

CS447: Natural Language Processing

<http://courses.engr.illinois.edu/cs447>

Lecture 15:

The CKY parsing algorithm

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Last lecture's key concepts

Natural language syntax

- Constituents

- Dependencies

- Context-free grammar

- Arguments and modifiers

- Recursion in natural language

Today's class

Parsing with CFGs:

The CKY (Cocke Kasami Younger) algorithm

An example CFG

DT \rightarrow {the, a}

N \rightarrow {ball, garden, house, sushi }

P \rightarrow {in, behind, with}

NP \rightarrow DT N

NP \rightarrow NP PP

PP \rightarrow P NP

N: noun

P: preposition

NP: “noun phrase”

PP: “prepositional phrase”

Reminder: Context-free grammars

A CFG is a 4-tuple $\langle \mathbf{N}, \mathbf{\Sigma}, \mathbf{R}, S \rangle$ consisting of:

A set of nonterminals \mathbf{N}

(e.g. $\mathbf{N} = \{S, NP, VP, PP, Noun, Verb, \dots\}$)

A set of terminals $\mathbf{\Sigma}$

(e.g. $\mathbf{\Sigma} = \{I, you, he, eat, drink, sushi, ball, \}$)

A set of rules \mathbf{R}

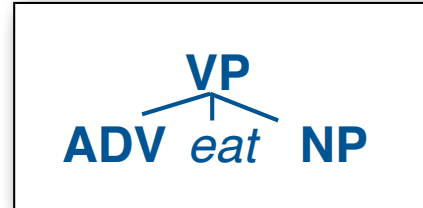
$\mathbf{R} \subseteq \{A \rightarrow \beta \text{ with left-hand-side (LHS) } A \in \mathbf{N}$
and right-hand-side (RHS) $\beta \in (\mathbf{N} \cup \mathbf{\Sigma})^* \}$

A start symbol $S \in \mathbf{N}$

Chomsky Normal Form

The right-hand side of a standard CFG can have an **arbitrary number of symbols** (terminals and nonterminals):

$VP \rightarrow ADV \text{ eat } NP$



A CFG in **Chomsky Normal Form** (CNF) allows only two kinds of right-hand sides:

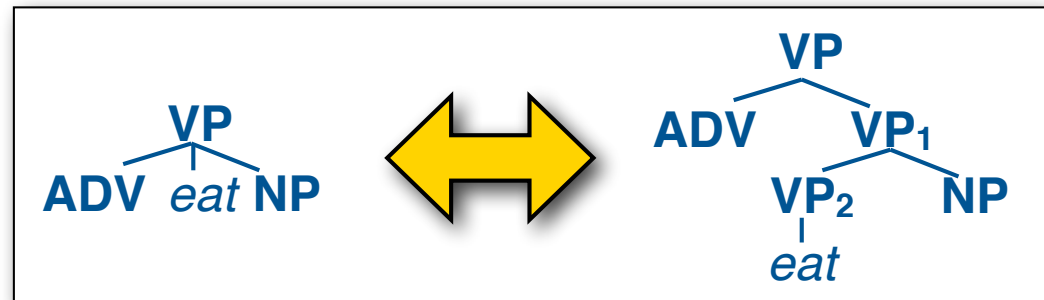
- **Two nonterminals:** $VP \rightarrow ADV VP$
- **One terminal:** $VP \rightarrow eat$

Any CFG can be transformed into an equivalent CNF:

$VP \rightarrow ADV VP_1$

$VP_1 \rightarrow VP_2 NP$

$VP_2 \rightarrow eat$



A note about ϵ -productions

Formally, context-free grammars are allowed to have **empty productions** (ϵ = the empty string):

$VP \rightarrow V NP$ $NP \rightarrow DT Noun$ $NP \rightarrow \epsilon$

These can always be **eliminated** without changing the language generated by the grammar:

$VP \rightarrow V NP$ $NP \rightarrow DT Noun$ $NP \rightarrow \epsilon$

becomes

$VP \rightarrow V NP$ $VP \rightarrow V \epsilon$ $NP \rightarrow DT Noun$

which in turn becomes

$VP \rightarrow V NP$ $VP \rightarrow V$ $NP \rightarrow DT Noun$

We will assume that our grammars don't have ϵ -productions

CKY chart parsing algorithm

Bottom-up parsing:

- start with the words

Dynamic programming:

- save the results in a table/chart

- re-use these results in finding larger constituents

Complexity: $O(n^3|G|)$

- n : length of string, $|G|$: size of grammar)

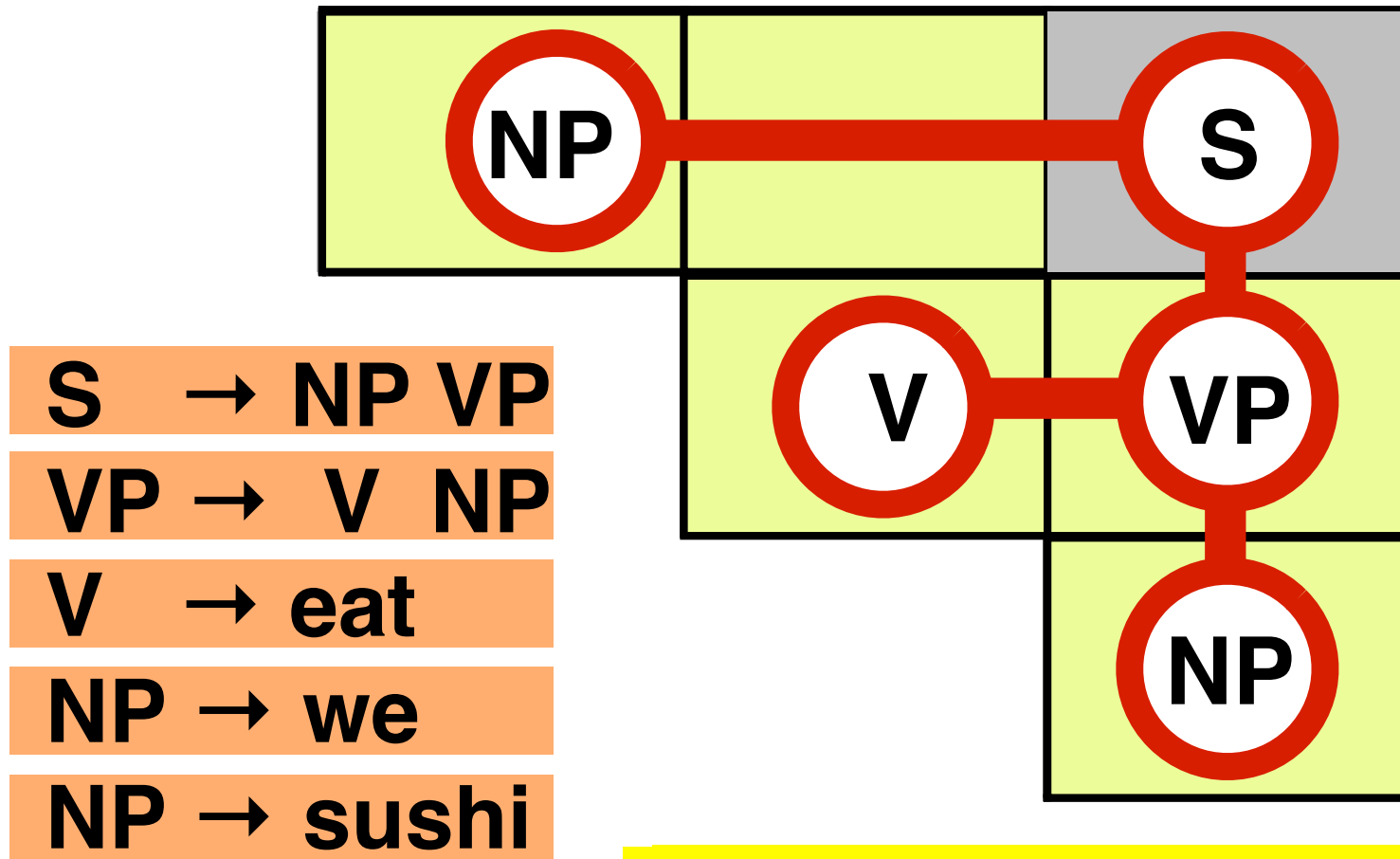
Presumes a CFG in **Chomsky Normal Form**:

- Rules are all either $A \rightarrow B C$ or $A \rightarrow a$

- (with A, B, C nonterminals and a a terminal)

The CKY parsing algorithm

To recover the parse tree, each entry needs **pairs** of backpointers.



We eat sushi

CKY algorithm

1. Create the chart

(an $n \times n$ upper triangular matrix for an sentence with n words)

- Each cell $\text{chart}[i][j]$ corresponds to the substring $w^{(i)} \dots w^{(j)}$

2. Initialize the chart (fill the diagonal cells $\text{chart}[i][i]$):

For all rules $X \rightarrow w^{(i)}$, add an entry X to $\text{chart}[i][i]$

3. Fill in the chart:

Fill in all cells $\text{chart}[i][i+1]$, then $\text{chart}[i][i+2]$, ...,

until you reach $\text{chart}[1][n]$ (the top right corner of the chart)

- To fill $\text{chart}[i][j]$, consider all binary splits $w^{(i)} \dots w^{(k)} | w^{(k+1)} \dots w^{(j)}$
- If the grammar has a rule $X \rightarrow YZ$, $\text{chart}[i][k]$ contains a Y and $\text{chart}[k+1][j]$ contains a Z , add an X to $\text{chart}[i][j]$ with two backpointers to the Y in $\text{chart}[i][k]$ and the Z in $\text{chart}[k+1][j]$

4. Extract the parse trees from the S in $\text{chart}[1][n]$.

CKY: filling the chart

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

CKY: filling one cell

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

chart[2][6]:

w_1 **w_2** **w_3** **w_4** **w_5** **w_6** w_7

chart[2][6]:

w_1 **w_2** **w_3** **w_4** **w_5** **w_6** w_7

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

chart[2][6]:

w_1 **w_2** **w_3** **w_4** **w_5** **w_6** w_7

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

chart[2][6]:

w_1 **w_2** **w_3** **w_4** **w_5** **w_6** w_7

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

chart[2][6]:

w_1 **w_2** **w_3** **w_4** **w_5** **w_6** w_7

w	...			w_i	...	w	
							w
							...
							..
							w_i
							...
							w

The CKY parsing algorithm

V buy	VP buy drinks	buy drinks with	VP buy drinks with milk
	V, NP drinks	drinks with	VP, NP drinks with milk
		P	PP

Each cell may have **one entry**
for each nonterminal

We buy drinks with milk

$S \rightarrow NP VP$

$VP \rightarrow V NP$

$VP \rightarrow VP PP$

$V \rightarrow \text{drinks}$

$NP \rightarrow NP PP$

$NP \rightarrow \text{we}$

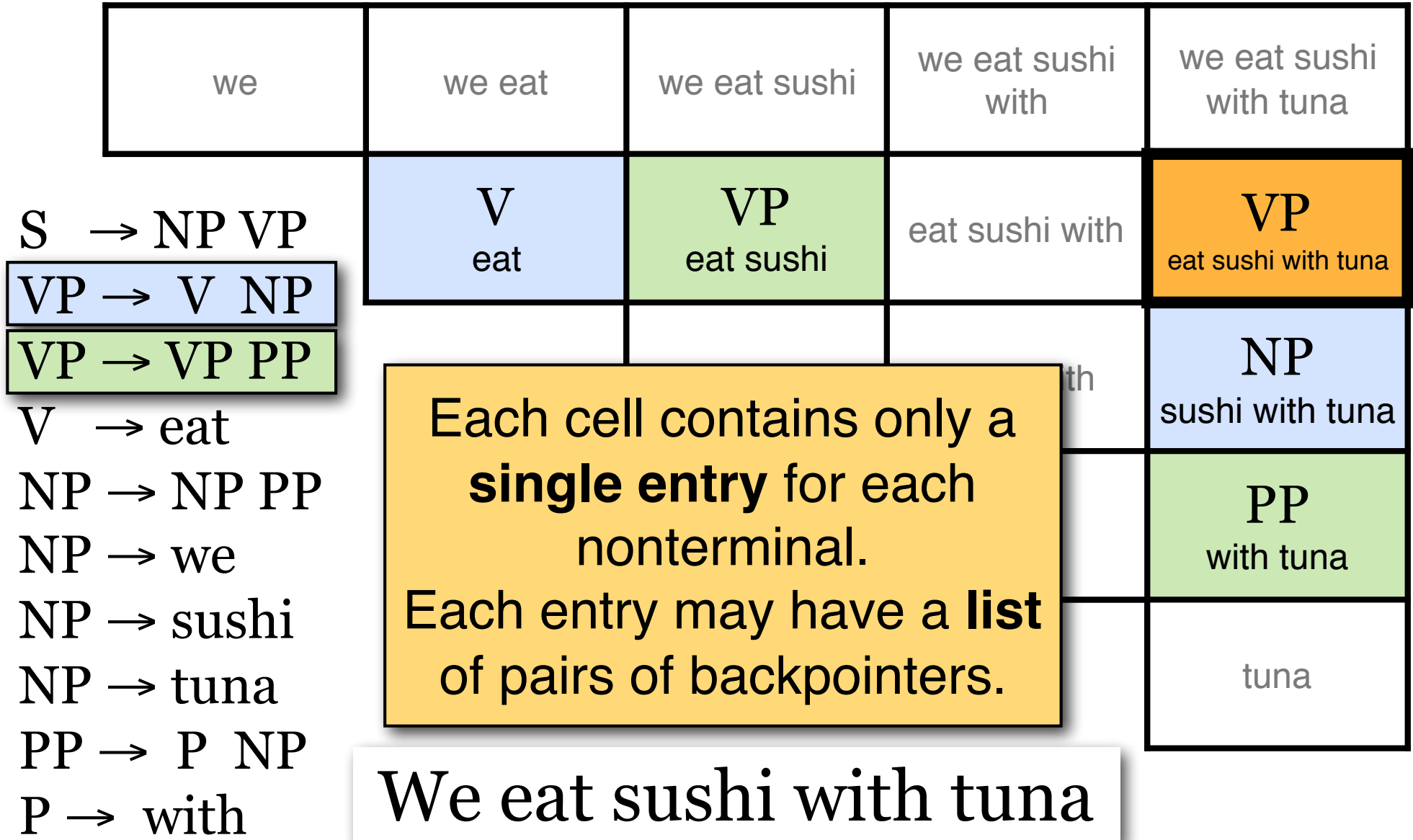
$NP \rightarrow \text{drinks}$

$NP \rightarrow \text{milk}$

$PP \rightarrow P NP$

$P \rightarrow \text{with}$

The CKY parsing algorithm



What are the terminals in NLP?

Are the “terminals”: words or POS tags?

For toy examples (e.g. on slides), it's typically the words

With POS-tagged input, we may either treat the POS tags as the terminals, or we assume that the unary rules in our grammar are of the form

POS-tag \rightarrow word

(so POS tags are the only nonterminals that can be rewritten as words; some people call POS tags “preterminals”)

Additional unary rules

In practice, we may allow other unary rules, e.g.

$NP \rightarrow \text{Noun}$

(where Noun is also a nonterminal)

In that case, we apply all unary rules to the entries in $\text{chart}[i][j]$ after we've checked all binary splits $(\text{chart}[i][k], \text{chart}[k+1][j])$

Unary rules are fine as long as there are no “loops” that could lead to an infinite chain of unary productions, e.g.:

$\mathbf{X} \rightarrow Y$ and $Y \rightarrow \mathbf{X}$

or: $\mathbf{X} \rightarrow Y$ and $Y \rightarrow Z$ and $Z \rightarrow \mathbf{X}$

CKY so far...

Each entry in a cell $\text{chart}[i][j]$ is associated with a nonterminal X .

If there is a rule $X \rightarrow YZ$ in the grammar, and there is a pair of cells $\text{chart}[i][k]$, $\text{chart}[k+1][j]$ with a Y in $\text{chart}[i][k]$ and a Z in $\text{chart}[k+1][j]$, we can add an entry X to cell $\text{chart}[i][j]$, and associate one pair of backpointers with the X in cell $\text{chart}[i][k]$

Each entry might have multiple pairs of backpointers.

When we extract the parse trees at the end,
we can get **all possible trees**.

We will need probabilities to find the single best tree!

Exercise: CKY parser

I eat sushi with chopsticks with you

S → NP VP
NP → NP PP
NP → sushi
NP → I
NP → chopsticks
NP → you
VP → VP PP
VP → Verb NP
Verb → eat
PP → Prep NP
Prep → with

How do you count the **number of parse trees** for a sentence?

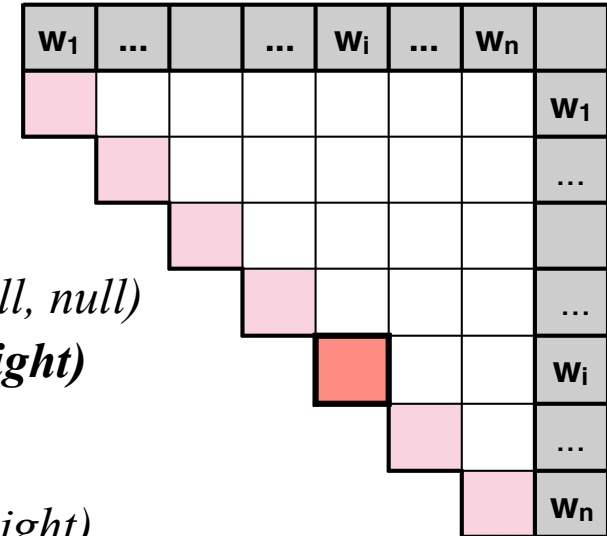
1. For each **pair of backpointers**
(e.g. $VP \rightarrow V \ NP$): **multiply** #trees of children
$$\text{trees}(VP_{VP \rightarrow V \ NP}) = \text{trees}(V) \times \text{trees}(NP)$$

2. For each **list of pairs of backpointers**
(e.g. $VP \rightarrow V \ NP$ and $VP \rightarrow VP \ PP$): **sum** #trees
$$\text{trees}(VP) = \text{trees}(VP_{VP \rightarrow V \ NP}) + \text{trees}(VP_{VP \rightarrow VP \ PP})$$

Cocke Kasami Younger (1)

```
ckyParse(n):
  initChart(n)
  fillChart(n)
```

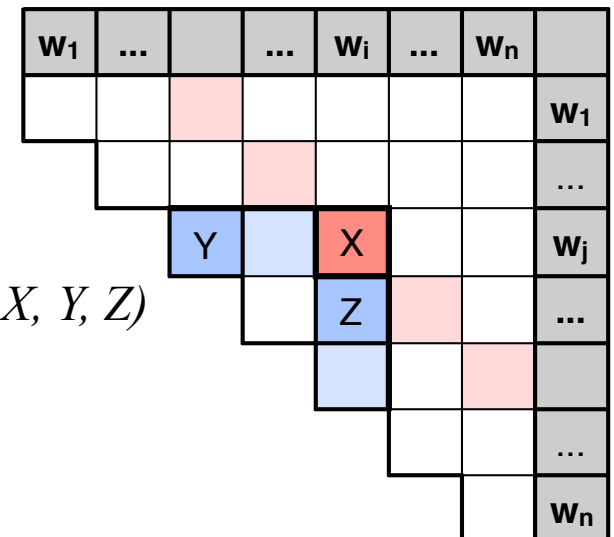
```
initChart(n):
  for  $i = 1 \dots n$ :
    initCell(i,i)
initCell(i,i):
  for  $c$  in  $\text{lex}(\text{word}[i])$ :
    addToCell(cell[i][i], c, null, null)
addToCell(Parent, cell, Left, Right)
  if (cell.hasEntry(Parent)):
     $P = \text{cell.getEntry(Parent)}$ 
     $P.\text{addBackpointers(Left, Right)}$ 
  else cell.addEntry(Parent, Left, Right)
```



```
fillChart(n):
  for span = 1...n-1:
    for  $i = 1 \dots n - \text{span}$ :
      fillCell(i, i+span)
```

```
fillCell(i,j):
  for  $k = i \dots j-1$ :
    combineCells(i, k, j)
```

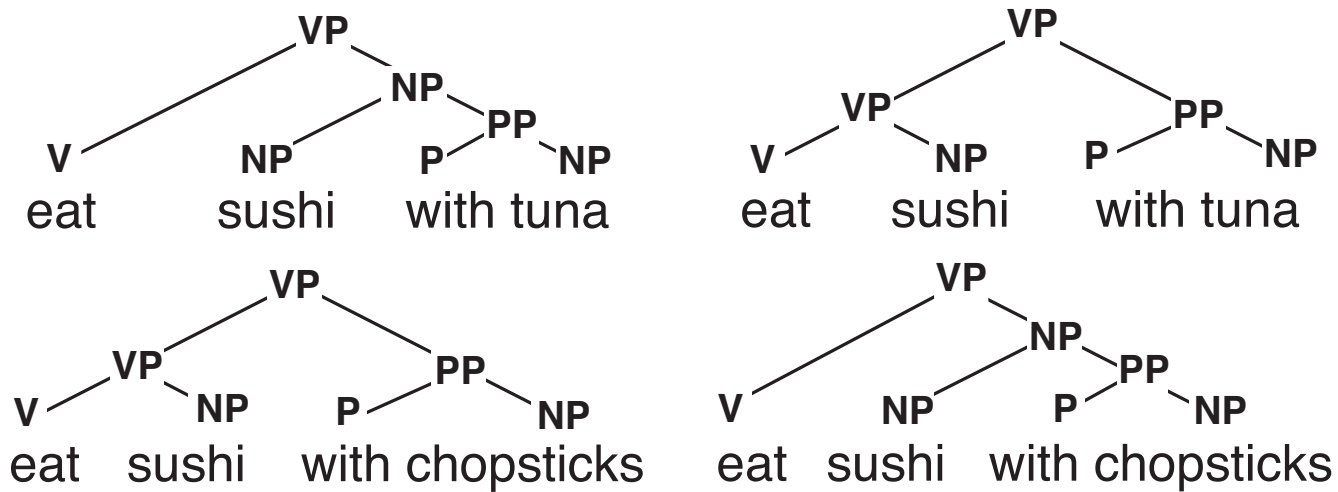
```
combineCells(i,k,j):
  for  $Y$  in  $\text{cell}[i][k]$ :
    for  $Z$  in  $\text{cell}[k+1][j]$ :
      for  $X$  in Nonterminals:
        if  $X \rightarrow YZ$  in Rules:
          addToCell(cell[i][j], X, Y, Z)
```



Dealing with ambiguity: Probabilistic Context-Free Grammars (PCFGs)

Grammars are ambiguous

A grammar might generate multiple trees for a sentence:



What's the most likely parse τ for sentence S ?

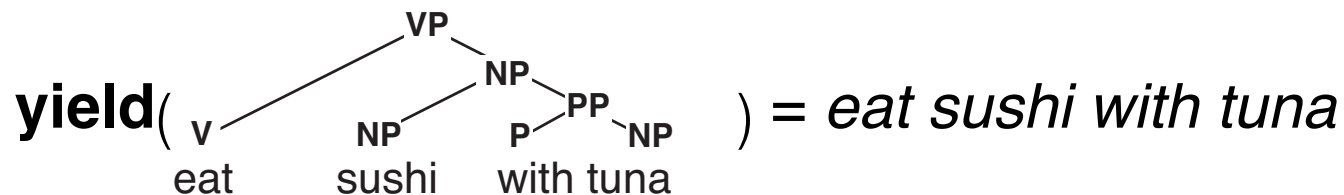
We need a model of $P(\tau \mid S)$

Computing $P(\tau \mid S)$

Using Bayes' Rule:

$$\begin{aligned}\arg \max_{\tau} P(\tau \mid S) &= \arg \max_{\tau} \frac{P(\tau, S)}{P(S)} \\ &= \arg \max_{\tau} P(\tau, S) \\ &= \arg \max_{\tau} P(\tau) \quad \text{if } S = \text{yield}(\tau)\end{aligned}$$

The **yield of a tree** is the string of terminal symbols that can be read off the leaf nodes



Computing $P(\tau)$

T is the (infinite) set of all trees in the language:

$$L = \{s \in \Sigma^* \mid \exists \tau \in T : \text{yield}(\tau) = s\}$$

We need to define $P(\tau)$ such that:

$$\forall \tau \in T : 0 \leq P(\tau) \leq 1$$

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

The set T is generated by a context-free grammar

$S \rightarrow NP VP$	$VP \rightarrow Verb NP$	$NP \rightarrow Det Noun$
$S \rightarrow S conj S$	$VP \rightarrow VP PP$	$NP \rightarrow NP PP$
$S \rightarrow \dots\dots$	$VP \rightarrow \dots\dots$	$NP \rightarrow \dots\dots$

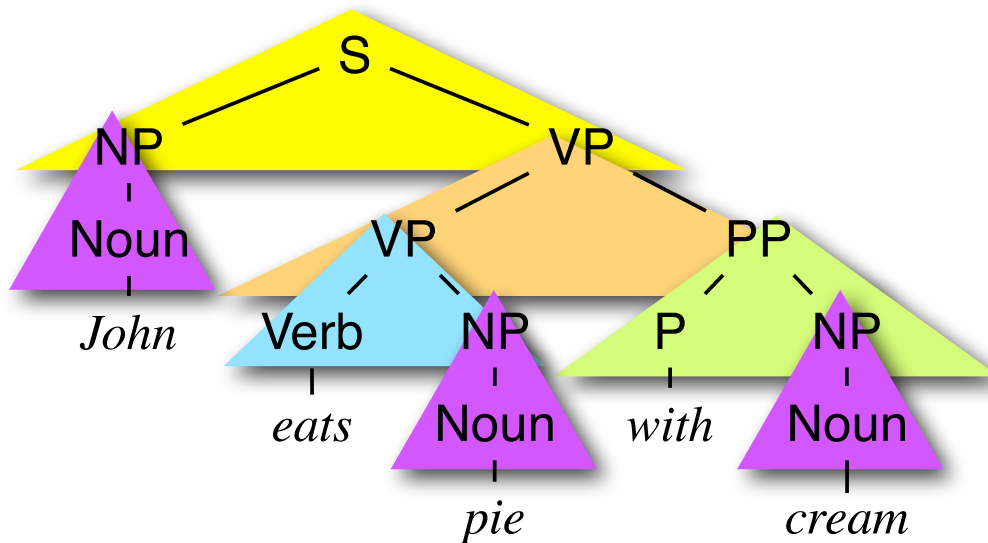
Probabilistic Context-Free Grammars

For every nonterminal X , define a probability distribution $P(X \rightarrow \alpha \mid X)$ over all rules with the same LHS symbol X :

S	\rightarrow	NP VP	0.8
S	\rightarrow	S conj S	0.2
NP	\rightarrow	Noun	0.2
NP	\rightarrow	Det Noun	0.4
NP	\rightarrow	NP PP	0.2
NP	\rightarrow	NP conj NP	0.2
VP	\rightarrow	Verb	0.4
VP	\rightarrow	Verb NP	0.3
VP	\rightarrow	Verb NP NP	0.1
VP	\rightarrow	VP PP	0.2
PP	\rightarrow	P NP	1.0

Computing $P(\tau)$ with a PCFG

The probability of a tree τ is the product of the probabilities of all its rules:



$$P(\tau) = 0.8 \times 0.3 \times 0.2 \times 1.0 \times 0.2^3$$
$$= 0.00384$$

S	→ NP VP	0.8
S	→ S conj S	0.2
NP	→ Noun	0.2
NP	→ Det Noun	0.4
NP	→ NP PP	0.2
NP	→ NP conj NP	0.2
VP	→ Verb	0.4
VP	→ Verb NP	0.3
VP	→ Verb NP NP	0.1
VP	→ VP PP	0.2
PP	→ P NP	1.0

PCFG parsing (decoding): Probabilistic CKY

Probabilistic CKY: Viterbi

Like standard CKY, but with probabilities.

Finding the most likely tree $\operatorname{argmax}_{\tau} P(\tau, s)$ is similar to Viterbi for HMMs:

Initialization: every chart entry that corresponds to a **terminal** (entries X in $\text{cell}[i][i]$) has a Viterbi probability $P_{\text{VIT}}(X_{[i][i]}) = 1$

Recurrence: For every entry that corresponds to a **non-terminal** x in $\text{cell}[i][j]$, keep only the highest-scoring pair of backpointers to any pair of children (Y in $\text{cell}[i][k]$ and Z in $\text{cell}[k+1][j]$):
$$P_{\text{VIT}}(X_{[i][j]}) = \operatorname{argmax}_{Y,Z,k} P_{\text{VIT}}(Y_{[i][k]}) \times P_{\text{VIT}}(Z_{[k+1][j]}) \times P(X \rightarrow YZ \mid X)$$

Final step: Return the Viterbi parse for the start symbol S in the top $\text{cell}[1][n]$.

Probabilistic CKY

Input: POS-tagged sentence

John_N eats_V pie_N with_P cream_N

John	eats	pie	with	cream	
N NP 1.0 0.2	S $0.8 \cdot 0.2 \cdot 0.3$	S $0.8 \cdot 0.2 \cdot 0.06$	S $0.2 \cdot 0.0036 \cdot 0.8$		John
	V VP 1.0 0.3	VP $1 \cdot 0.3 \cdot 0.2 = 0.06$	VP $\max(1.0 \cdot 0.008 \cdot 0.3, 0.06 \cdot 0.2 \cdot 0.3)$		eats
		N NP 1.0 0.2		NP $0.2 \cdot 0.2 \cdot 0.2 = 0.008$	pie
			P 1.0	PP $1 \cdot 1 \cdot 0.2$	with
				N NP 1.0 0.2	cream

S	→ NP VP	0.8
S	→ S conj S	0.2
NP	→ Noun	0.2
NP	→ Det Noun	0.4
NP	→ NP PP	0.2
NP	→ NP conj NP	0.2
VP	→ Verb	0.3
VP	→ Verb NP	0.3
VP	→ Verb NP NP	0.1
VP	→ VP PP	0.3
PP	→ P NP	1.0