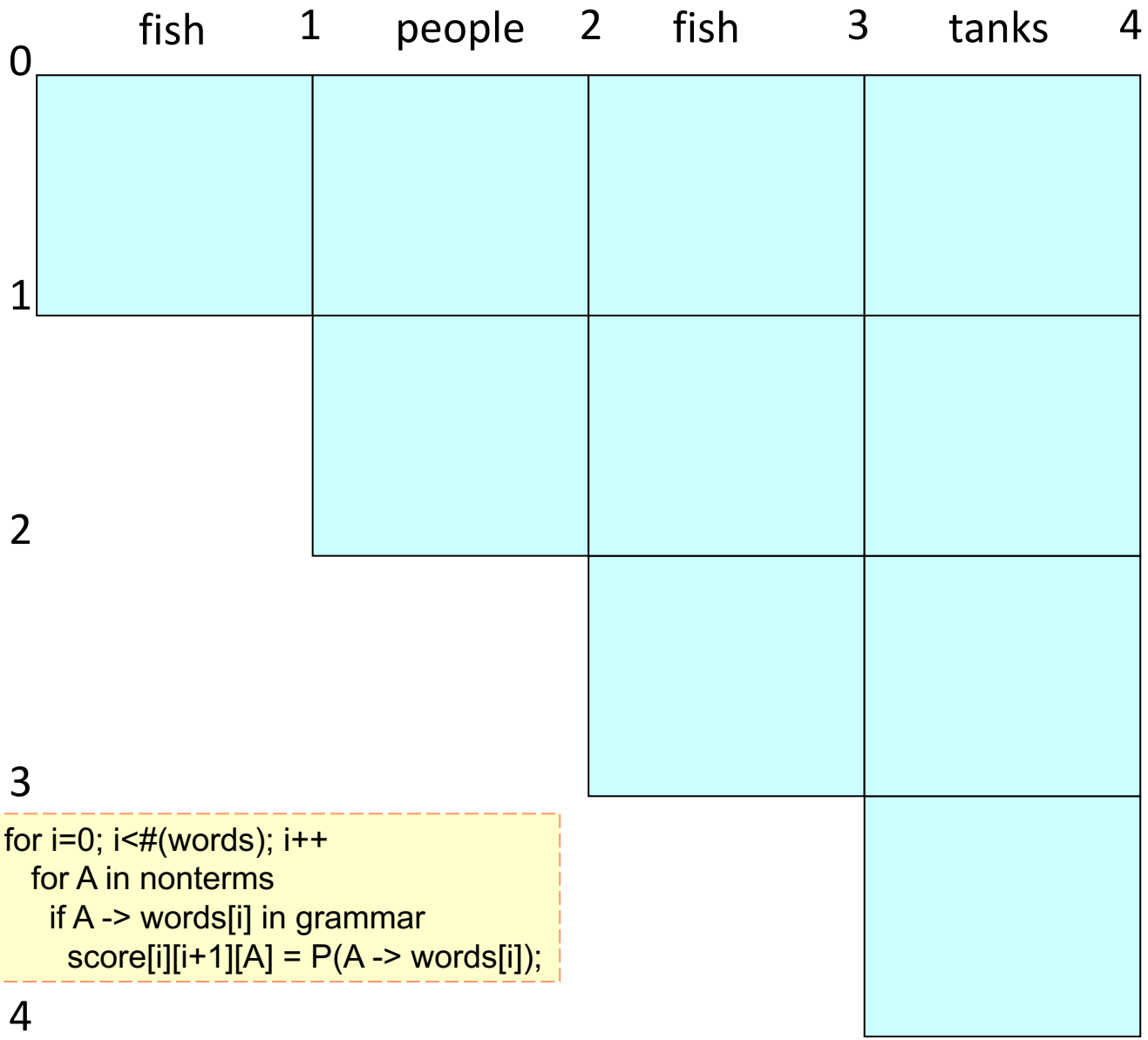
The grammar: Binary, no epsilons,

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

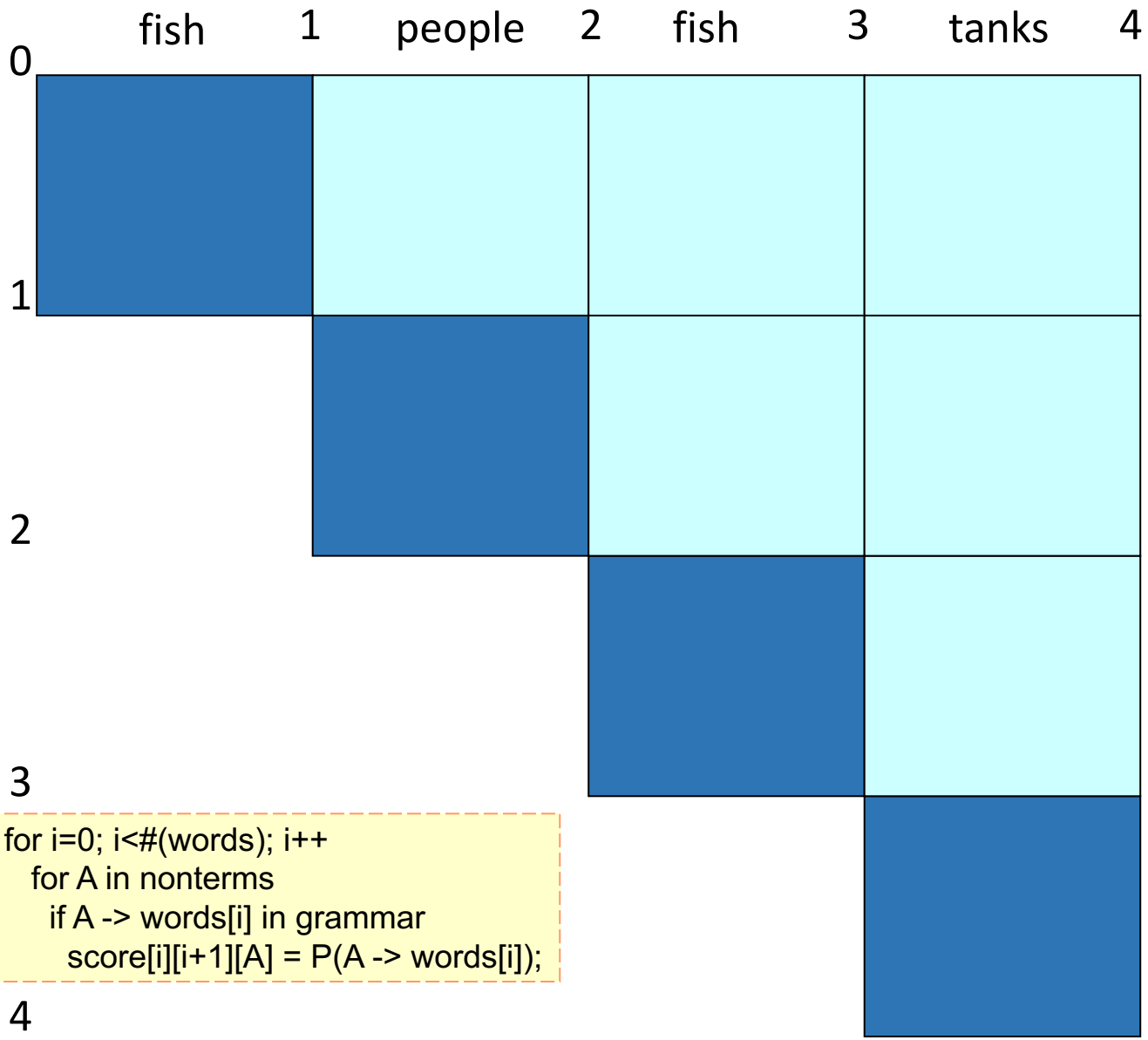

fish
1
people
2
fish
3
tanks
4
0
1
2
3
4
score[0][1]
score[0][2]
score[0][3]
score[0][4]
score[1][2]
score[1][3]
score[1][4]
score[2][3]
score[2][4]
score[3][4]

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



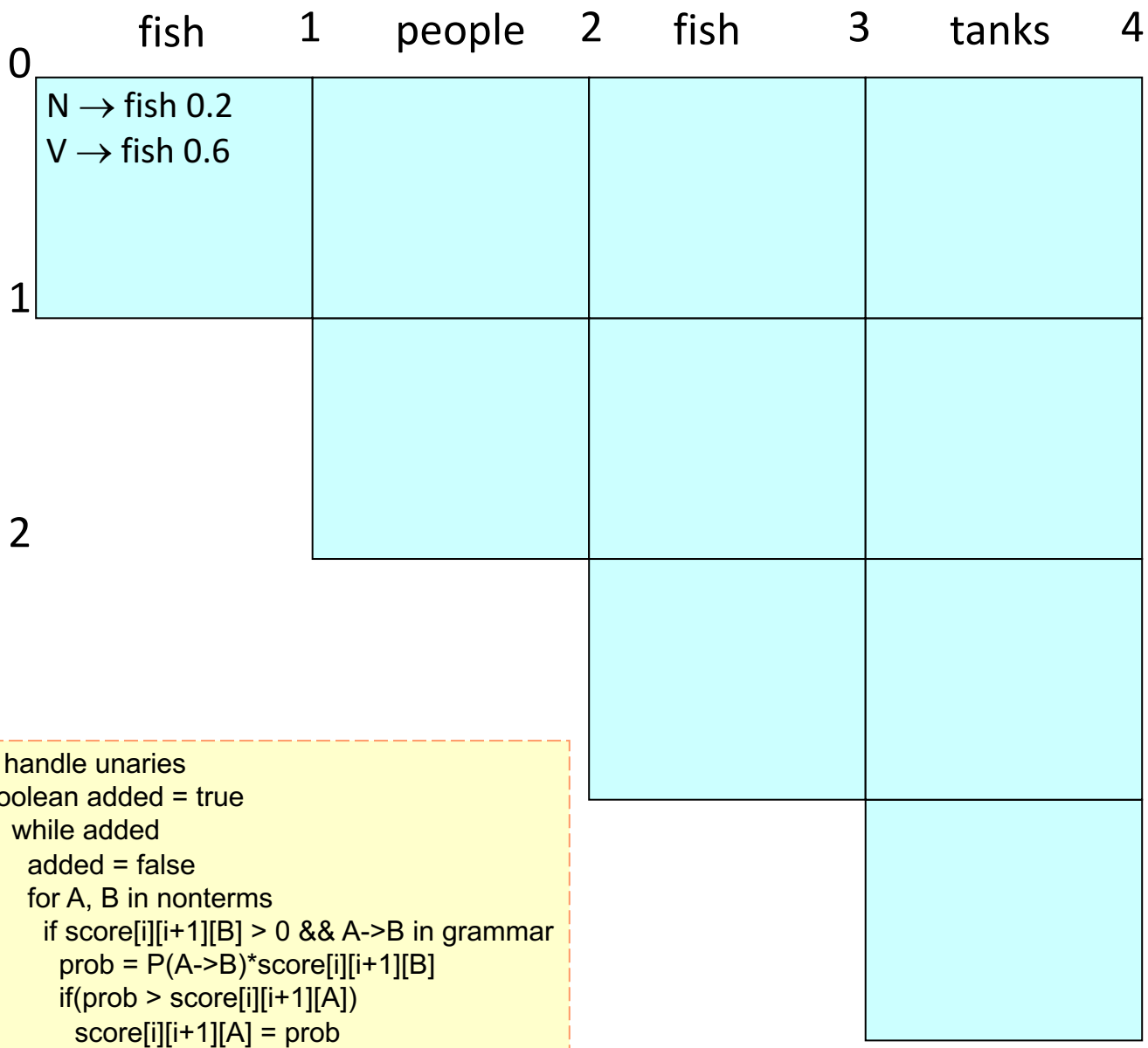

```
for i=0; i<#(words); i++
  for A in nonterms
    if A -> words[i] in grammar
      score[i][i+1][A] = P(A -> words[i]);
```

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



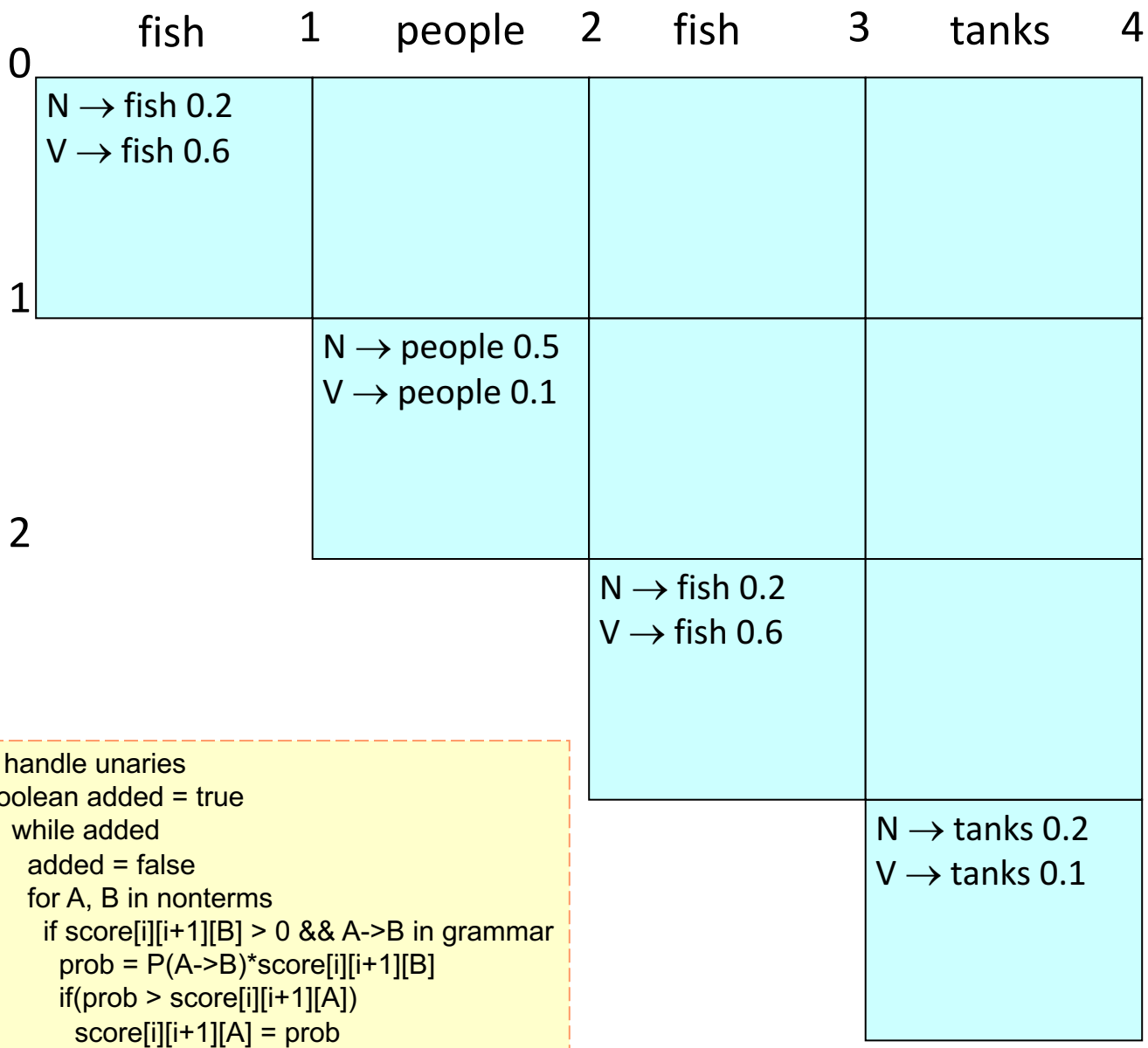

```
for i=0; i<#(words); i++
  for A in nonterms
    if A -> words[i] in grammar
      score[i][i+1][A] = P(A -> words[i]);
```

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



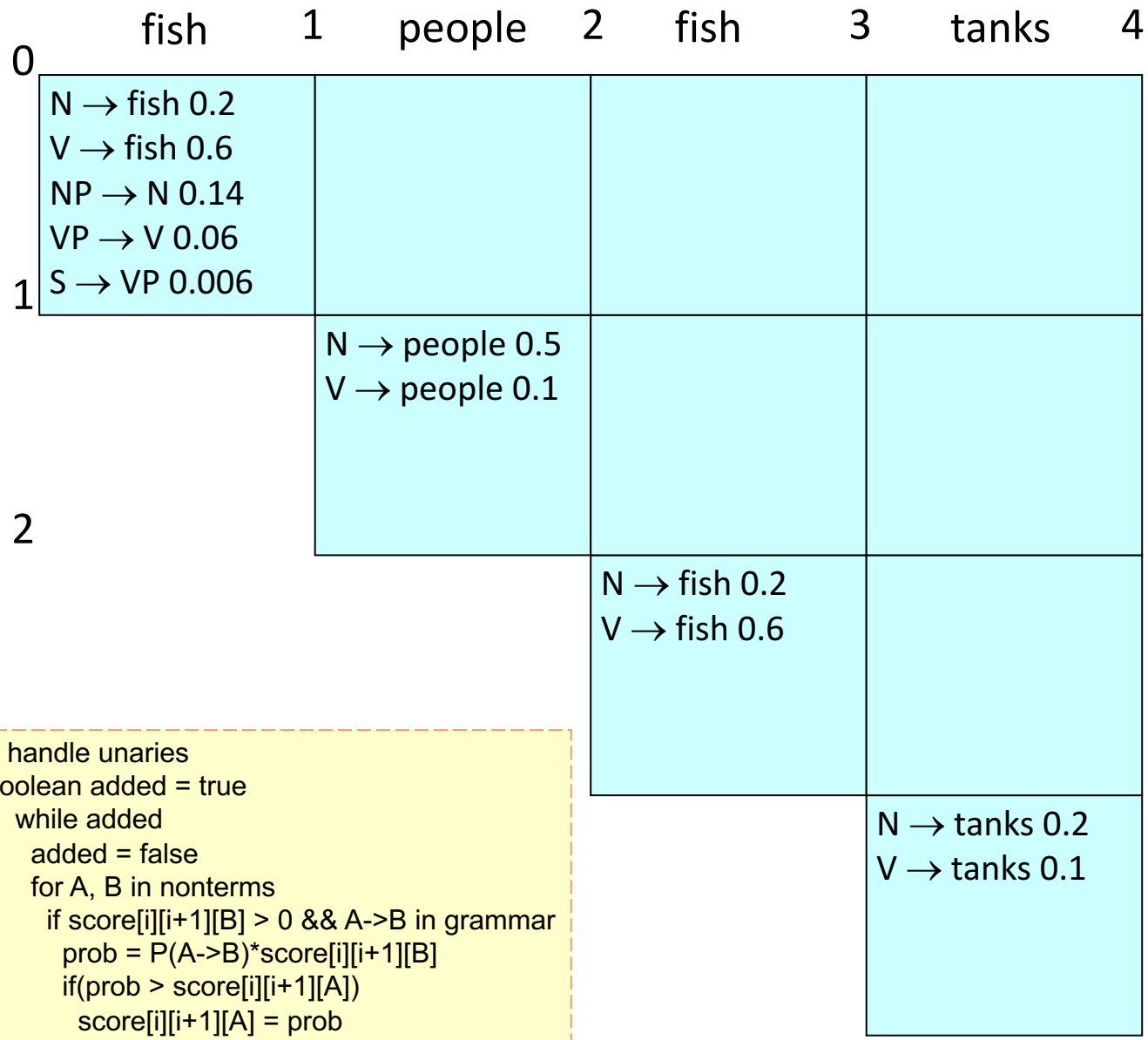

```
// handle unaries
boolean added = true
    while added
      added = false
      for A, B in nonterms
        if score[i][i+1][B] > 0 && A->B in grammar
          prob = P(A->B)*score[i][i+1][B]
          if(prob > score[i][i+1][A])
            score[i][i+1][A] = prob
            back[i][i+1][A]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



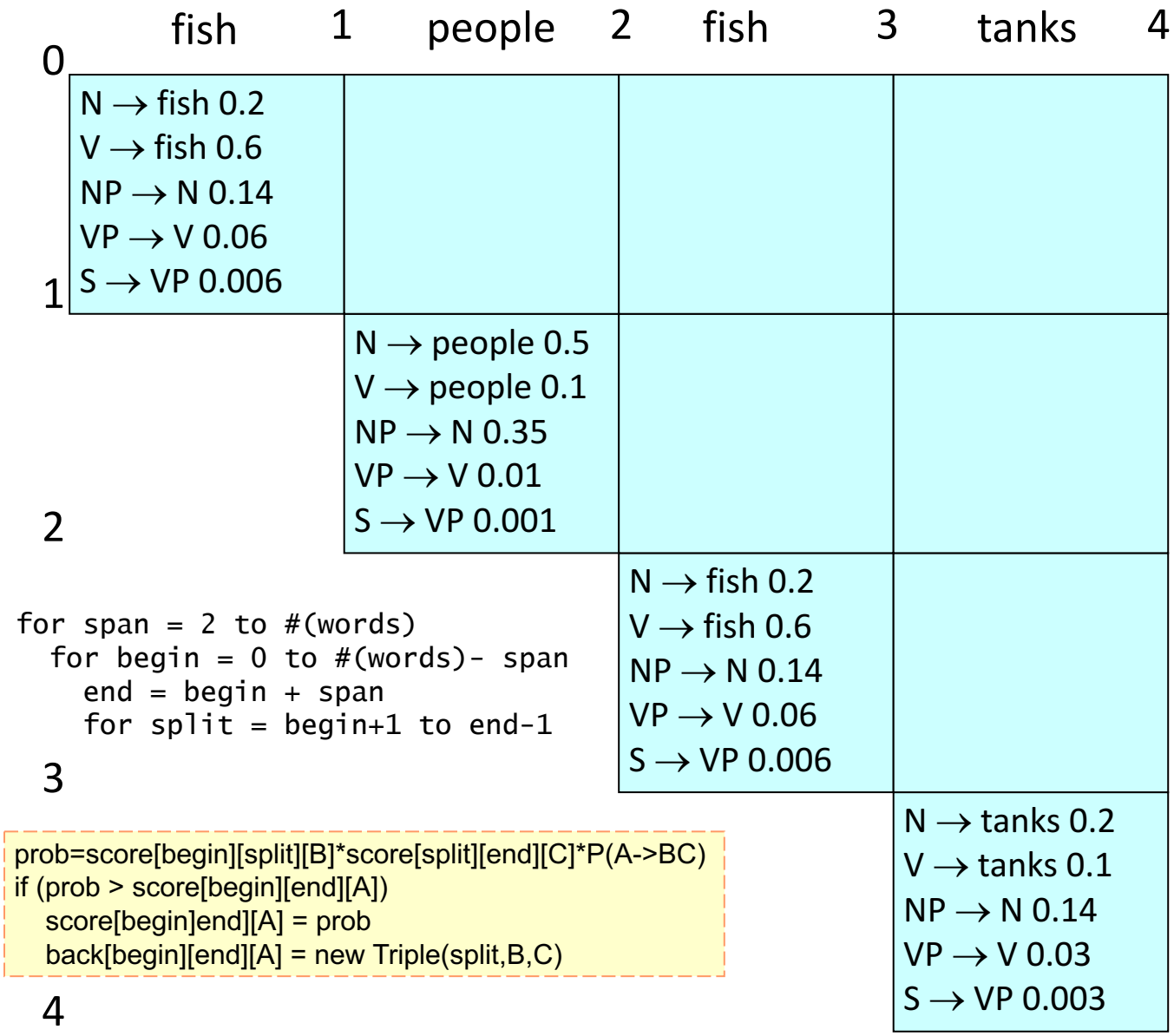

```
// handle unaries
boolean added = true
    while added
      added = false
      for A, B in nonterms
        if score[i][i+1][B] > 0 && A->B in grammar
          prob = P(A->B)*score[i][i+1][B]
          if(prob > score[i][i+1][A])
            score[i][i+1][A] = prob
            back[i][i+1][A]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

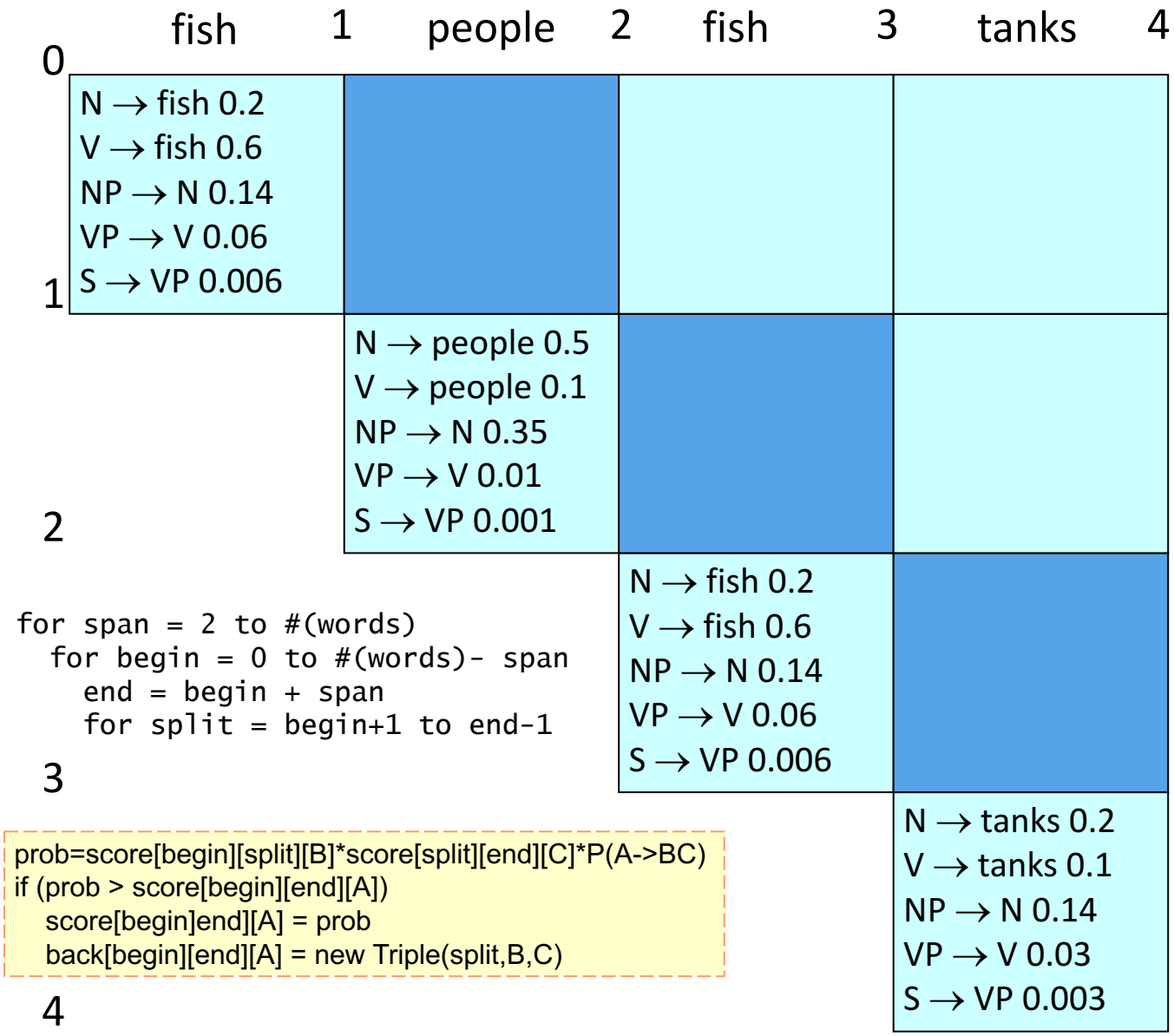



```
// handle unaries
boolean added = true
    while added
      added = false
      for A, B in nonterms
        if score[i][i+1][B] > 0 && A->B in grammar
          prob = P(A->B)*score[i][i+1][B]
          if(prob > score[i][i+1][A])
            score[i][i+1][A] = prob
            back[i][i+1][A]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



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 span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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					
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	
4								

```
for span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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
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	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					
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 span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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					
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 span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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					
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 span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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					
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 span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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 span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
```

```
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		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								

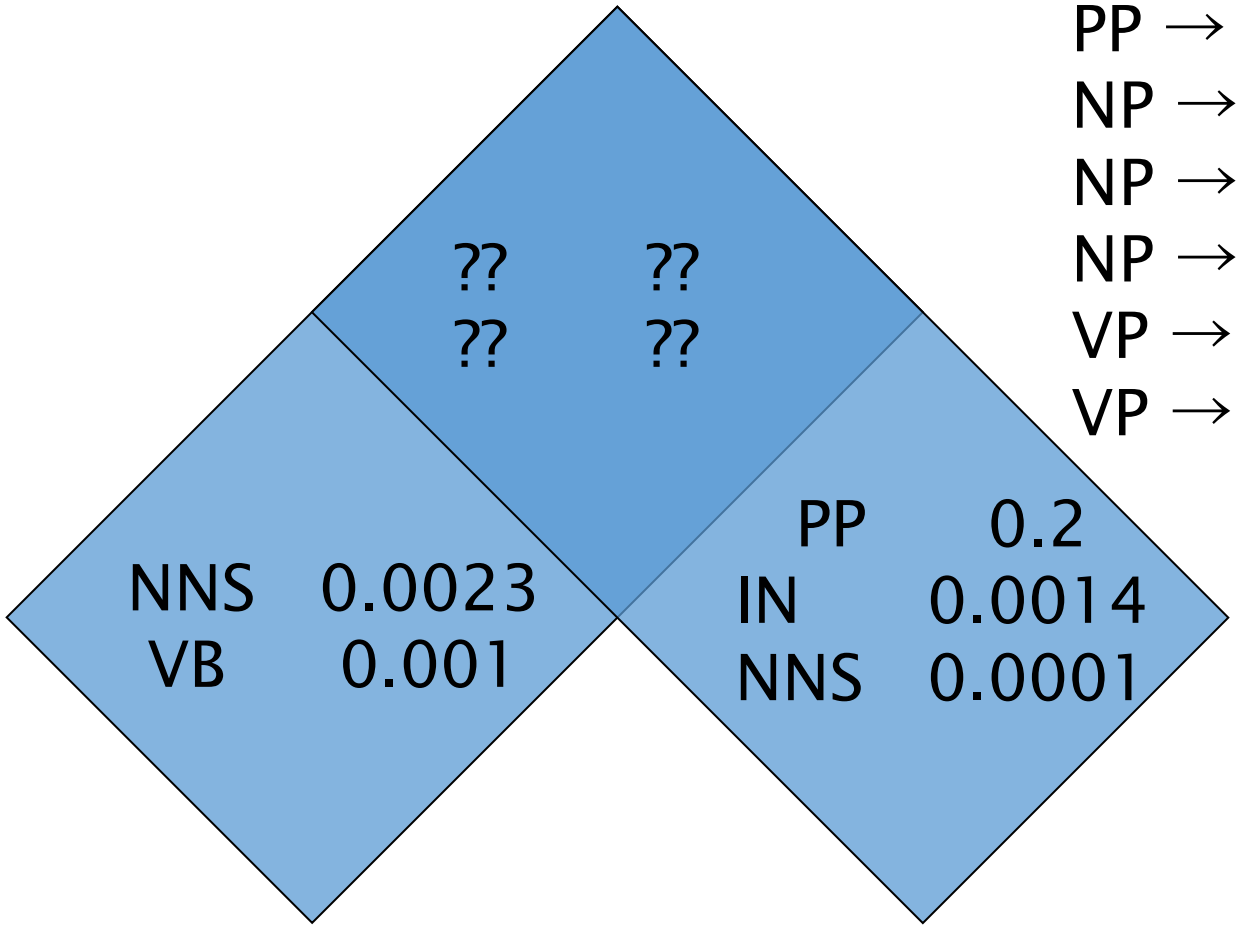
```
for split = begin+1 to end-1
   for A,B,C in nonterms
      prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
      if prob > score[begin][end][A]
         score[begin]end][A] = prob
         back[begin][end][A] = new Triple(split,B,C)
```

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	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
4				

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

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

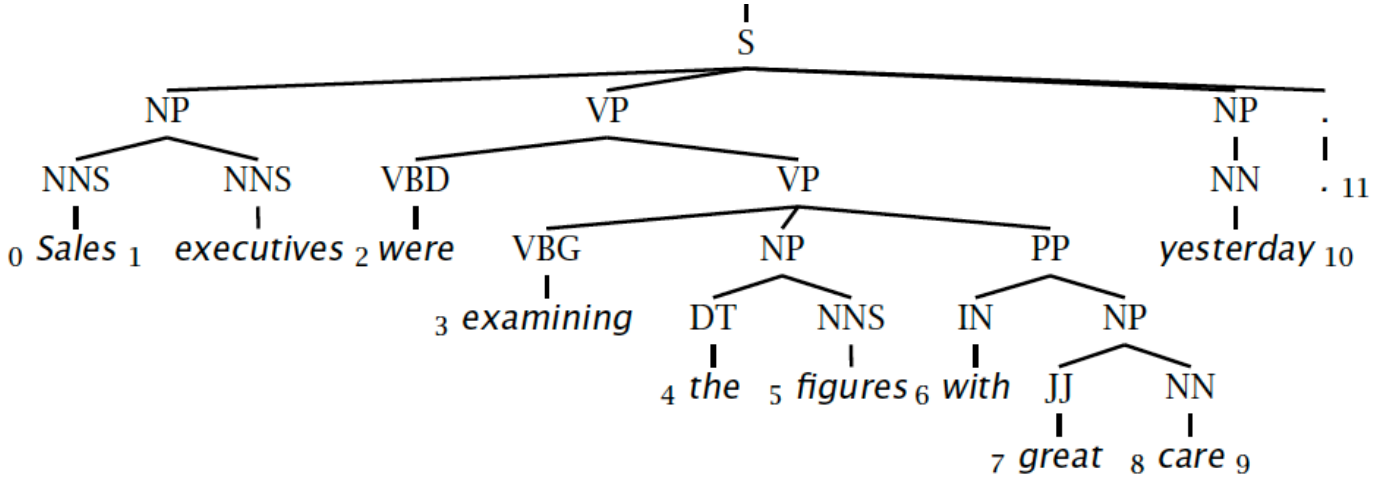
What constituents (with what probability can you make?

Outline

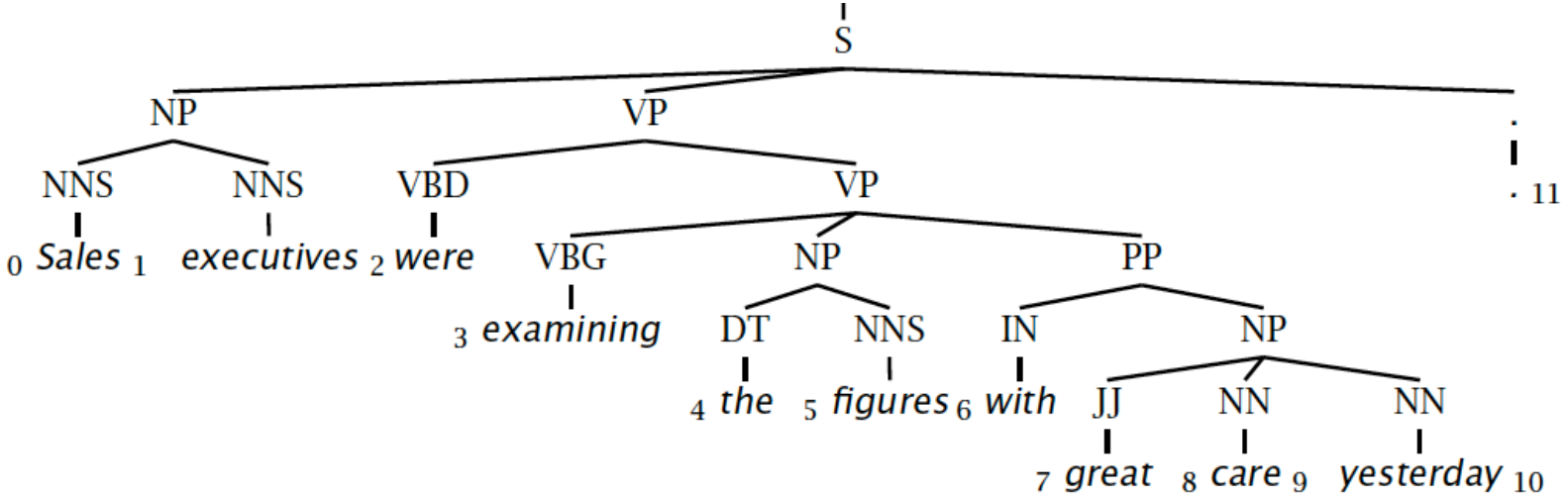
- Restricting the grammar form for efficient parsing
- Exact polynomial time parsing of (P)CFGs
- Constituency Parser Evaluation

Evaluating constituency parsing

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)



Evaluating constituency parsing

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%
LP/LR F1	40.0%
Tagging Accuracy	11/11 = 100.0%

How good are PCFGs?

- Penn WSJ parsing accuracy: about 73% LP/LR F1
- Robust
  - Usually admit everything, but with low probability
- Partial solution for grammar ambiguity
  - A PCFG gives some idea of the plausibility of a parse
  - But not so good because the independence assumptions are too strong
- Give a probabilistic language model
  - But in the simple case it performs worse than a trigram model
- The problem seems to be that PCFGs lack the lexicalization of a trigram model