

Syntax and Parsing

Yoav Goldberg
Bar Ilan University

Goals

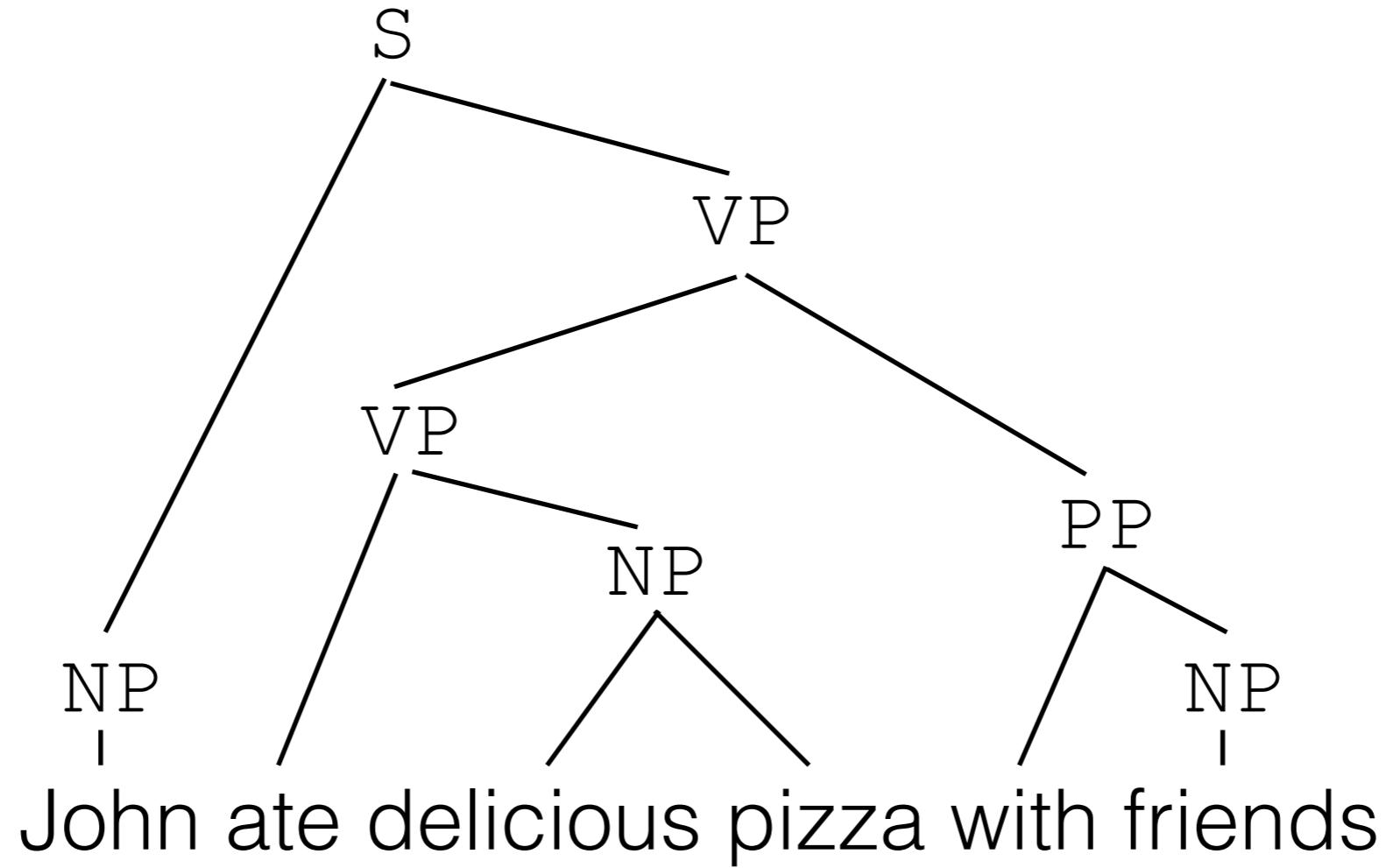
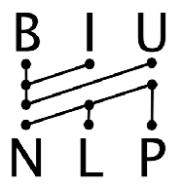
- What is parsing? why do we care?
- Phrase-based (constituency) trees
 - PCFG, CKY
- Dependency trees
 - Graph parsers, transition parsers

Natural Language Parsing

- ▶ Sentences in natural language have structure.
- ▶ Linguists create Linguistic Theories for defining this structure.
- ▶ The parsing problem is recovering that structure.

B I U
N L P

John ate delicious pizza with friends

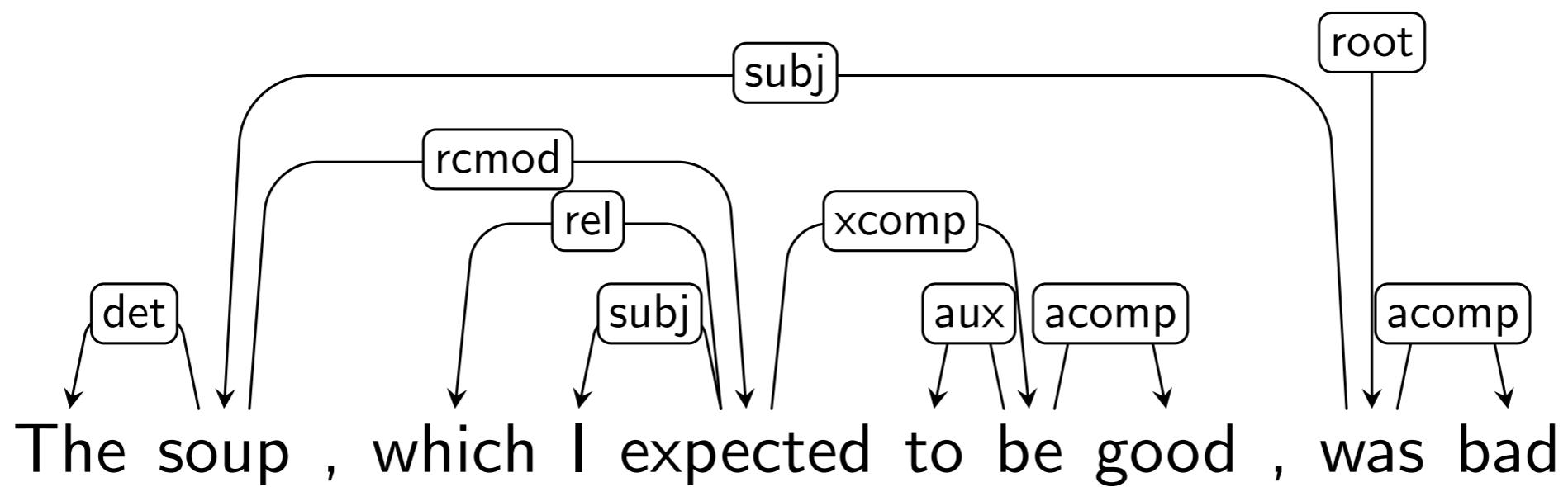


parsing

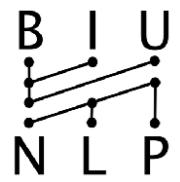
B I U
N L P

The soup , which I expected to be good , was bad

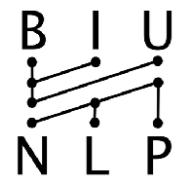
B I U
N L P



parsing

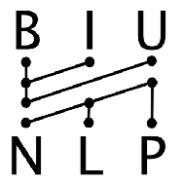


Why Parsing?



sentiment analysis

The soup , which I expected to be good , was bad

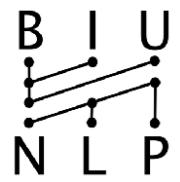


sentiment analysis

The soup , which I expected to be good , was bad



negative



sentiment analysis

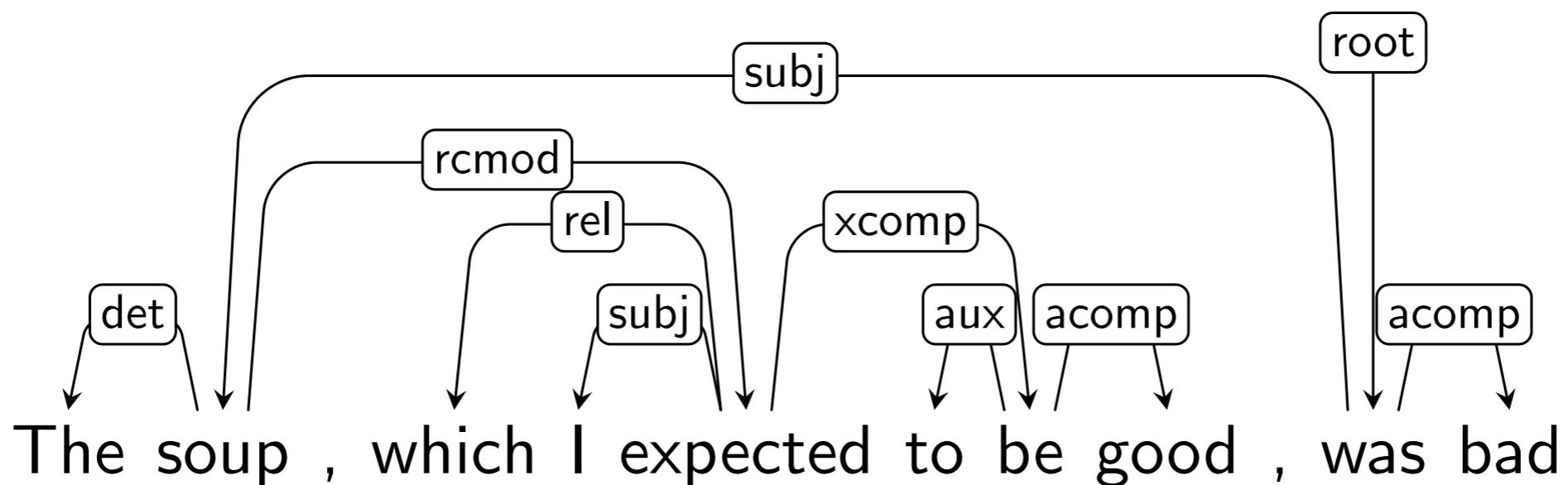
The soup , which I expected to be good , was bad



negative

B I U
N L P

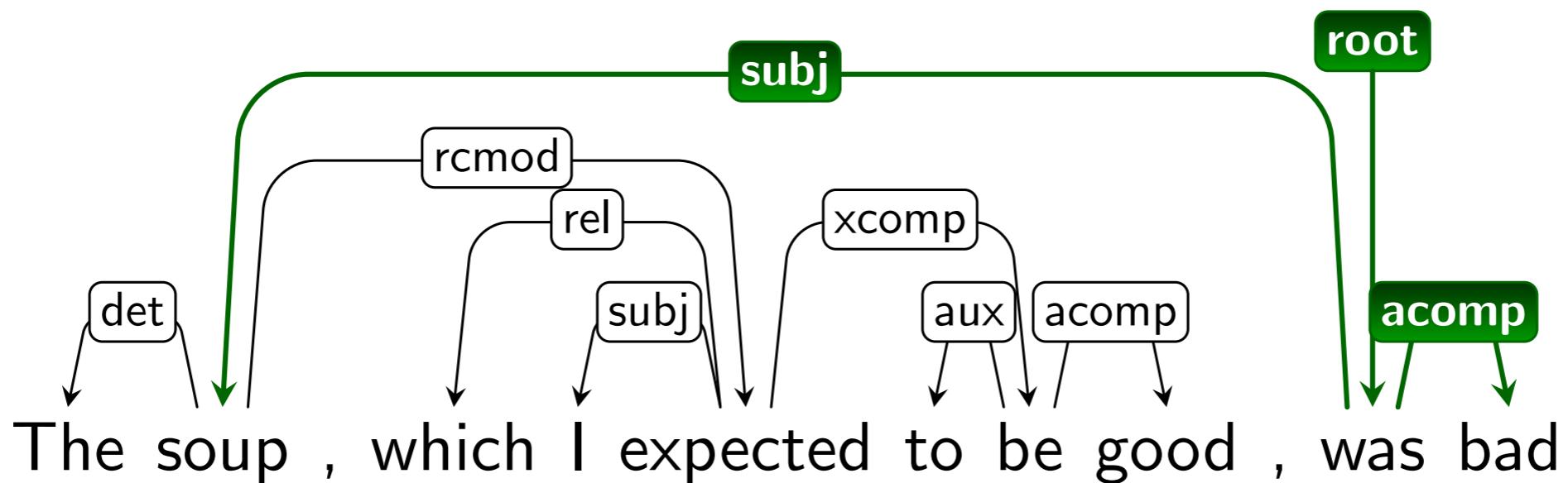
sentiment analysis



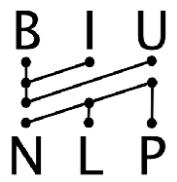
+
parsing

B I U
N L P

sentiment analysis



+
parsing

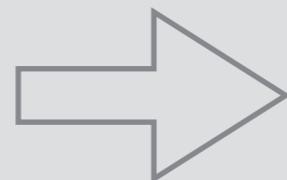


knowing the structure of the sentence
helps sentiment analysis

B | U
N L P

machine translation

German



English



über mehrere Jahre hatte niemand in dem Haus gelebt .



over several years , no one had lived in the house .

B | U
N L P

machine translation

German



English

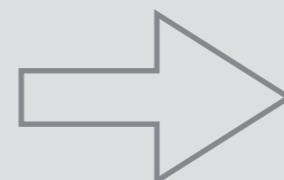


no one lived in the house for several years .
über mehrere Jahre hatte niemand in dem Haus gelebt .
↓
over several years , no one had lived in the house .

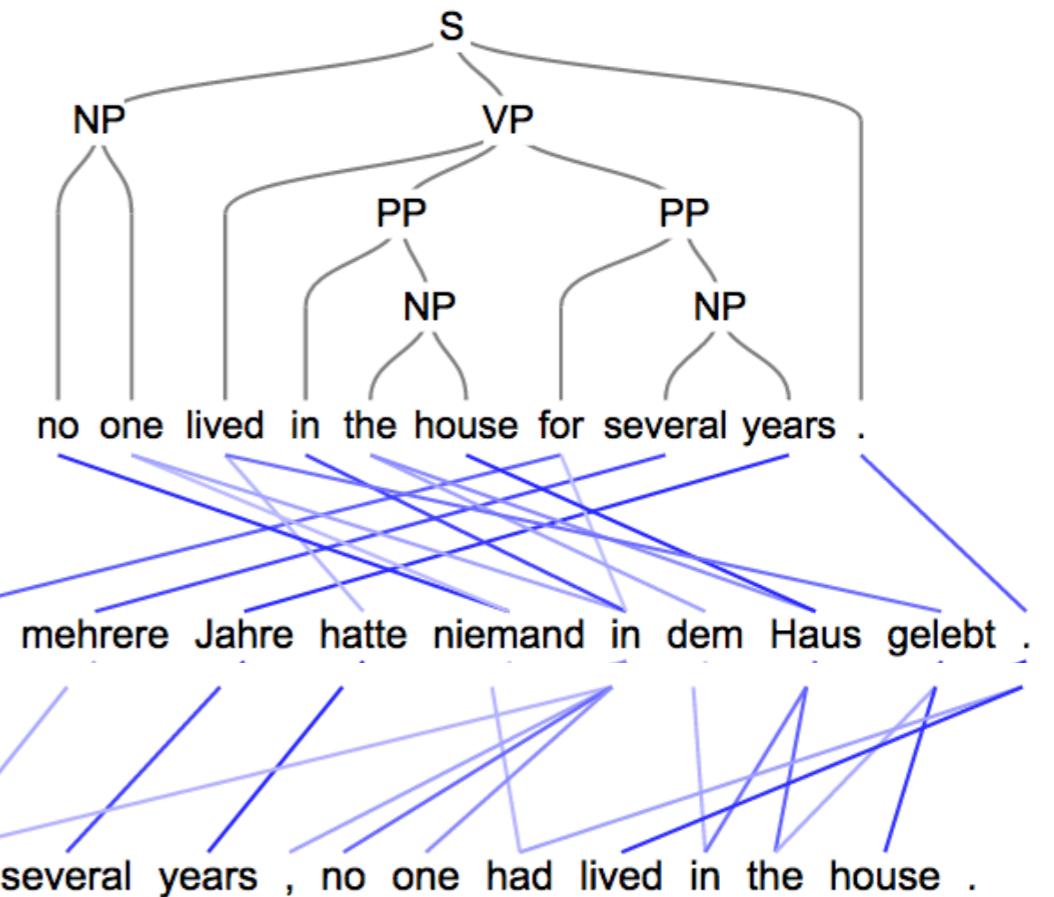
B | U
N L P

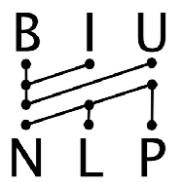
machine translation

German

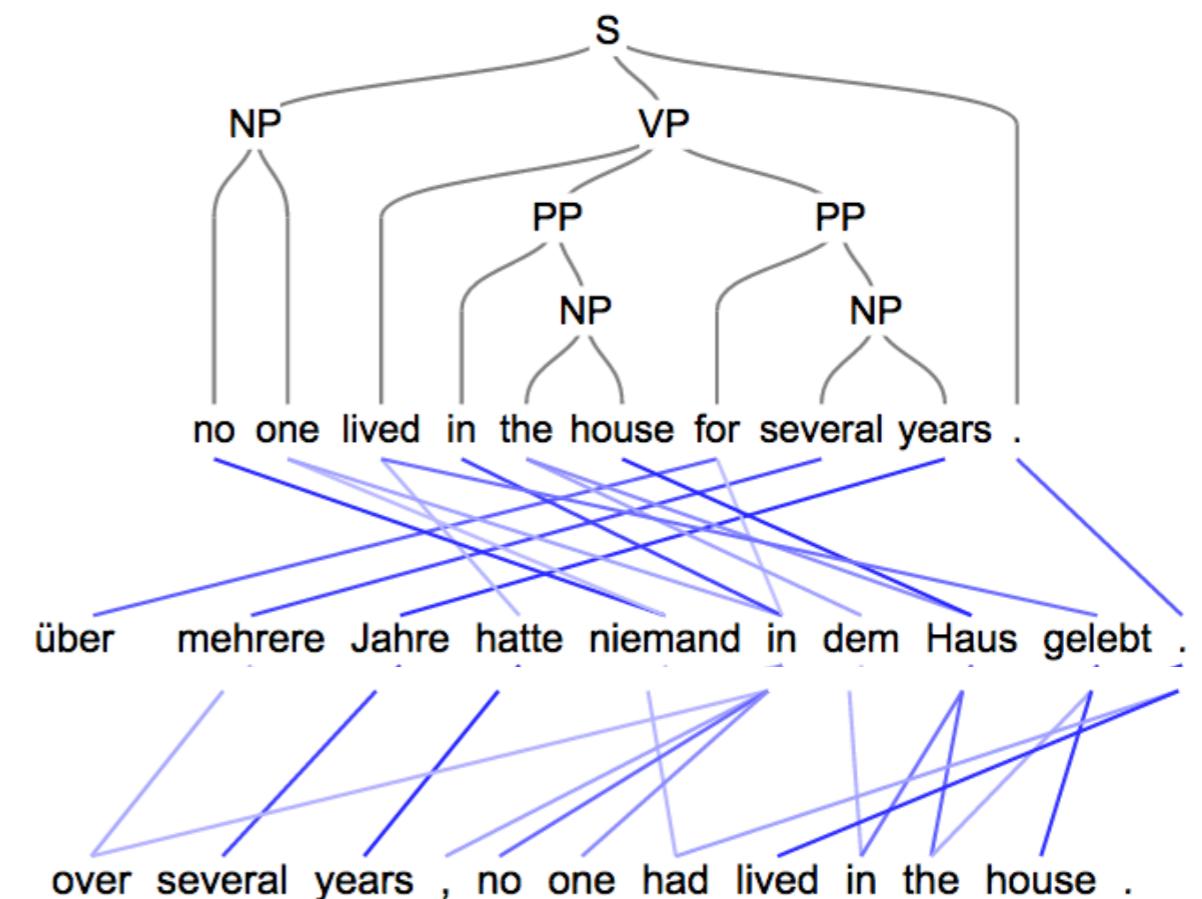


English



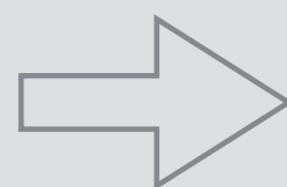


Knowing the structure of a sentence helps to translate better



B I U
N L P

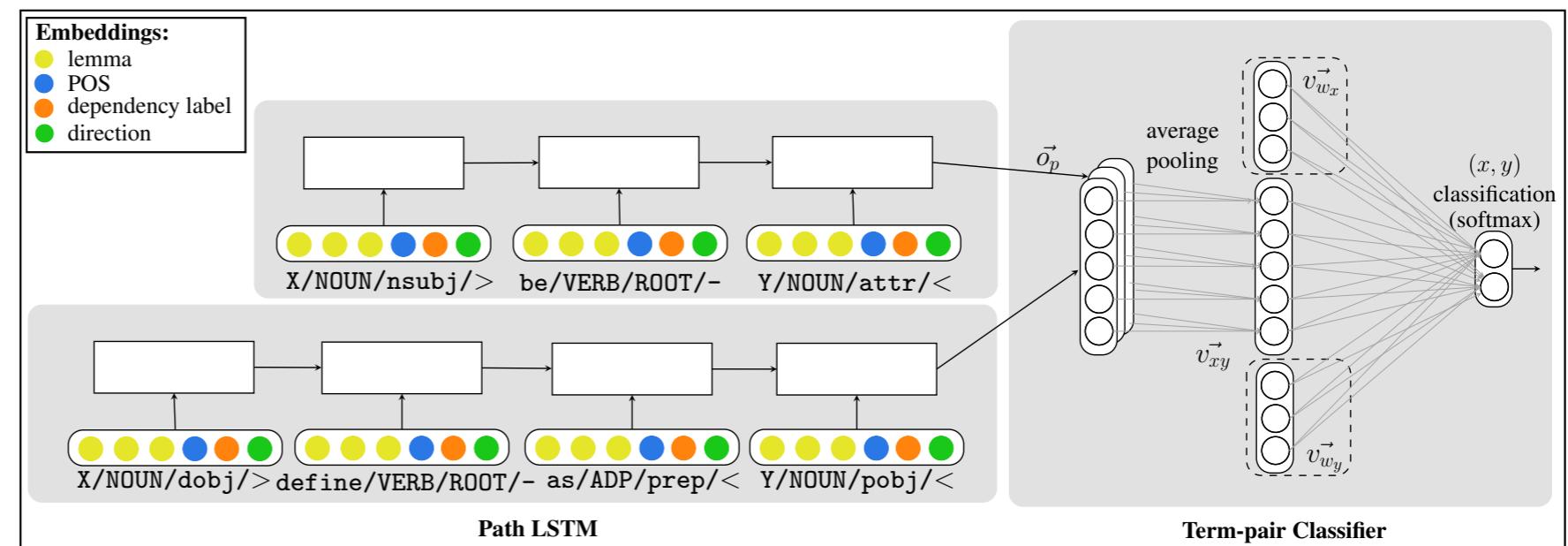
Hypernymy Extraction



"tuvalu" is a country
"ninjaken" is a weapon
"chlamydophila" is a bacteria



**hypernymy
detection
and
extraction**

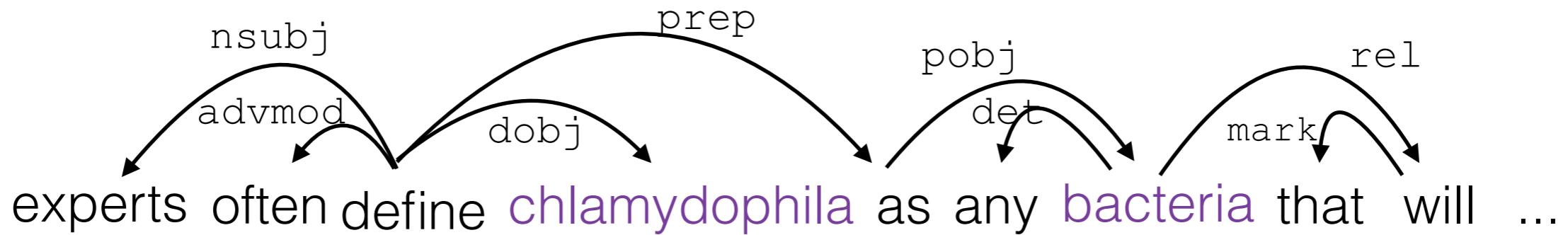


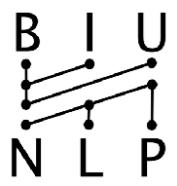
B I U
N L P

experts often define chlamydophila as any bacteria that will ...

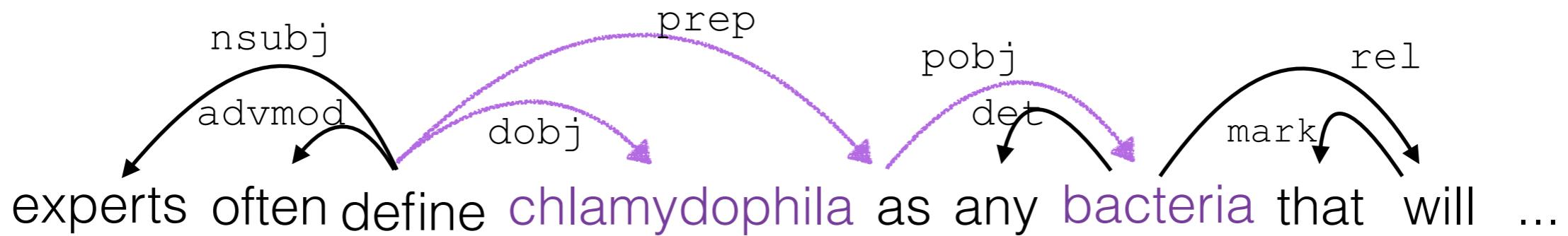
B I U
N L P

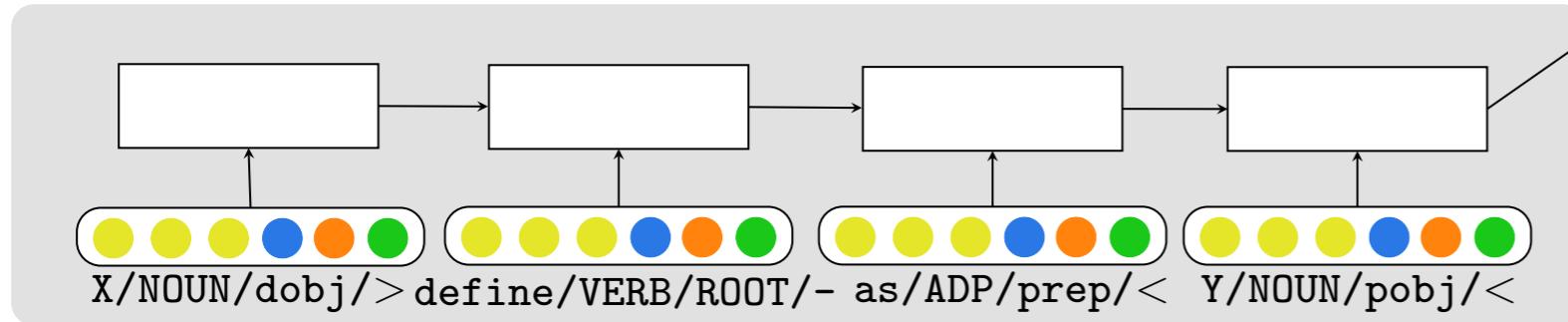
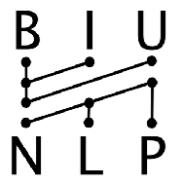
parse





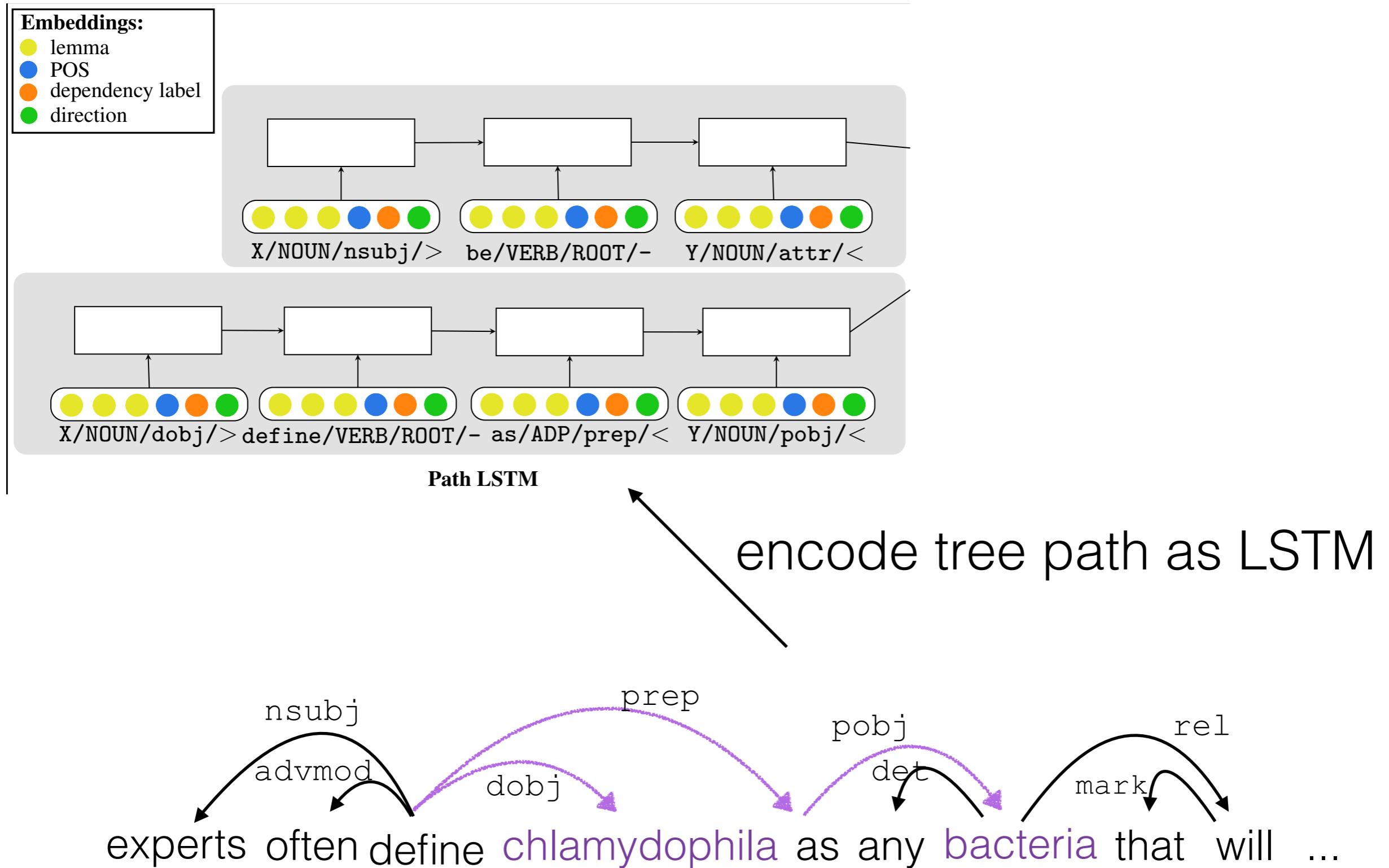
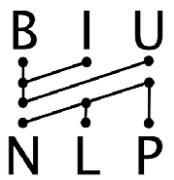
find tree path



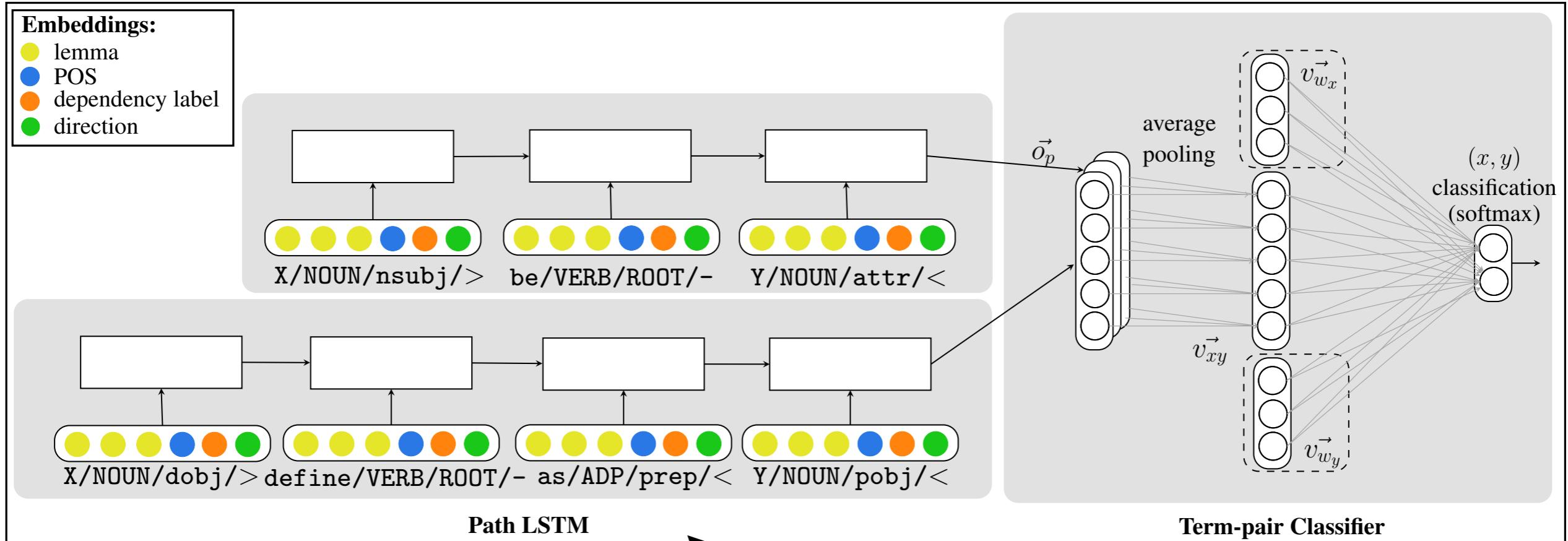


encode tree path as LSTM

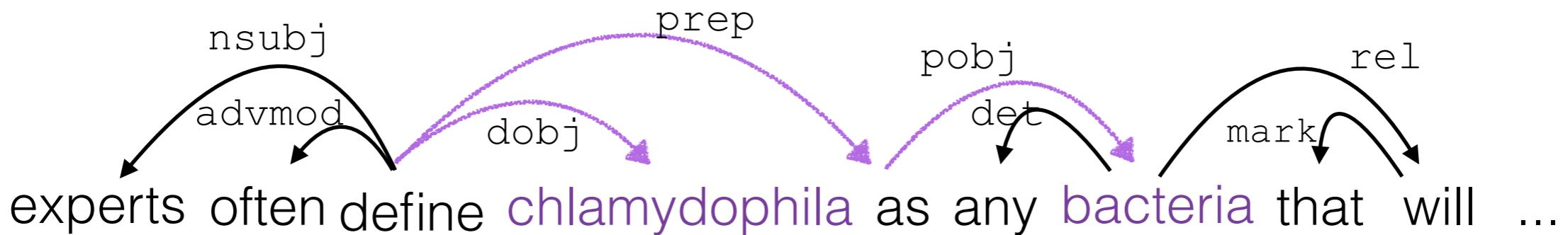
experts often define **chlamydophila** as any **bacteria** that will ...

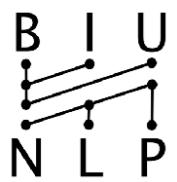


B I U
N L P



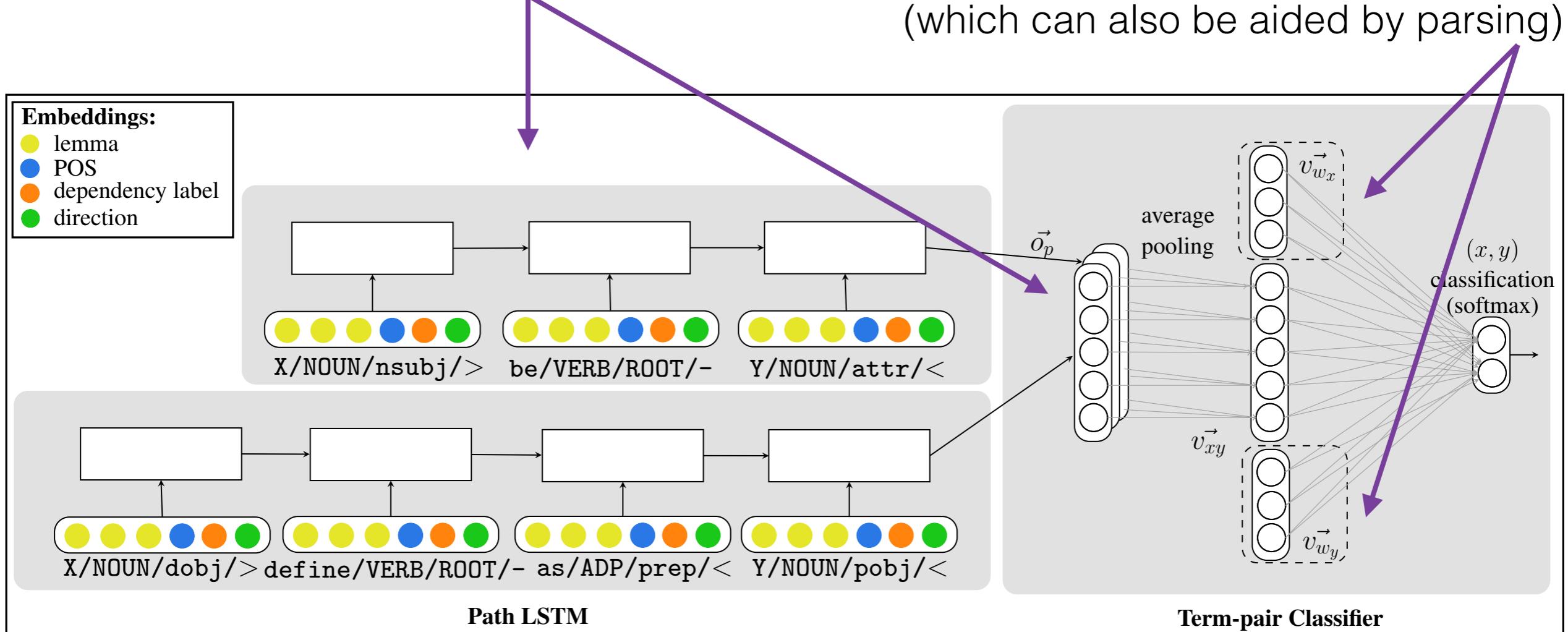
encode tree path as LSTM





based on parse-tree paths

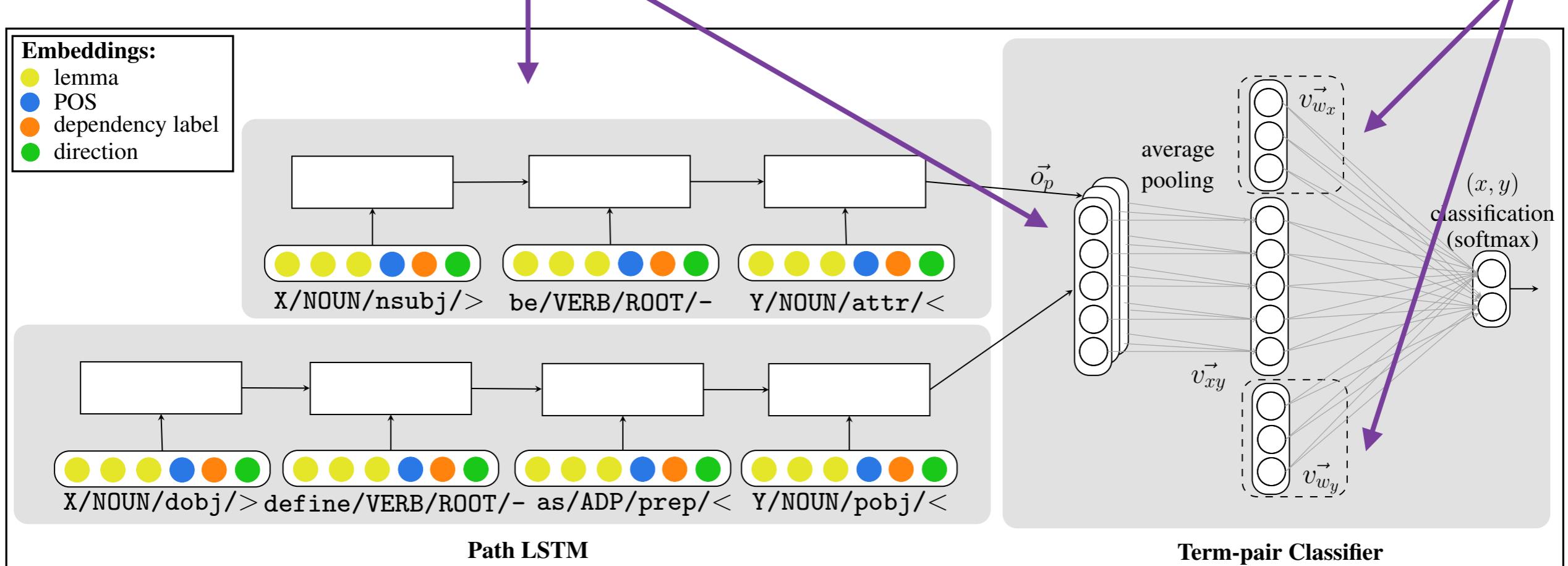
based on
distributional semantics
(which can also be aided by parsing)



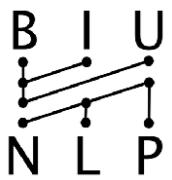
B I U
N L P

based on parse-tree paths

based on
distributional semantics
(which can also be aided by parsing)



**substantially improves results on many lexical
relation extraction tasks and datasets**

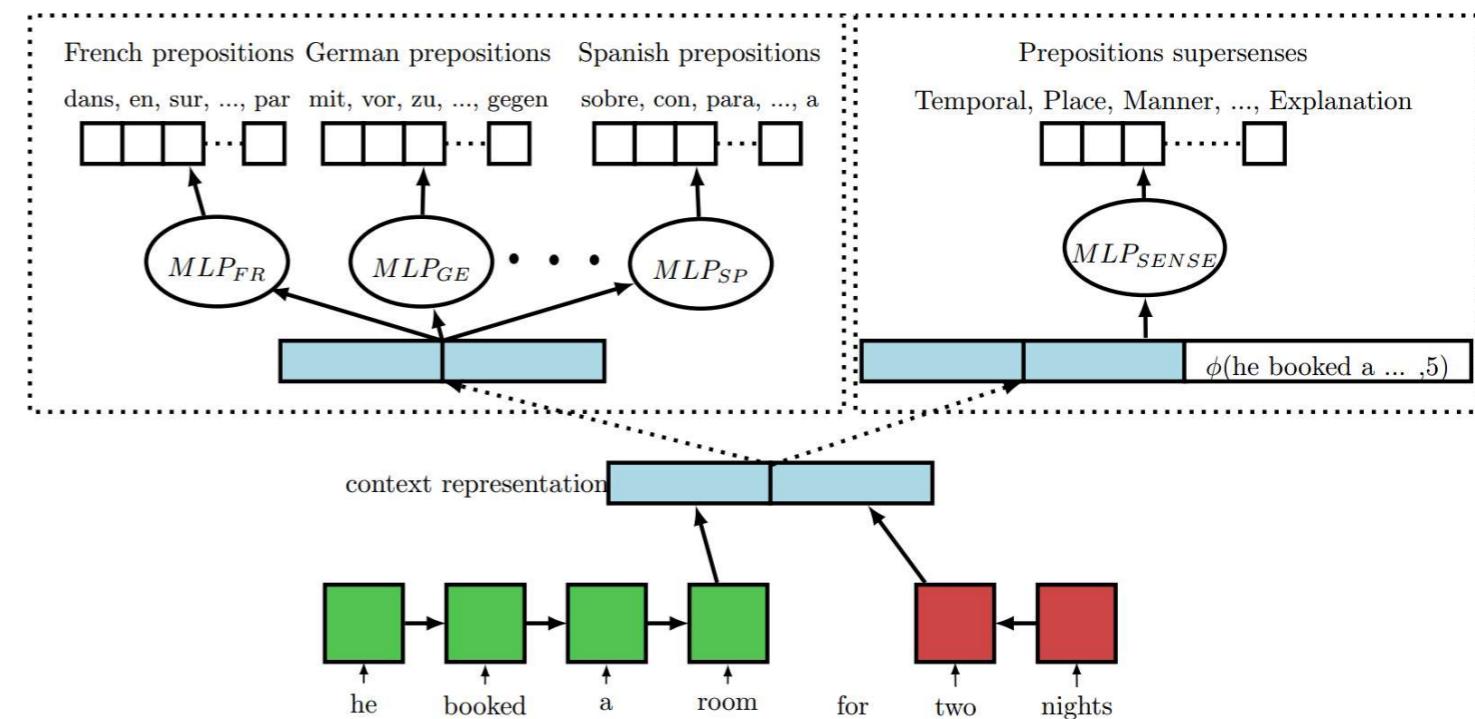


Preposition Sense Disambiguation

I met him **for** lunch → Purpose
He paid **for** me → Beneficiary
We sat there **for** hours → Duration



**preposition
sense
disambiguation**
+
semisup on multilingual data

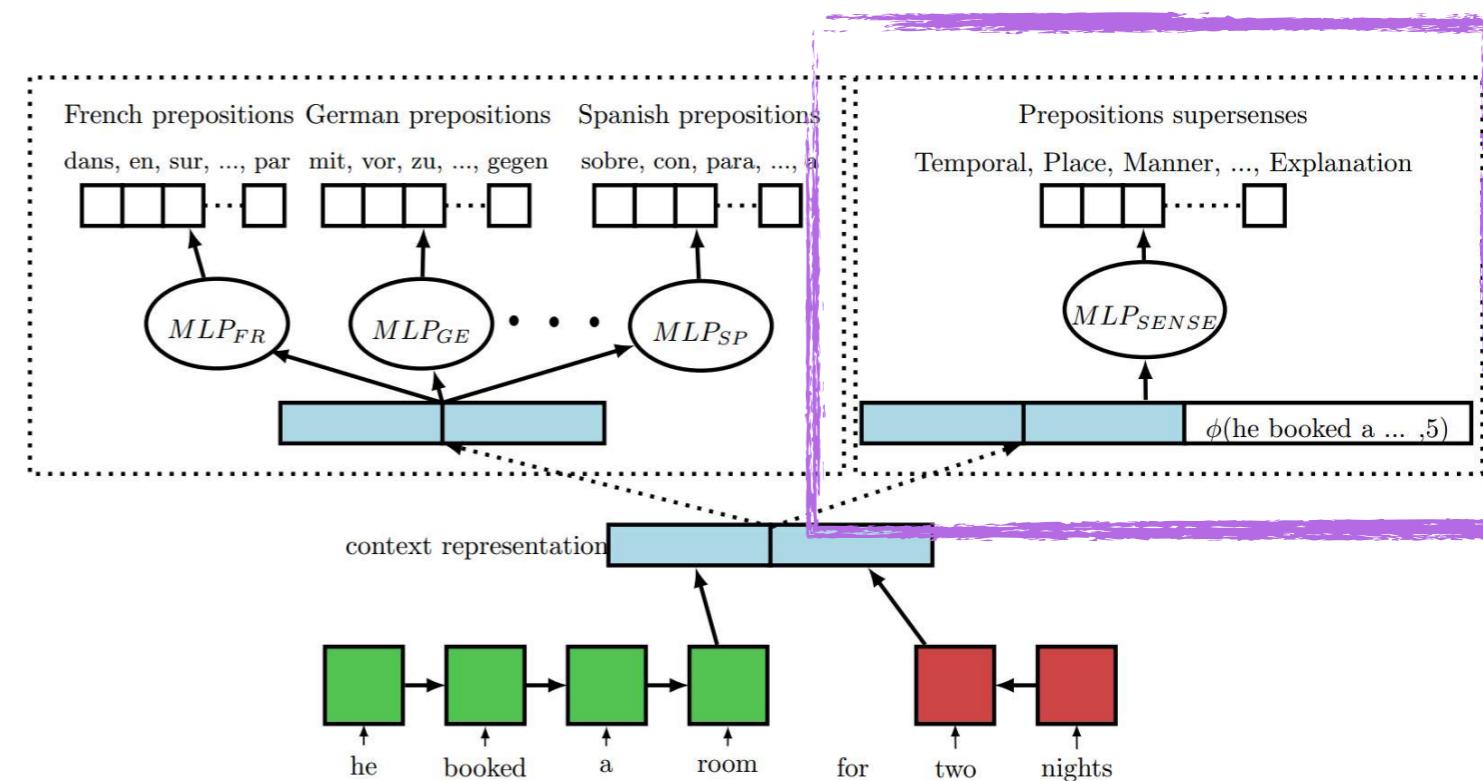


Preposition Sense Disambiguation

I met him for lunch	→	Purpose
He paid for me	→	Beneficiary
We sat there for hours	→	Duration



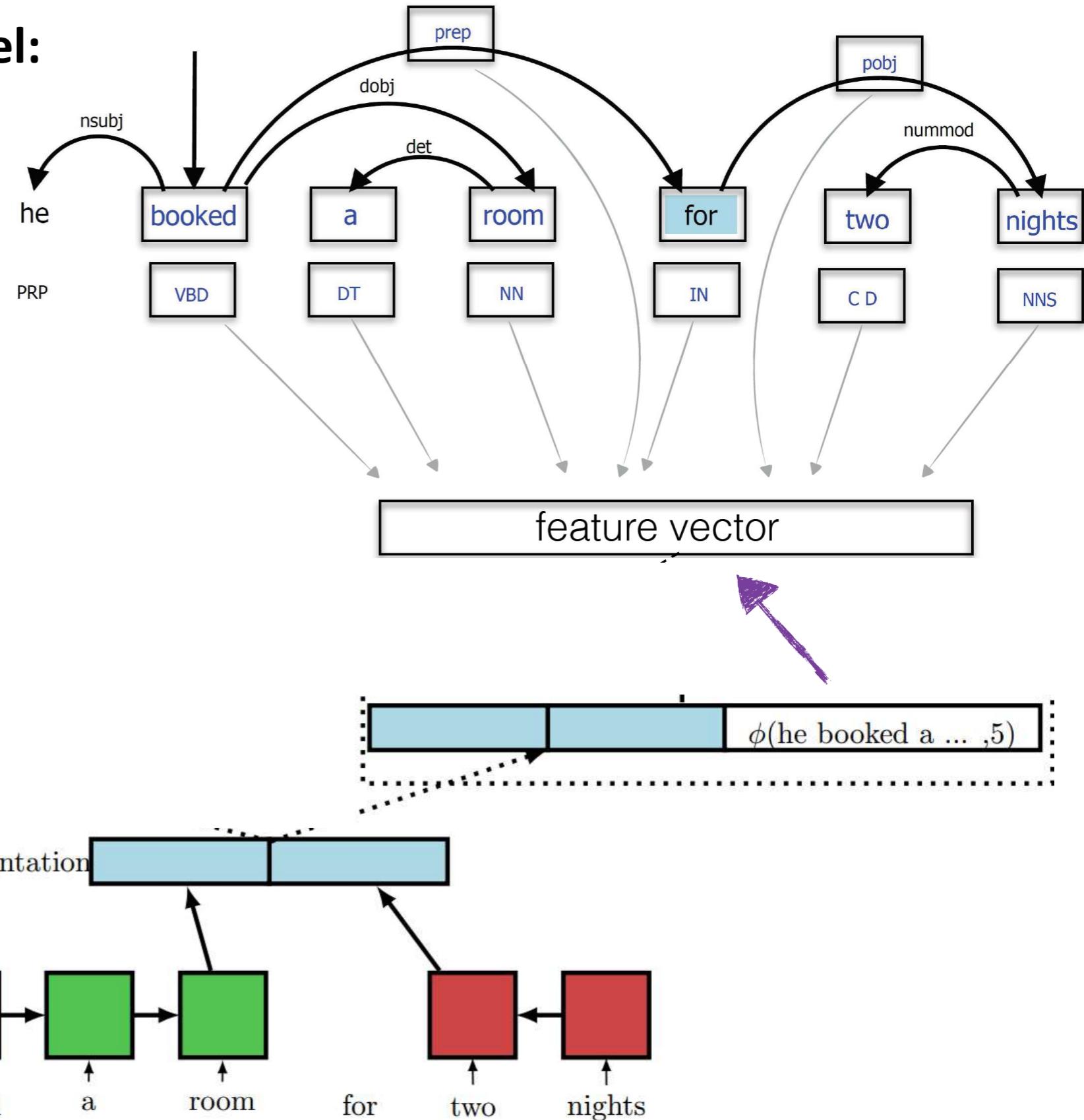
**preposition
sense
disambiguation**
+
semisup on multilingual data

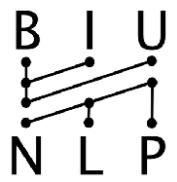


The features and the model:

The features are similar to those used in previous works. Features are extracted from:

- 2-words-window
- Head and modifier of the preposition





parse-trees play a central role
in many applications.

Nice property of syntax – no need to understand

Can recover structure without understanding the words

“the plumpets and goomps ghoked the kolp”

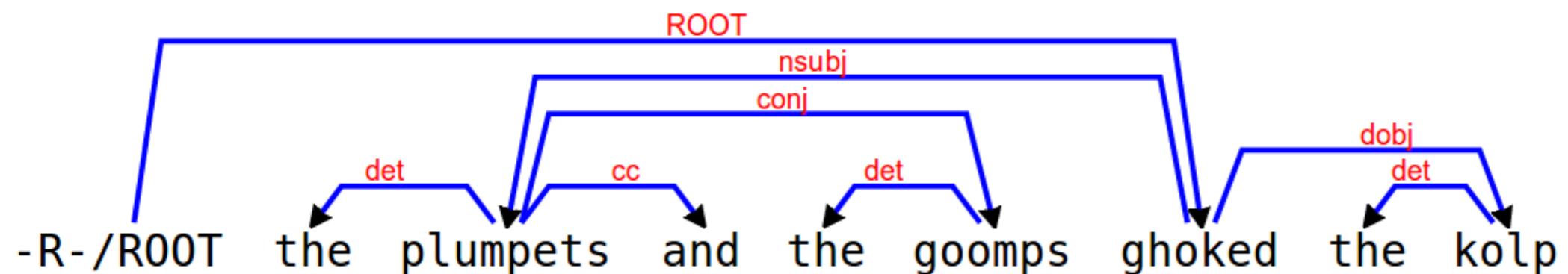
- ▶ there is something called a plumpet. more than 1.
- ▶ there is a similar thing called a goomp. more than 1.
- ▶ plumpets and goomps are together.
- ▶ there is an action called ghoking.
- ▶ ...

[u.cs.biu.ac.il/~yogo/parse/index?text=the+plumpets+and+the+goomps+ghoked+the+kolp](#)

This page is in Hebrew Would you like to translate it? [Translate](#) [Nope](#) [Never translate Hebrew](#)

the plumpets and the goomps ghoked the kolp

Parse



Automatic parsers get it right too

Parsing is
structured prediction

Natural Language Parsing

Previously

- ▶ Structure is a sequence.
- ▶ Each item can be tagged.
- ▶ We can mark some spans.

Natural Language Parsing

Previously

- ▶ Structure is a sequence.
- ▶ Each item can be tagged.
- ▶ We can mark some spans.

Today

- ▶ Hierarchical Structure.

Hierarchical Structure?

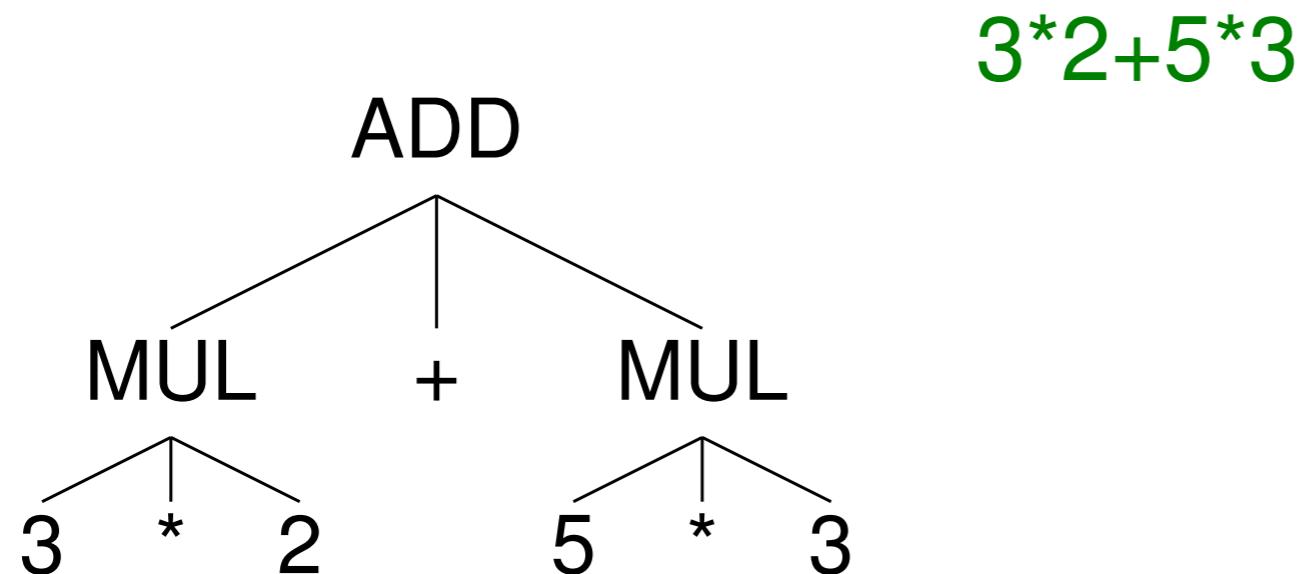
Structure

Example 1: math

3*2+5*3

Structure

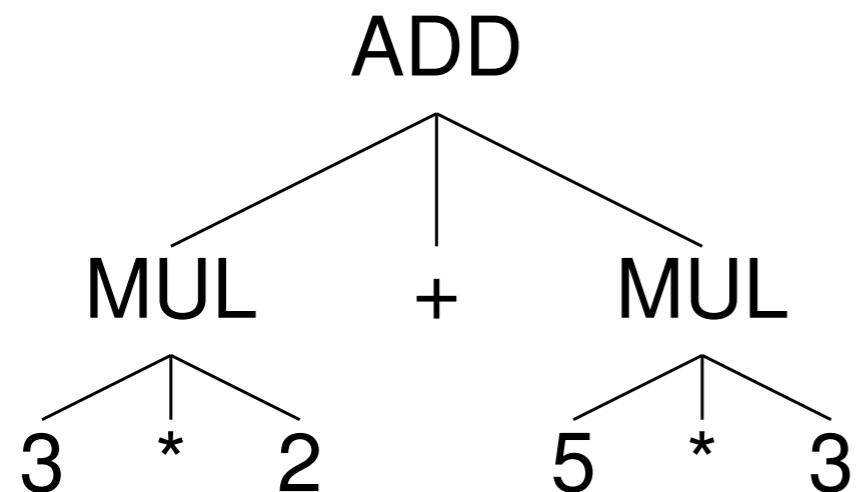
Example 1: math



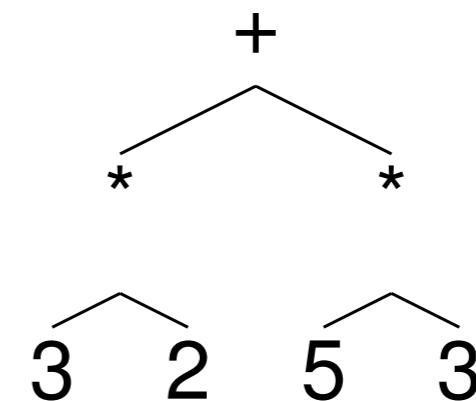
$3 * 2 + 5 * 3$

Structure

Example 1: math



$3 * 2 + 5 * 3$



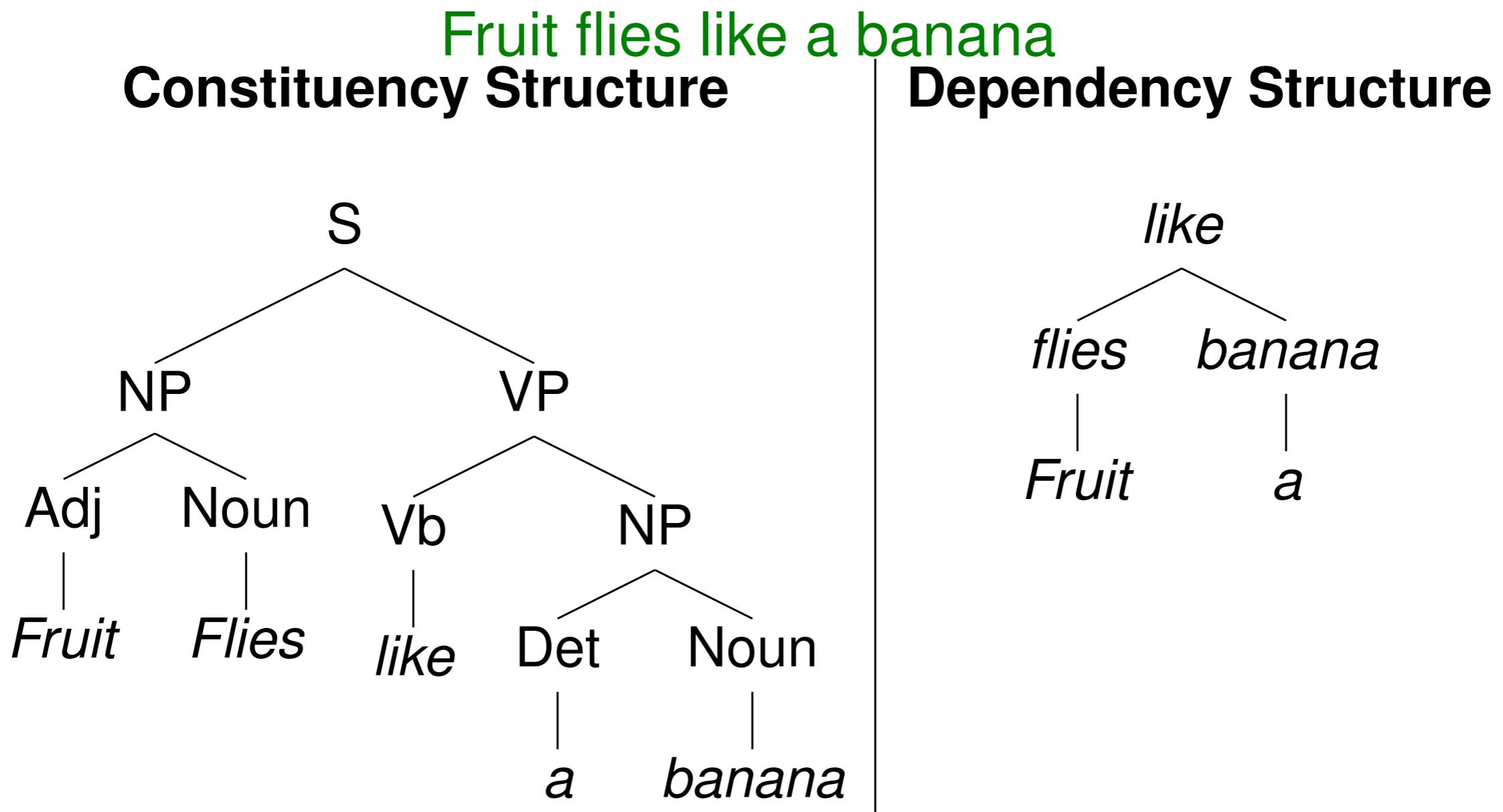
Structure

Example 2: Language Data

Fruit flies like a banana

Structure

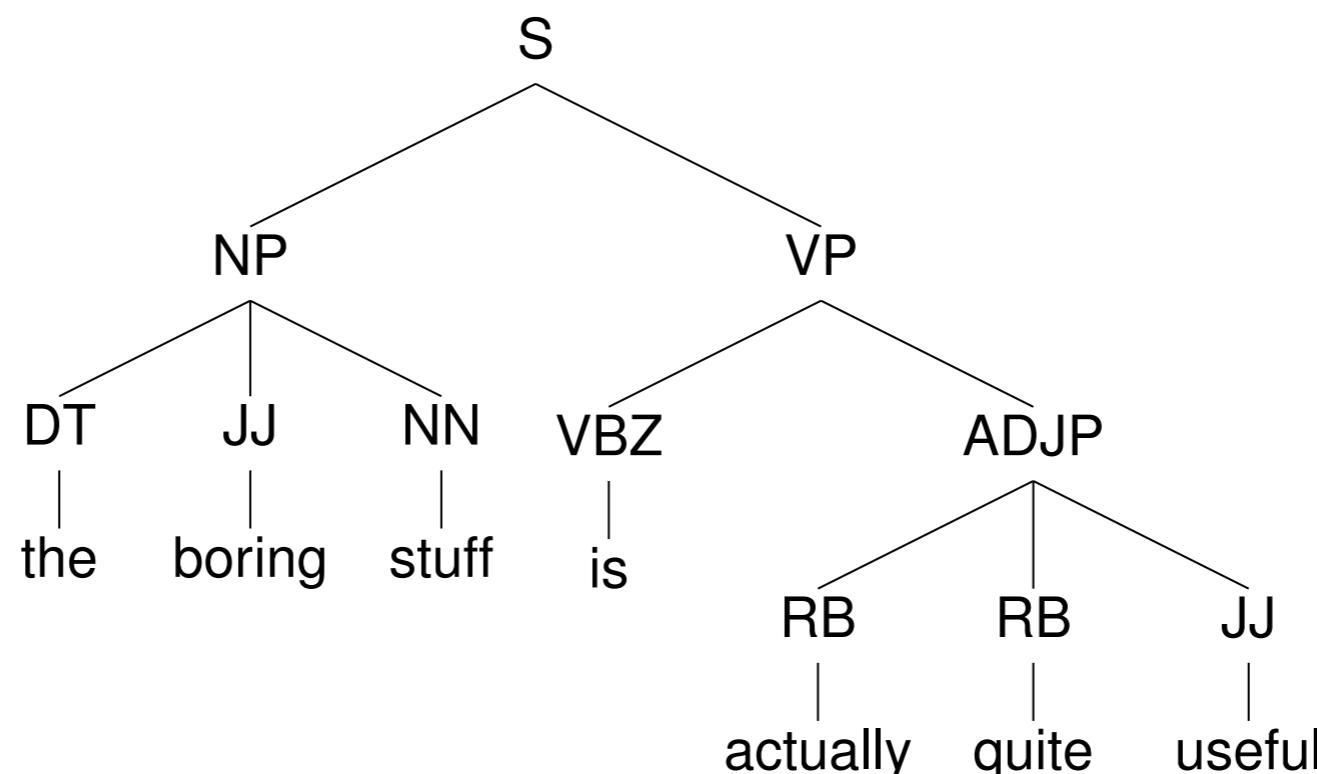
Example 2: Language Data



Structure

Constituency Structure

- ▶ In this part we concentrate on **Constituency Parsing**: mapping from sentences to trees with labeled nodes and the sentence words at the leaves.



Why is parsing hard?

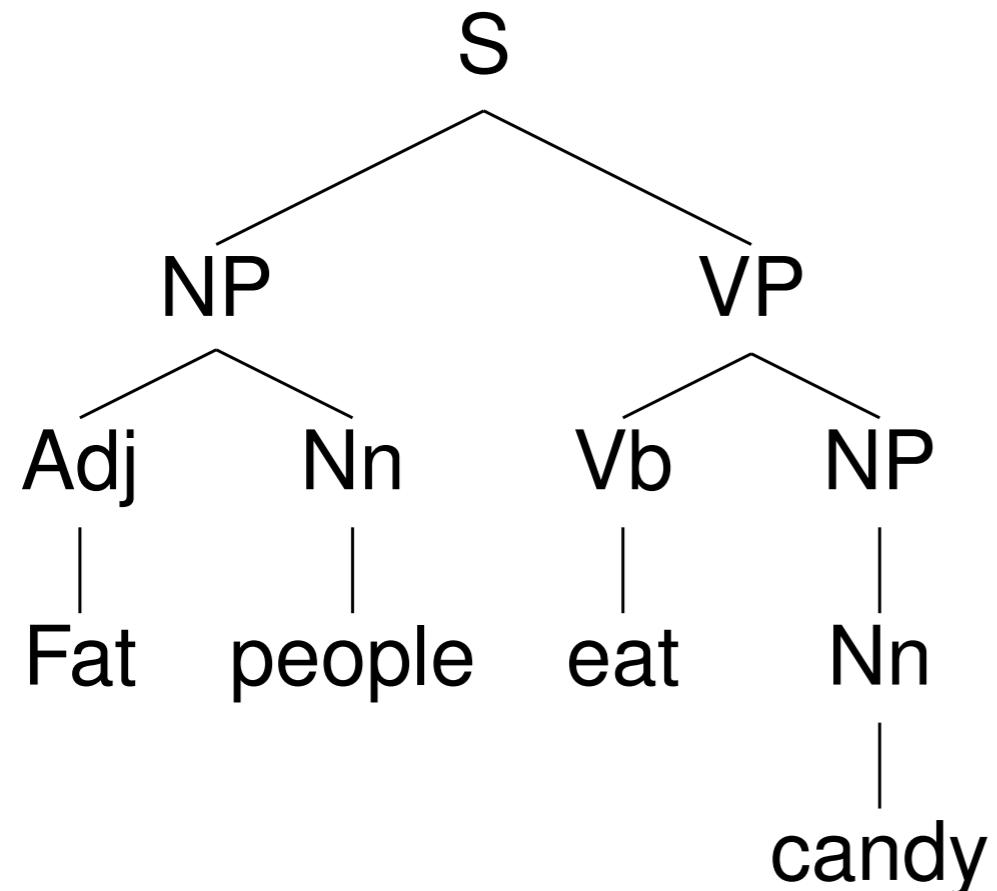
Ambiguity

Fat people eat candy

Why is parsing hard?

Ambiguity

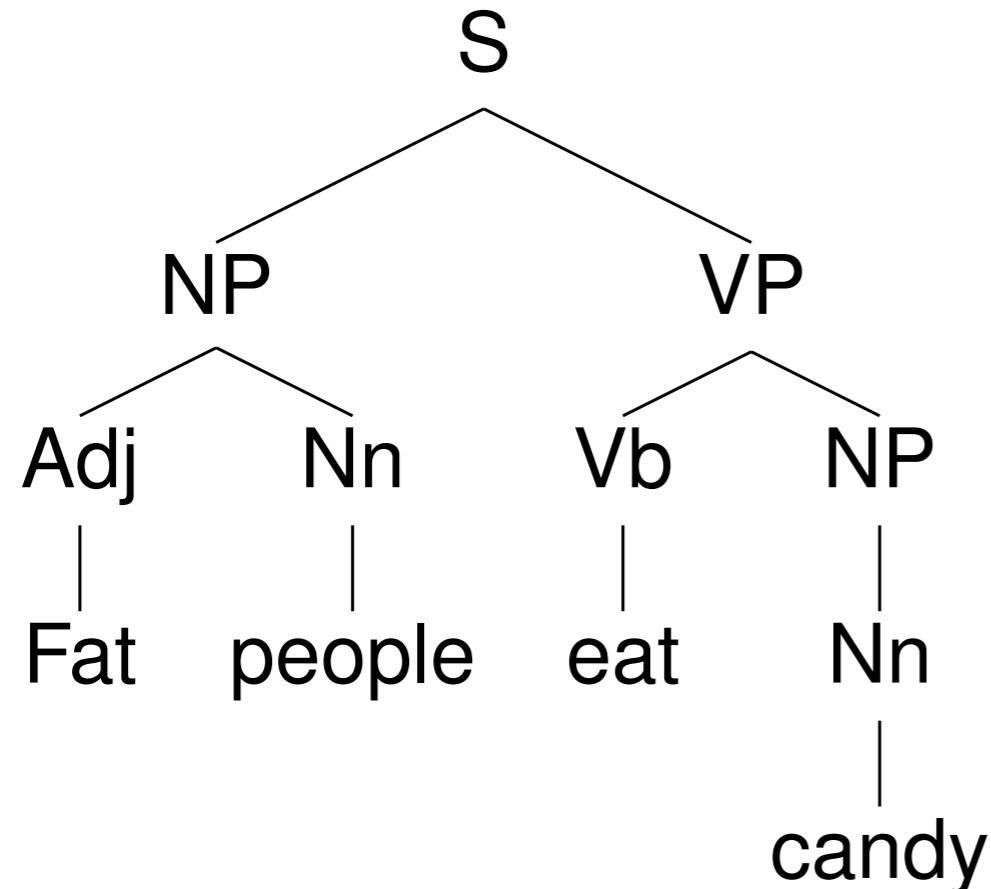
Fat people eat candy



Why is parsing hard?

Ambiguity

Fat people eat candy

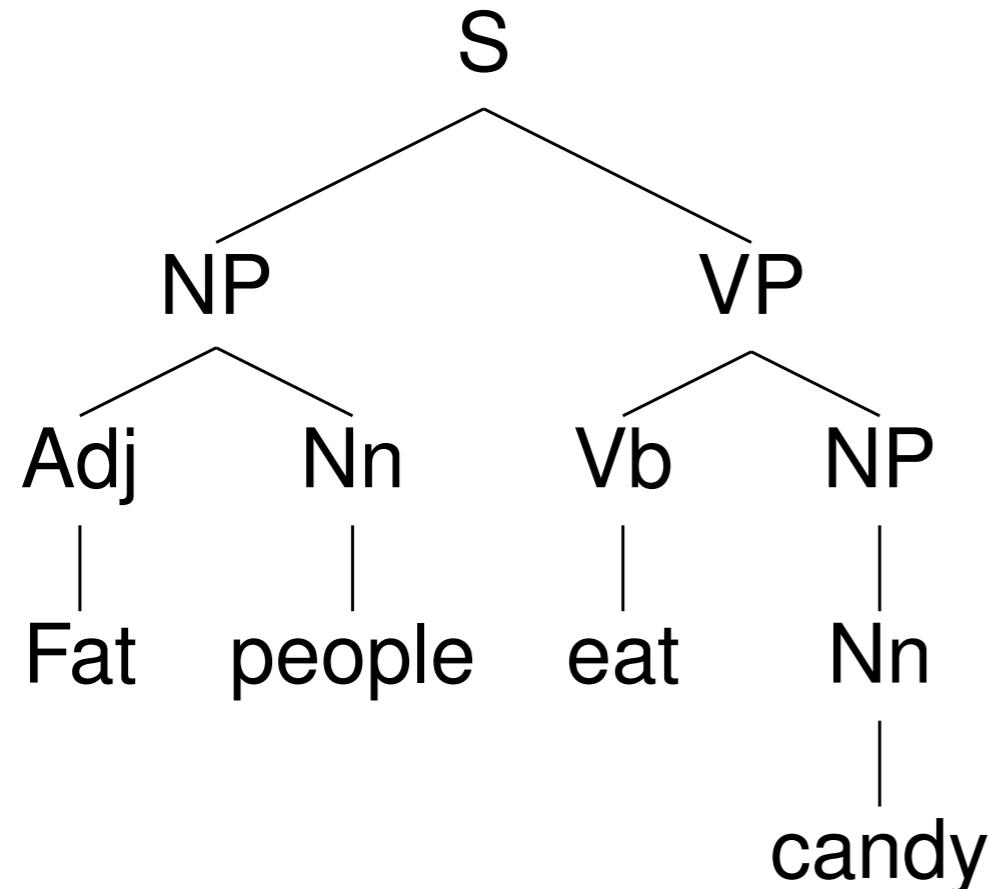


Fat people eat accumulates

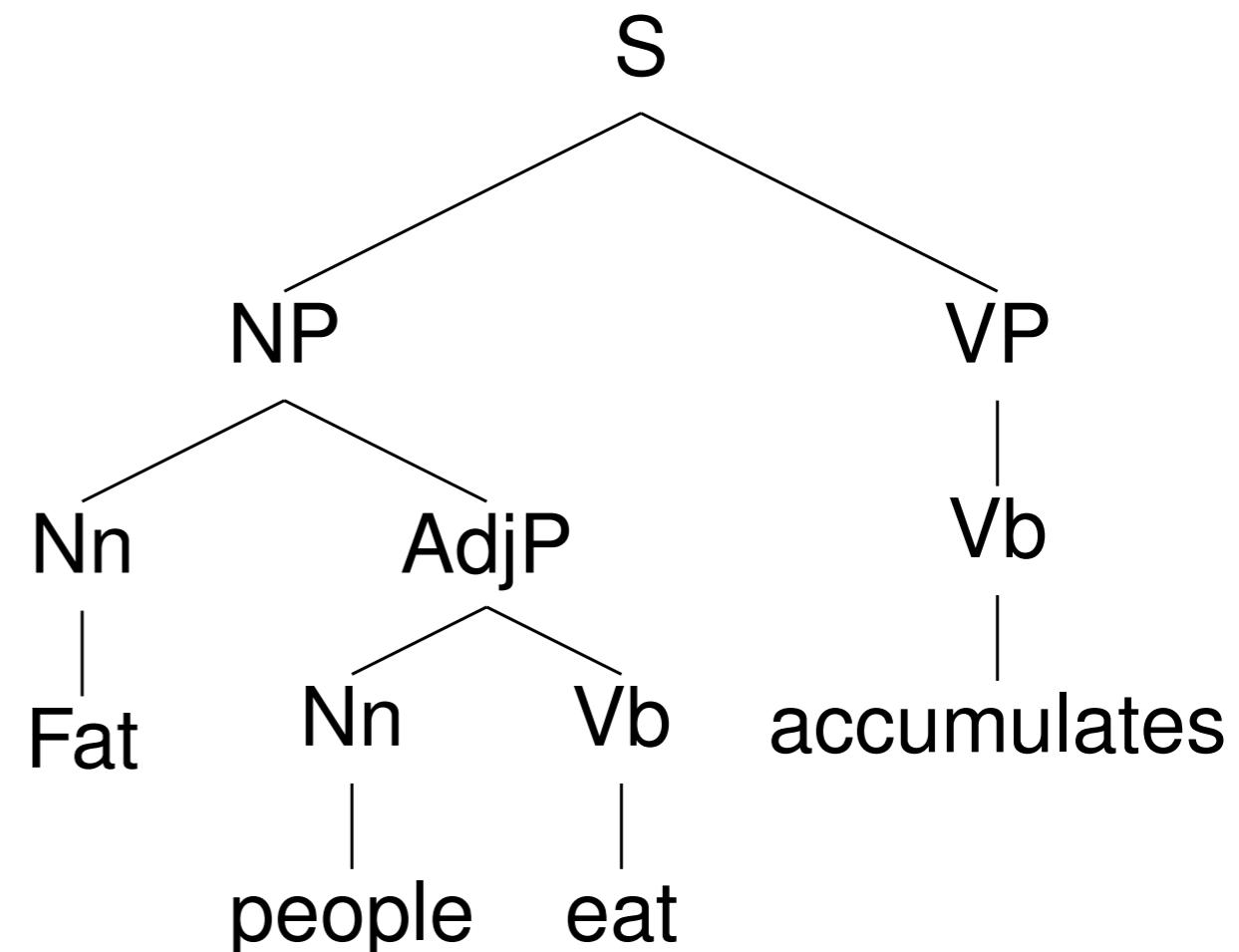
Why is parsing hard?

Ambiguity

Fat people eat candy



Fat people eat accumulates



Why is parsing hard?

Ambiguity

- ▶ I ate pizza with anchovies
- ▶ I ate pizza with friends

Why is parsing hard?

Real Sentences are long...

“Former Beatle Paul McCartney today was ordered to pay nearly \$50M to his estranged wife as their bitter divorce battle came to an end . ”

“Welcome to our Columbus hotels guide, where you’ll find honest, concise hotel reviews, all discounts, a lowest rate guarantee, and no booking fees.”

Let's learn how to parse

Let's learn how to parse . . . but first lets review some stuff from
Automata and Formal Languages.

Context Free Grammars

A context free grammar $G = (N, \Sigma, R, S)$ where:

- ▶ N is a set of non-terminal symbols
- ▶ Σ is a set of terminal symbols
- ▶ R is a set of rules of the form $X \rightarrow Y_1 Y_2 \cdots Y_n$
for $n \geq 0, X \in N, Y_i \in (N \cup \Sigma)$
- ▶ $S \in N$ is a special start symbol

Context Free Grammars

a simple grammar

$$N = \{S, NP, VP, Adj, Det, Vb, Noun\}$$

$$\Sigma = \{fruit, flies, like, a, banana, tomato, angry\}$$

$$S = 'S'$$

$$R =$$

$$S \rightarrow NP \ VP$$

$$NP \rightarrow Adj \ Noun$$

$$NP \rightarrow Det \ Noun$$

$$VP \rightarrow Vb \ NP$$

$$Adj \rightarrow fruit$$

$$Noun \rightarrow flies$$

$$Vb \rightarrow like$$

$$Det \rightarrow a$$

$$Noun \rightarrow banana$$

$$Noun \rightarrow tomato$$

$$Adj \rightarrow angry$$

Left-most derivations

Left-most derivation is a sequence of strings s_1, \dots, s_n where

- ▶ $s_1 = S$ the start symbol
- ▶ $s_n \in \Sigma^*$, meaning s_n is only terminal symbols
- ▶ Each s_i for $i = 2 \dots n$ is derived from s_{i-1} by picking the left-most non-terminal X in s_{i-1} and replacing it by some β where $X \rightarrow \beta$ is a rule in R .

Left-most derivations

Left-most derivation is a sequence of strings s_1, \dots, s_n where

- ▶ $s_1 = S$ the start symbol
- ▶ $s_n \in \Sigma^*$, meaning s_n is only terminal symbols
- ▶ Each s_i for $i = 2 \dots n$ is derived from s_{i-1} by picking the left-most non-terminal X in s_{i-1} and replacing it by some β where $X \rightarrow \beta$ is a rule in R .

For example: [S],[NP VP],[Adj Noun VP], [fruit Noun VP], [fruit flies VP],[fruit flies Vb NP],[fruit flies like NP], [fruit flies like Det Noun], [fruit flies like a], [fruit flies like a banana]

Left-most derivation example

S

Left-most derivation example

S
NP VP

$S \rightarrow NP\ VP$

Left-most derivation example

S
NP VP
Adj Noun VP $\text{NP} \rightarrow \text{Adj Noun}$

Left-most derivation example

S

NP VP

Adj Noun VP

fruit Noun VP

Adj → fruit

Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP

Noun → flies

Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP

VP → Vb NP

Left-most derivation example

S

NP VP

Adj Noun VP

fruit Noun VP

fruit flies VP

fruit flies Vb NP

fruit flies like NP

Vb → like

Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun

NP → Det Noun

Left-most derivation example

S

NP VP

Adj Noun VP

fruit Noun VP

fruit flies VP

fruit flies Vb NP

fruit flies like NP

fruit flies like Det Noun

fruit flies like a Noun

Det → a

Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun
fruit flies like a Noun
fruit flies like a banana

Noun → banana

Left-most derivation example

S

NP VP

Adj Noun VP

fruit Noun VP

fruit flies VP

fruit flies Vb NP

fruit flies like NP

fruit flies like Det Noun

fruit flies like a Noun

fruit flies like a banana

- ▶ The resulting derivation can be written as a tree.

Left-most derivation example

S

NP VP

Adj Noun VP

fruit Noun VP

fruit flies VP

fruit flies Vb NP

fruit flies like NP

fruit flies like Det Noun

fruit flies like a Noun

fruit flies like a banana

- ▶ The resulting derivation can be written as a tree.
- ▶ Many trees can be generated.

Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-

$Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-

$Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

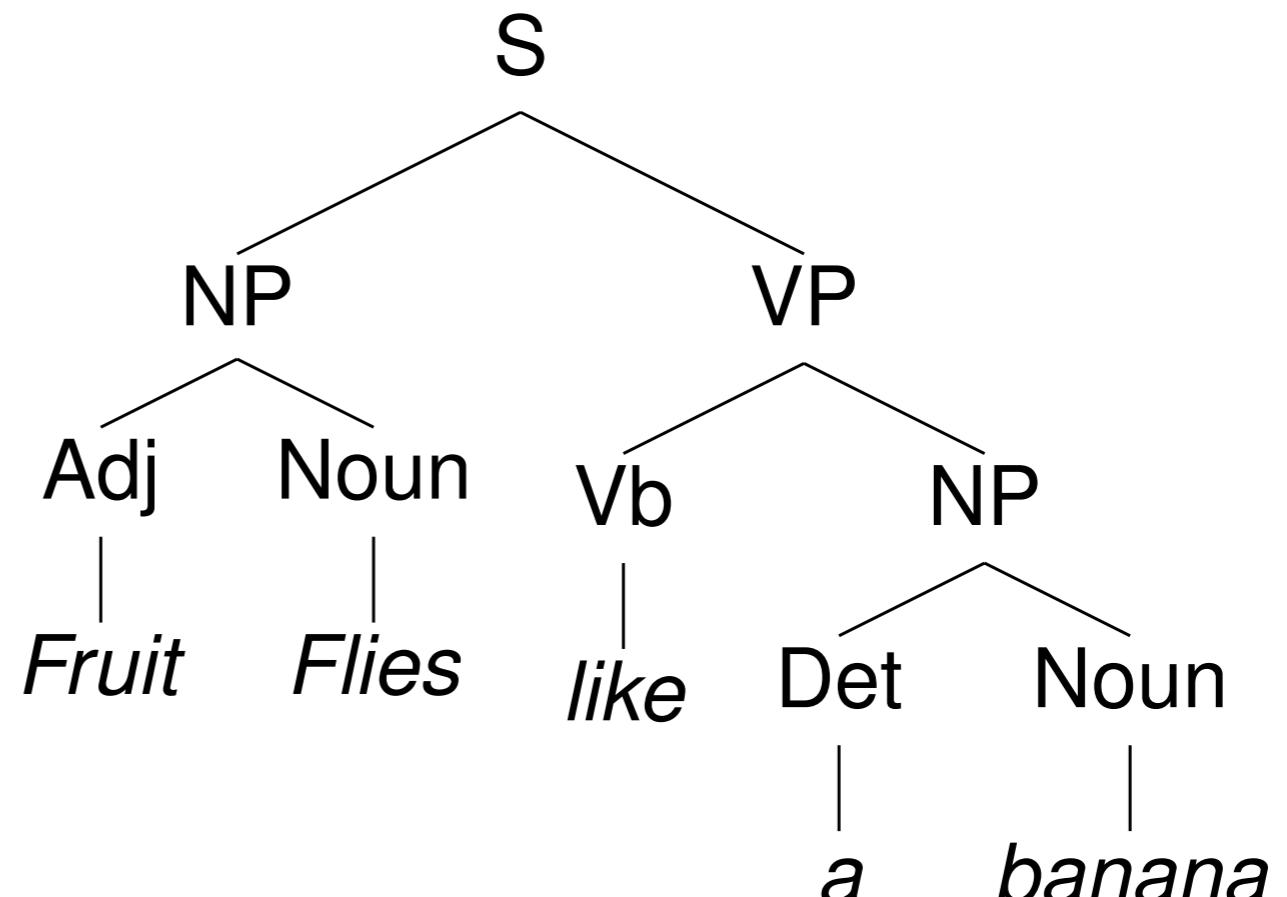
$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Example



Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-

$Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

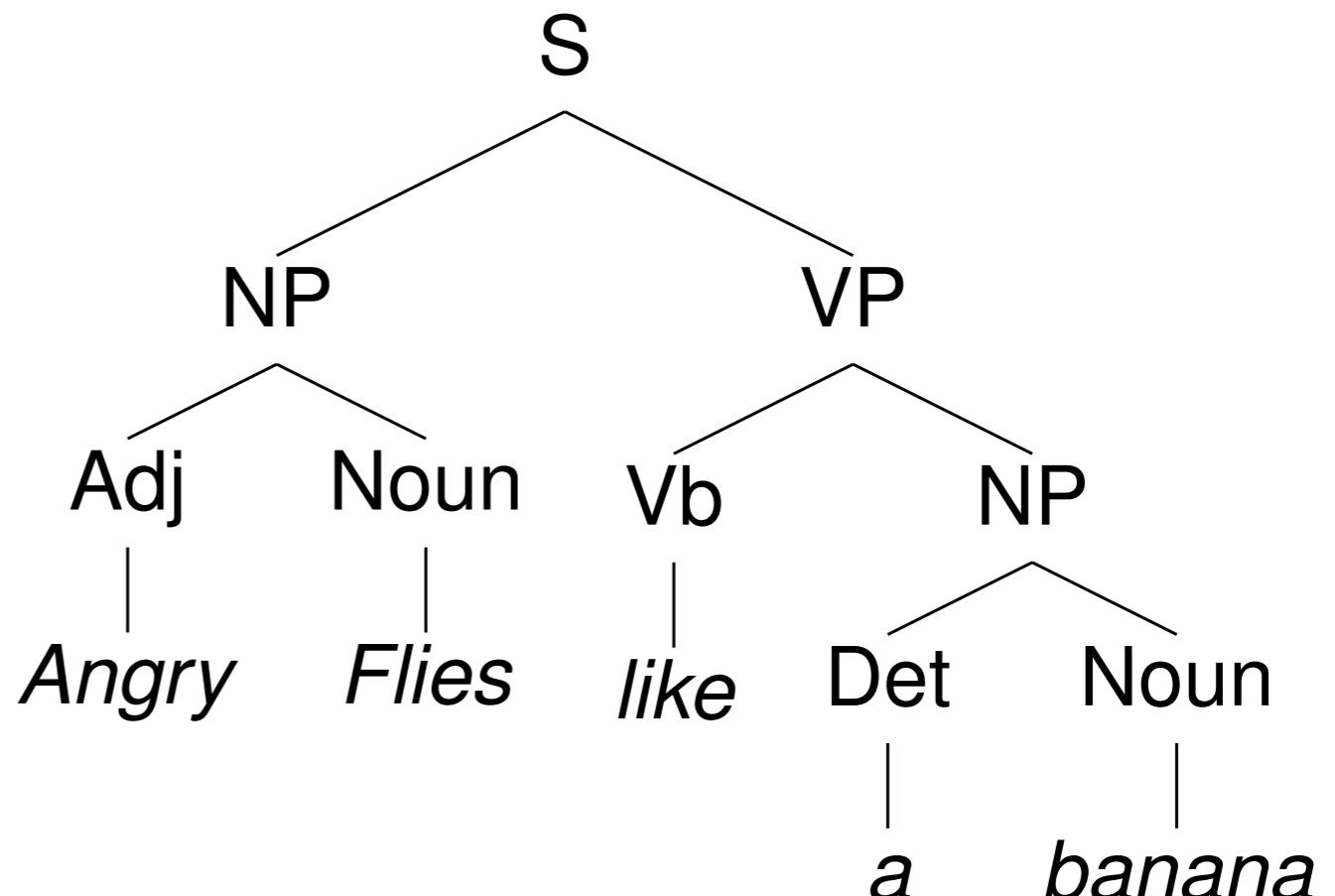
$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Example



Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-

$Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

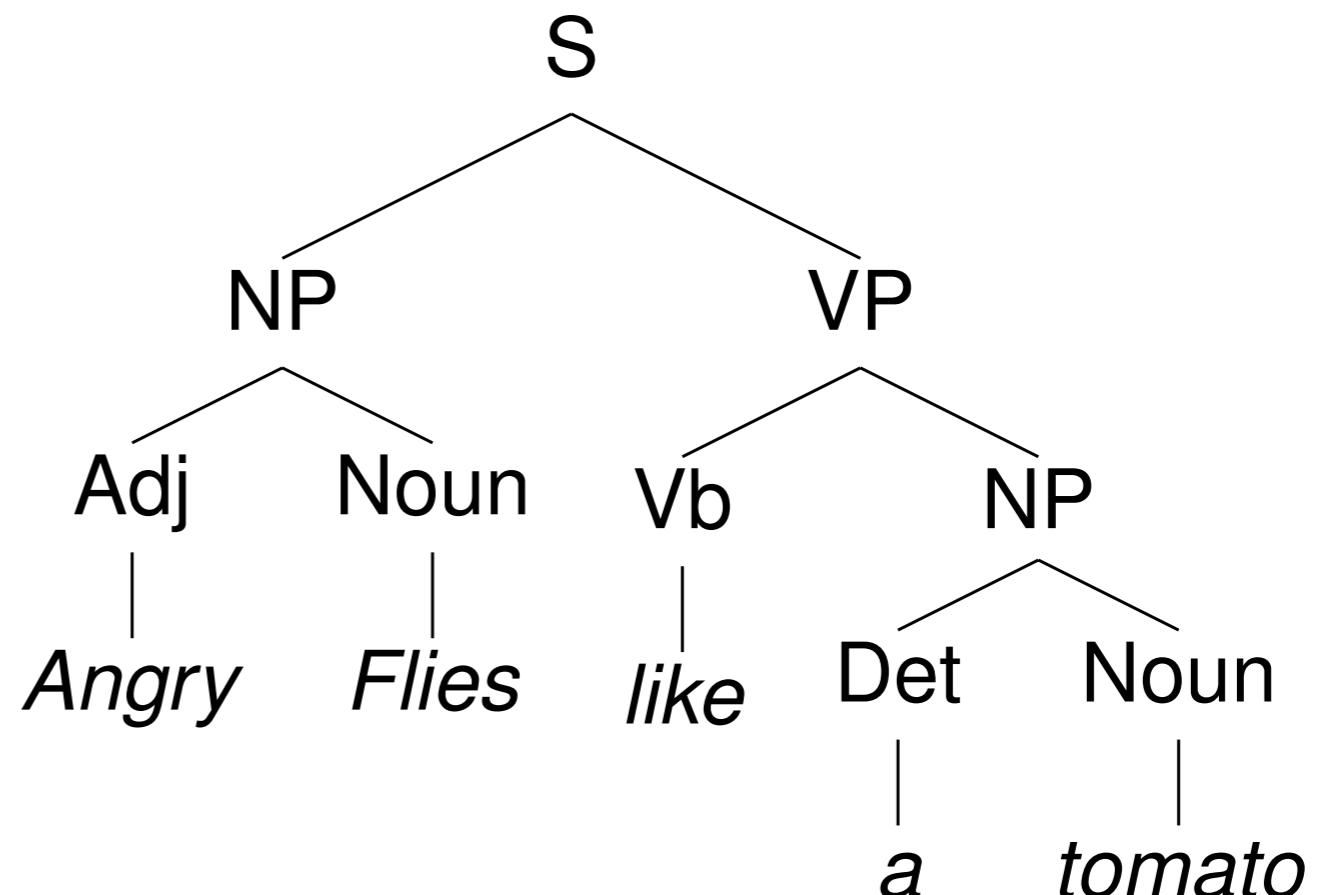
$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Example



Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-

$Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

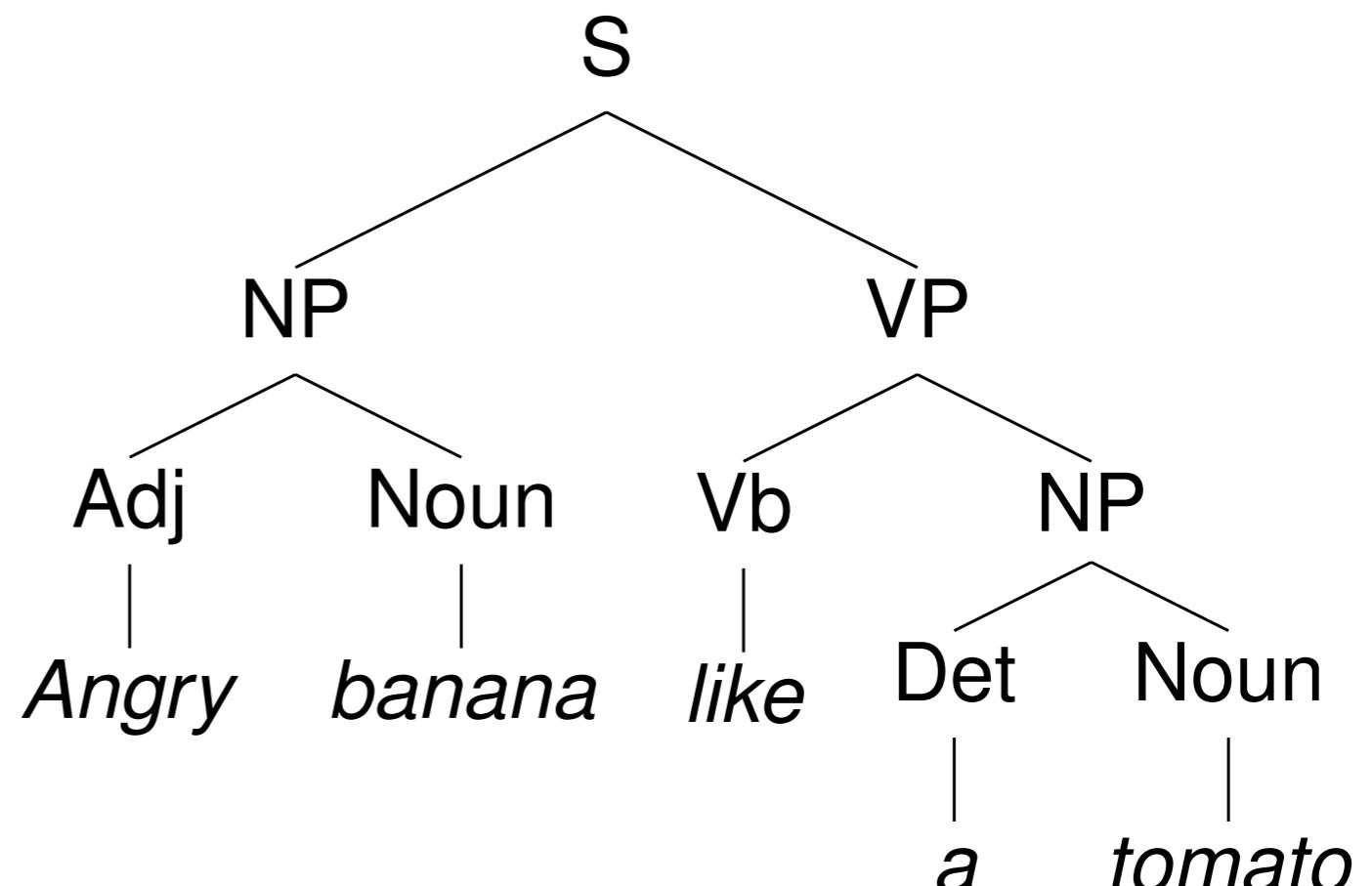
$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Example



Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-
 $Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

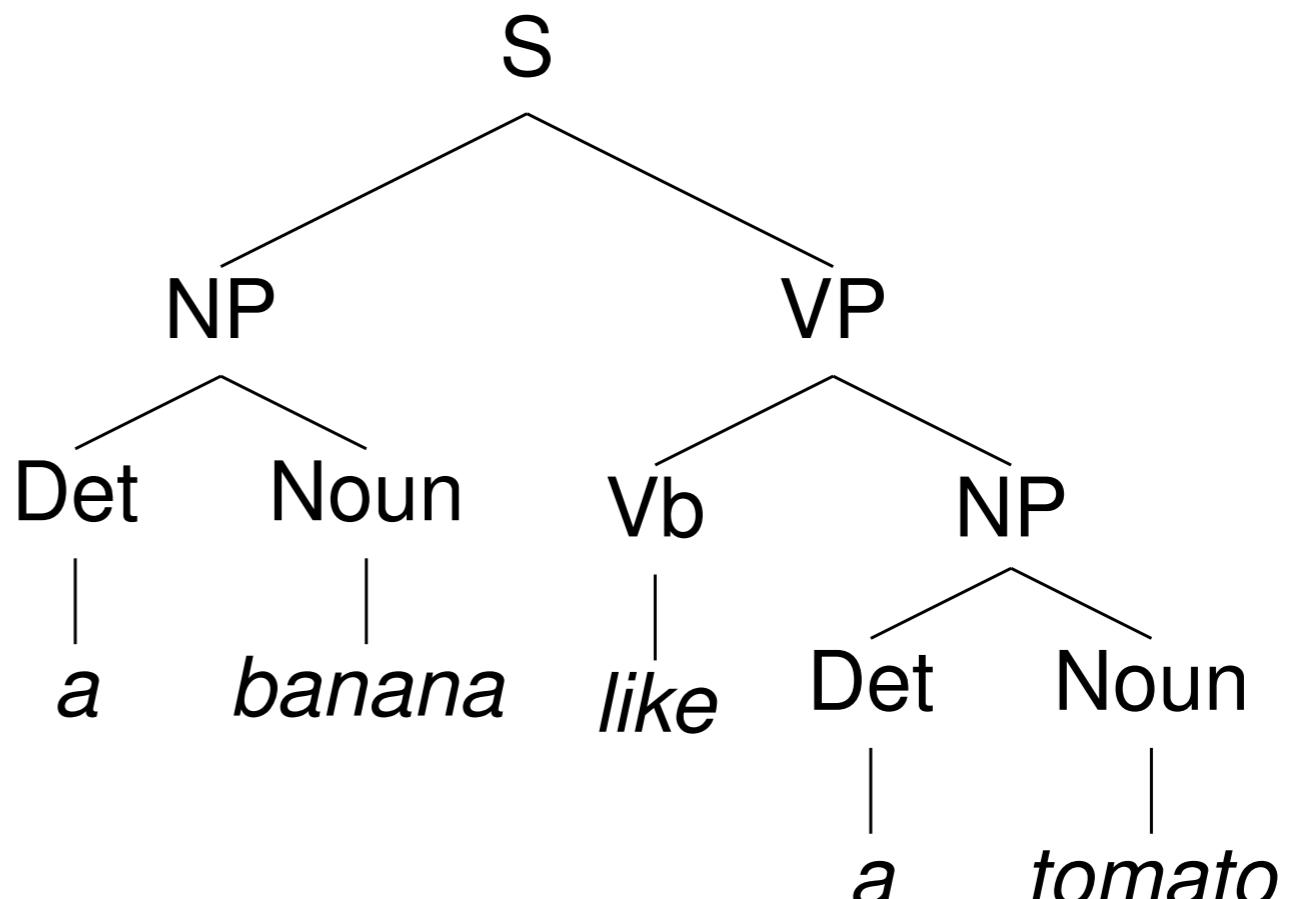
$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Example



Context Free Grammars

a simple grammar

$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$NP \rightarrow Det\ Noun$

$VP \rightarrow Vb\ NP$

-

$Adj \rightarrow fruit$

$Noun \rightarrow flies$

$Vb \rightarrow like$

$Det \rightarrow a$

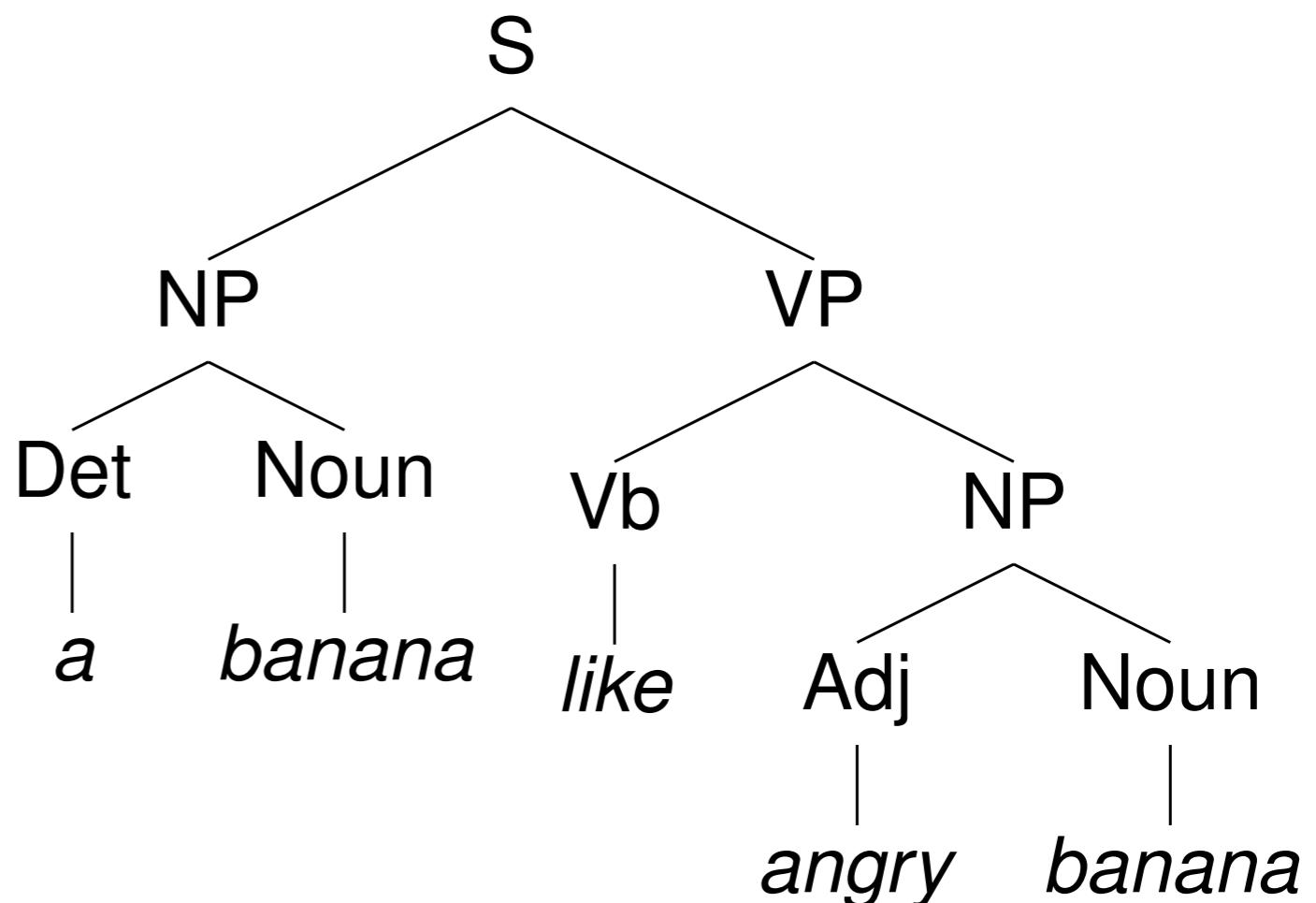
$Noun \rightarrow banana$

$Noun \rightarrow tomato$

$Adj \rightarrow angry$

...

Example



The parsing problem

Given a string, recover the derivation.

Parsing with (P)CFGs

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ Natural Language is NOT generated by a CFG.
 - ▶ We can find $a^n b^n c^n$ structures, and many other arguments.

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ Natural Language is NOT generated by a CFG.
 - ▶ We can find $a^n b^n c^n$ structures, and many other arguments.

Solution

- ▶ We assume really hard that it is.

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ We don't have the grammar.

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ We don't have the grammar.

Solution

- ▶ We'll ask a genius linguist to write it!

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ How do we find the chain of derivations?

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ How do we find the chain of derivations?

Solution

- ▶ With dynamic programming! (soon)

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ Real grammar: hundreds of possible derivations per sentence.

Parsing with CFGs

Let's assume...

- ▶ Let's assume natural language is generated by a CFG.
- ▶ ... and let's assume we have the grammar.
- ▶ Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- ▶ Real grammar: hundreds of possible derivations per sentence.

Solution

- ▶ No problem! We'll choose the best one. (sooner)

Obtaining a Grammar

Let a genius linguist write it

- ▶ Hard. Many rules, many complex interactions.
- ▶ Genius linguists don't grow on trees !

Obtaining a Grammar

Let a genius linguist write it

- ▶ Hard. Many rules, many complex interactions.
- ▶ Genius linguists don't grow on trees !

An easier way: ask a linguist to grow trees

- ▶ Ask a linguist to annotate sentences with tree structure.
- ▶ (This need not be a genius—Smart is enough.)
- ▶ Then extract the rules from the annotated trees.

Obtaining a Grammar

Let a genius linguist write it

- ▶ Hard. Many rules, many complex interactions.
- ▶ Genius linguists don't grow on trees !

An easier way: ask a linguist to grow trees

- ▶ Ask a linguist to annotate sentences with tree structure.
- ▶ (This need not be a genius—Smart is enough.)
- ▶ Then extract the rules from the annotated trees.

Treebanks

- ▶ **English Treebank:** 40k sentences, manually annotated with tree structure.
- ▶ **Other languages:** often about 5k sentences.

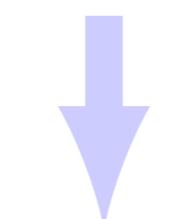
Treebank Sentence Example

```
( (S
  (NP-SBJ
    (NP (NNP Pierre) (NNP Vinken) )
    (, ,)
    (ADJP
      (NP (CD 61) (NNS years) )
      (JJ old) )
    (, ,) )
  (VP (MD will)
  (VP (VB join)
    (NP (DT the) (NN board) )
    (PP-CLR (IN as)
      (NP (DT a) (JJ nonexecutive) (NN director
        (NP-TMP (NNP Nov.) (CD 29) ) )
    (..) ) )
```

Supervised Learning from a Treebank



((fruit/ADJ flies/NN) (like/VB
(time/NN (flies/VB (like/IN
.....
.....



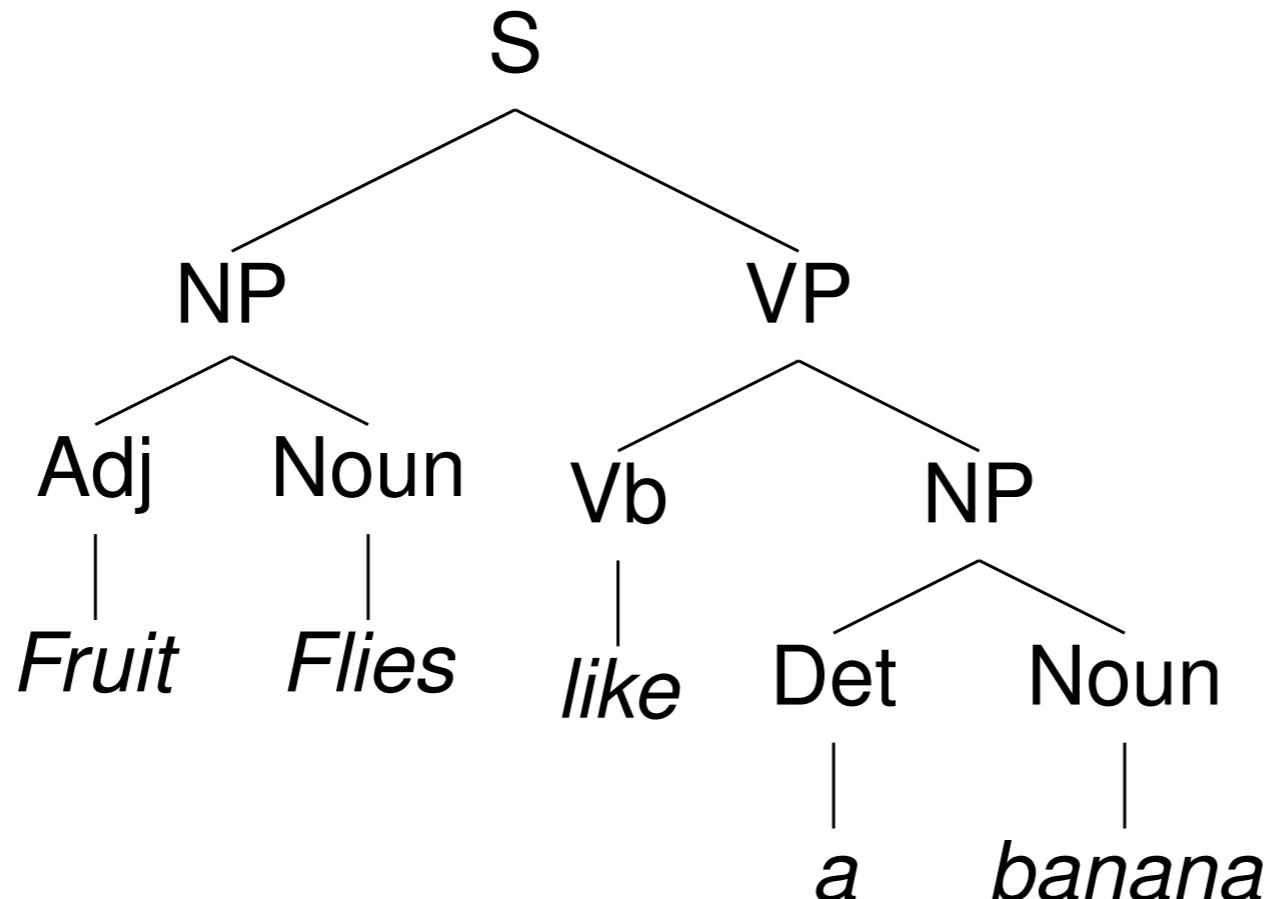
Extracting CFG from Trees

- ▶ The leafs of the trees define Σ
- ▶ The internal nodes of the trees define N
- ▶ Add a special S symbol on top of all trees
- ▶ Each node and its children is a rule in R

Extracting CFG from Trees

- ▶ The leafs of the trees define Σ
- ▶ The internal nodes of the trees define N
- ▶ Add a special S symbol on top of all trees
- ▶ Each node and its children is a rule in R

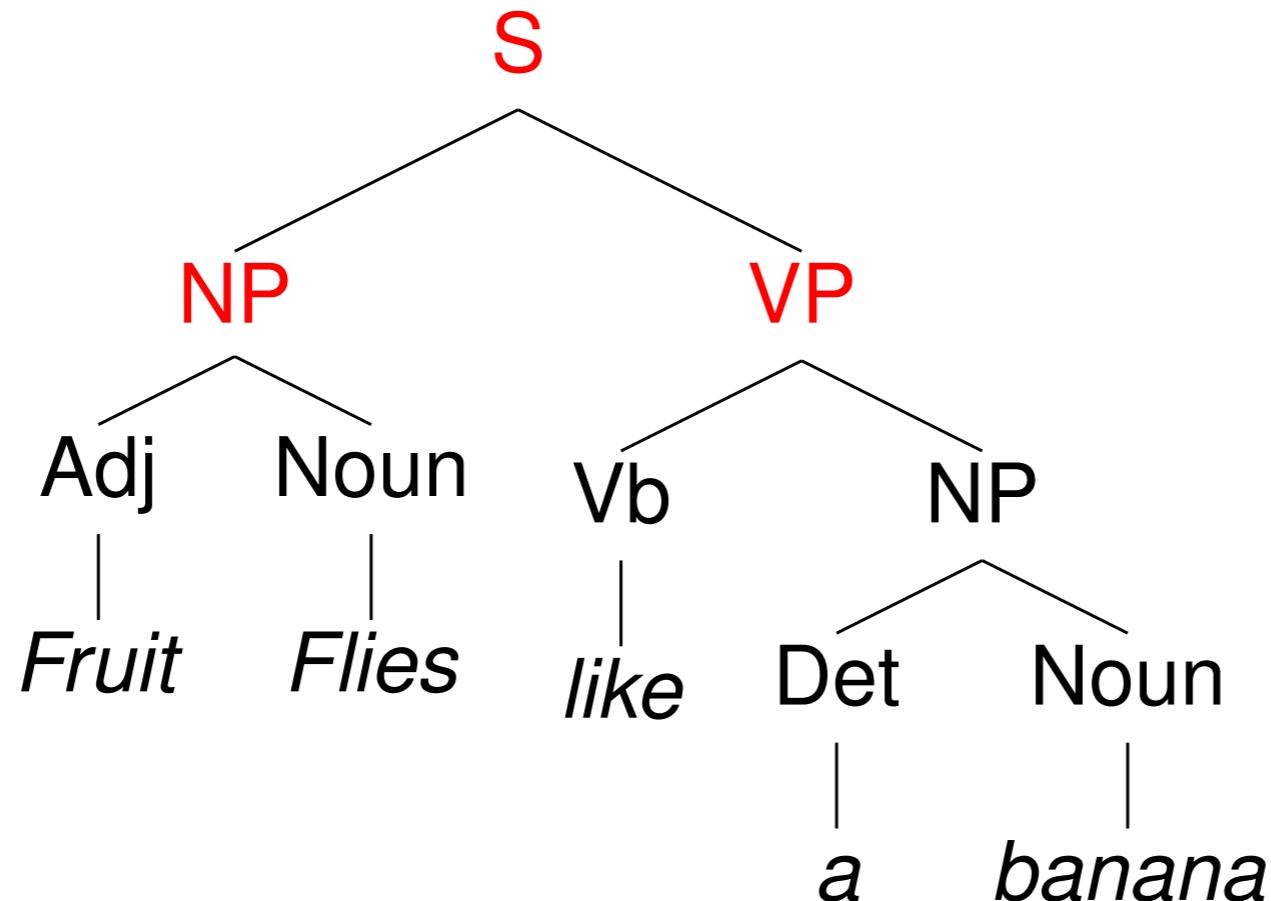
Extracting Rules



Extracting CFG from Trees

- ▶ The leafs of the trees define Σ
- ▶ The internal nodes of the trees define N
- ▶ Add a special S symbol on top of all trees
- ▶ Each node and its children is a rule in R

Extracting Rules

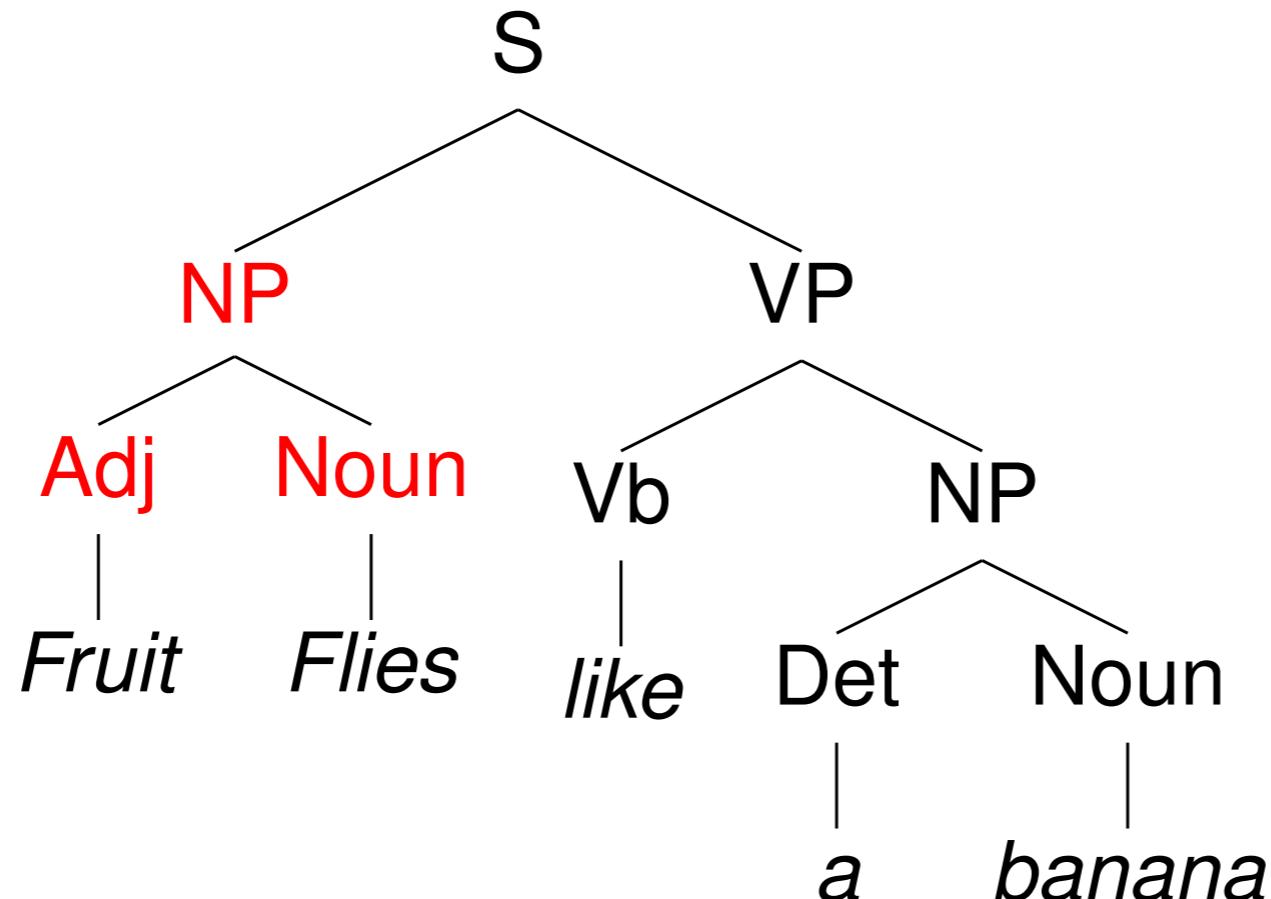


S → **NP VP**

Extracting CFG from Trees

- ▶ The leafs of the trees define Σ
- ▶ The internal nodes of the trees define N
- ▶ Add a special S symbol on top of all trees
- ▶ Each node and its children is a rule in R

Extracting Rules



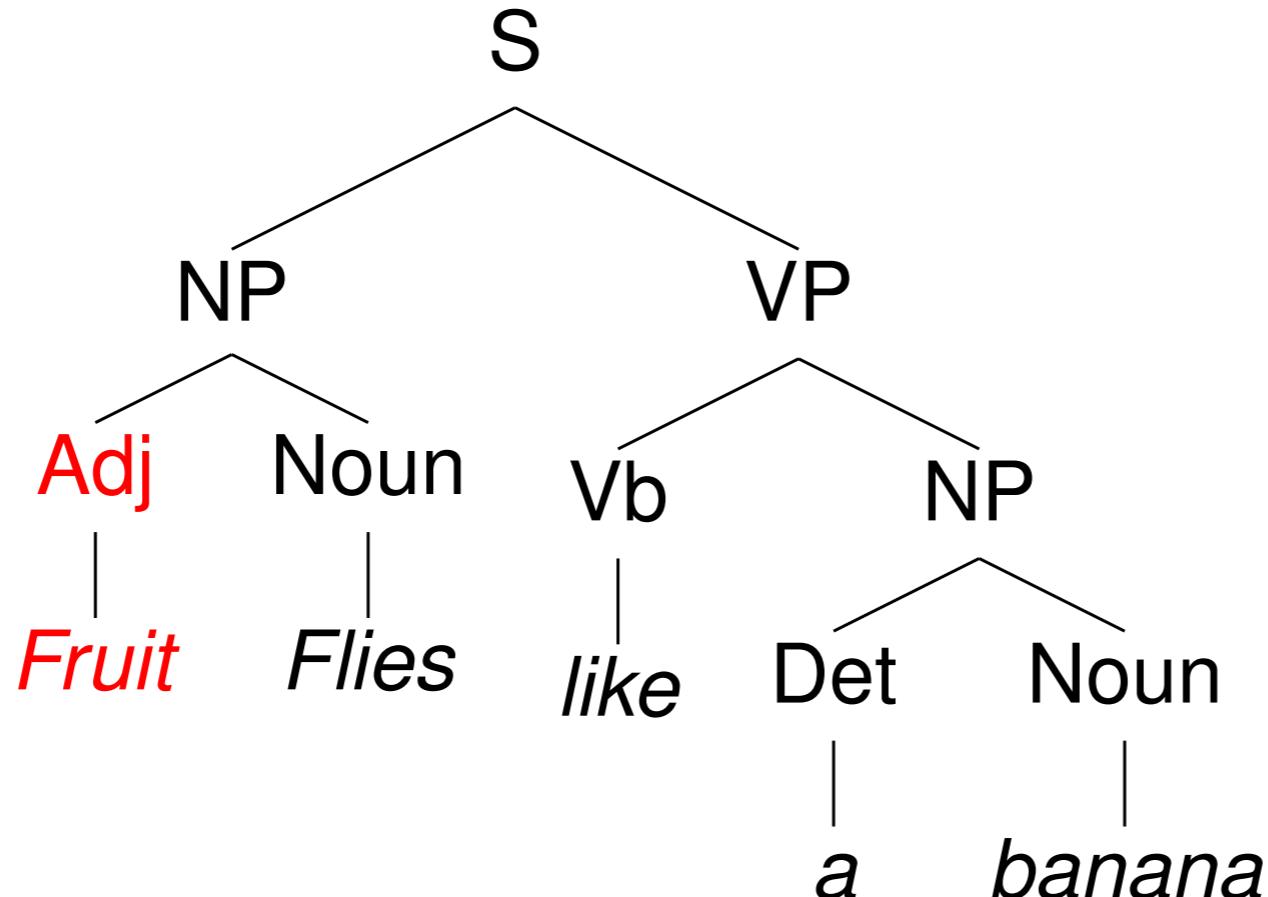
$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

Extracting CFG from Trees

- ▶ The leafs of the trees define Σ
- ▶ The internal nodes of the trees define N
- ▶ Add a special S symbol on top of all trees
- ▶ Each node and its children is a rule in R

Extracting Rules



$S \rightarrow NP\ VP$

$NP \rightarrow Adj\ Noun$

$Adj \rightarrow fruit$

From CFG to PCFG

- ▶ English is NOT generated from CFG \Rightarrow It's generated by a PCFG!

From CFG to PCFG

- ▶ English is NOT generated from CFG \Rightarrow It's generated by a PCFG!
- ▶ PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- ▶ All probabilities for the same LHS sum to 1.

From CFG to PCFG

- ▶ English is NOT generated from CFG \Rightarrow It's generated by a PCFG!
- ▶ PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- ▶ All probabilities for the same LHS sum to 1.
- ▶ Multiplying all the rule probs in a derivation gives the probability of the derivation.
- ▶ We want the tree with maximum probability.

From CFG to PCFG

- ▶ English is NOT generated from CFG \Rightarrow It's generated by a PCFG!
- ▶ PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- ▶ All probabilities for the same LHS sum to 1.
- ▶ Multiplying all the rule probs in a derivation gives the probability of the derivation.
- ▶ We want the tree with maximum probability.

More Formally

$$P(\text{tree}, \text{sent}) = \prod_{I \rightarrow r \in \text{deriv}(\text{tree})} p(I \rightarrow r)$$

From CFG to PCFG

- ▶ English is NOT generated from CFG \Rightarrow It's generated by a PCFG!
- ▶ PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- ▶ All probabilities for the same LHS sum to 1.
- ▶ Multiplying all the rule probs in a derivation gives the probability of the derivation.
- ▶ We want the tree with maximum probability.

More Formally

$$P(\text{tree}, \text{sent}) = \prod_{I \rightarrow r \in \text{deriv}(\text{tree})} p(I \rightarrow r)$$

$$\text{tree} = \arg \max_{\text{tree} \in \text{trees}(\text{sent})} P(\text{tree} | \text{sent}) = \arg \max_{\text{tree} \in \text{trees}(\text{sent})} P(\text{tree}, \text{sent})$$

Just structure prediction, really

$$\text{score}(\text{tree}) = \prod_{l \rightarrow r \in \text{deriv}(\text{tree})} p(l \rightarrow r)$$

$$\text{score}(\text{tree}) = \sum_{l \rightarrow r \in \text{deriv}(\text{tree})} \log p(l \rightarrow r)$$

$$\text{score}(\text{tree}) = \sum_{l \rightarrow r \in \text{deriv}(\text{tree})} \text{score}(l \rightarrow r)$$

decompose

represent and score parts

PCFG Example

a simple PCFG

1.0 $S \rightarrow NP\ VP$

0.3 $NP \rightarrow Adj\ Noun$

0.7 $NP \rightarrow Det\ Noun$

1.0 $VP \rightarrow Vb\ NP$

-
0.2 $Adj \rightarrow fruit$

0.2 $Noun \rightarrow flies$

1.0 $Vb \rightarrow like$

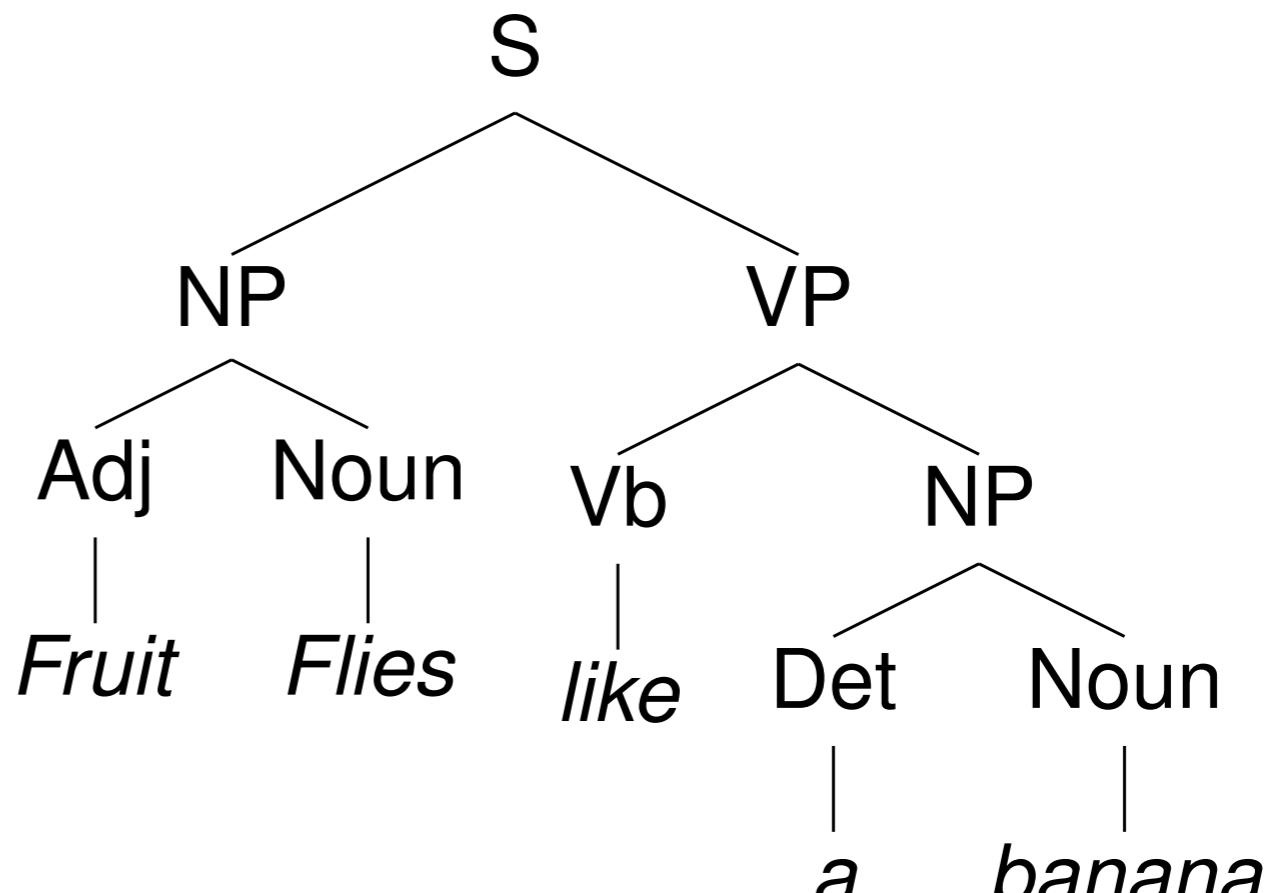
1.0 $Det \rightarrow a$

0.4 $Noun \rightarrow banana$

0.4 $Noun \rightarrow tomato$

0.8 $Adj \rightarrow angry$

Example



$$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$$

PCFG Example

a simple PCFG

1.0 $S \rightarrow NP\ VP$

0.3 $NP \rightarrow Adj\ Noun$

0.7 $NP \rightarrow Det\ Noun$

1.0 $VP \rightarrow Vb\ NP$

-
0.2 $Adj \rightarrow fruit$

0.2 $Noun \rightarrow flies$

1.0 $Vb \rightarrow like$

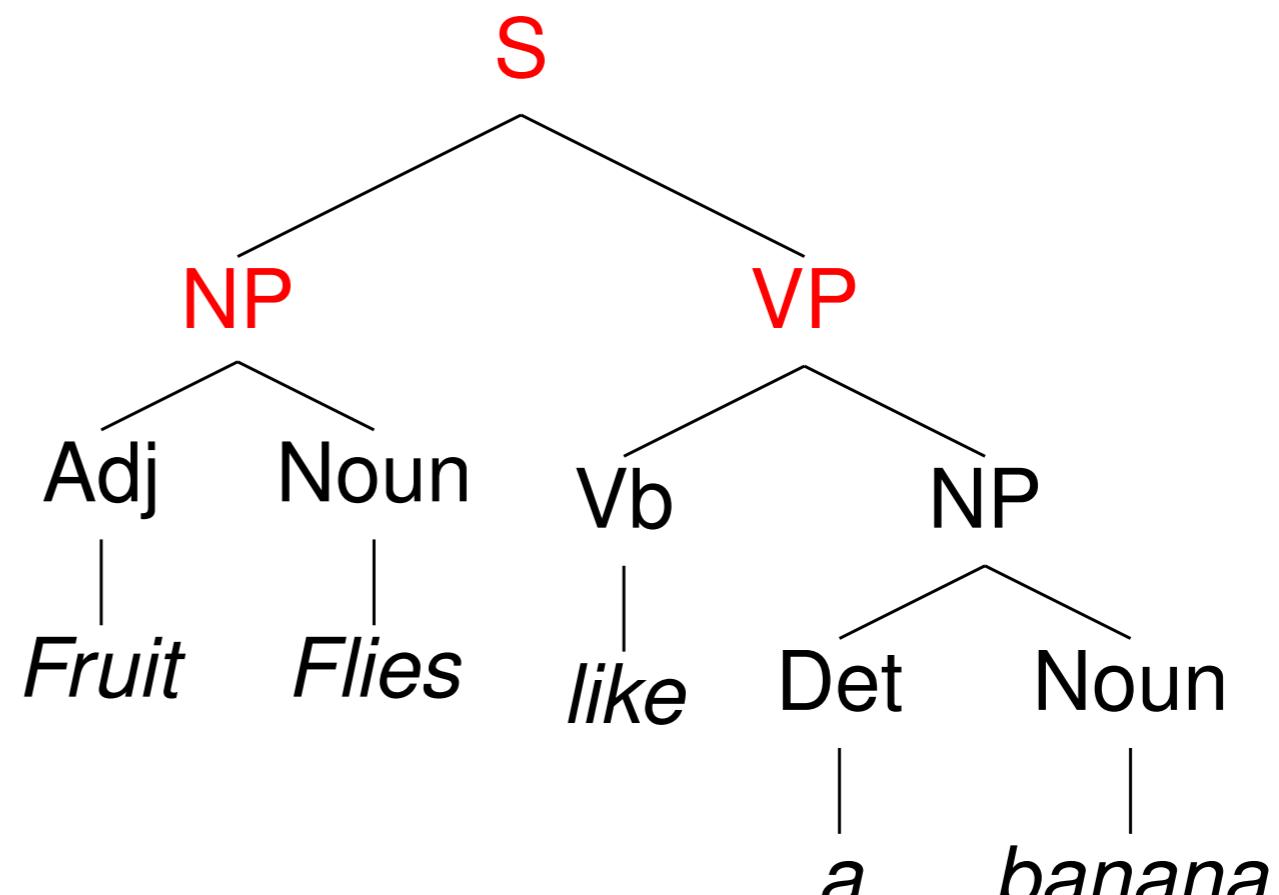
1.0 $Det \rightarrow a$

0.4 $Noun \rightarrow banana$

0.4 $Noun \rightarrow tomato$

0.8 $Adj \rightarrow angry$

Example



$$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$$

PCFG Example

a simple PCFG

1.0 $S \rightarrow NP\ VP$

0.3 $NP \rightarrow Adj\ Noun$

0.7 $NP \rightarrow Det\ Noun$

1.0 $VP \rightarrow Vb\ NP$

-
0.2 $Adj \rightarrow fruit$

0.2 $Noun \rightarrow flies$

1.0 $Vb \rightarrow like$

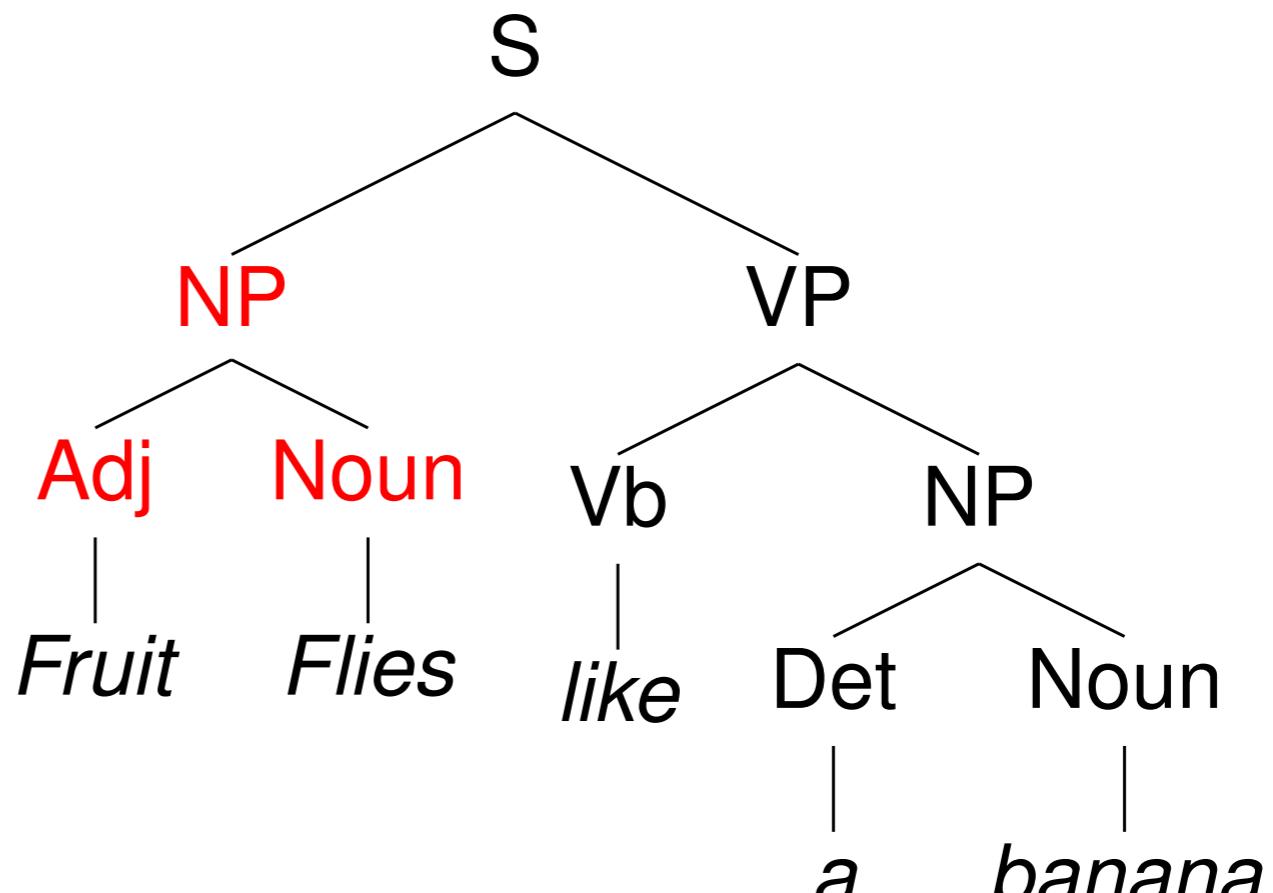
1.0 $Det \rightarrow a$

0.4 $Noun \rightarrow banana$

0.4 $Noun \rightarrow tomato$

0.8 $Adj \rightarrow angry$

Example



$$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$$

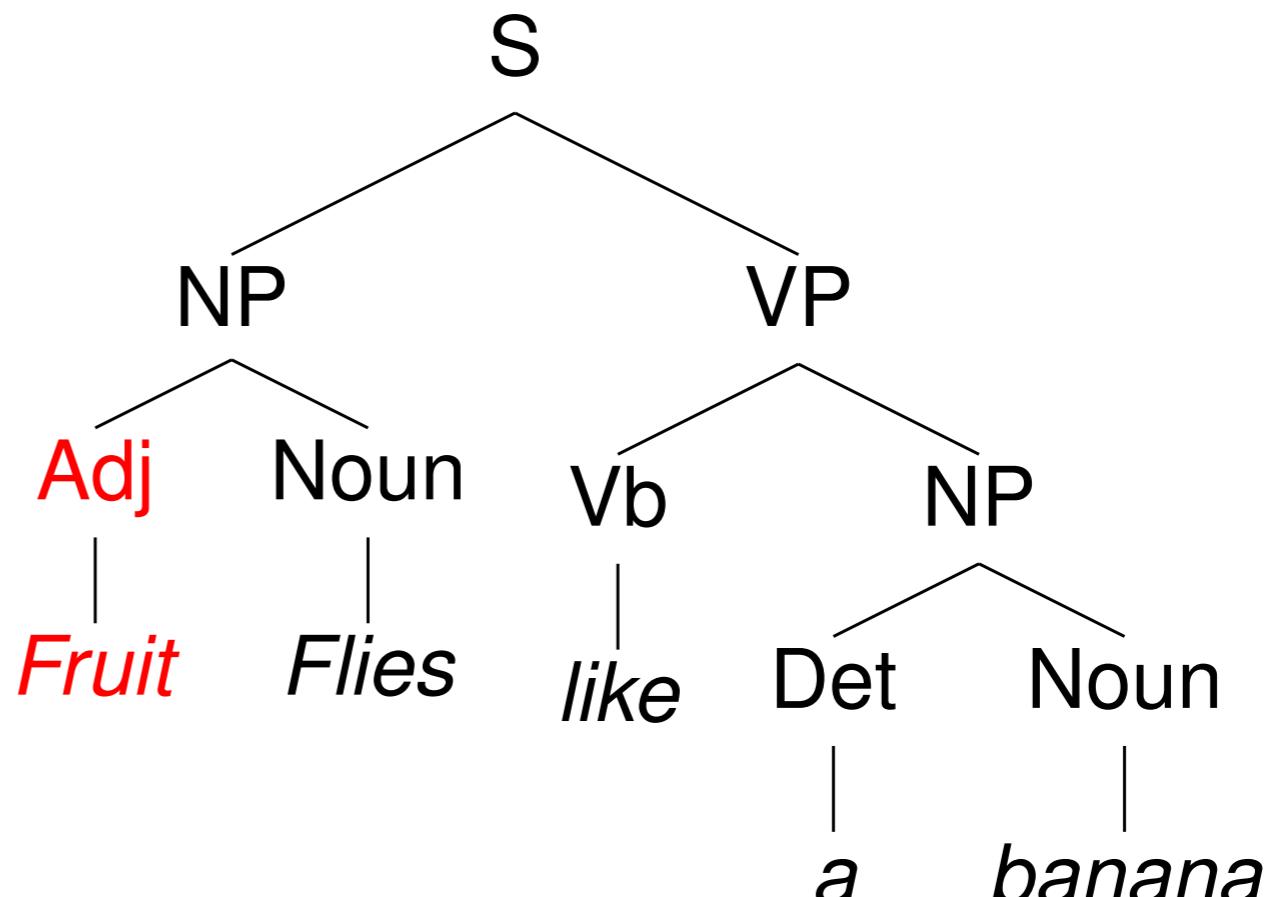
PCFG Example

a simple PCFG

1.0 $S \rightarrow NP\ VP$
0.3 $NP \rightarrow Adj\ Noun$
0.7 $NP \rightarrow Det\ Noun$
1.0 $VP \rightarrow Vb\ NP$

-
0.2 $Adj \rightarrow fruit$
0.2 $Noun \rightarrow flies$
1.0 $Vb \rightarrow like$
1.0 $Det \rightarrow a$
0.4 $Noun \rightarrow banana$
0.4 $Noun \rightarrow tomato$
0.8 $Adj \rightarrow angry$

Example



$$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$$

Parsing with PCFG

- ▶ Parsing with a PCFG is finding the most probable derivation for a given sentence.
- ▶ This can be done quite efficiently with dynamic programming (the CKY algorithm)

Parsing with PCFG

- ▶ Parsing with a PCFG is finding the most probable derivation for a given sentence.
- ▶ This can be done quite efficiently with dynamic programming (the CKY algorithm)

Obtaining the probabilities

- ▶ We estimate them from the Treebank.
- ▶ $P(LHS \rightarrow RHS) = \frac{\text{count}(LHS \rightarrow RHS)}{\text{count}(LHS \rightarrow \diamond)}$
- ▶ We can also add smoothing and backoff, as before.
- ▶ Dealing with unknown words - like in the HMM

The CKY algorithm

The Problem

Input

- ▶ Sentence (a list of words)
 - ▶ n – sentence length
- ▶ CFG Grammar (with weights on rules)
 - ▶ g – number of non-terminal symbols

Output

- ▶ A parse tree / the best parse tree

But . . .

- ▶ Exponentially many possible parse trees!

Solution

- ▶ Dynamic Programming!

Cocke Kasami Younger

Cocke Kasami Younger
196?

Cocke Kasami Younger
196? 1965

Cocke	Kasami	Younger
196?	1965	1967

3 Interesting Problems

- ▶ Recognition
- ▶ Parsing
- ▶ Disambiguation

3 Interesting Problems

- ▶ Recognition
 - ▶ Can this string be generated by the grammar?
- ▶ Parsing
- ▶ Disambiguation

3 Interesting Problems

- ▶ Recognition
 - ▶ Can this string be generated by the grammar?
- ▶ Parsing
 - ▶ Show me a possible derivation...
- ▶ Disambiguation

3 Interesting Problems

- ▶ Recognition
 - ▶ Can this string be generated by the grammar?
- ▶ Parsing
 - ▶ Show me a possible derivation...
- ▶ Disambiguation
 - ▶ Show me THE BEST derivation

3 Interesting Problems

- ▶ Recognition
 - ▶ Can this string be generated by the grammar?
- ▶ Parsing
 - ▶ Show me a possible derivation...
- ▶ Disambiguation
 - ▶ Show me THE BEST derivation

CKY can do all of these in polynomial time

3 Interesting Problems

- ▶ Recognition
 - ▶ Can this string be generated by the grammar?
- ▶ Parsing
 - ▶ Show me a possible derivation...
- ▶ Disambiguation
 - ▶ Show me THE BEST derivation

CKY can do all of these in polynomial time

- ▶ For any **CNF** grammar

CNF

Chomsky Normal Form

Definition

A CFG is in CNF form if it only has rules like:

- ▶ $A \rightarrow B C$
- ▶ $A \rightarrow \alpha$

A, B, C are non terminal symbols

α is a terminal symbol (a word...)

- ▶ All terminal symbols are RHS of unary rules
- ▶ All non terminal symbols are RHS of **binary** rules

CKY can be easily extended to handle also unary rules: $A \rightarrow B$

Binarization

Fact

- ▶ Any CFG grammar can be converted to CNF form

Binarization

Fact

- ▶ Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- ▶ We already have $A \rightarrow \alpha$

Binarization

Fact

- ▶ Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- ▶ We already have $A \rightarrow \alpha$
 - ▶ ($A \rightarrow \alpha \beta$ is also easy to handle)

Binarization

Fact

- ▶ Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- ▶ We already have $A \rightarrow \alpha$
 - ▶ ($A \rightarrow \alpha \beta$ is also easy to handle)
- ▶ Unary rules ($A \rightarrow B$) are OK

Binarization

Fact

- ▶ Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- ▶ We already have $A \rightarrow \alpha$
 - ▶ ($A \rightarrow \alpha \beta$ is also easy to handle)
- ▶ Unary rules ($A \rightarrow B$) are OK
- ▶ Only problem: $S \rightarrow NP\ PP\ VP\ PP$

Binarization

Fact

- ▶ Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- ▶ We already have $A \rightarrow \alpha$
 - ▶ ($A \rightarrow \alpha \beta$ is also easy to handle)
- ▶ Unary rules ($A \rightarrow B$) are OK
- ▶ Only problem: $S \rightarrow NP\ PP\ VP\ PP$

Binarization

S	\rightarrow	$NP\ NP_PP-VP-PP$
$NP_PP-VP-PP$	\rightarrow	$PP\ NP-PP_VP-PP$
$NP_PP-VP-PP$	\rightarrow	$VP\ NP-PP-VP_PP$

Finally, CKY

Recognition

- ▶ Main idea:
 - ▶ Build parse tree from bottom up
 - ▶ Combine built trees to form bigger trees using grammar rules
 - ▶ When left with a single tree, verify root is S
- ▶ Exponentially many possible trees...
 - ▶ Search over all of them in polynomial time using DP
 - ▶ Shared structure – smaller trees

Main Idea

If we know:

- ▶ $w_i \dots w_j$ is an NP
- ▶ $w_{j+1} \dots w_k$ is a VP

and grammar has rule:

- ▶ $S \rightarrow NP \ VP$

Then we know:

- ▶ S can derive $w_i \dots w_k$

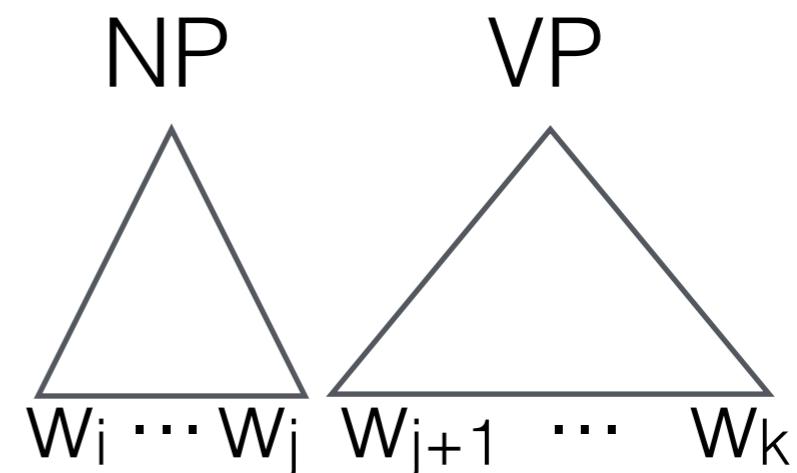
Main Idea

If we know:

- ▶ $w_i \dots w_j$ is an *NP*
- ▶ $w_{j+1} \dots w_k$ is a *VP*

and grammar has rule:

- ▶ $S \rightarrow NP \ VP$



Then we know:

- ▶ S can derive $w_i \dots w_k$

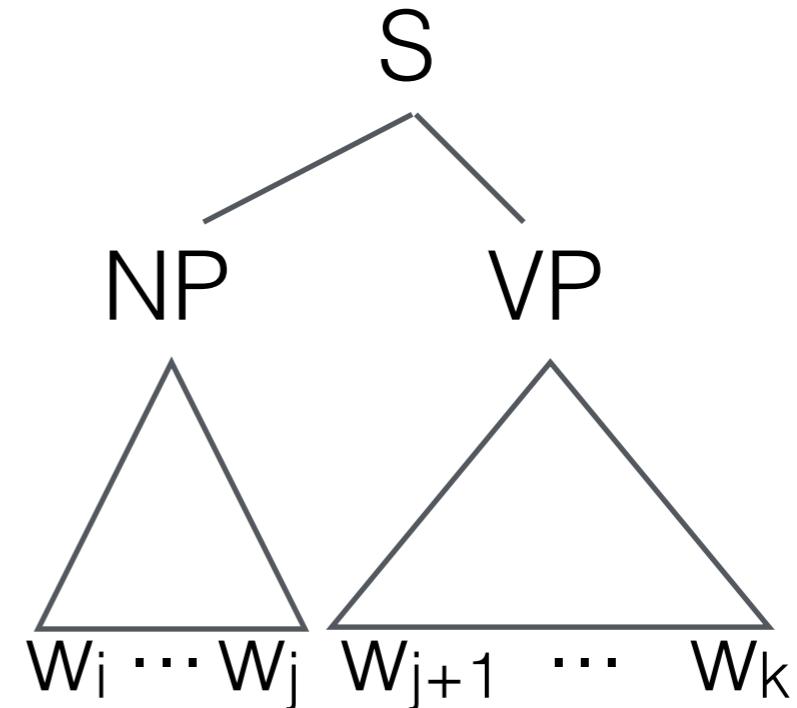
Main Idea

If we know:

- ▶ $w_i \dots w_j$ is an *NP*
- ▶ $w_{j+1} \dots w_k$ is a *VP*

and grammar has rule:

- ▶ $S \rightarrow NP \ VP$



Then we know:

- ▶ S can derive $w_i \dots w_k$

Main Idea

If we know:

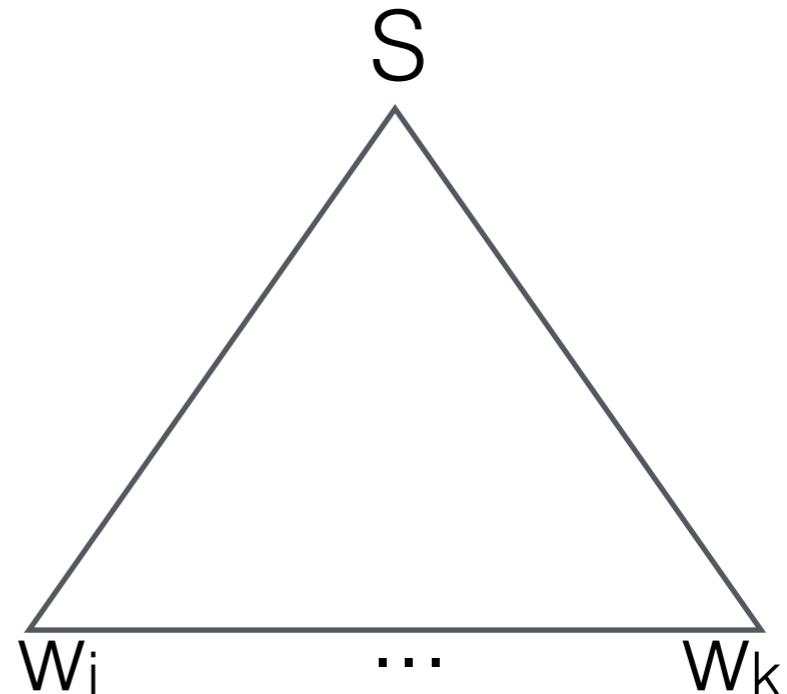
- ▶ $w_i \dots w_j$ is an *NP*
- ▶ $w_{j+1} \dots w_k$ is a *VP*

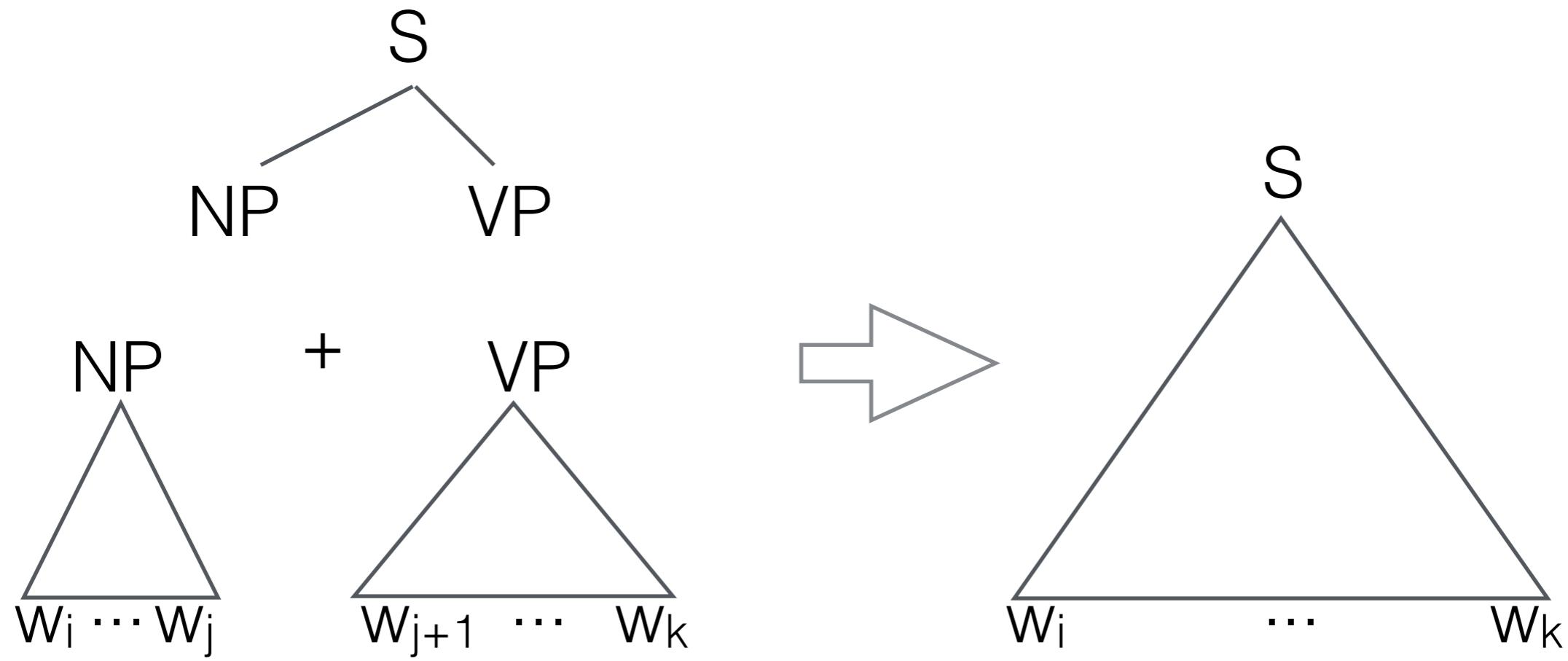
and grammar has rule:

- ▶ $S \rightarrow NP \ VP$

Then we know:

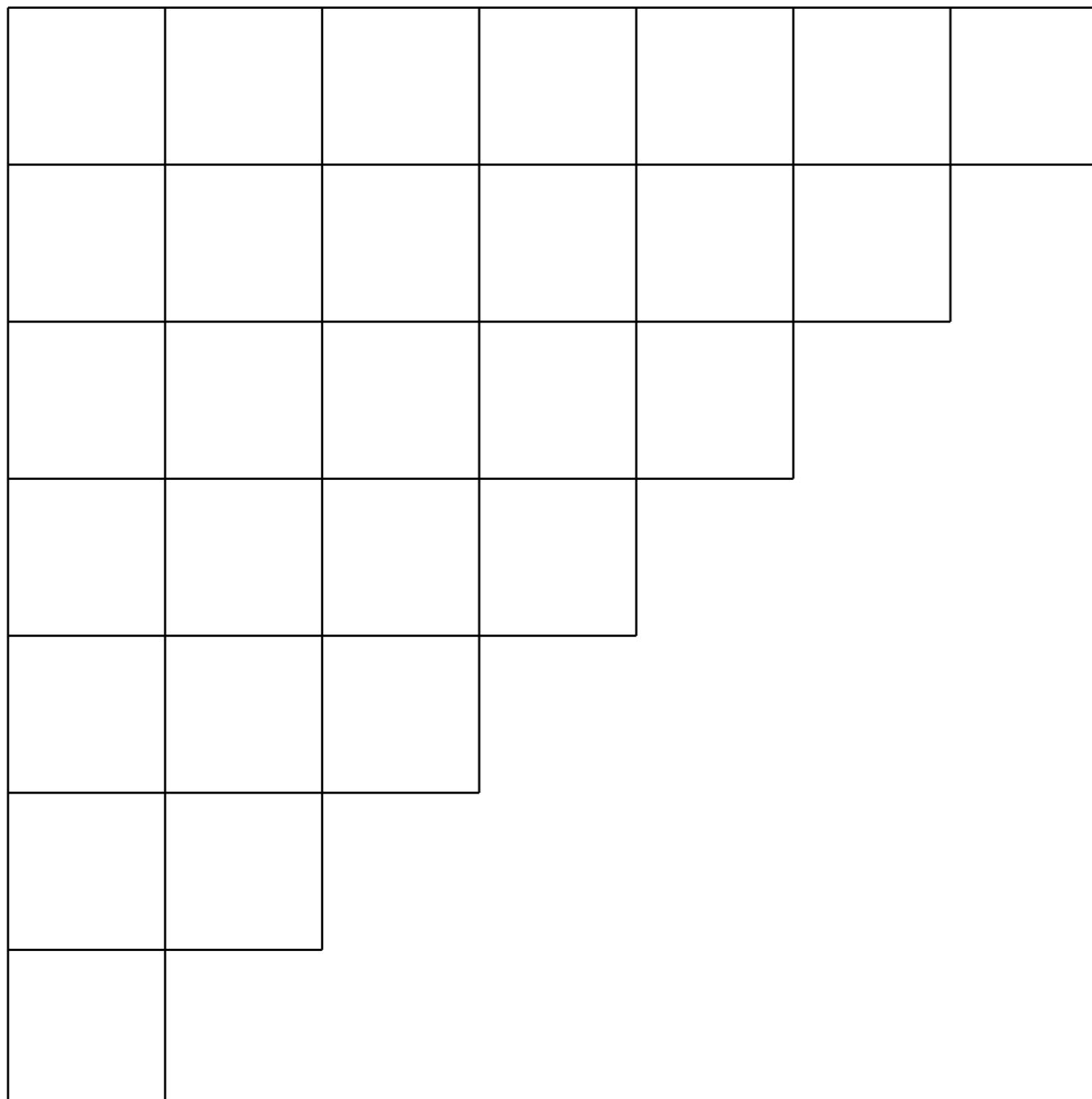
- ▶ S can derive $w_i \dots w_k$





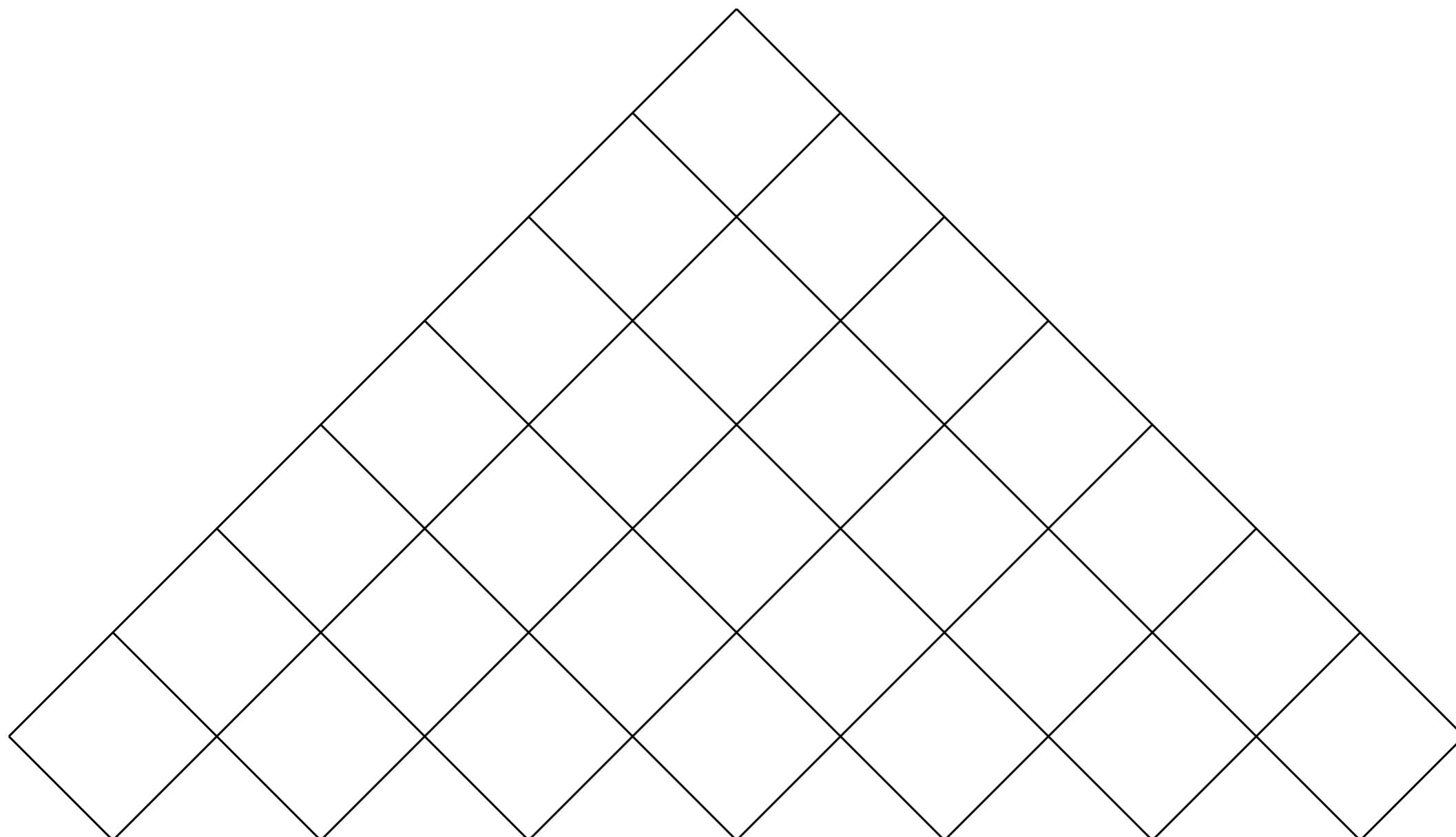
Data Structure

(Half a) two dimensional array ($n \times n$)



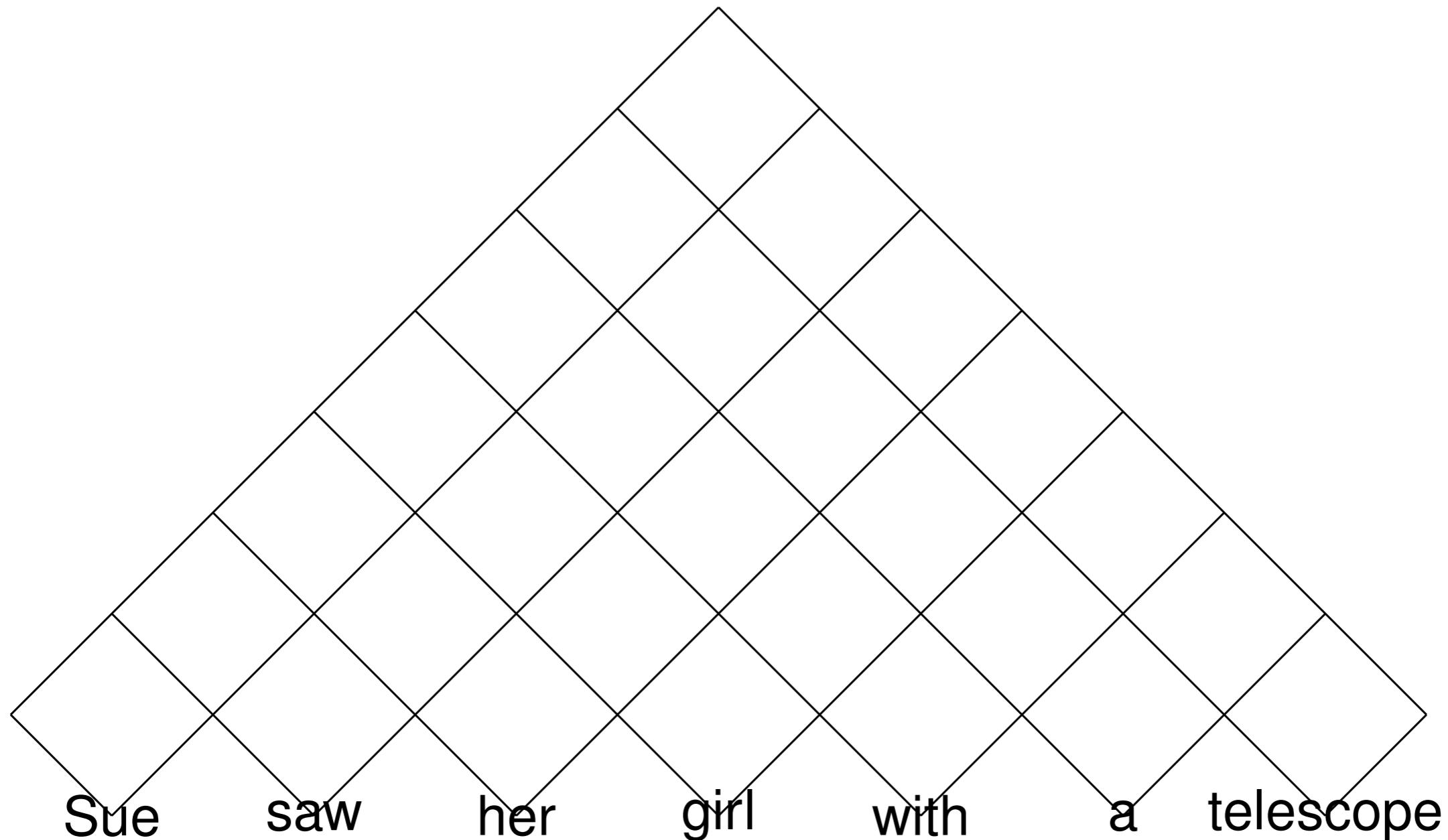
Data Structure

On its side



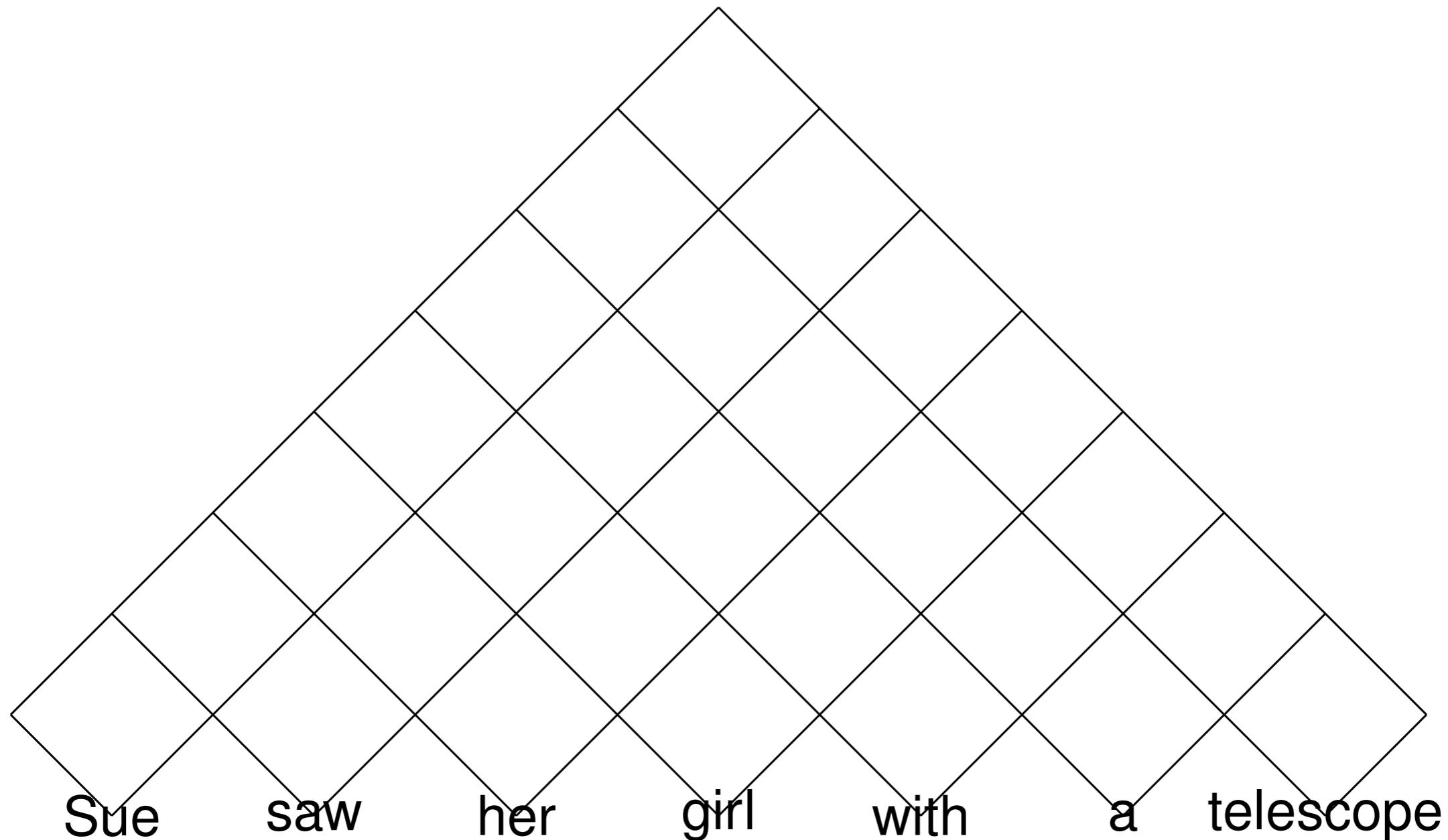
Data Structure

Each cell: all nonterminals than can derive word i to word j



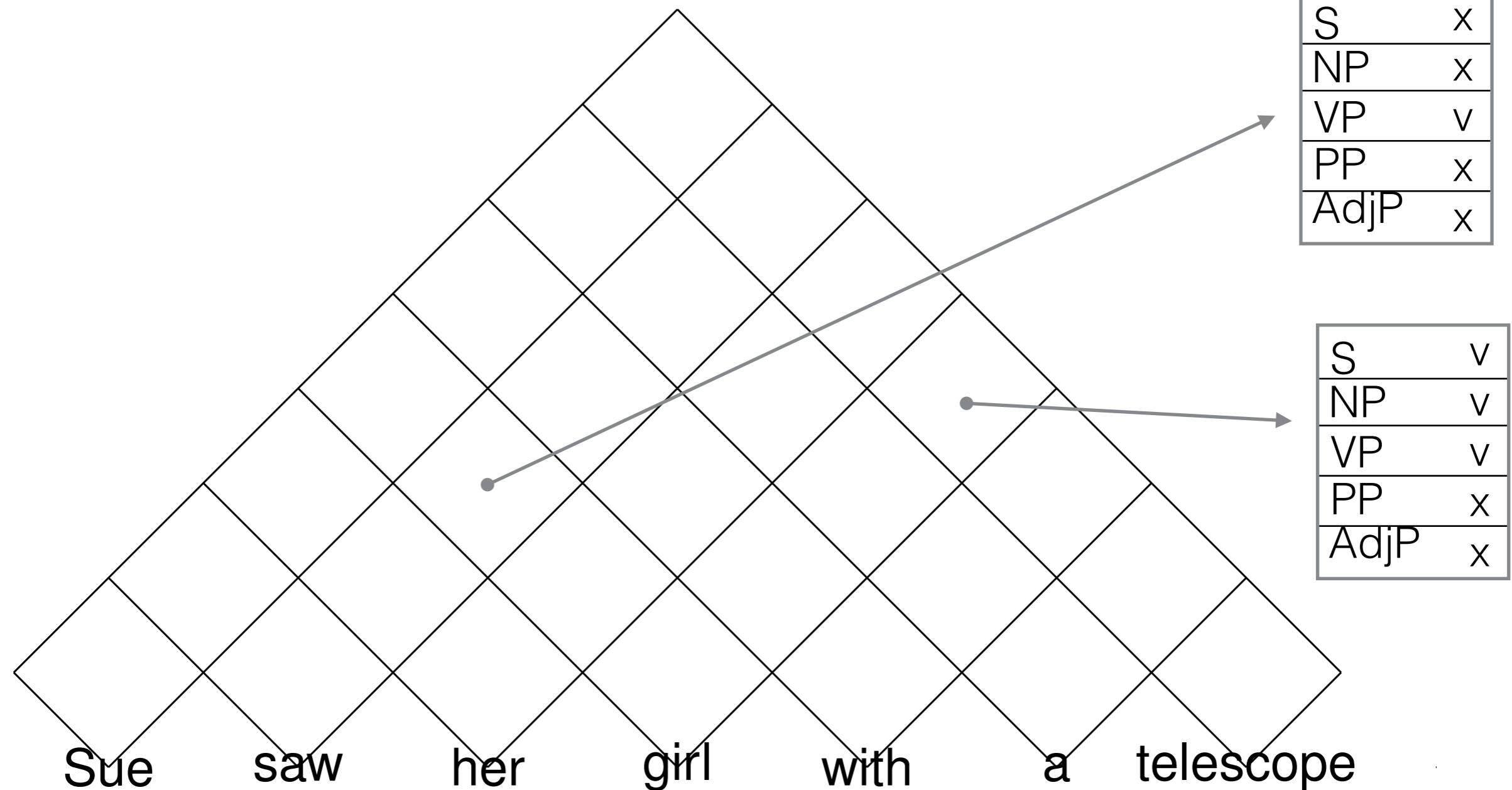
Data Structure

Each cell: all nonterminals than can derive word i to word j
imagine each cell as a g dimensional array

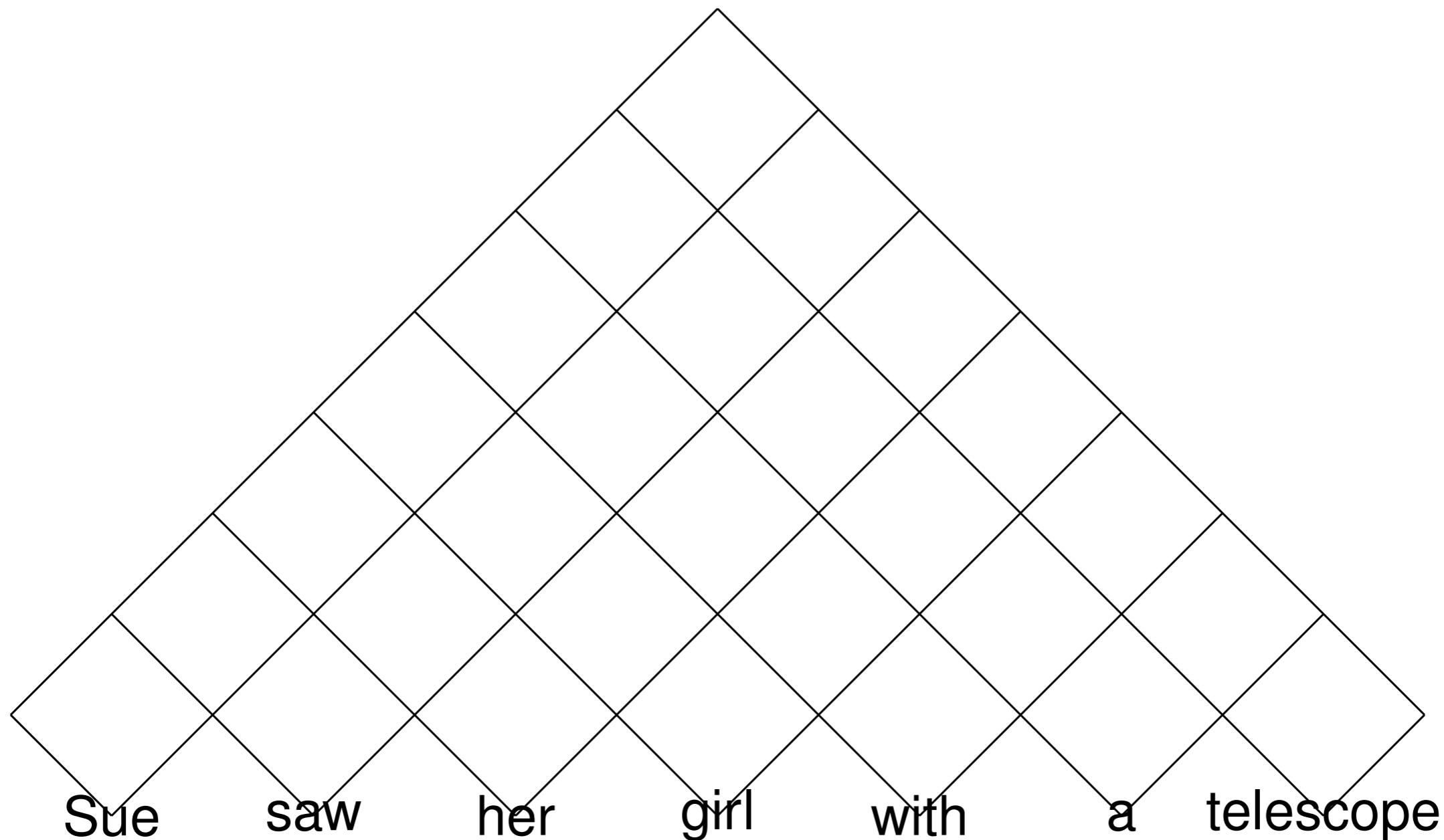


Data Structure

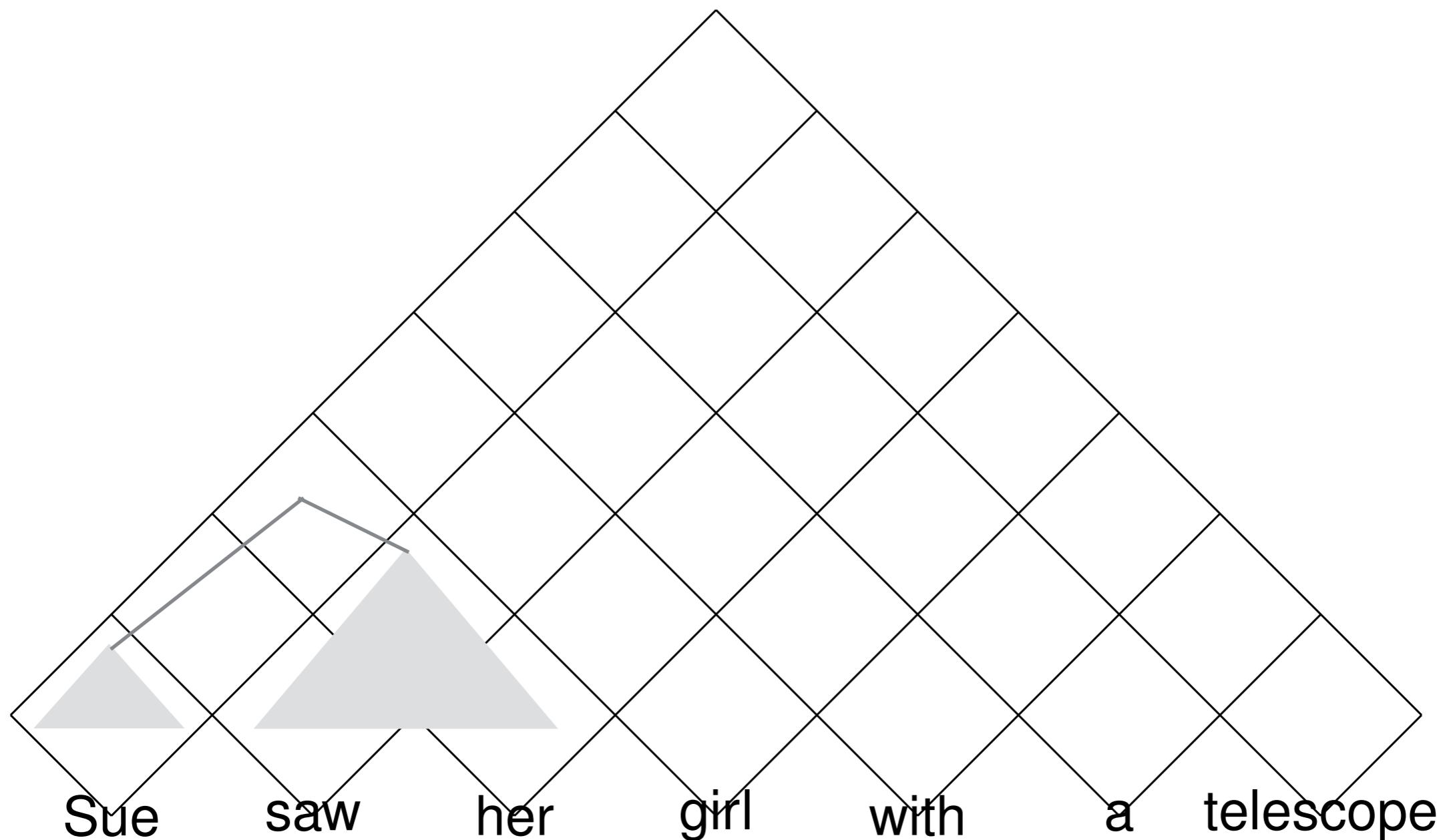
Each cell: all nonterminals that can derive word i to word j
imagine each cell as a g dimensional array



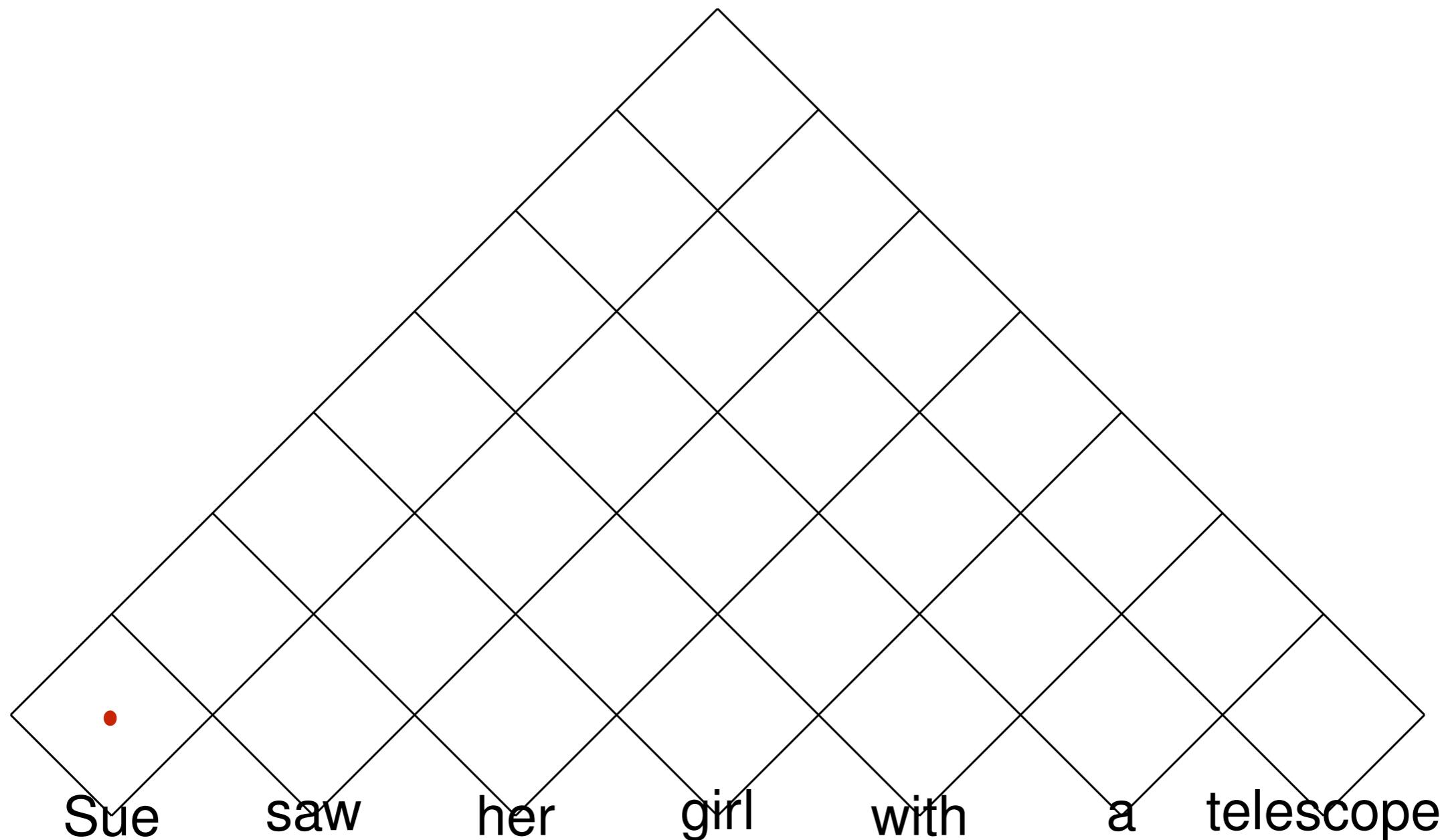
Filling the table



Filling the table

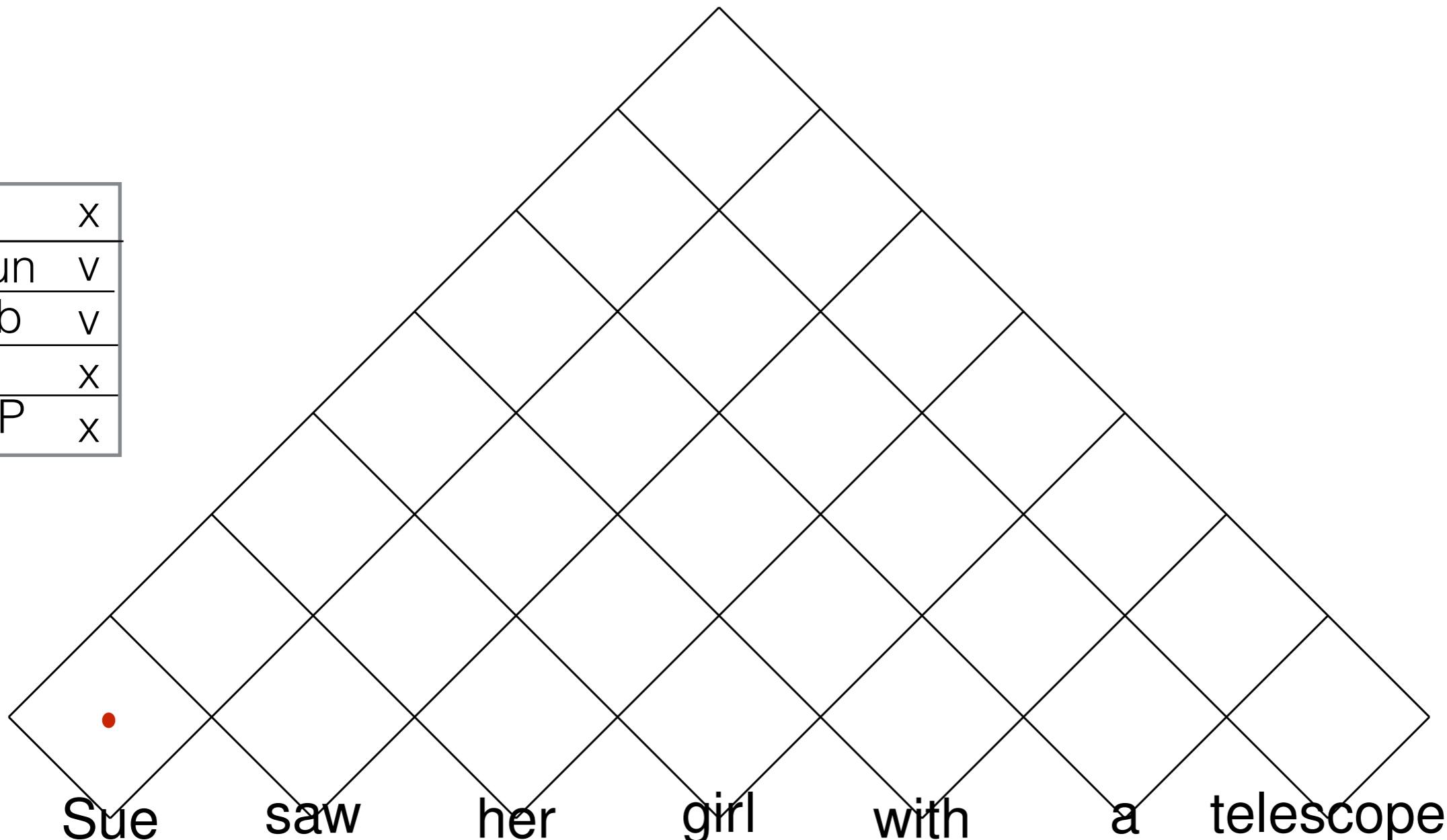


Filling the table

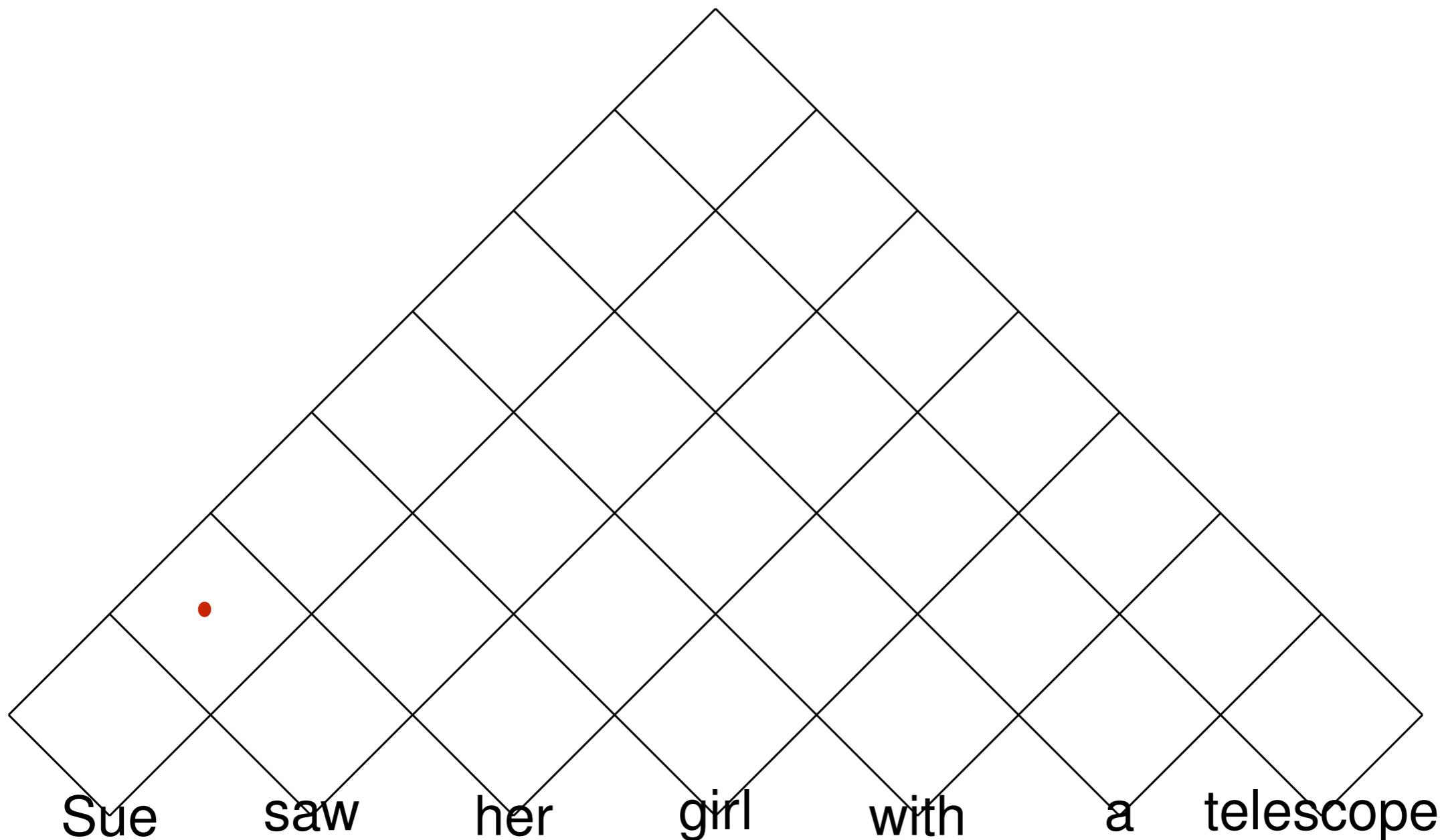


Filling the table

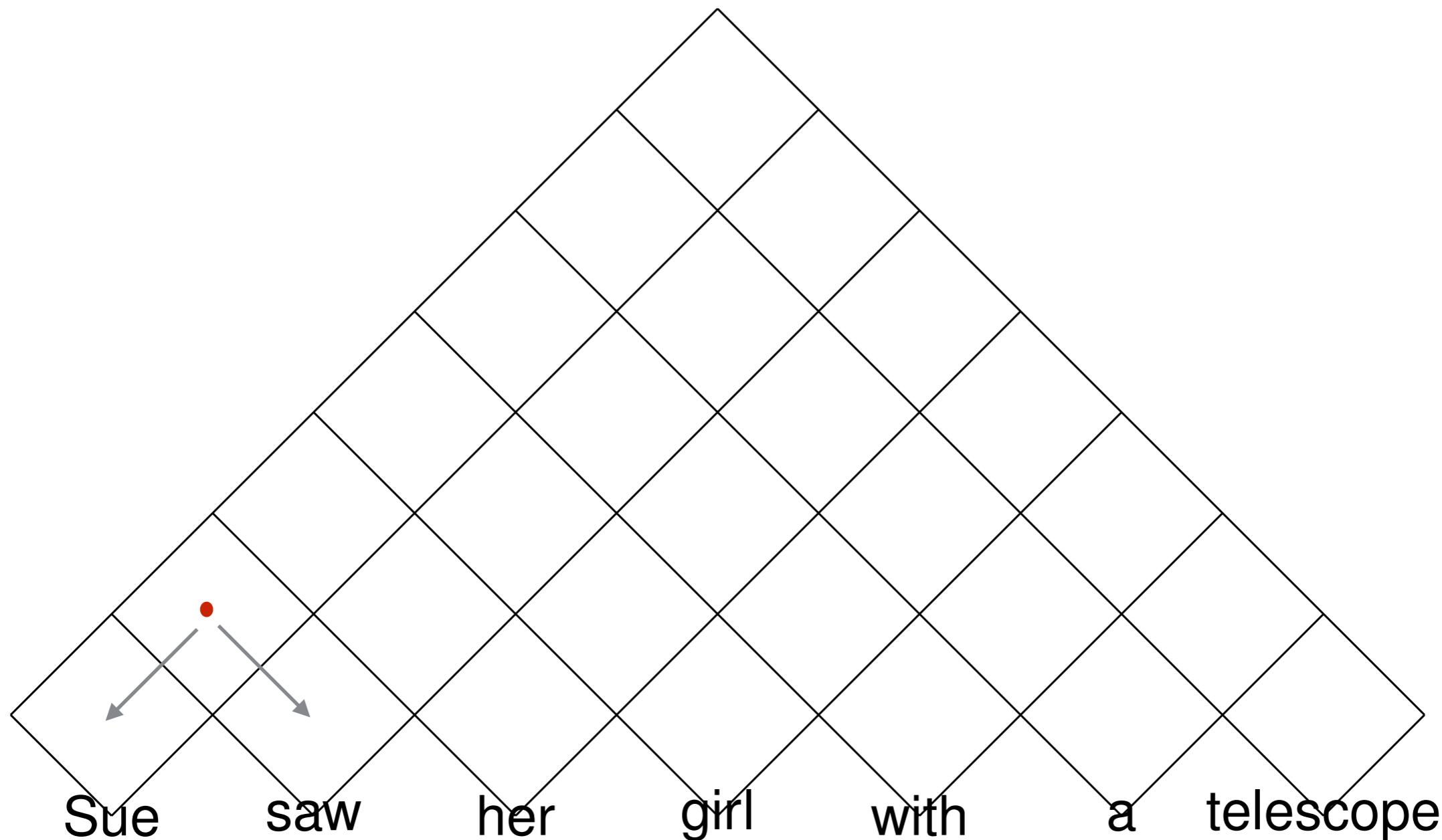
S	x
Noun	v
Verb	v
PP	x
AdjP	x



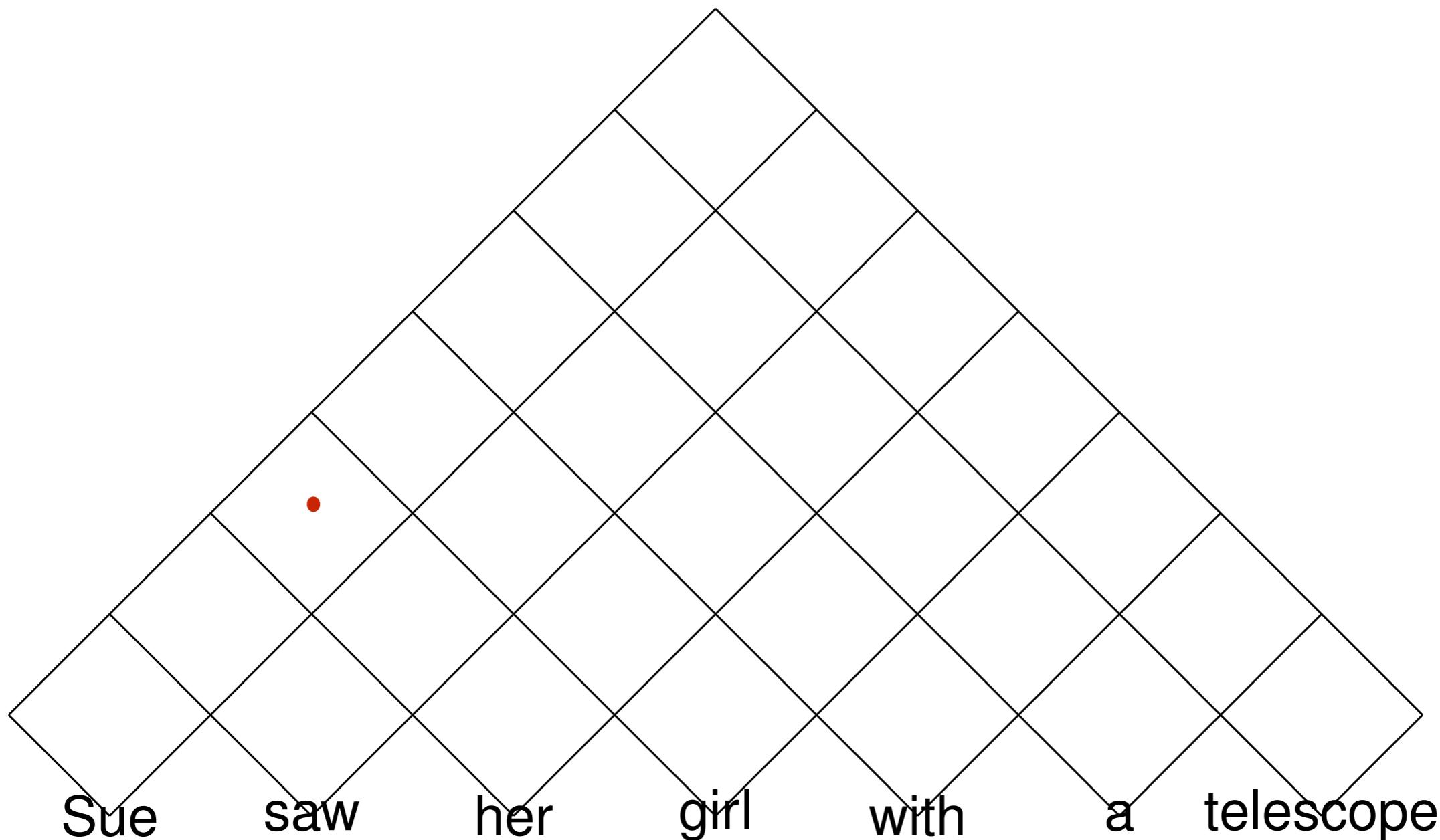
Filling the table



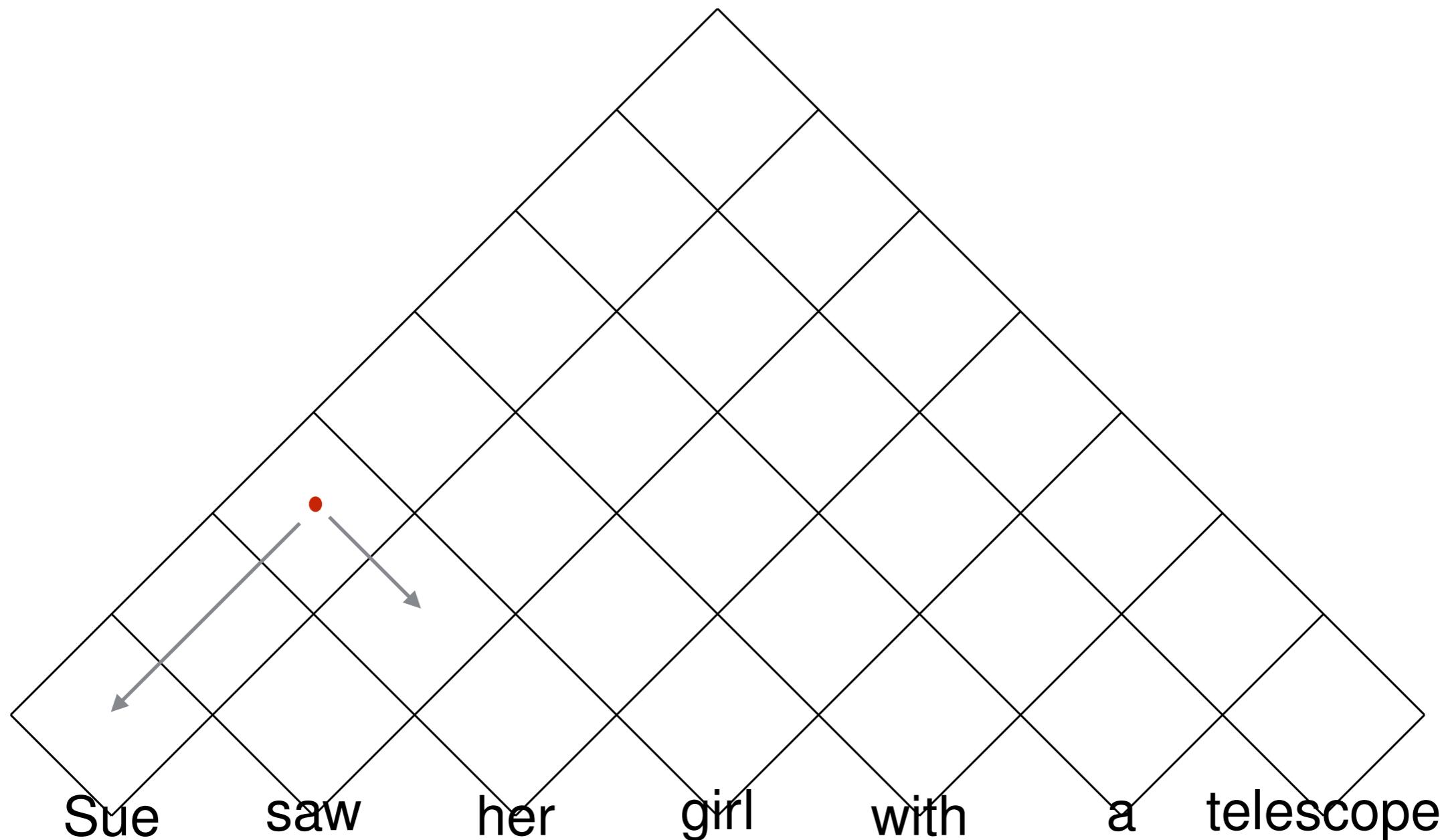
Filling the table



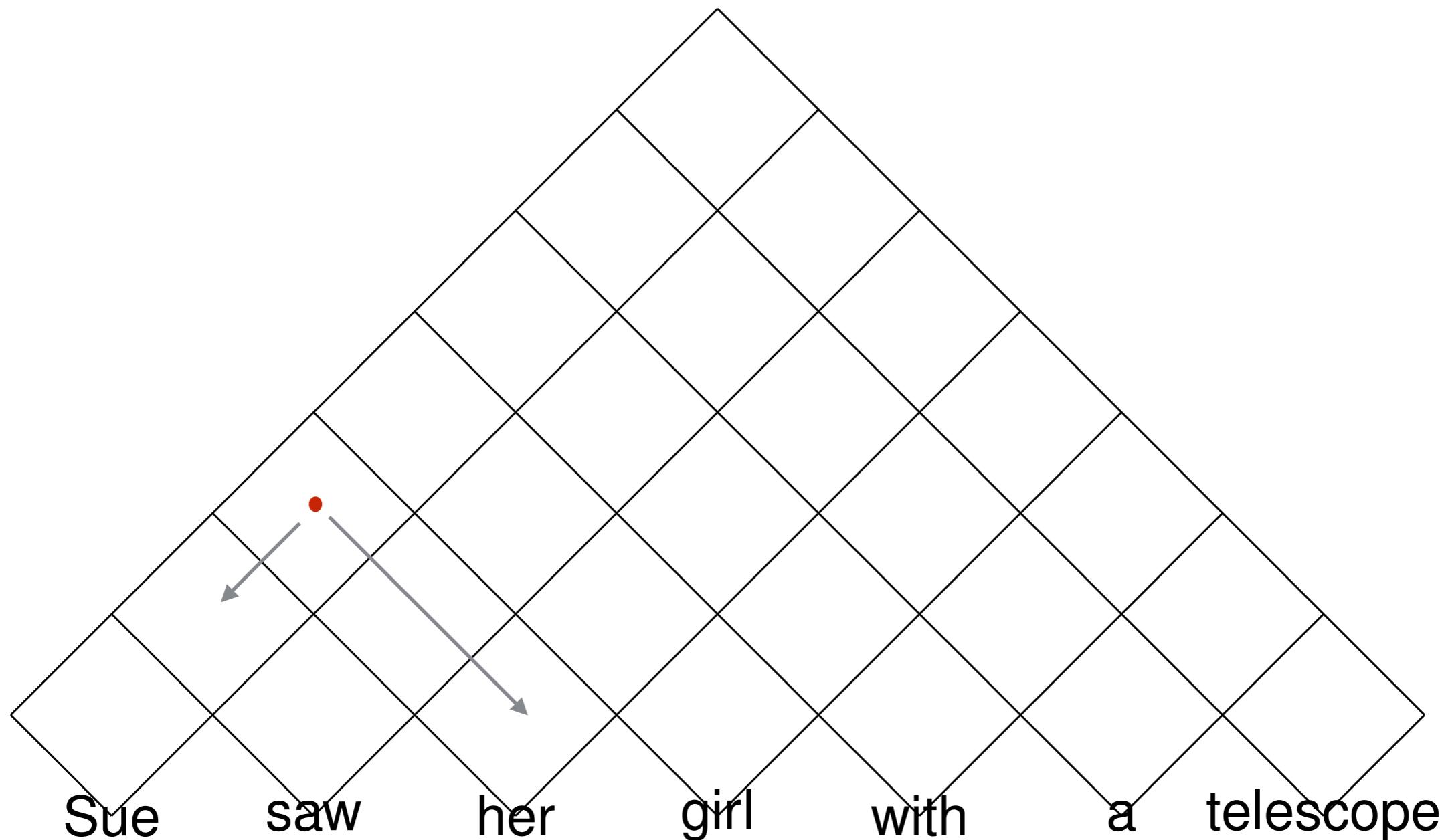
Filling the table



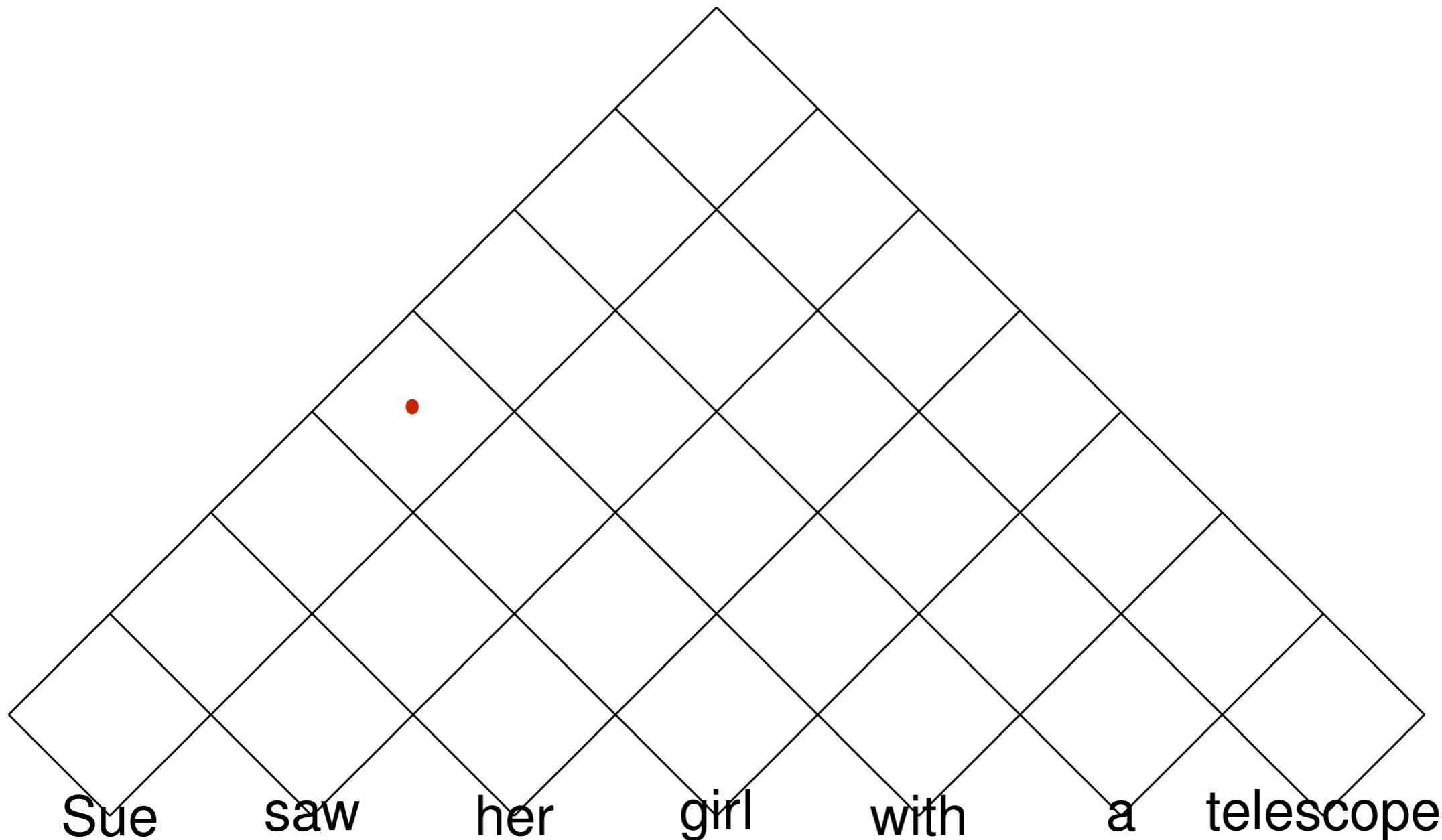
Filling the table



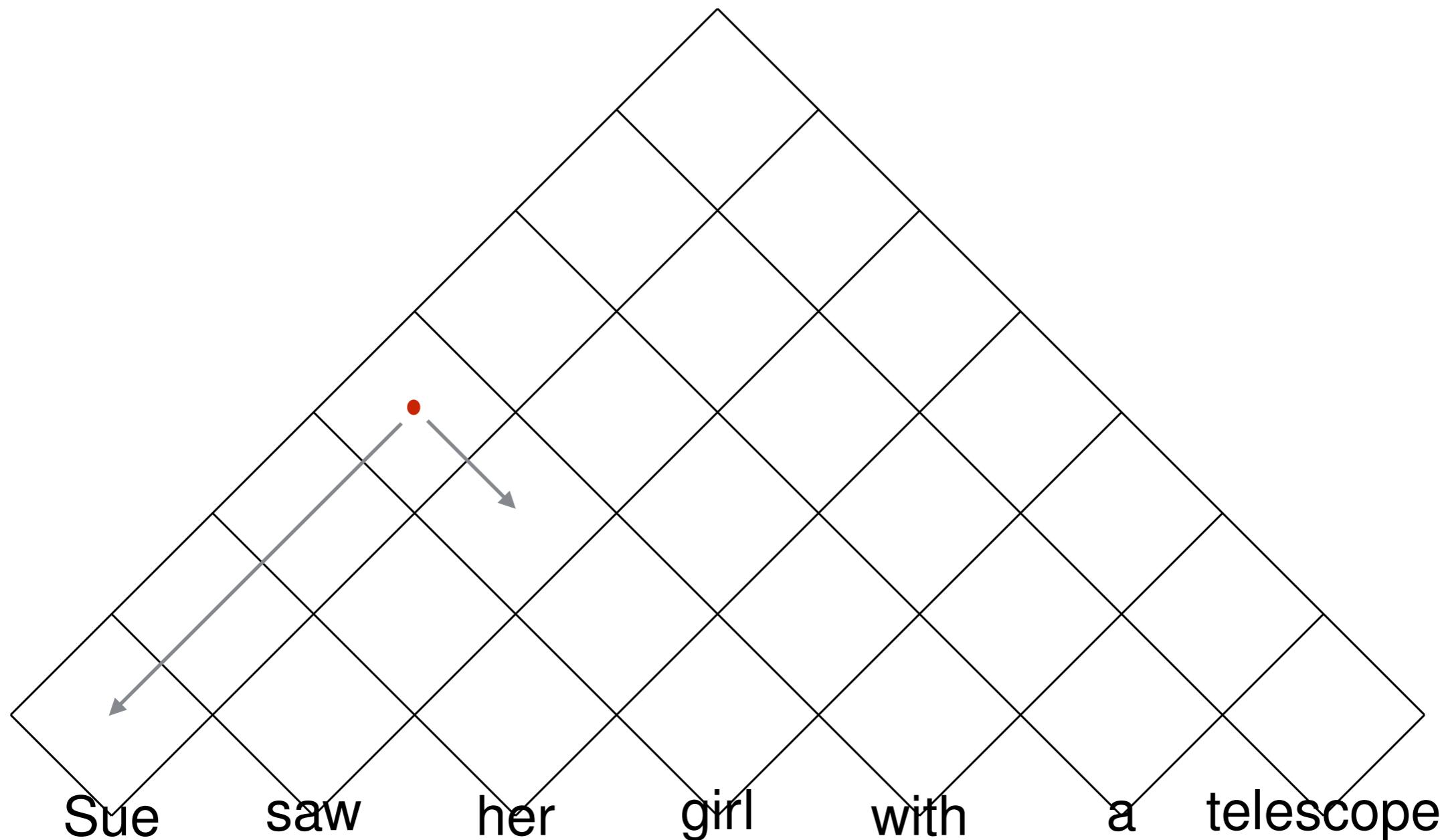
Filling the table



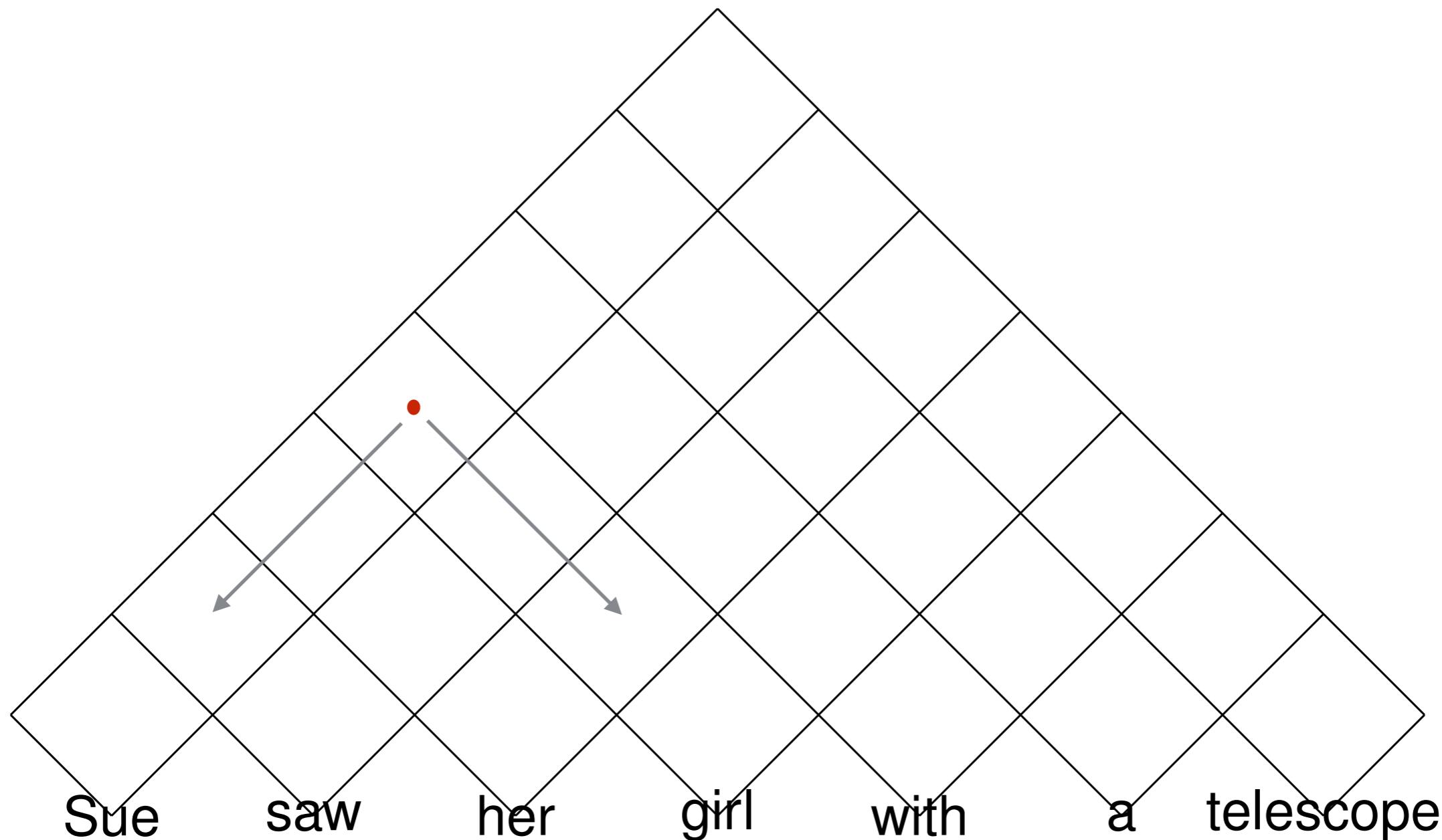
Filling the table



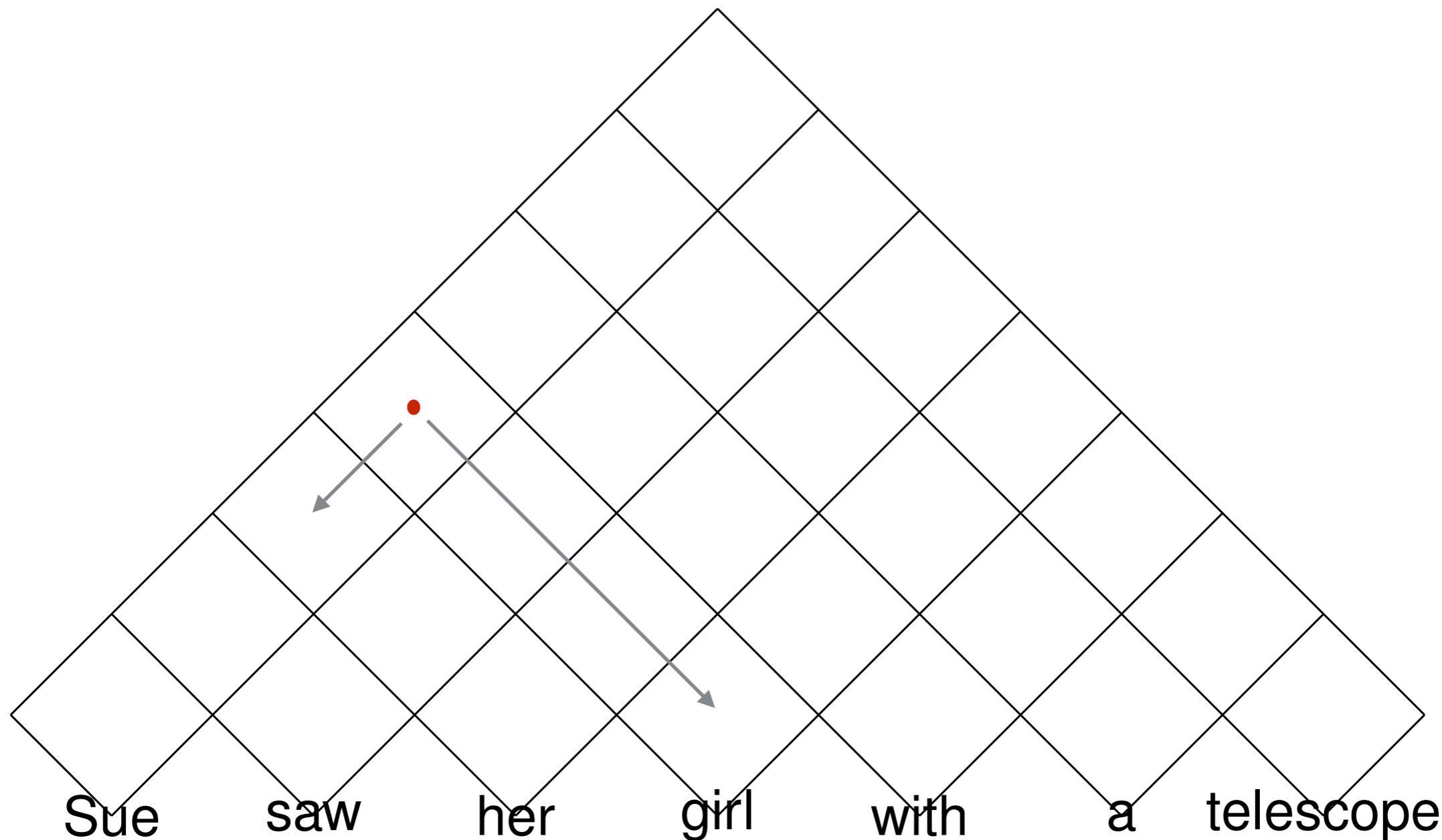
Filling the table



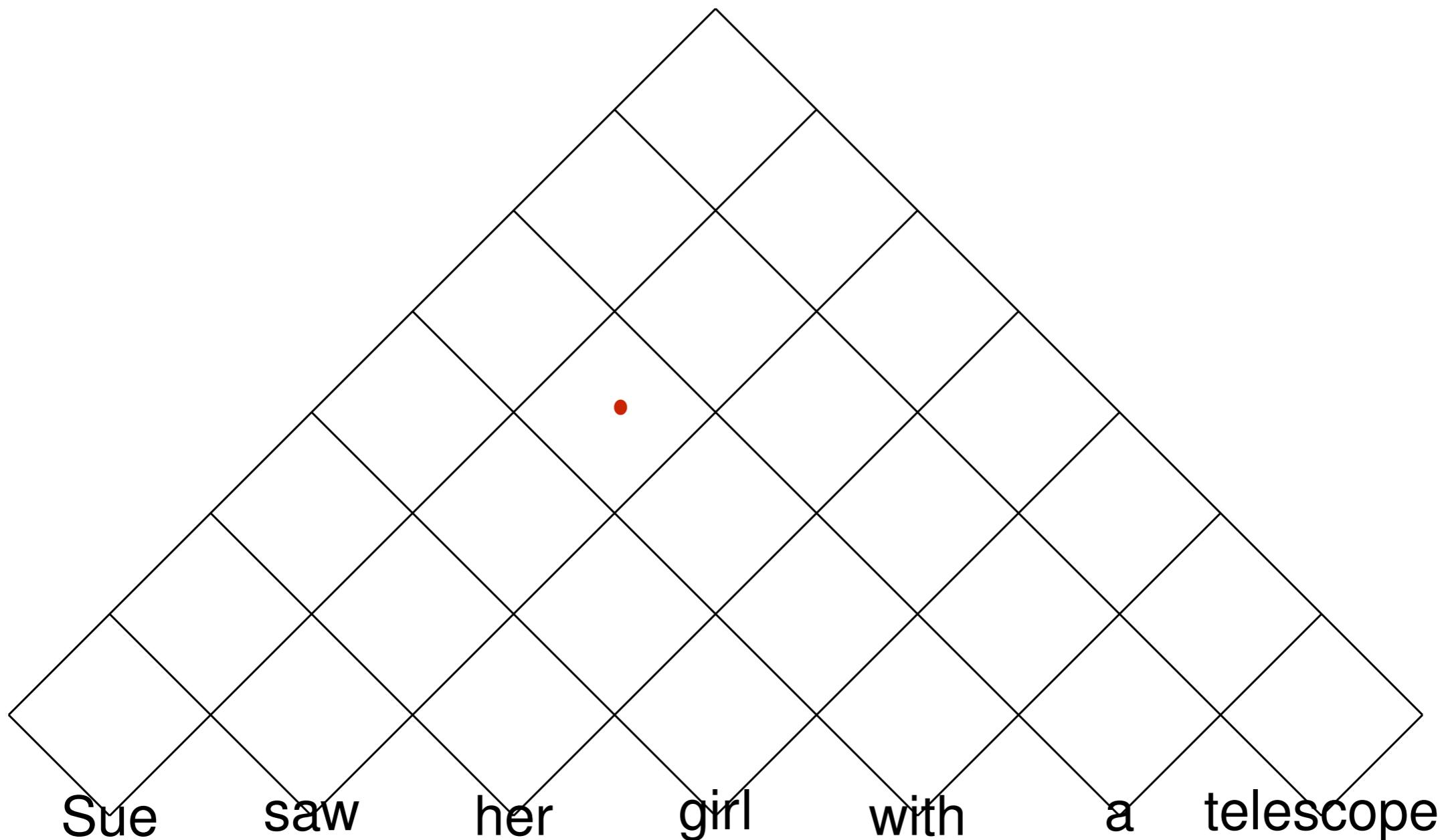
Filling the table



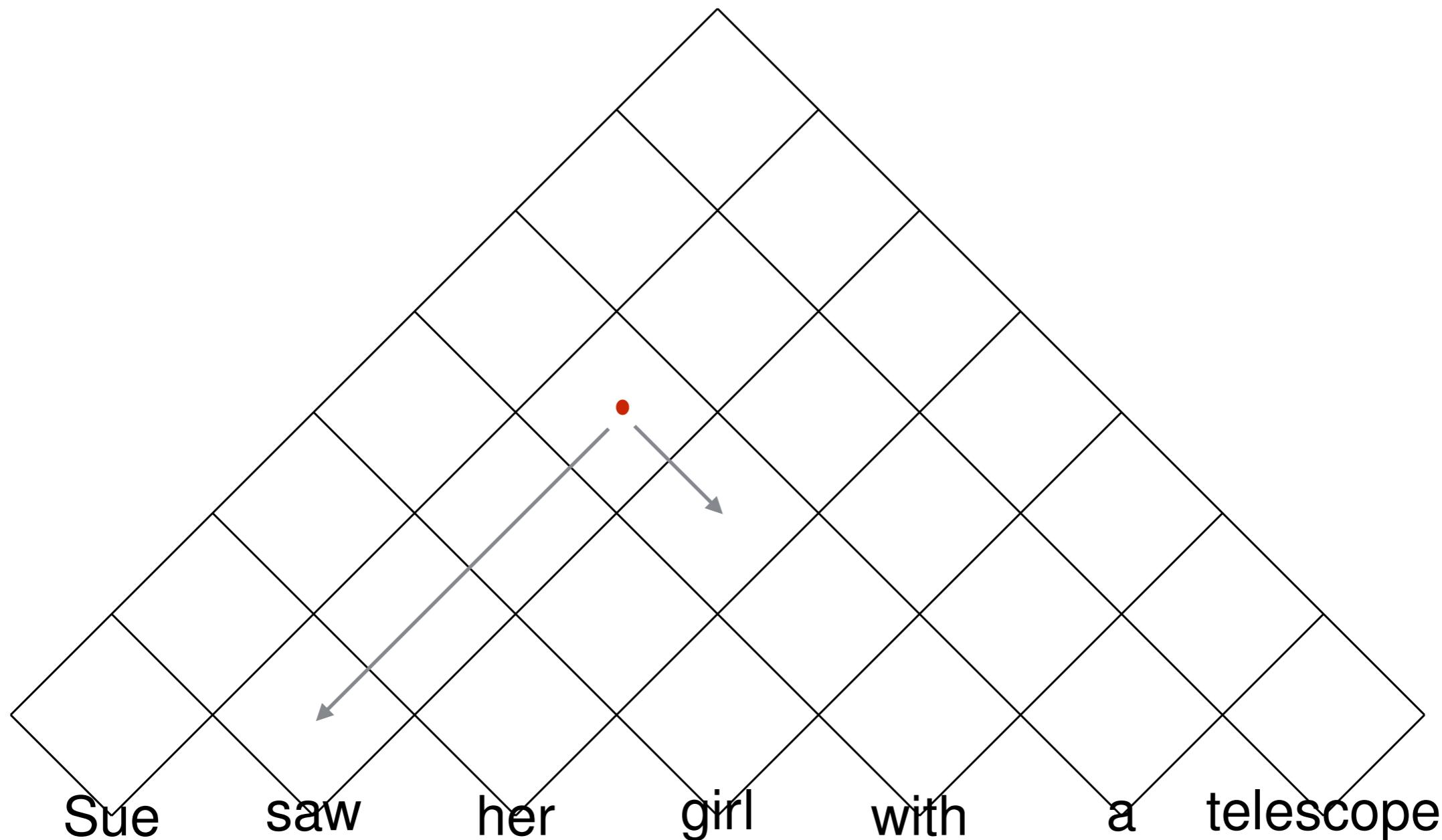
Filling the table



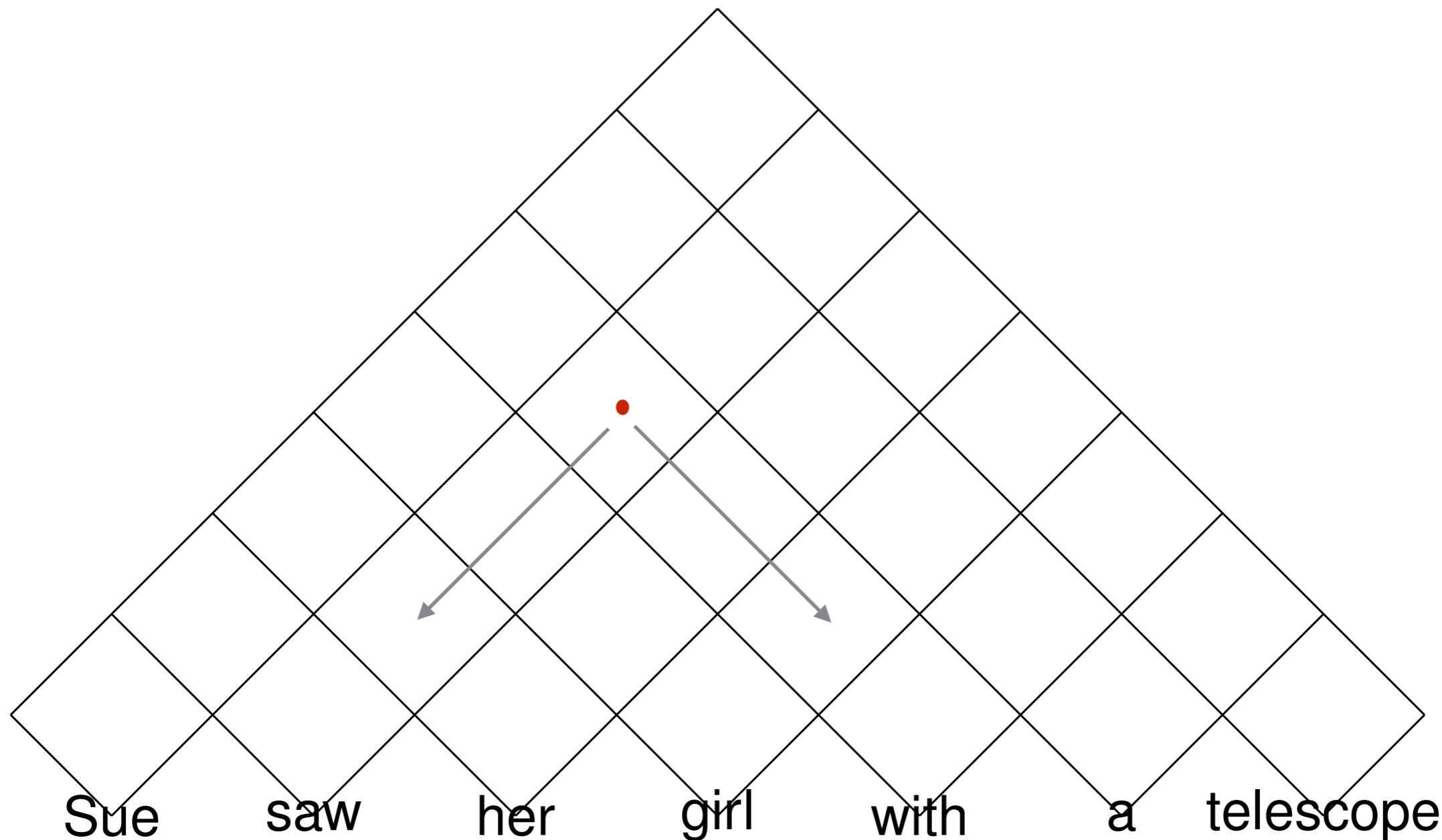
Filling the table



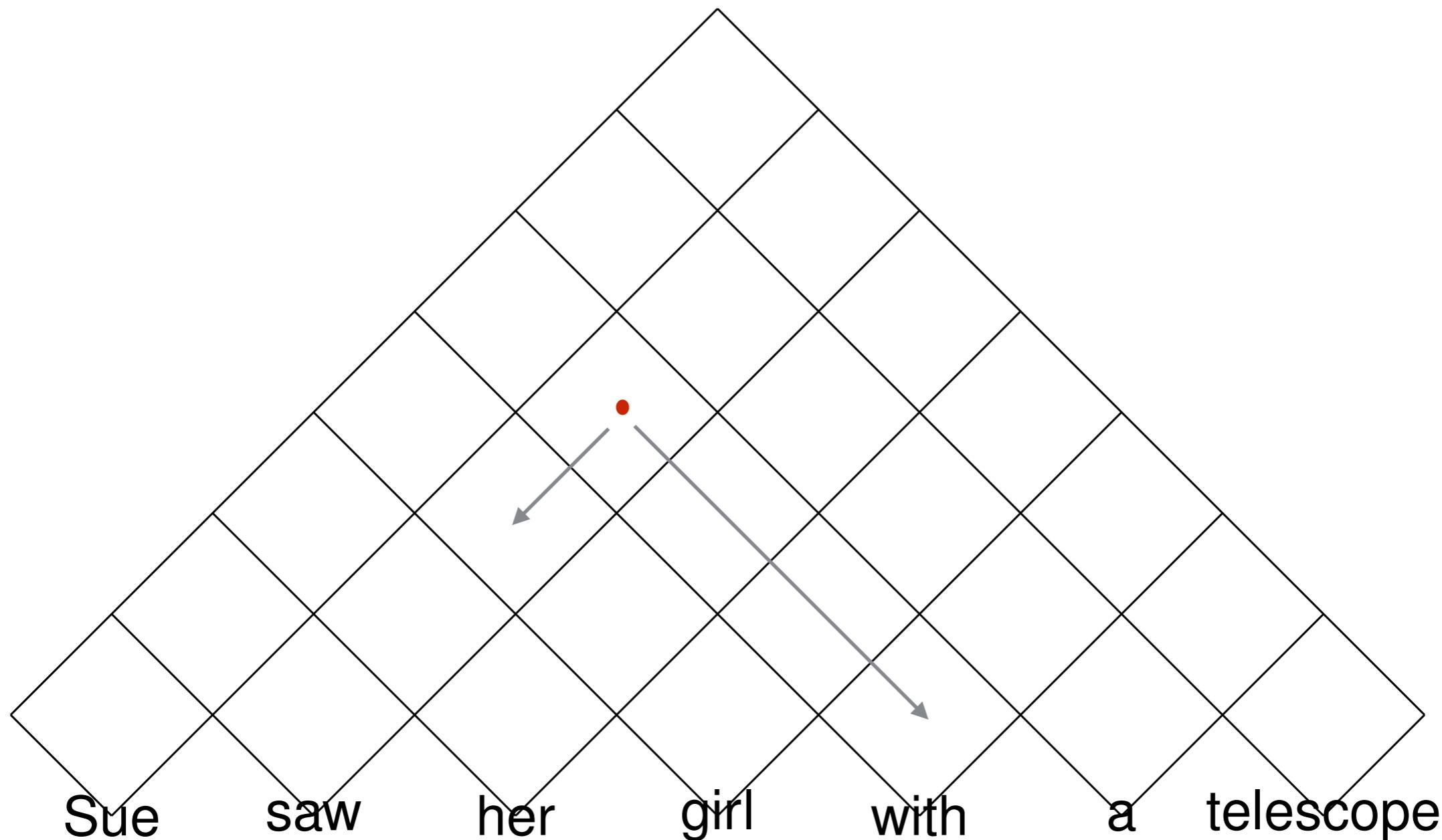
Filling the table



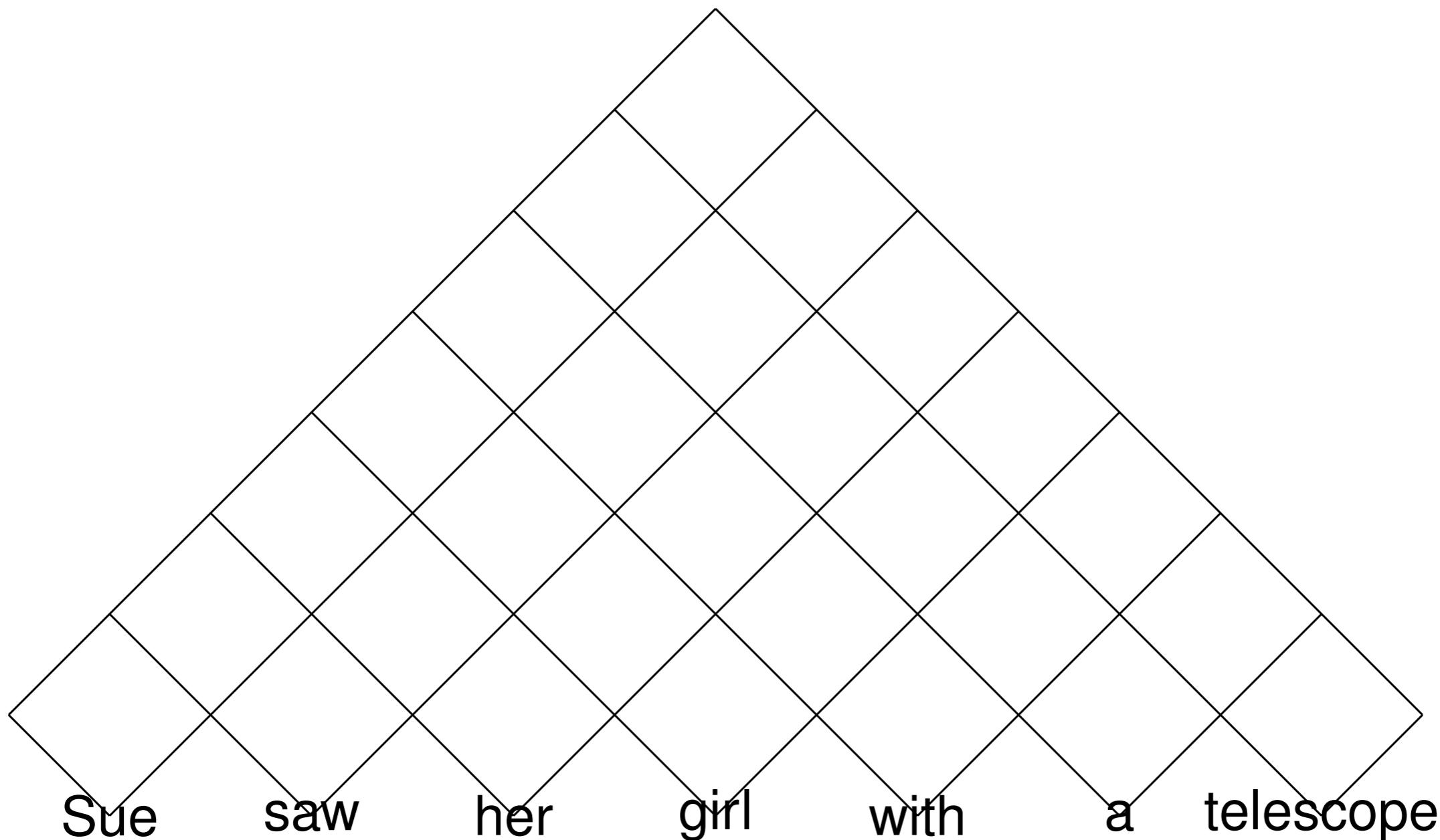
Filling the table



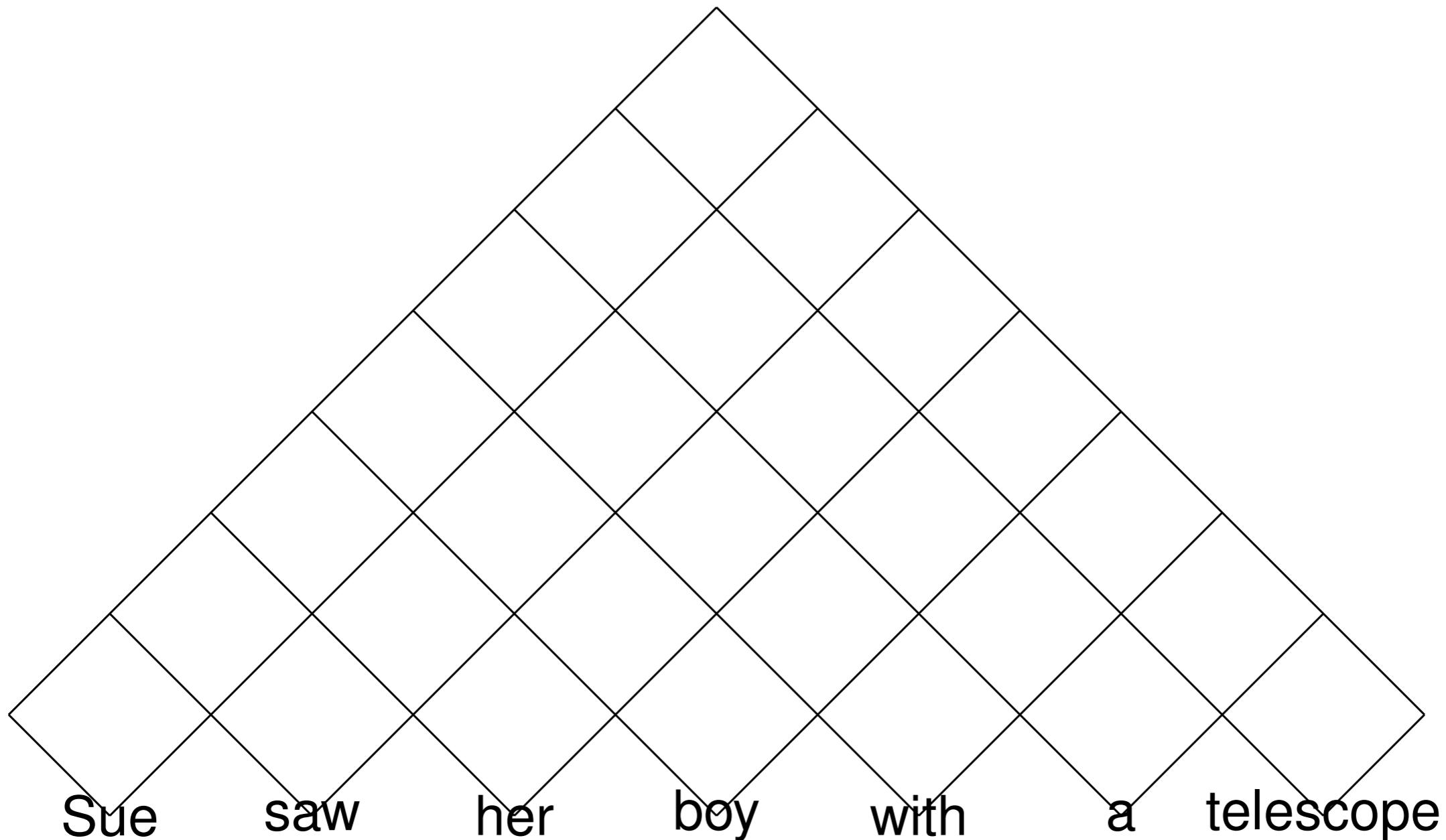
Filling the table



Handling Unary rules?

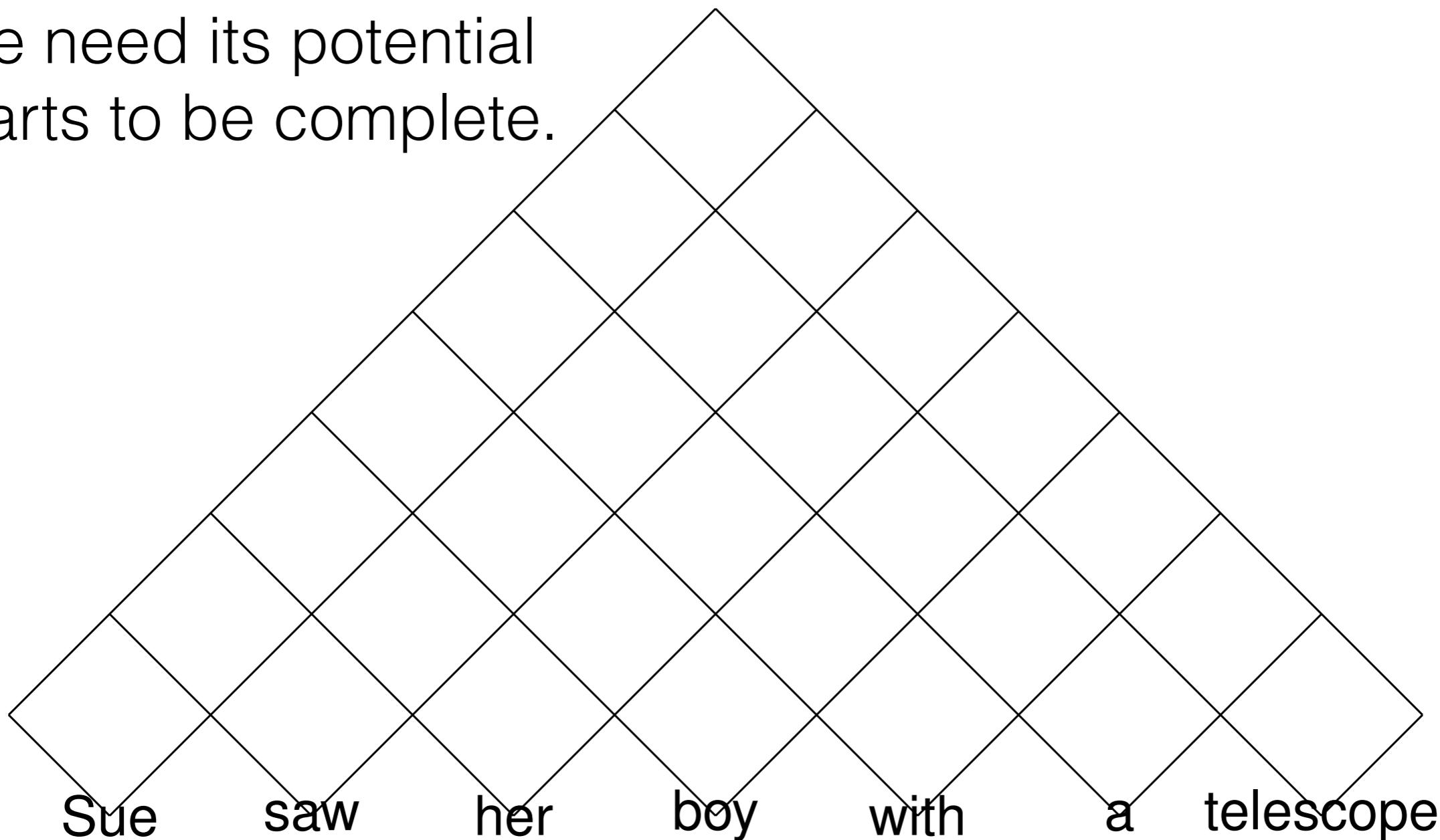


Which Order?

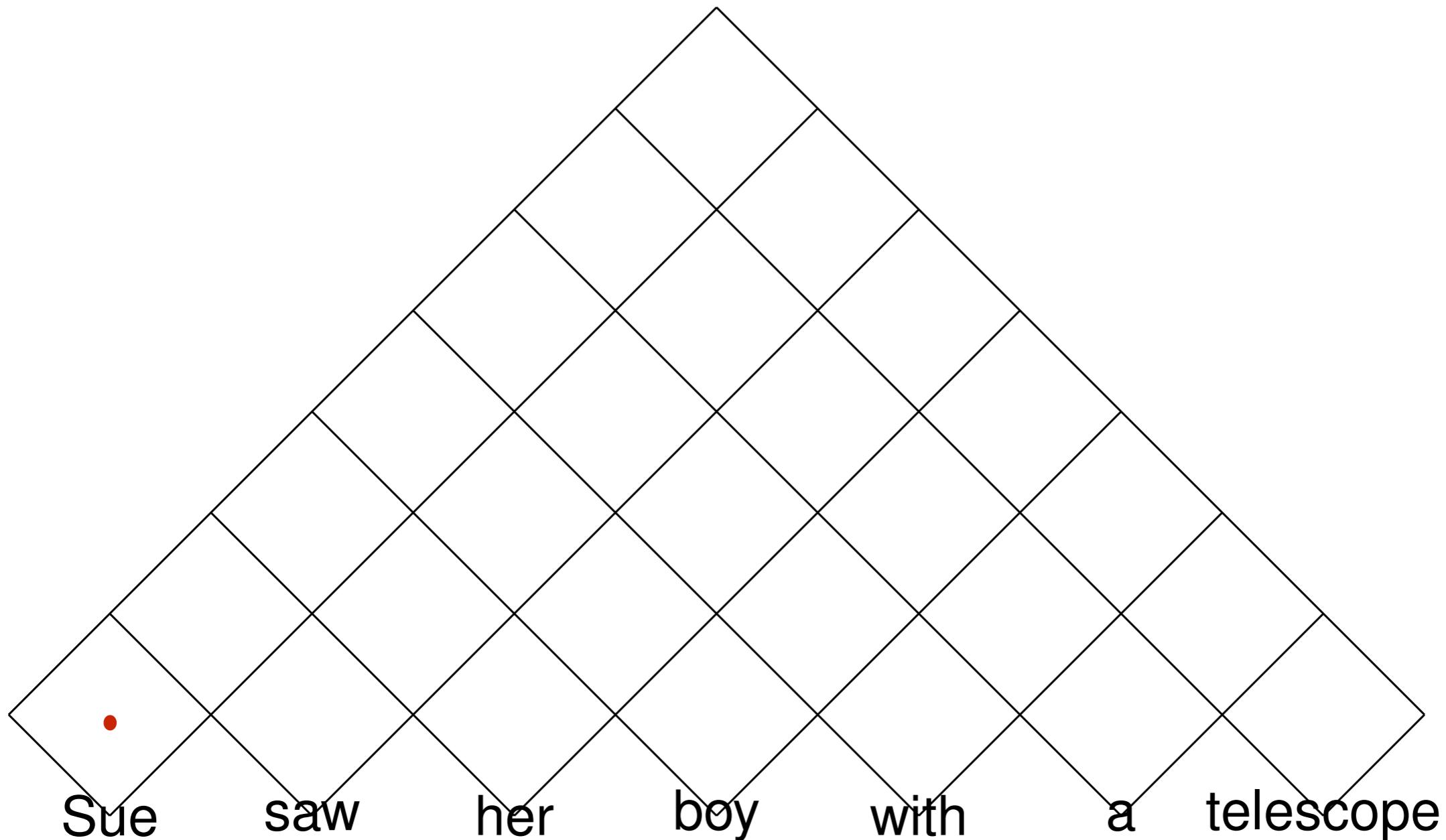


Which Order?

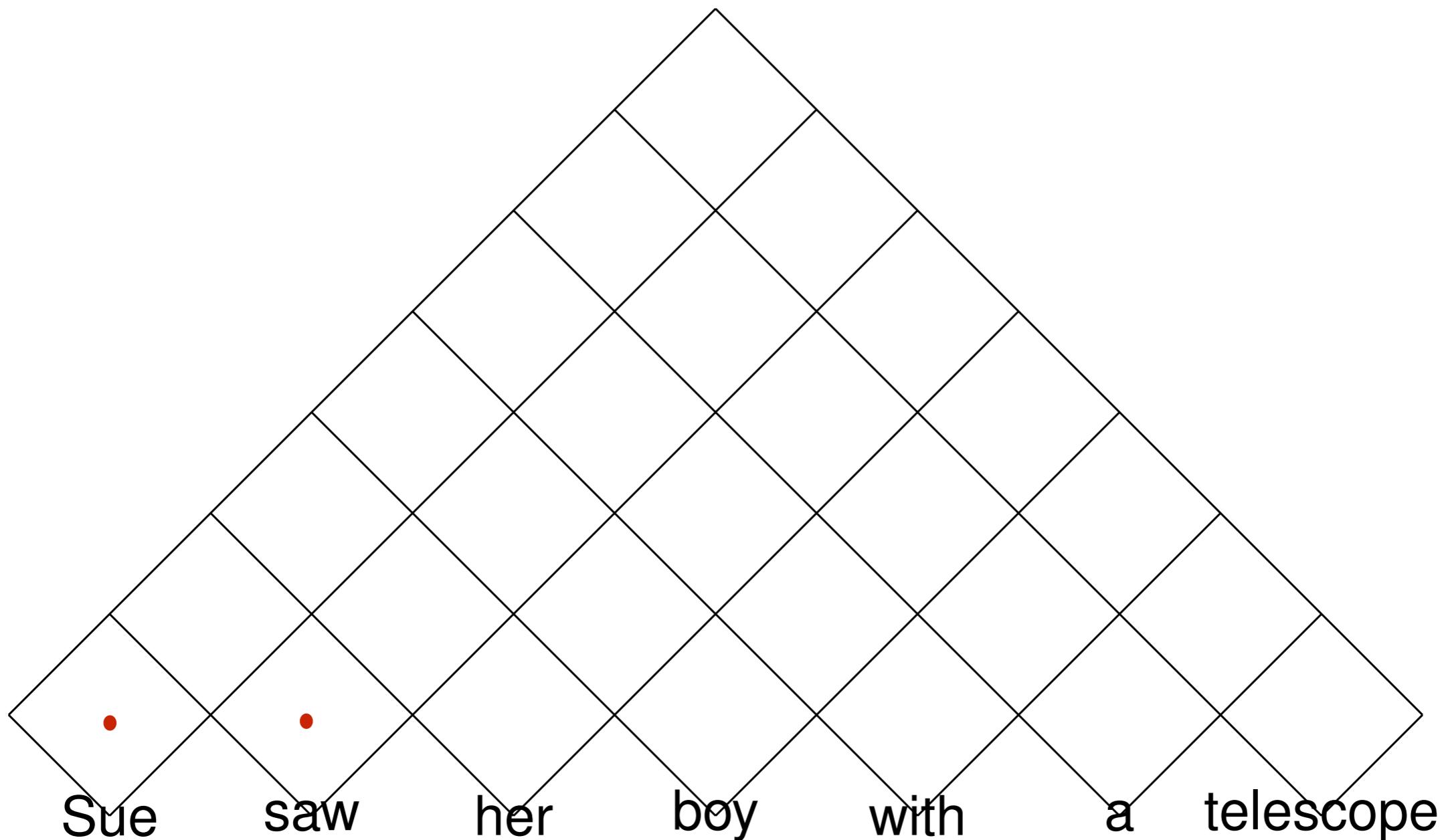
When handling a cell,
we need its potential
parts to be complete.



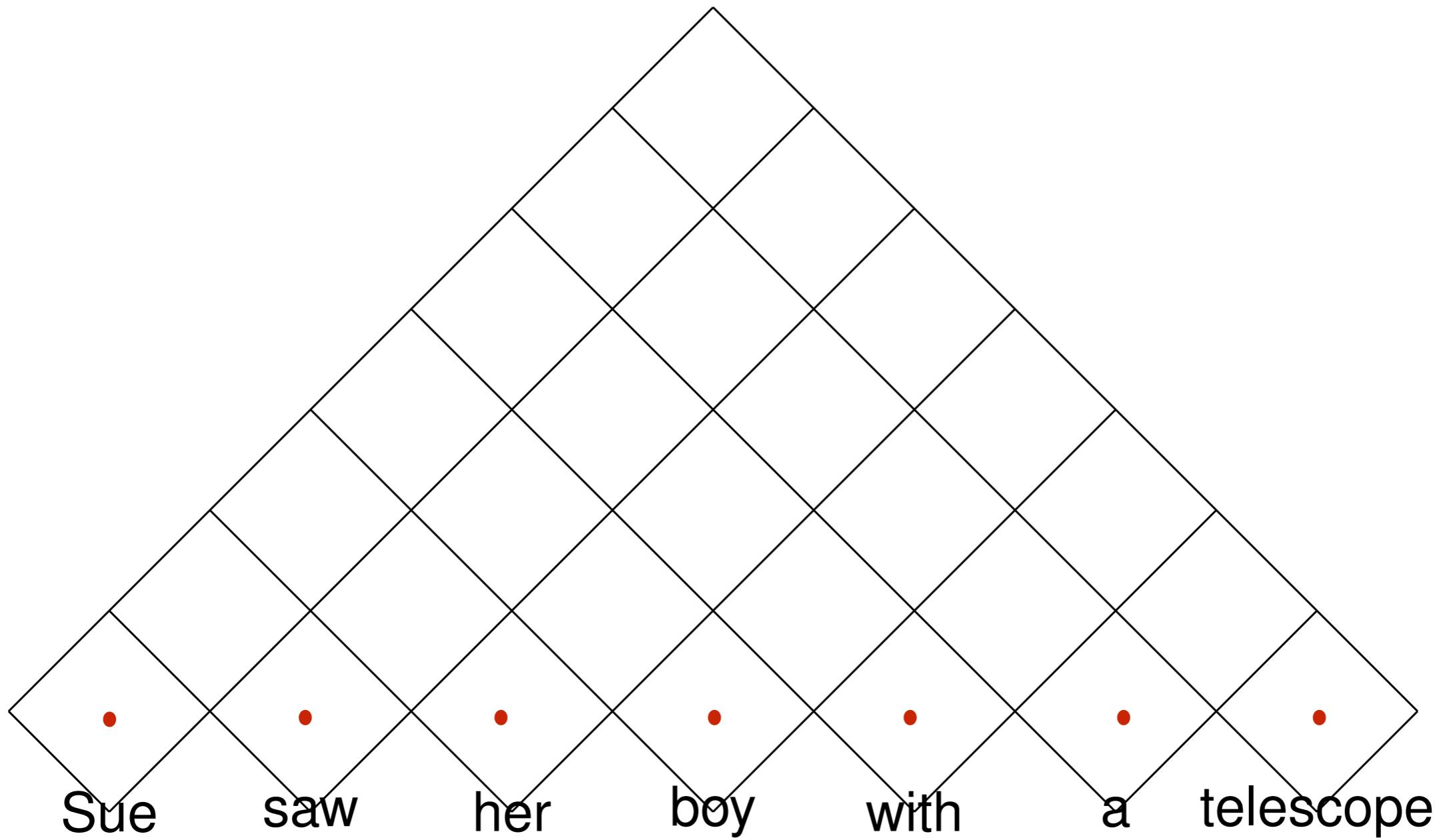
Which Order?



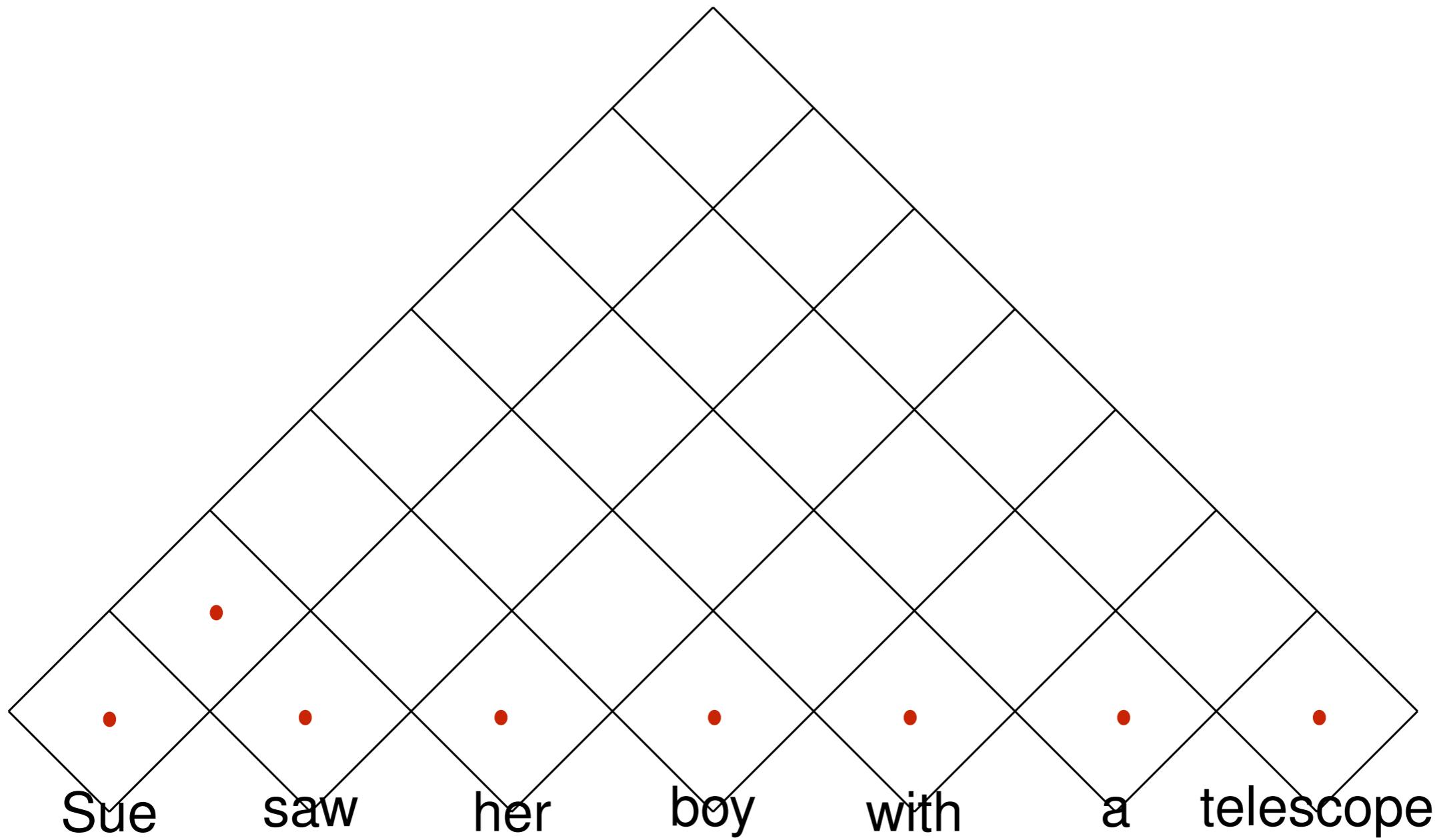
Which Order?



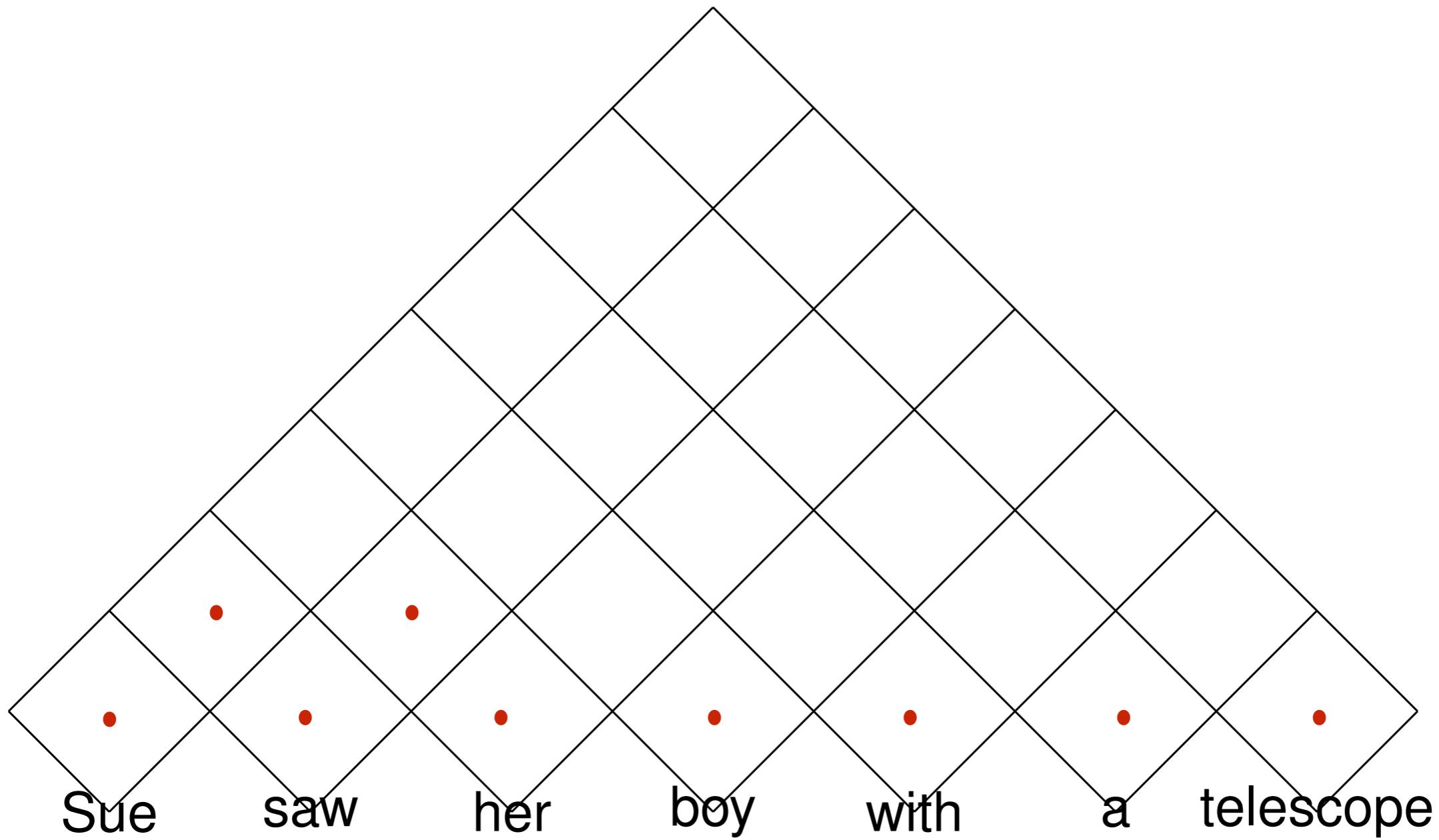
Which Order?



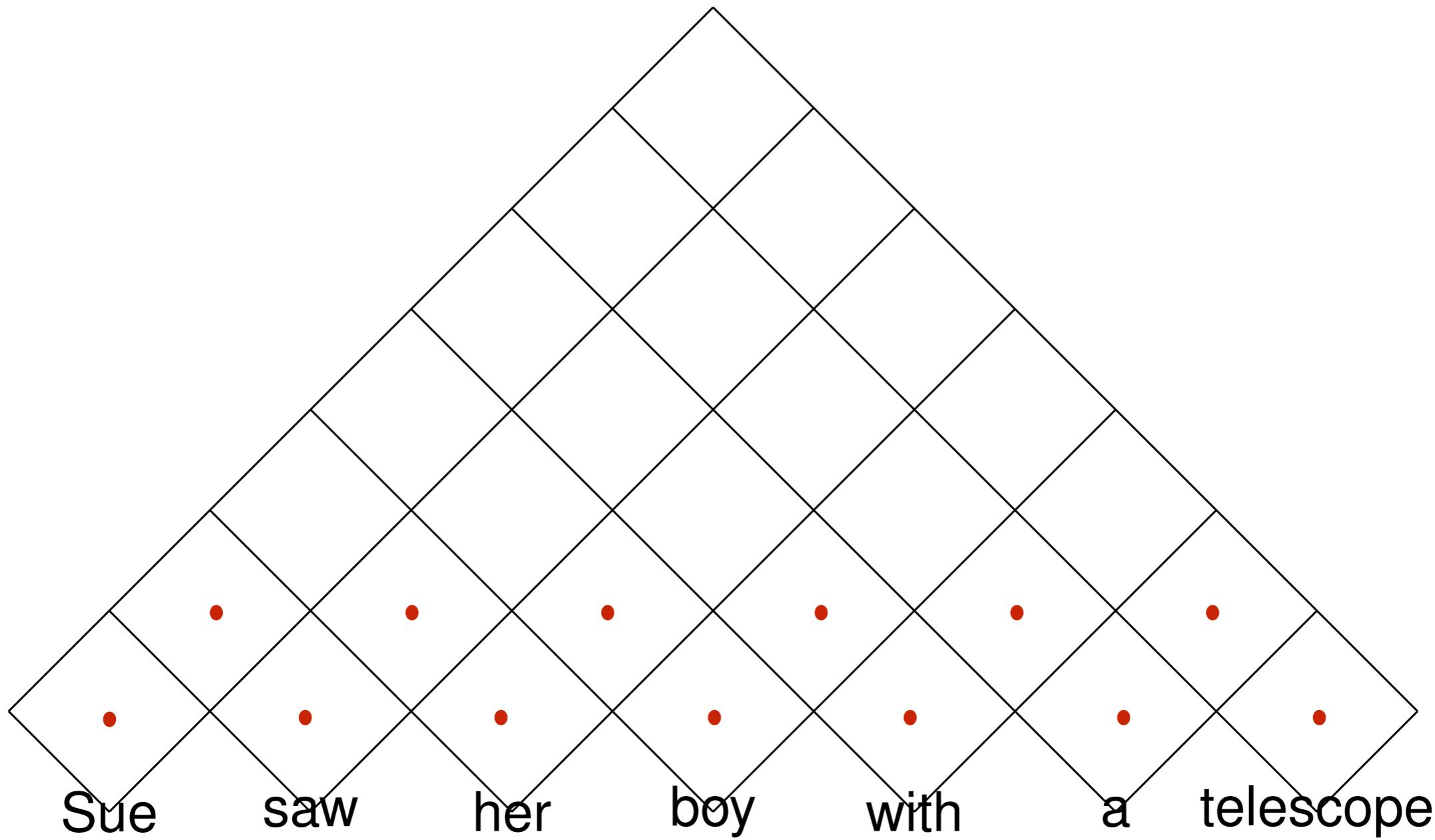
Which Order?



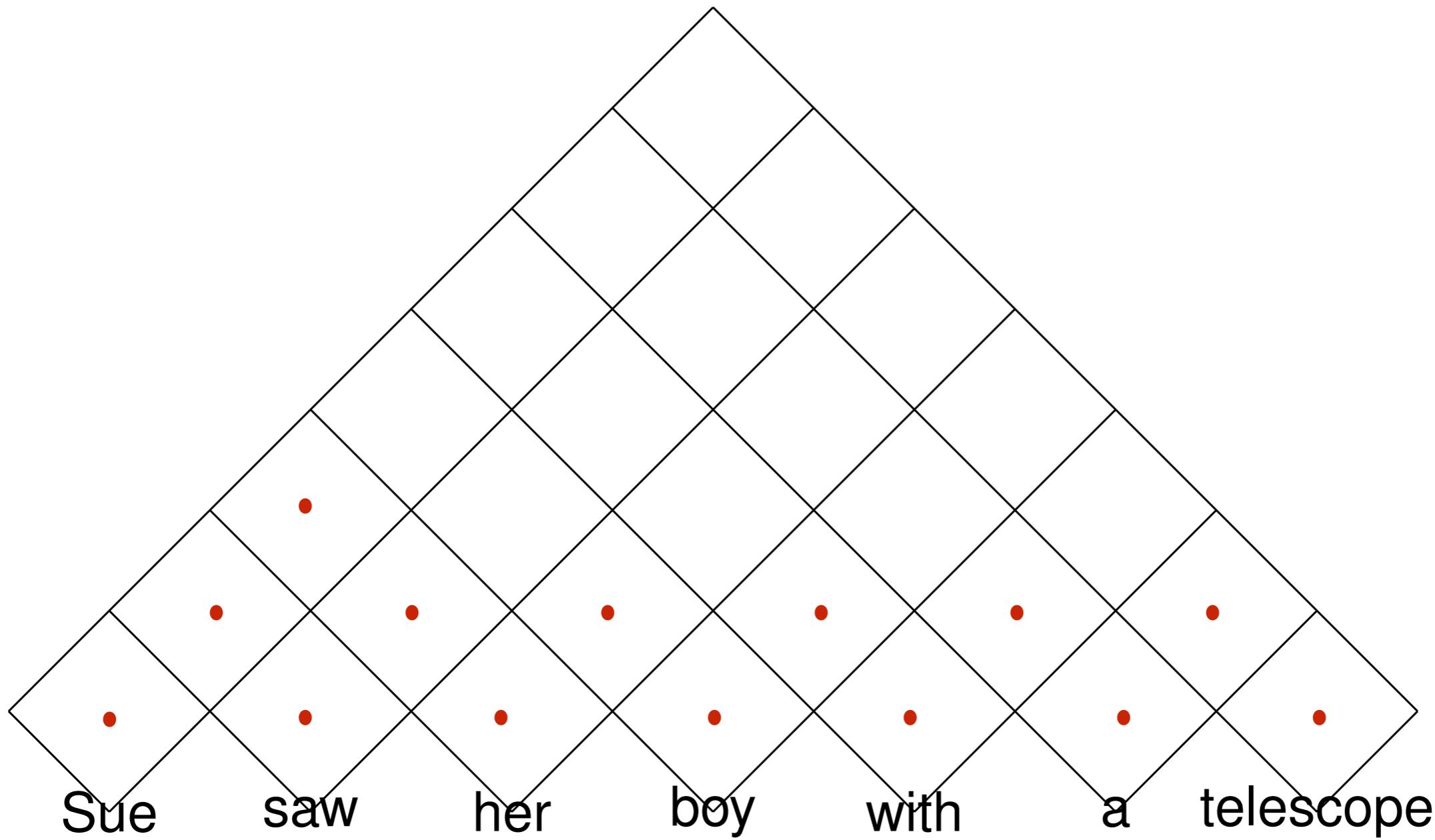
Which Order?



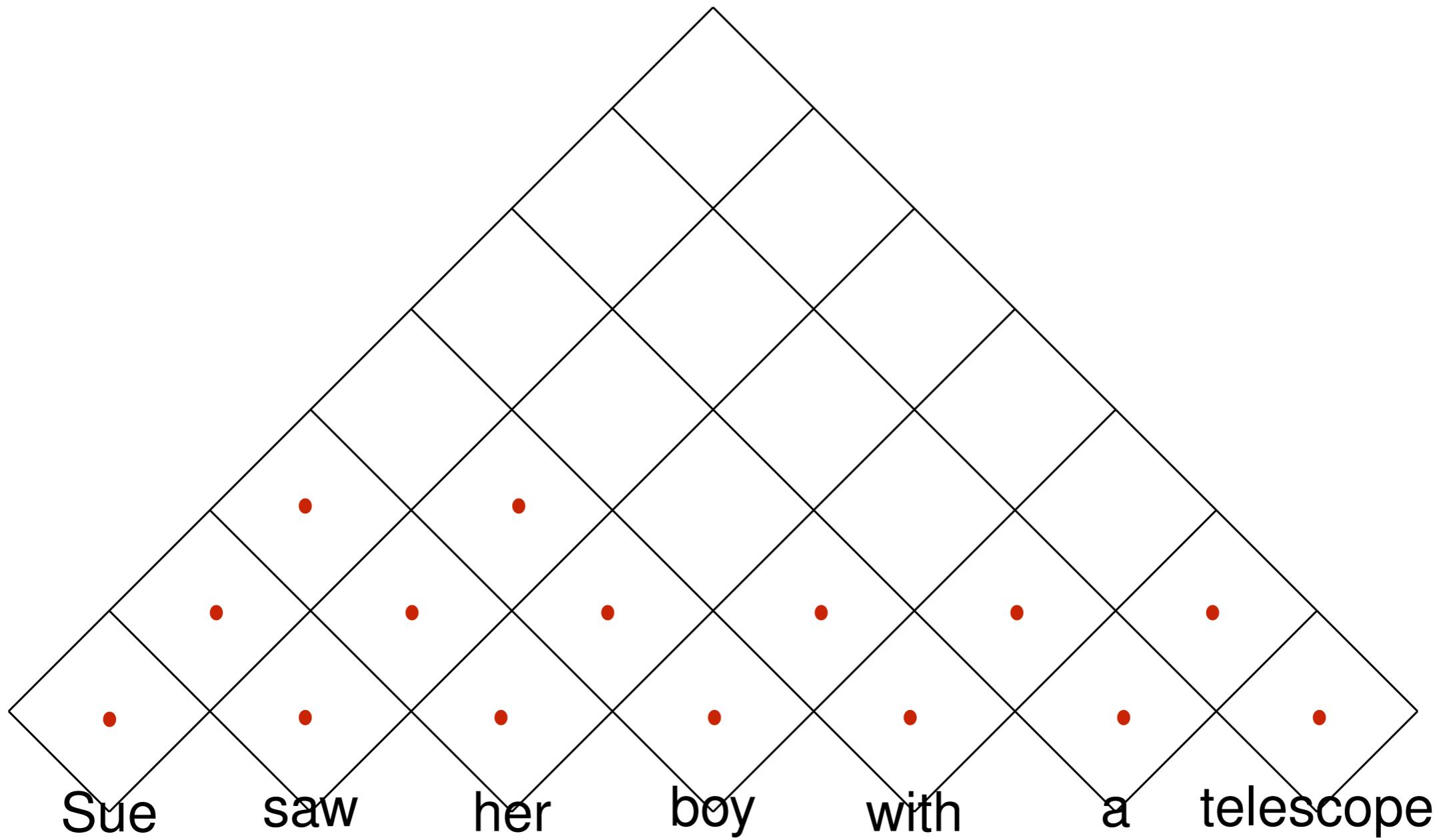
Which Order?



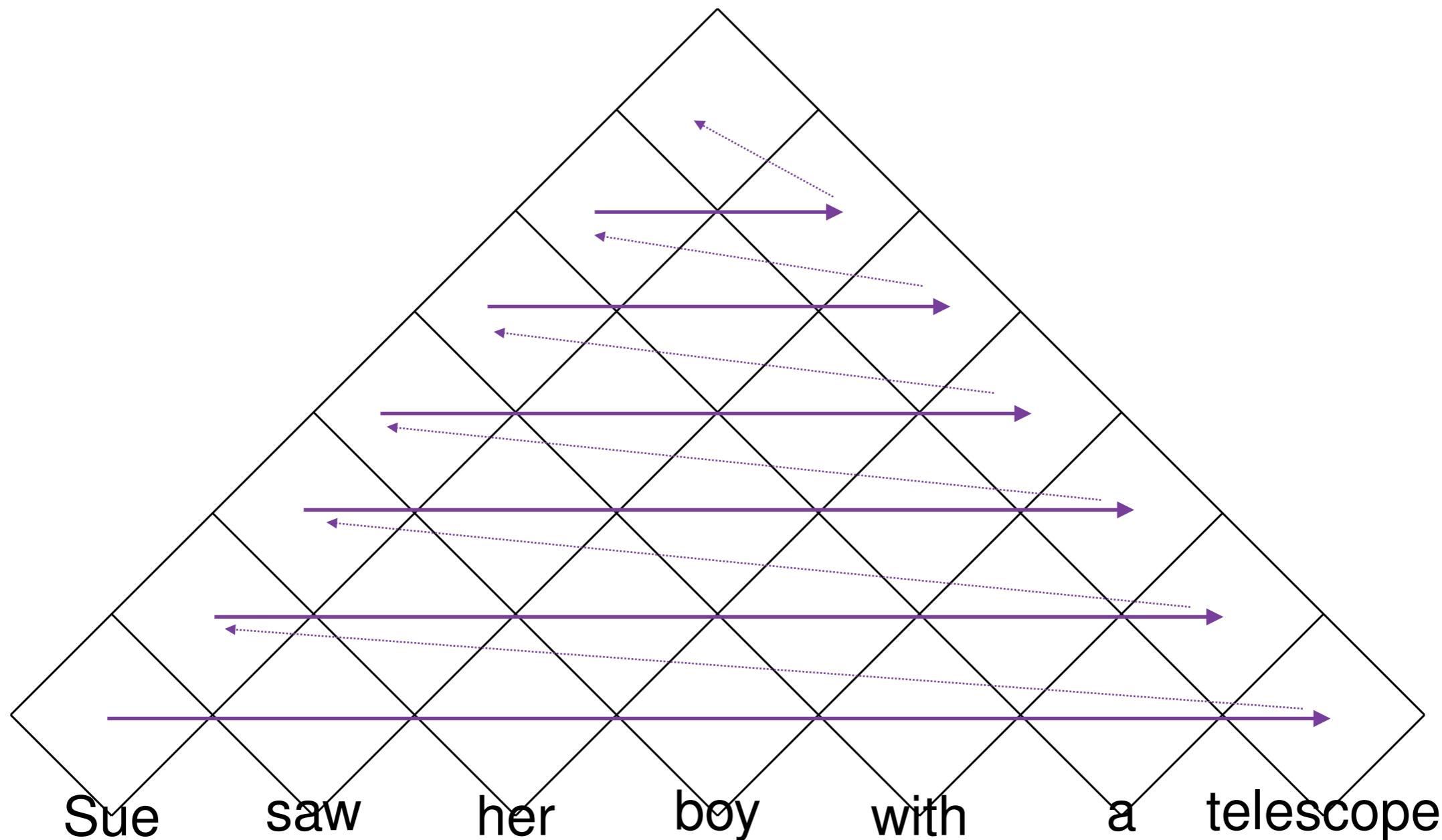
Which Order?



Which Order?

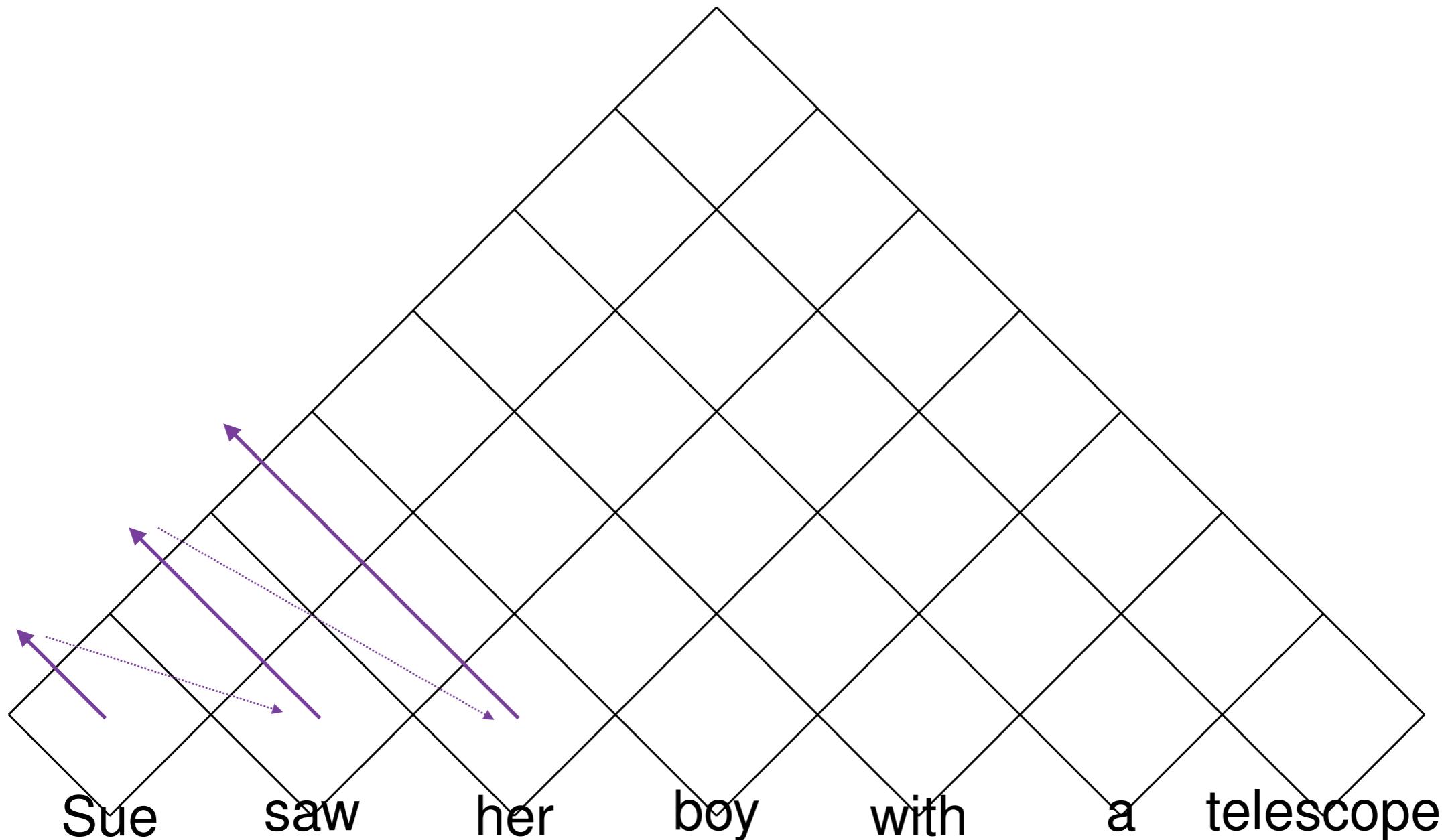


Which Order?



Which Order?

"left corner parsing"



Complexity?

Complexity?

- ▶ n^2g cells to fill

Complexity?

- ▶ n^2g cells to fill
- ▶ g^2n ways to fill each one

Complexity?

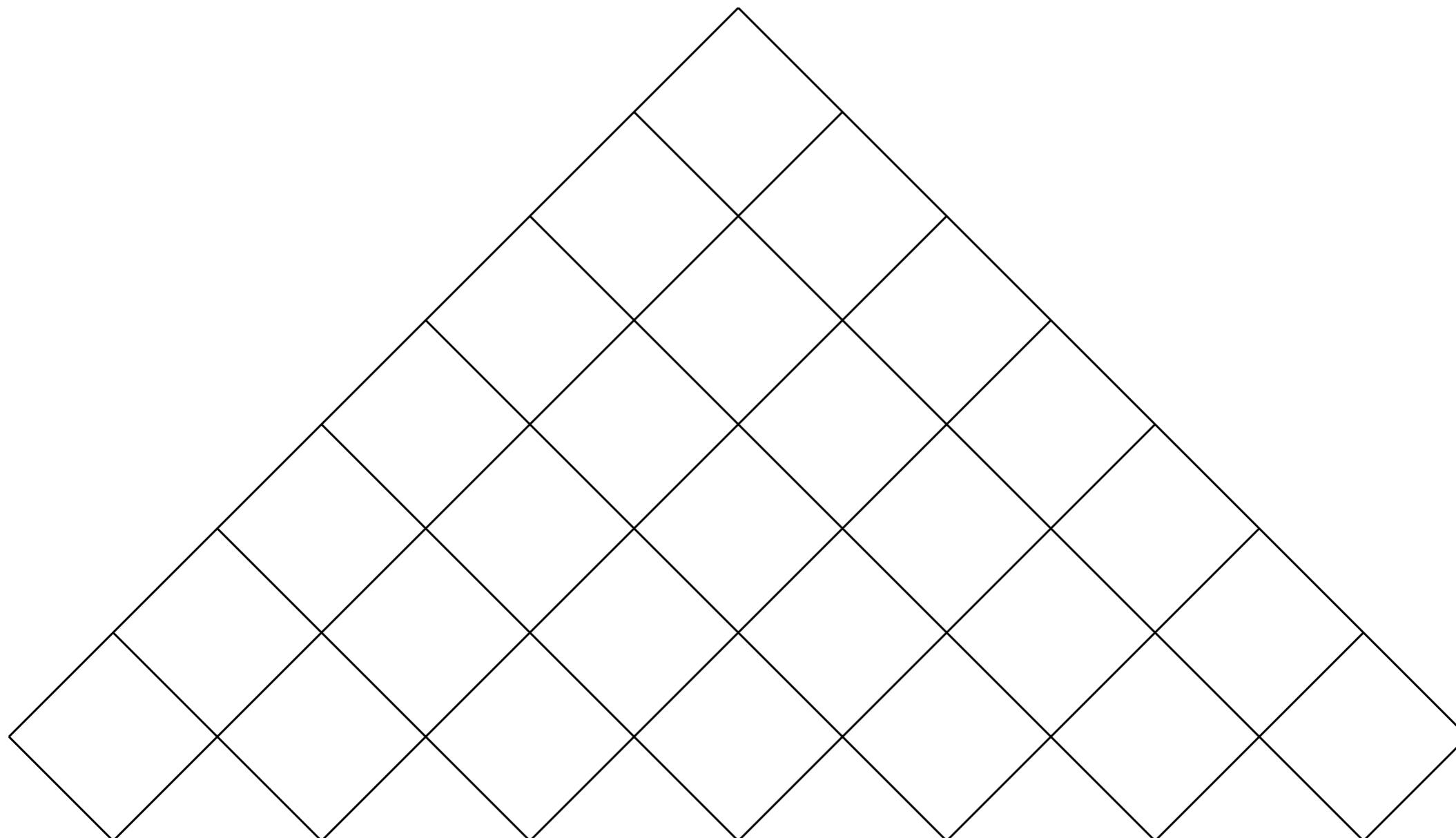
- ▶ n^2g cells to fill
- ▶ g^2n ways to fill each one

$$O(g^3 n^3)$$

Finding a parse

Parsing – we want to actually find a parse tree

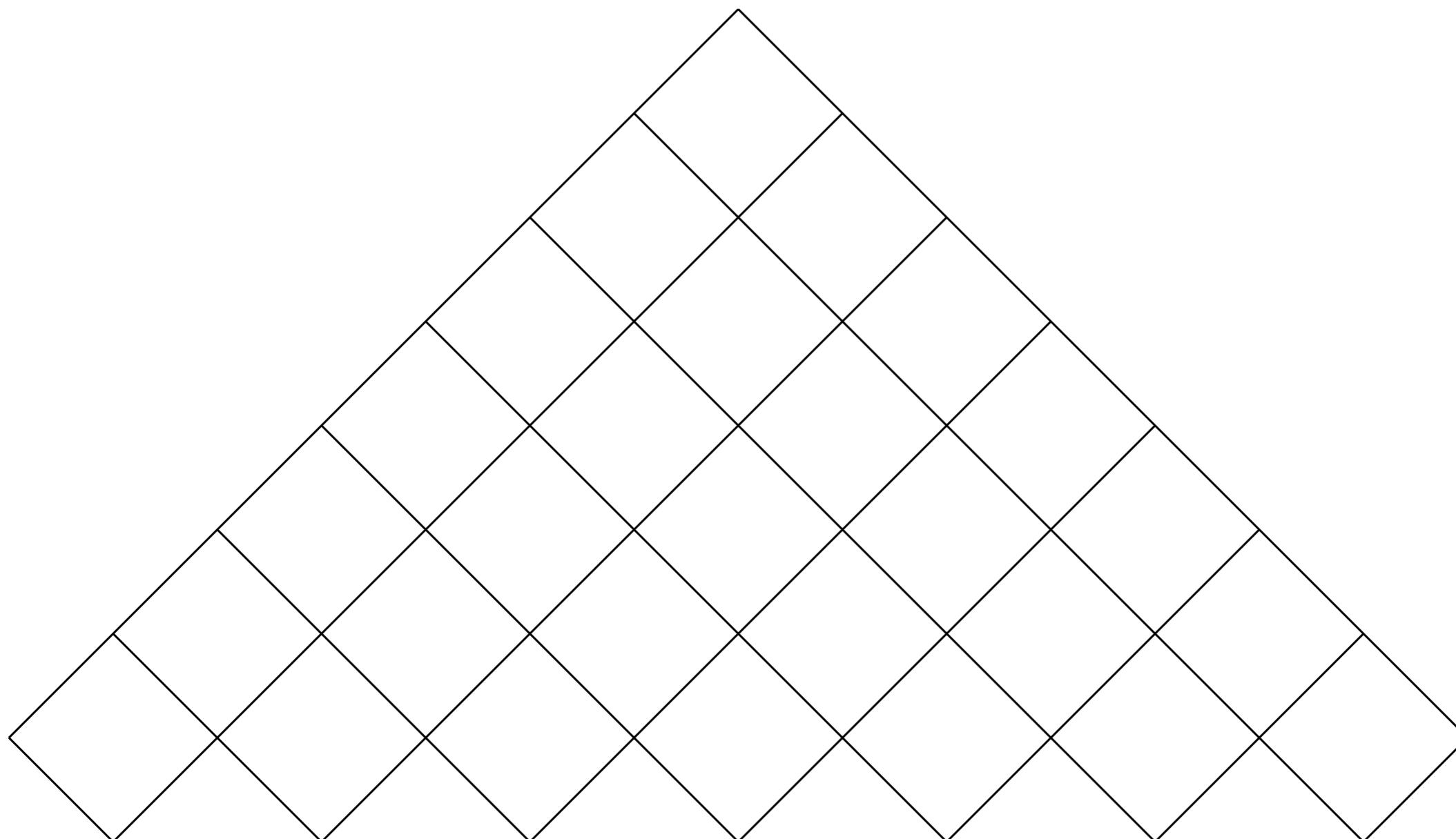
Easy: also keep a possible split point for each NT



PCFG Parsing and Disambiguation

Disambiguation – we want THE BEST parse tree

Easy: for each NT, keep best split point, and score.



PCFG Parsing Recap

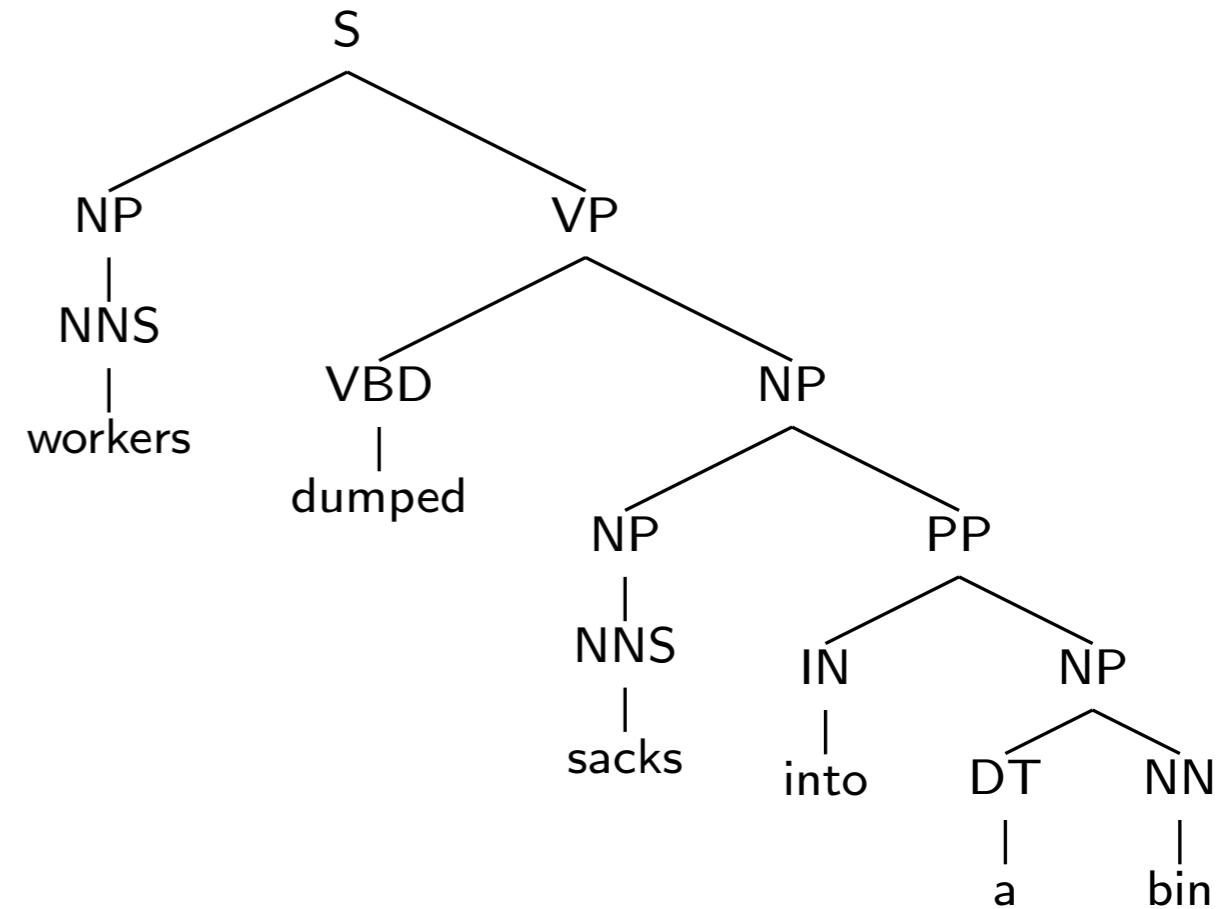
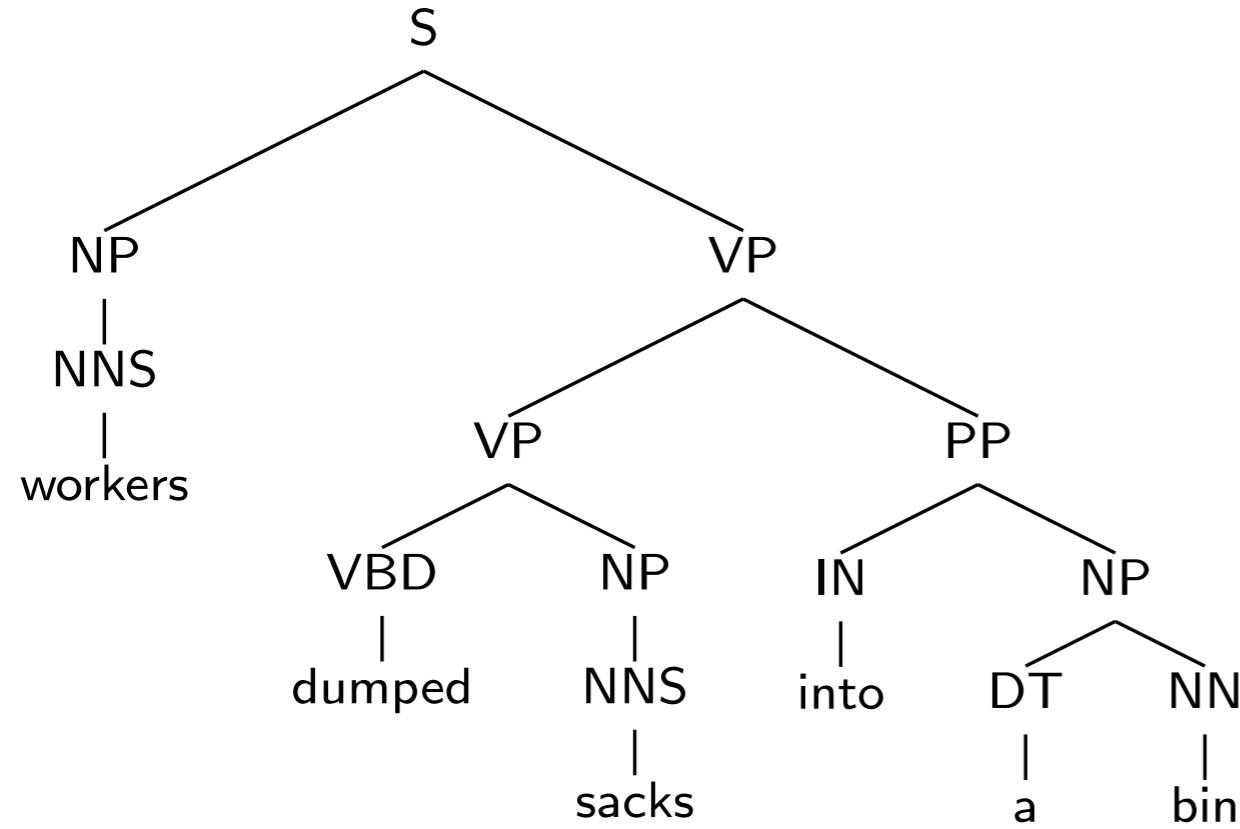
- Extract grammar + probabilities from treebank.
- Given a sentence, use CKY to recover to highest scoring tree.

Doesn't really work

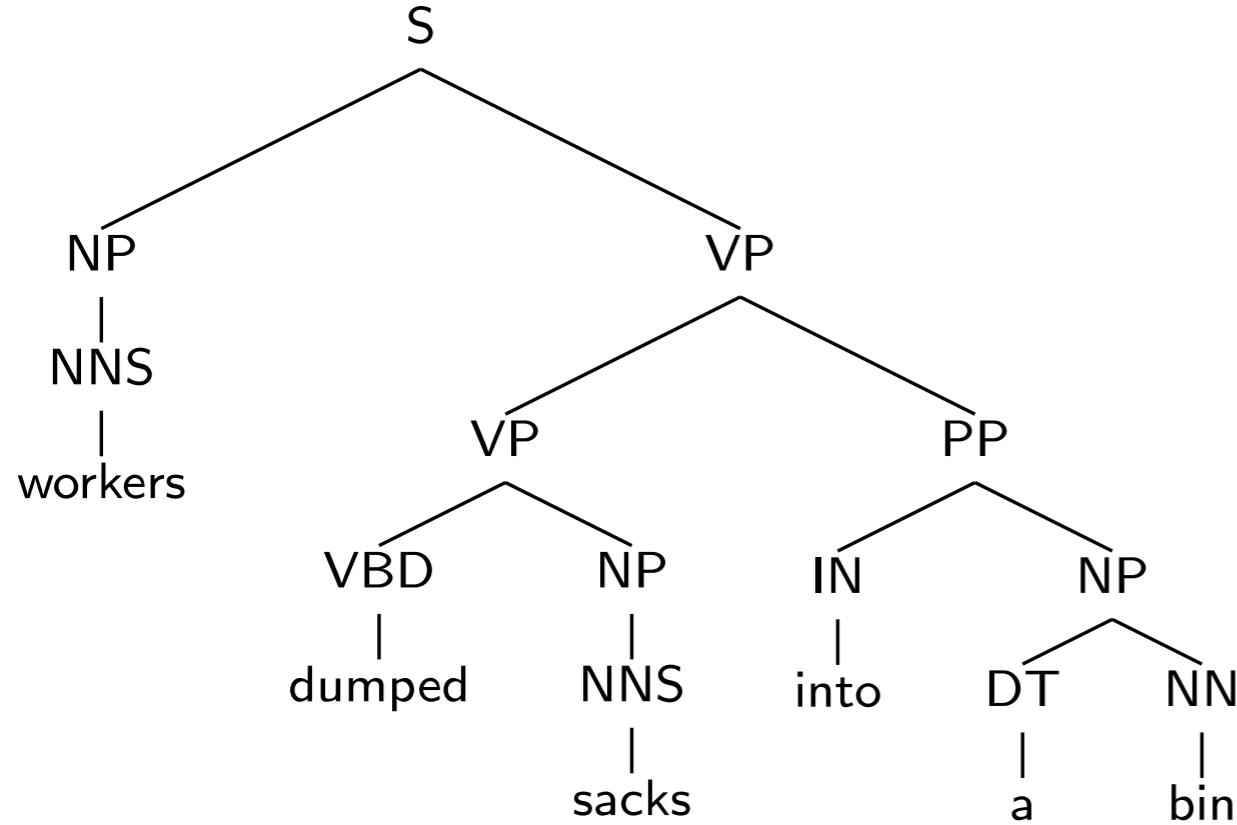
- It recovers the best tree according to the grammar.
- But these best trees are quite bad.
 - ~73 F1 score.

Some limitations of PCFGs

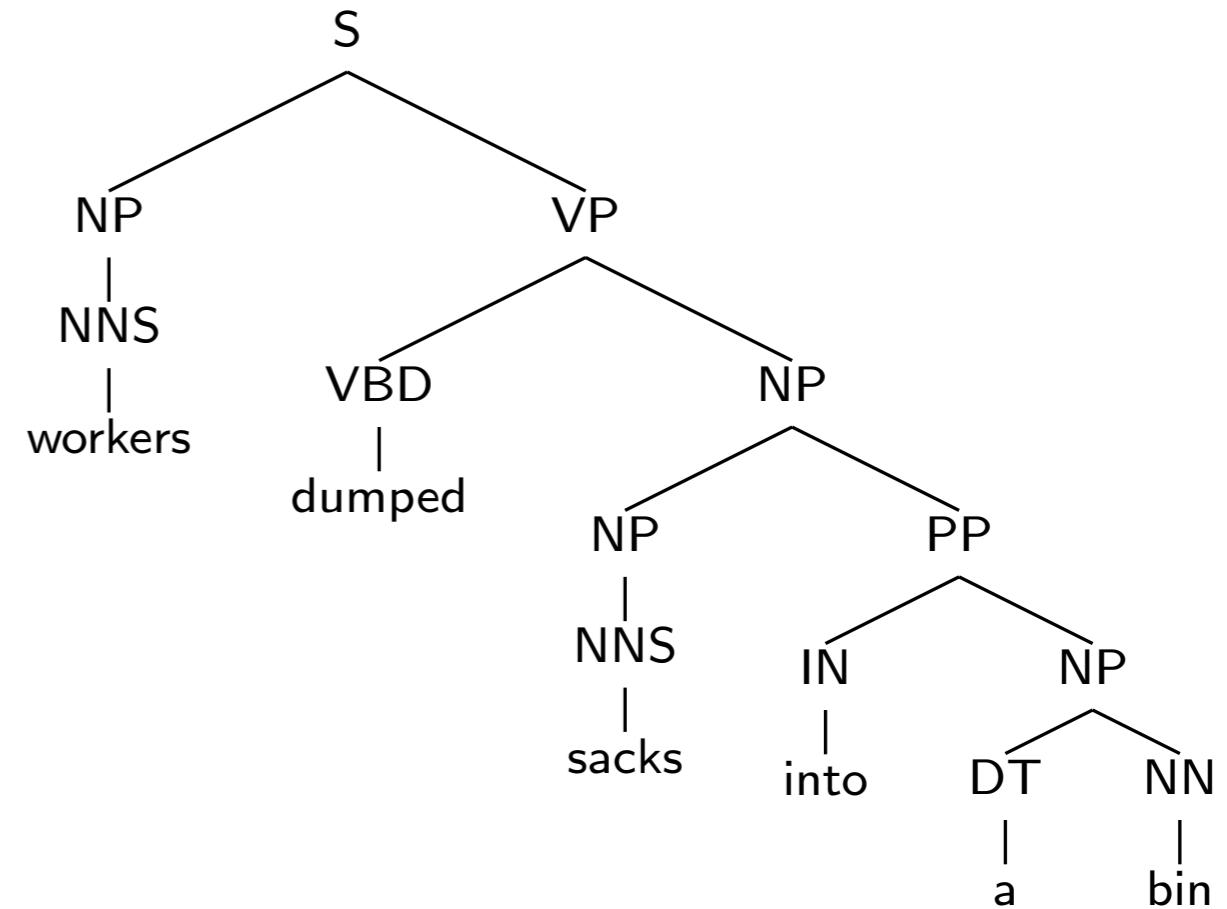
- Not sensitive to words.
- Not sensitive to structural frequencies.



(example from Mike Collins)

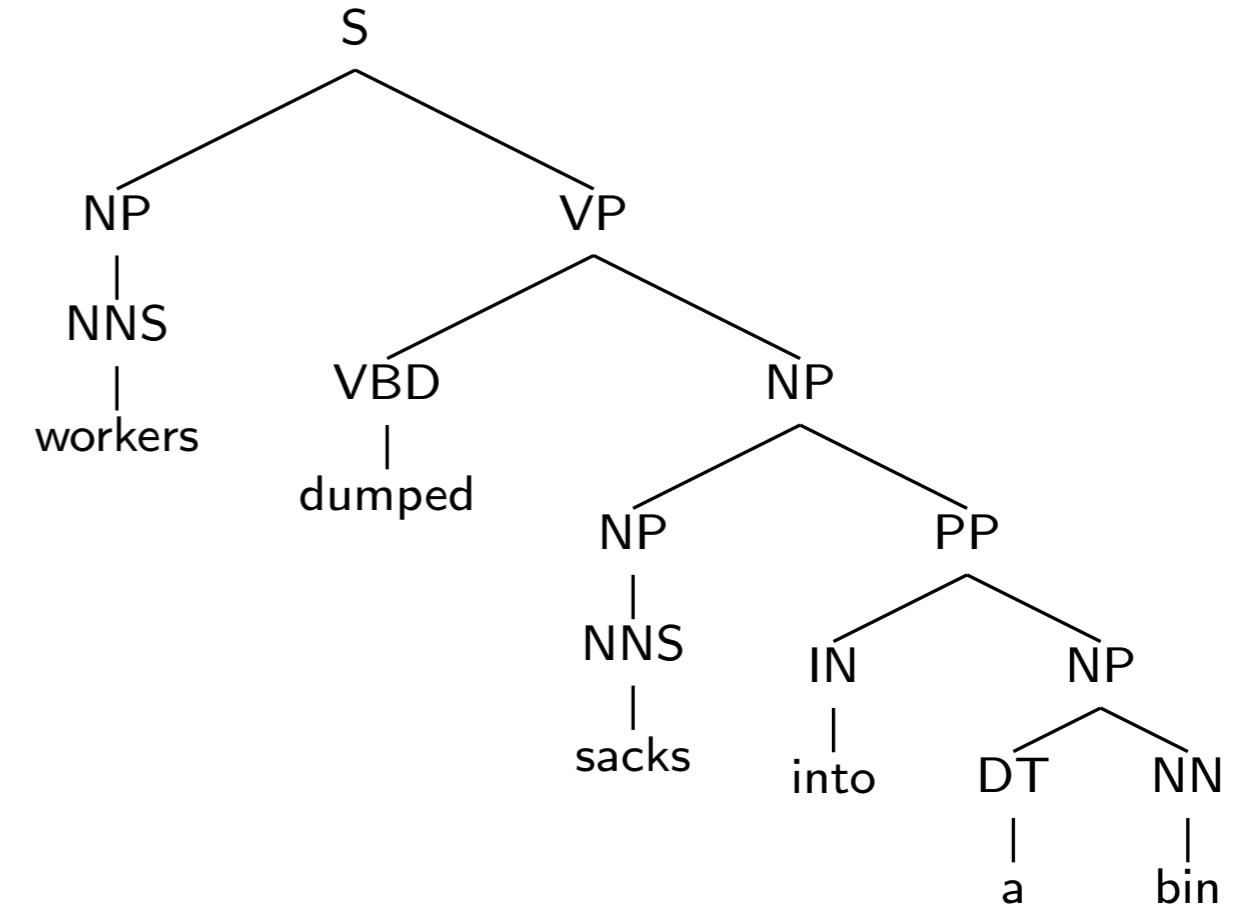
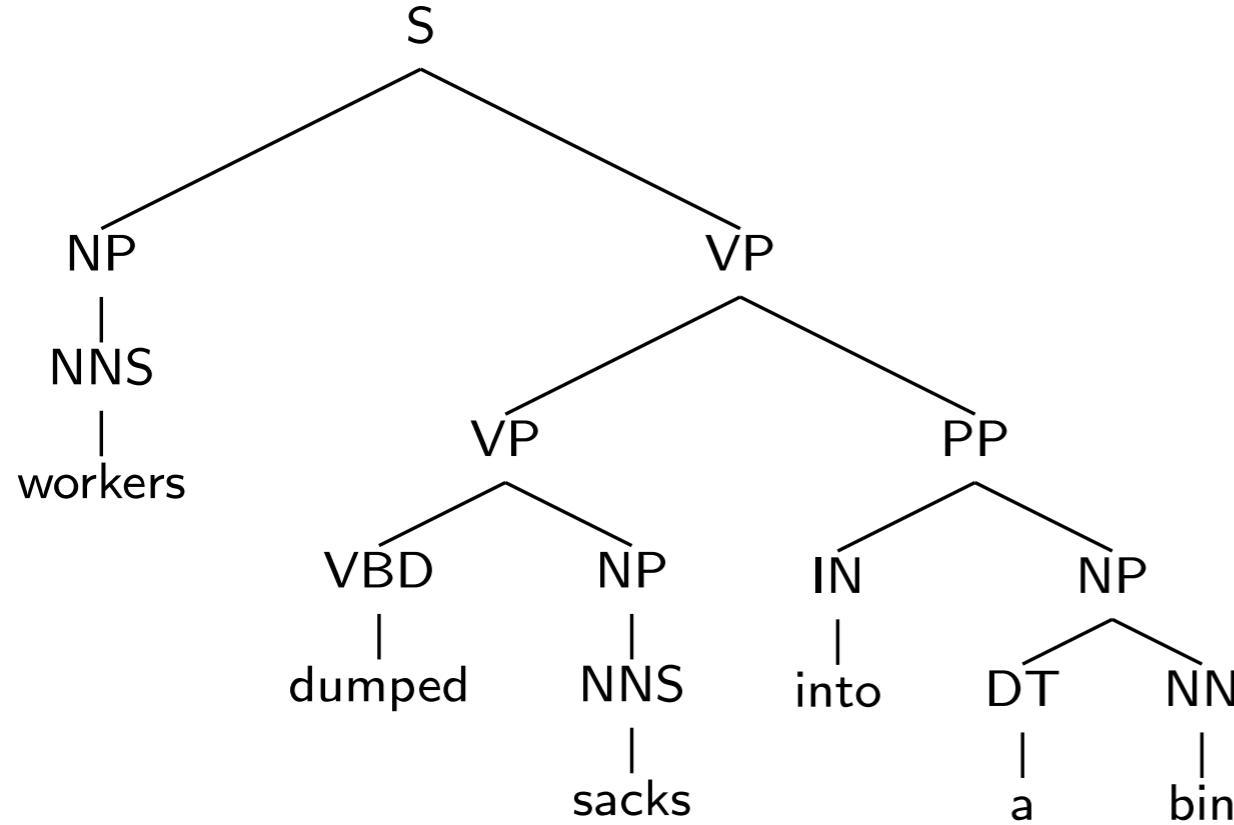


Rules
$S \rightarrow NP\ VP$
$NP \rightarrow NNS$
$VP \rightarrow VP\ PP$
$VP \rightarrow VBD\ NP$
$NP \rightarrow NNS$
$PP \rightarrow IN\ NP$
$NP \rightarrow DT\ NN$
$NNS \rightarrow workers$
$VBD \rightarrow dumped$
$NNS \rightarrow sacks$
$IN \rightarrow into$
$DT \rightarrow a$
$NN \rightarrow bin$



Rules
$S \rightarrow NP\ VP$
$NP \rightarrow NNS$
$NP \rightarrow NP\ PP$
$VP \rightarrow VBD\ NP$
$NP \rightarrow NNS$
$PP \rightarrow IN\ NP$
$NP \rightarrow DT\ NN$
$NNS \rightarrow workers$
$VBD \rightarrow dumped$
$NNS \rightarrow sacks$
$IN \rightarrow into$
$DT \rightarrow a$
$NN \rightarrow bin$

(example from Mike Collins)

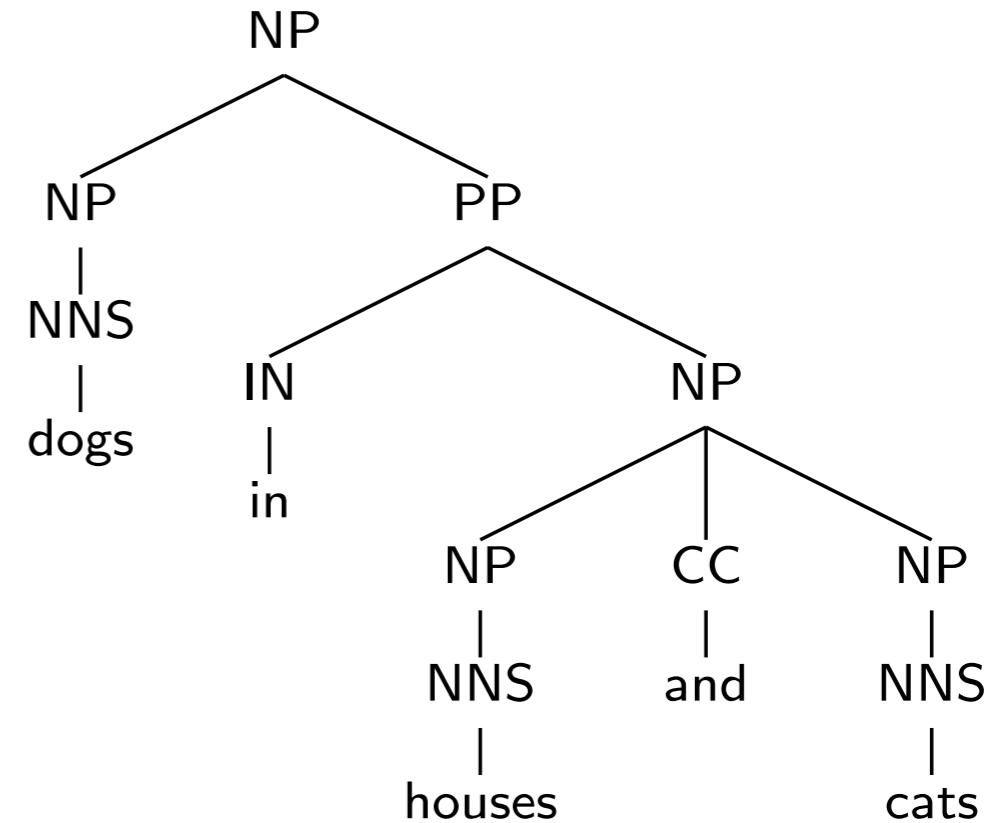
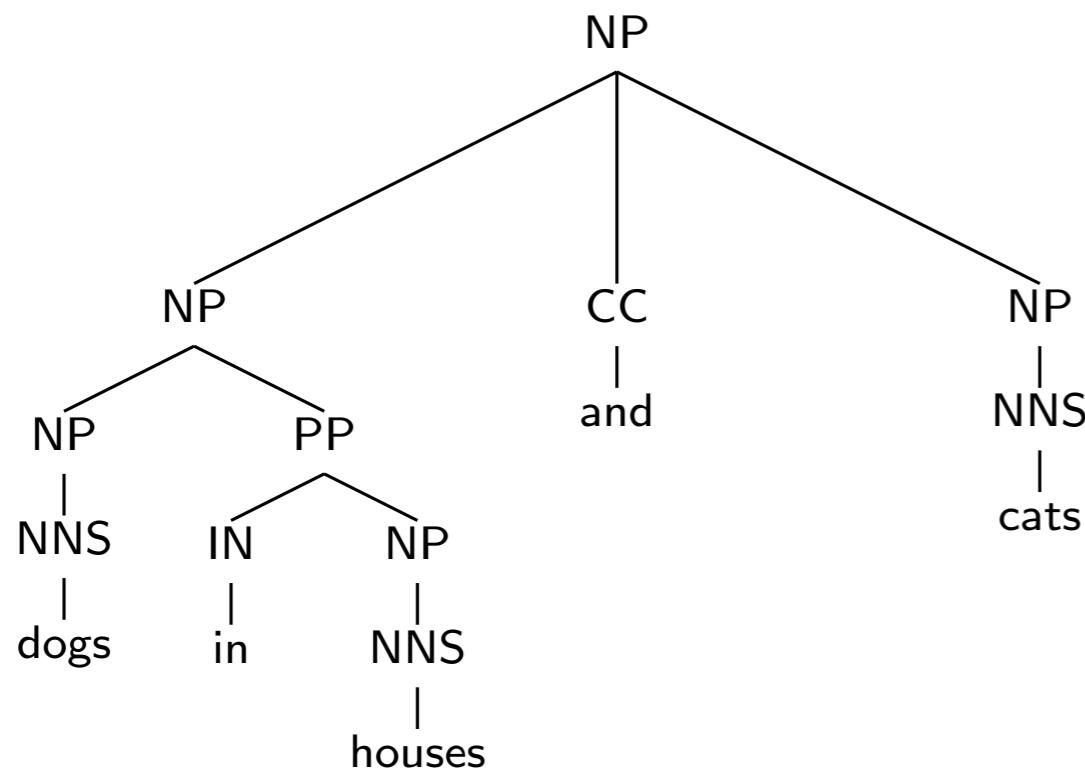


Rules
$S \rightarrow NP\ VP$
$NP \rightarrow NNS$
$VP \rightarrow VP\ PP$
$VP \rightarrow VBD\ NP$
$NP \rightarrow NNS$
$PP \rightarrow IN\ NP$
$NP \rightarrow DT\ NN$
$NNS \rightarrow workers$
$VBD \rightarrow dumped$
$NNS \rightarrow sacks$
$IN \rightarrow into$
$DT \rightarrow a$
$NN \rightarrow bin$

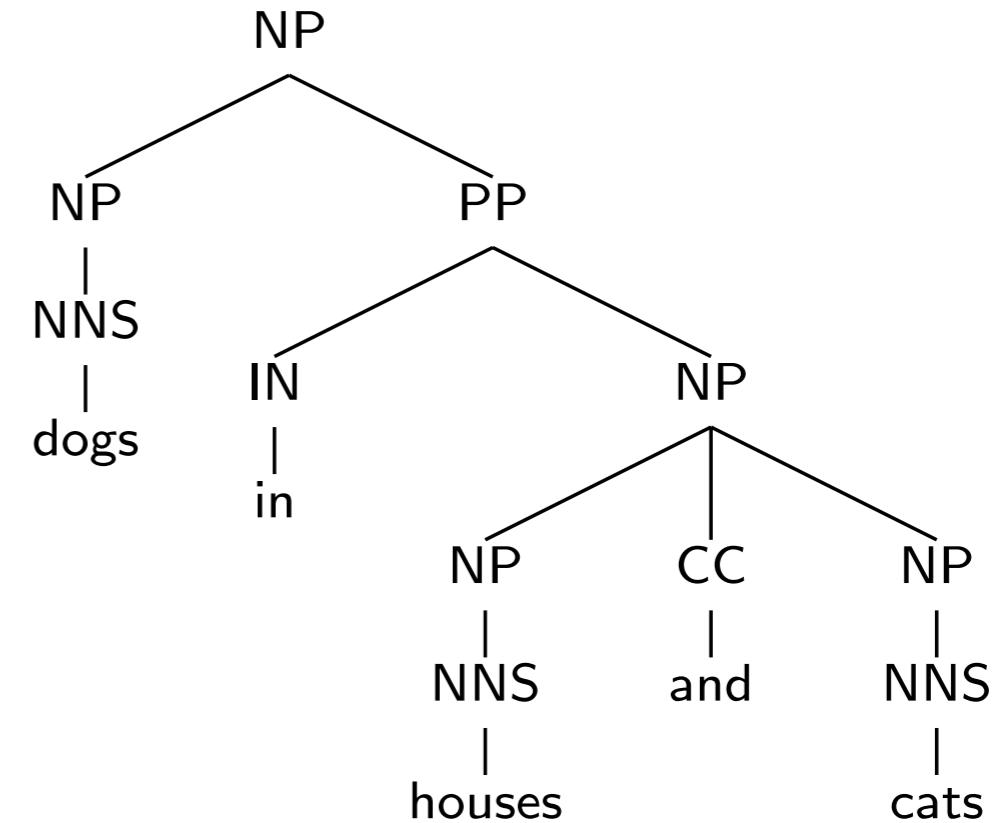
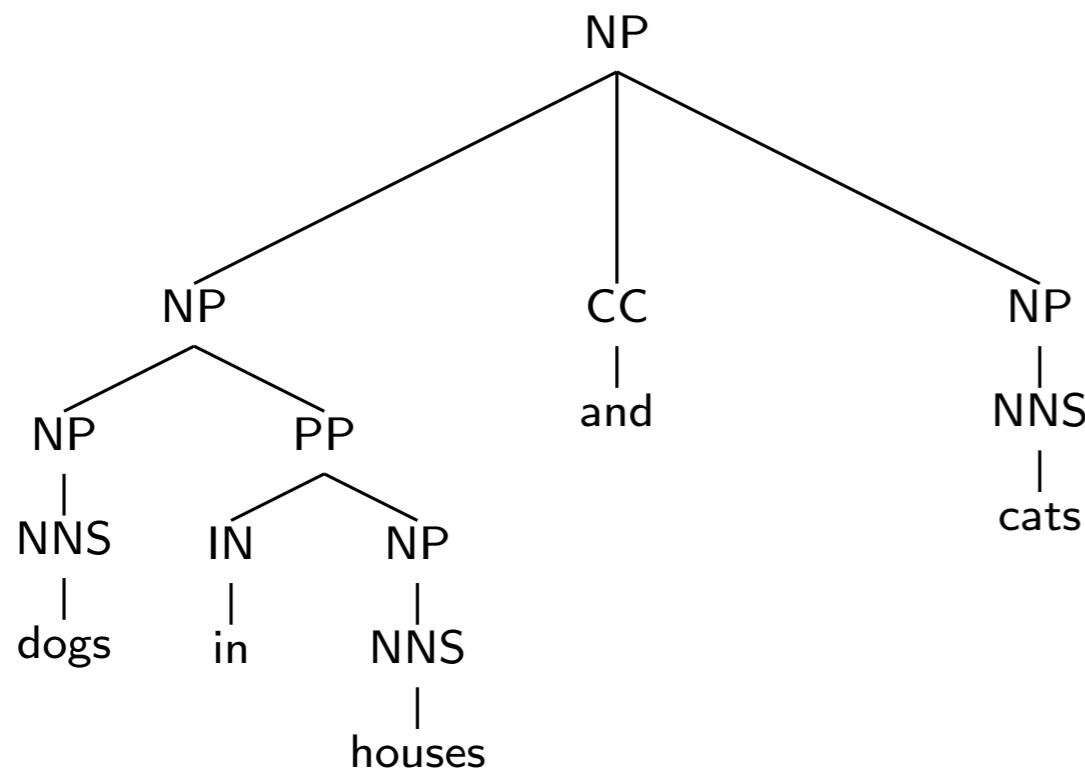
Rules
$S \rightarrow NP\ VP$
$NP \rightarrow NNS$
$NP \rightarrow NP\ PP$
$VP \rightarrow VBD\ NP$
$NP \rightarrow NNS$
$PP \rightarrow IN\ NP$
$NP \rightarrow DT\ NN$
$NNS \rightarrow workers$
$VBD \rightarrow dumped$
$NNS \rightarrow sacks$
$IN \rightarrow into$
$DT \rightarrow a$
$NN \rightarrow bin$

only difference
attachment decision
independent of words!

(example from Mike Collins)



(example from Mike Collins)



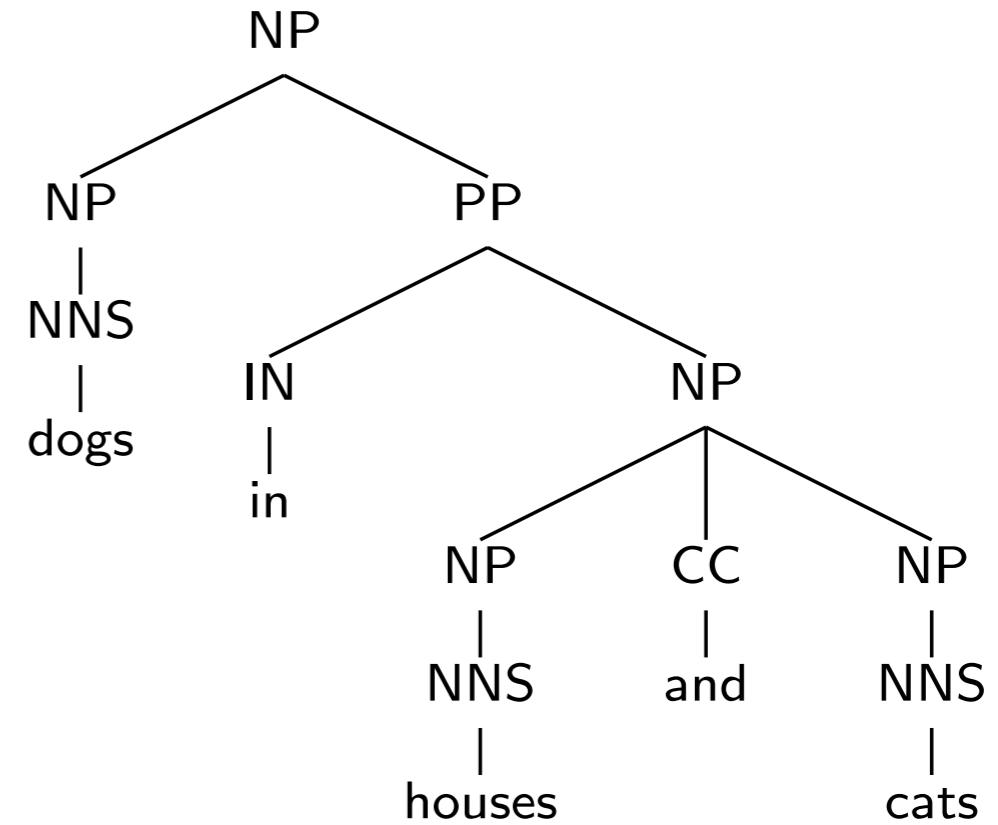
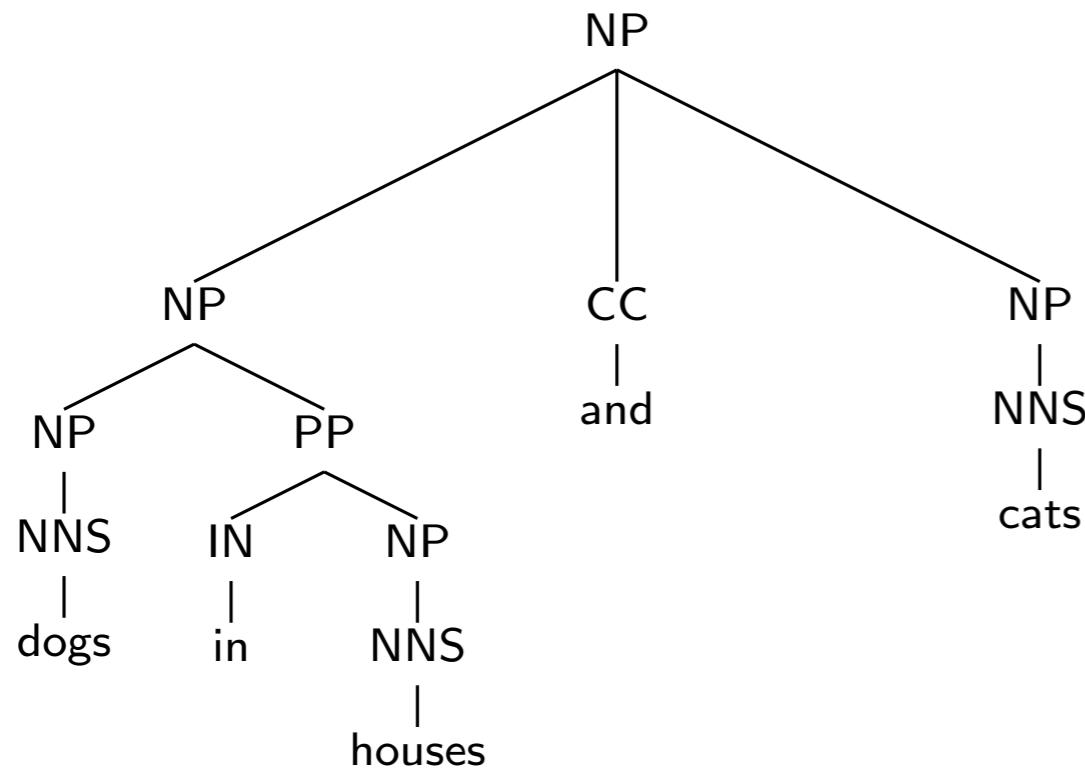
Rules

$NP \rightarrow NP\ CC\ NP$
 $NP \rightarrow NP\ PP$
 $NP \rightarrow NNS$
 $PP \rightarrow IN\ NP$
 $NP \rightarrow NNS$
 $NP \rightarrow NNS$
 $NNS \rightarrow dogs$
 $IN \rightarrow in$
 $NNS \rightarrow houses$
 $CC \rightarrow and$
 $NNS \rightarrow cats$

Rules

$NP \rightarrow NP\ CC\ NP$
 $NP \rightarrow NP\ PP$
 $NP \rightarrow NNS$
 $PP \rightarrow IN\ NP$
 $NP \rightarrow NNS$
 $NP \rightarrow NNS$
 $NNS \rightarrow dogs$
 $IN \rightarrow in$
 $NNS \rightarrow houses$
 $CC \rightarrow and$
 $NNS \rightarrow cats$

(example from Mike Collins)



Rules

$NP \rightarrow NP\ CC\ NP$
 $NP \rightarrow NP\ PP$
 $NP \rightarrow NNS$
 $PP \rightarrow IN\ NP$
 $NP \rightarrow NNS$
 $NP \rightarrow NNS$
 $NNS \rightarrow dogs$
 $IN \rightarrow in$
 $NNS \rightarrow houses$
 $CC \rightarrow and$
 $NNS \rightarrow cats$

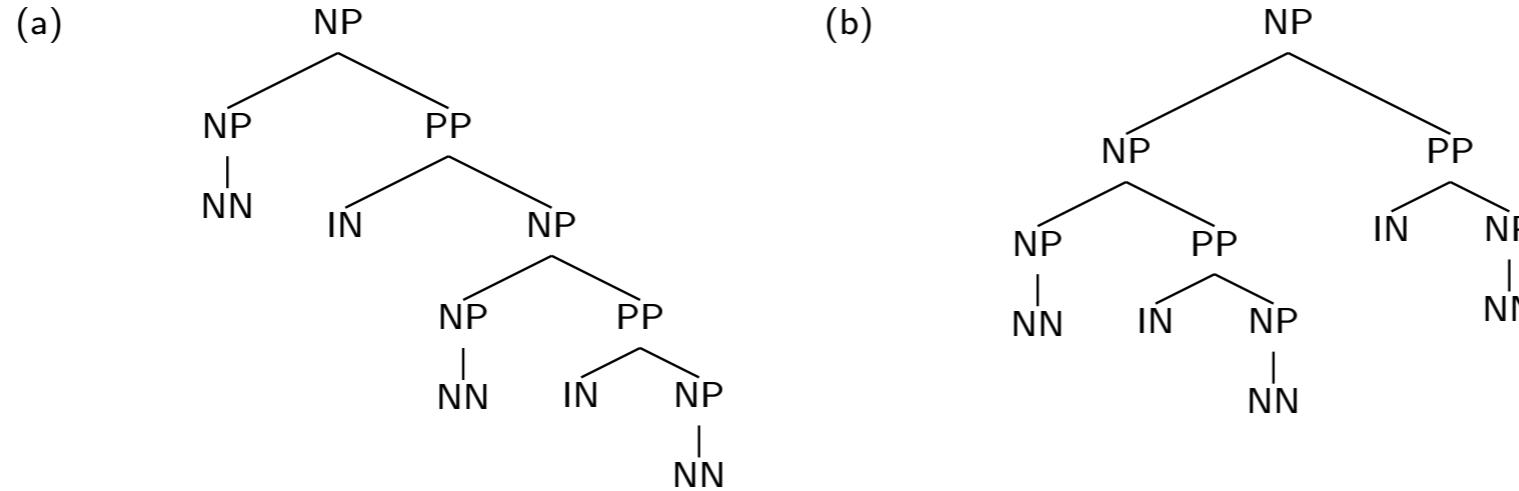
Identical set of rules.
Same score under any assignment of PCFG rule probabilities.

Rules

$NP \rightarrow NP\ CC\ NP$
 $NP \rightarrow NP\ PP$
 $NP \rightarrow NNS$
 $PP \rightarrow IN\ NP$
 $NP \rightarrow NNS$
 $NP \rightarrow NNS$
 $NNS \rightarrow dogs$
 $IN \rightarrow in$
 $NNS \rightarrow houses$
 $CC \rightarrow and$
 $NNS \rightarrow cats$

(example from Mike Collins)

Structural Preferences: Close Attachment



- ▶ Example: president of a company in Africa
- ▶ Both parses have the same rules, therefore receive same probability under a PCFG
- ▶ “Close attachment” (structure (a)) is twice as likely in Wall Street Journal text.

(slide from Mike Collins)

Lexicalized PCFGs

PCFG Problem 1

Lack of sensitivity to lexical information (words)

Solution

- ▶ Make PCFG aware of words (*lexicalized* PCFG)
- ▶ Main Idea: **Head Words**

Head Words

Each constituent has one words which captures its “essence”.

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John saw the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP saw the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large **hat**)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large **hat**)
- ▶ (PP with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large **hat**)
- ▶ (PP **with** the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large **hat**)
- ▶ (PP **with** the large **hat**)

Head Words

Each constituent has one word which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large **hat**)
- ▶ (PP **with** the large hat)
 - ▶ **hat** is the “semantic head”
 - ▶ **with** is the “functional head”
 - ▶ (it is common to choose the functional head)

More about Heads

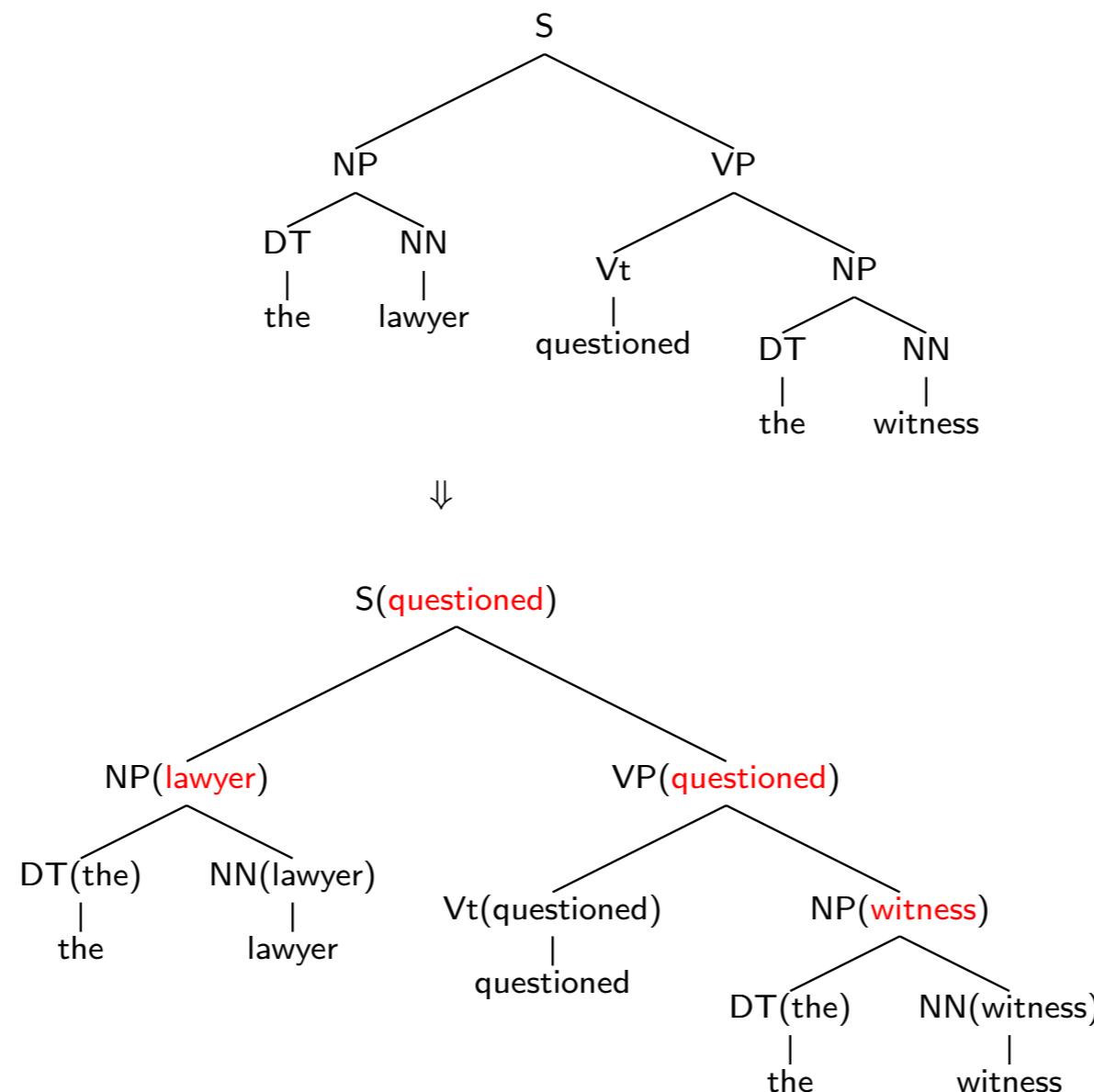
- ▶ Each context-free rule has one “special” child that is the head of the rule. e.g.,

$$\begin{array}{lll} S & \Rightarrow & NP \quad VP \\ & & \quad \quad \quad VP \text{ is the head} \\ VP & \Rightarrow & Vt \quad NP \\ & & \quad \quad \quad Vt \text{ is the head} \\ NP & \Rightarrow & DT \quad NN \quad NN \\ & & \quad \quad \quad NN \text{ is the head} \end{array}$$

- ▶ A core idea in syntax
(e.g., see X-bar Theory, Head-Driven Phrase Structure Grammar)
- ▶ Some intuitions:
 - ▶ The central sub-constituent of each rule.
 - ▶ The semantic predicate in each rule.

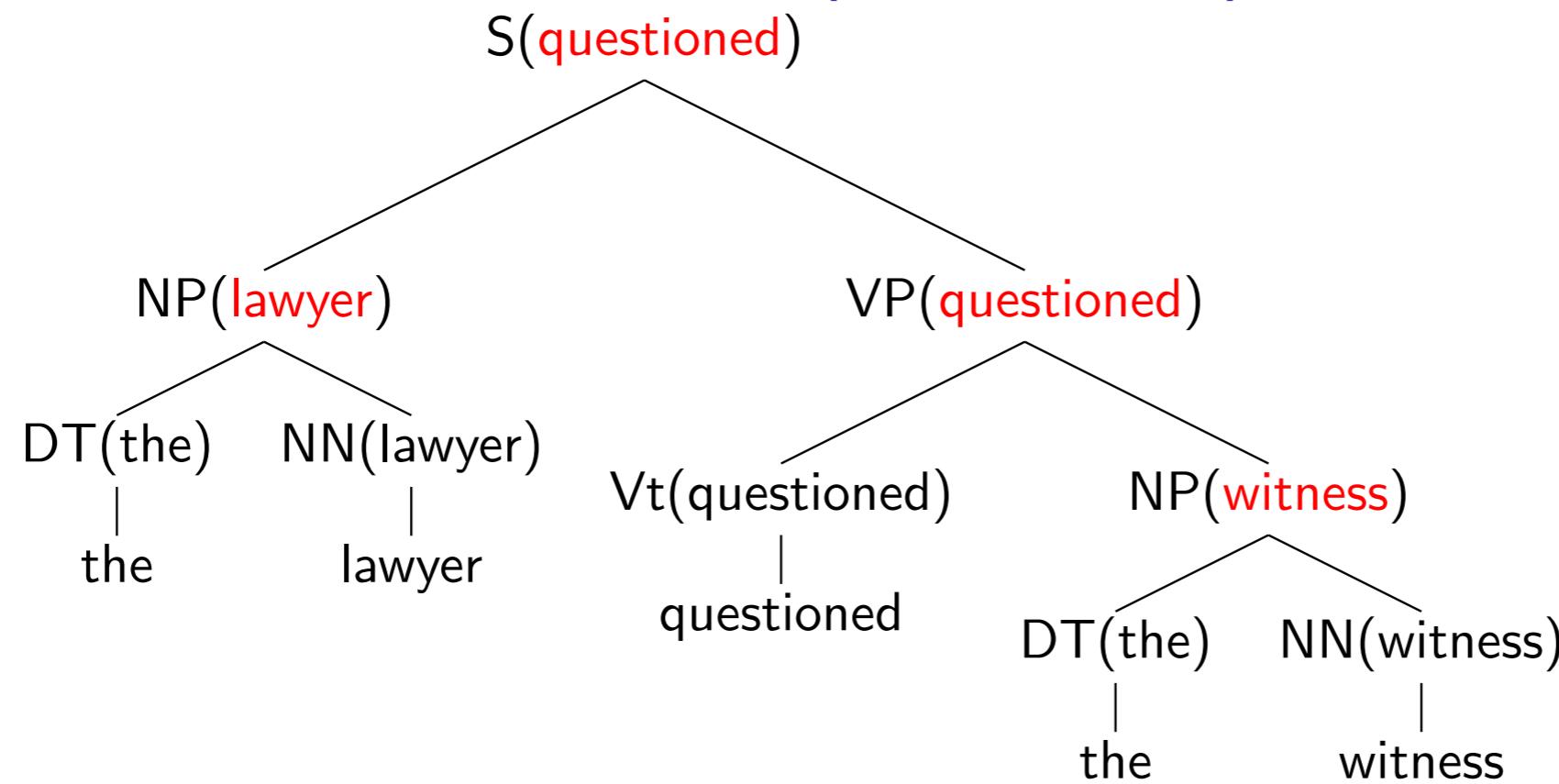
(slide from Mike Collins)

Adding Headwords to Trees



(slide from Mike Collins)

Adding Headwords to Trees (Continued)

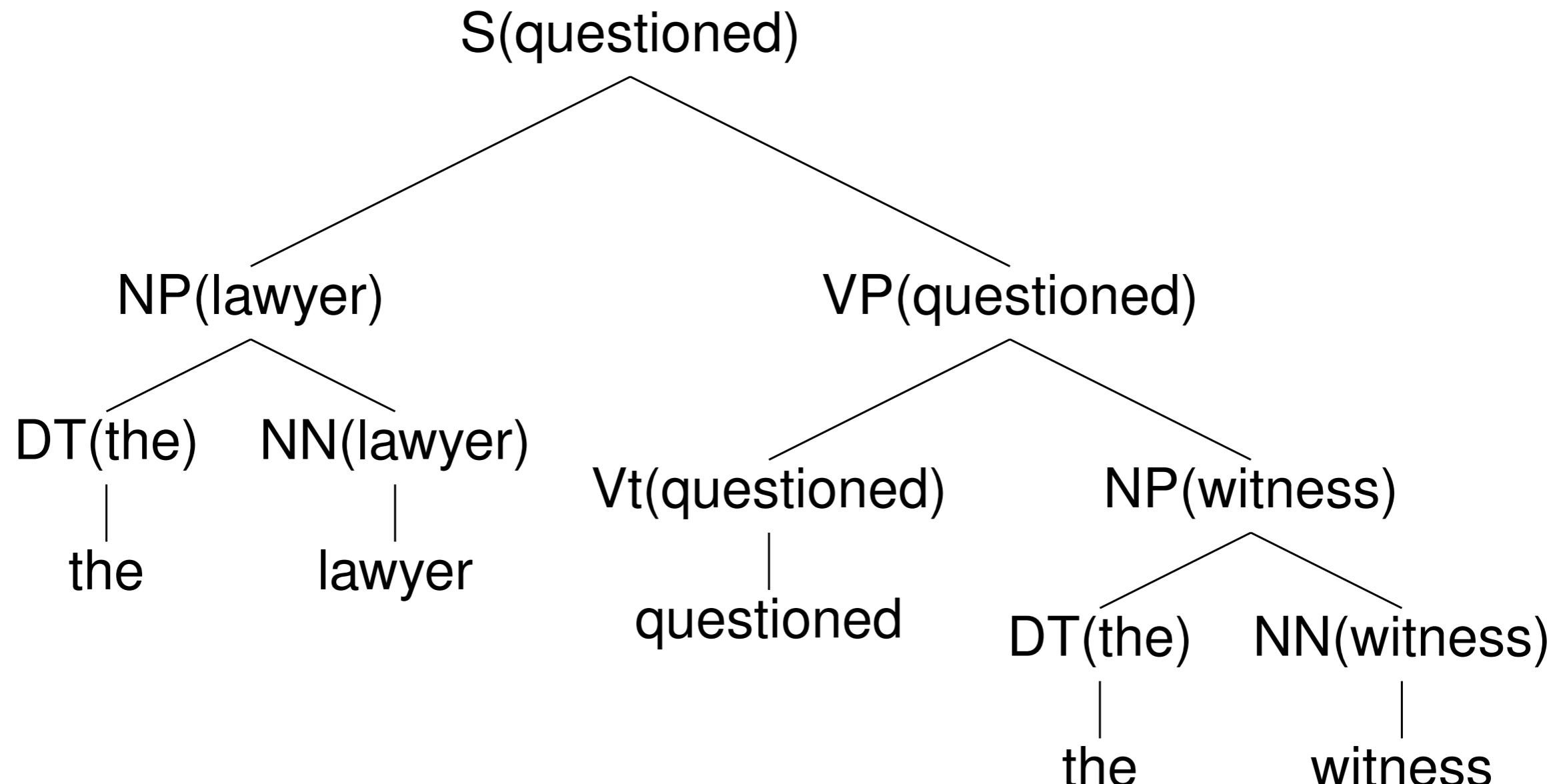


- ▶ A constituent receives its **headword** from its **head child**.

$$\begin{array}{lll} S & \Rightarrow & NP \quad VP \\ VP & \Rightarrow & Vt \quad NP \\ NP & \Rightarrow & DT \qquad \quad NN \end{array}$$

(S receives headword from VP)
(VP receives headword from Vt)
(NP receives headword from NN)

(slide from Mike Collins)



Chomsky Normal Form

A context free grammar $G = (N, \Sigma, R, S)$ in Chomsky Normal Form is as follows

- ▶ N is a set of non-terminal symbols
- ▶ Σ is a set of terminal symbols
- ▶ R is a set of rules which take one of two forms:
 - ▶ $X \rightarrow Y_1 Y_2$ for $X \in N$, and $Y_1, Y_2 \in N$
 - ▶ $X \rightarrow Y$ for $X \in N$, and $Y \in \Sigma$
- ▶ $S \in N$ is a distinguished start symbol

We can find the highest scoring parse under a PCFG in this form, in $O(n^3|N|^3)$ time where n is the length of the string being parsed.

(slide from Mike Collins)

Lexicalized Context-Free Grammars in Chomsky Normal Form

- ▶ N is a set of non-terminal symbols
- ▶ Σ is a set of terminal symbols
- ▶ R is a set of rules which take one of three forms:
 - ▶ $X(h) \rightarrow_1 Y_1(h) Y_2(w)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$
 - ▶ $X(h) \rightarrow_2 Y_1(w) Y_2(h)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$
 - ▶ $X(h) \rightarrow h$ for $X \in N$, and $h \in \Sigma$
- ▶ $S \in N$ is a distinguished start symbol

(slide from Mike Collins)

An Example

$S(saw)$	\rightarrow_2	$NP(man)$	$VP(saw)$
$VP(saw)$	\rightarrow_1	$Vt(saw)$	$NP(dog)$
$NP(man)$	\rightarrow_2	$DT(the)$	$NN(man)$
$NP(dog)$	\rightarrow_2	$DT(the)$	$NN(dog)$
$Vt(saw)$	\rightarrow	saw	
$DT(the)$	\rightarrow	the	
$NN(man)$	\rightarrow	man	
$NN(dog)$	\rightarrow	dog	

(slide from Mike Collins)

Parsing with Lexicalized CFGs

- ▶ The new form of grammar looks just like a Chomsky normal form CFG, but with potentially $O(|\Sigma|^2 \times |N|^3)$ possible rules.
- ▶ Naively, parsing an n word sentence using the dynamic programming algorithm will take $O(n^3|\Sigma|^2|N|^3)$ time. **But** $|\Sigma|$ can be huge!!
- ▶ Crucial observation: at most $O(n^2 \times |N|^3)$ rules can be applicable to a given sentence w_1, w_2, \dots, w_n of length n . This is because any rules which contain a lexical item that is not one of $w_1 \dots w_n$, can be safely discarded.
- ▶ The result: we can parse in $O(n^5|N|^3)$ time.

(slide from Mike Collins)

Accurate Unlexicalized Parsing

Accurate Unlexicalized Parsing

PCFG Problem 2

Lack of sensitivity to structural information

Accurate Unlexicalized Parsing

PCFG Problem 2

Lack of sensitivity to structural information

Solution

- ▶ This problem is also solved by lexicalization.
 - ▶ (maybe that's the main problem that's being solved by lexicalization)

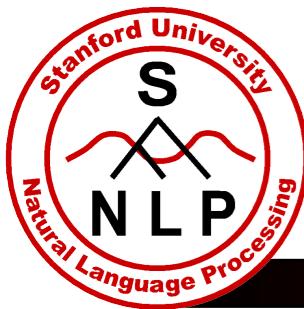
Accurate Unlexicalized Parsing

PCFG Problem 2

Lack of sensitivity to structural information

Solution

- ▶ This problem is also solved by lexicalization.
 - ▶ (maybe that's the main problem that's being solved by lexicalization)
- ▶ But can we do without lexicalizing the grammar?

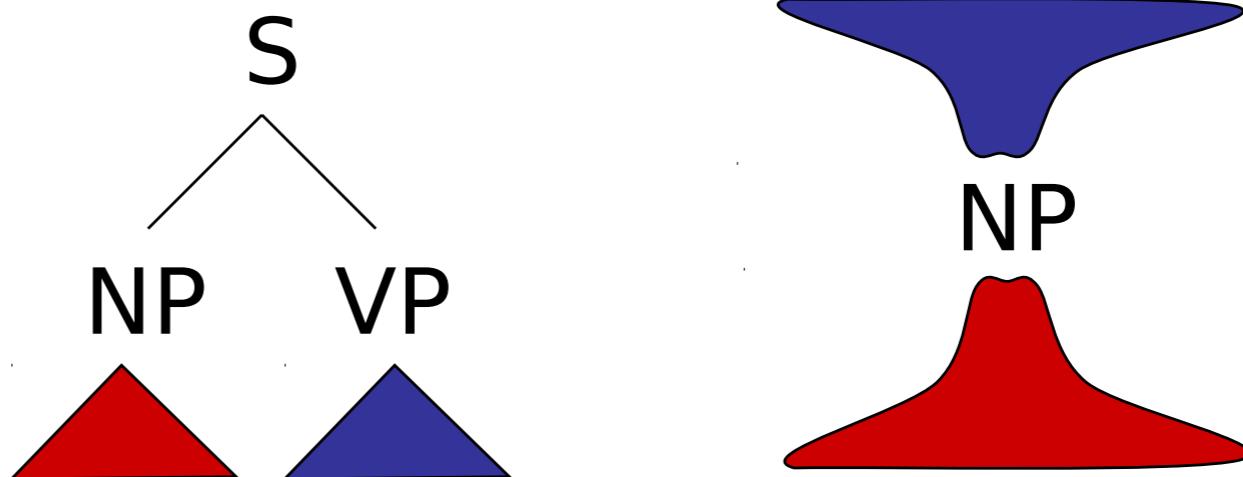


5. Accurate Unlexicalized Parsing: PCFGs and Independence

- The symbols in a PCFG define independence assumptions:

$S \rightarrow NP\ VP$

$NP \rightarrow DT\ NN$



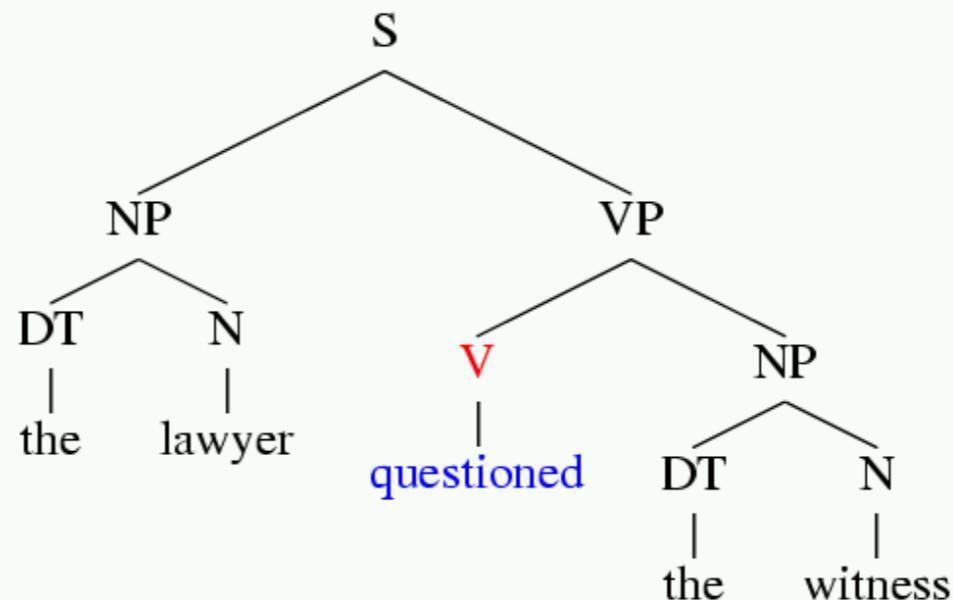
- At any node, the material inside that node is independent of the material outside that node, given the label of that node.
- Any information that statistically connects behavior inside and outside a node must flow through that node.



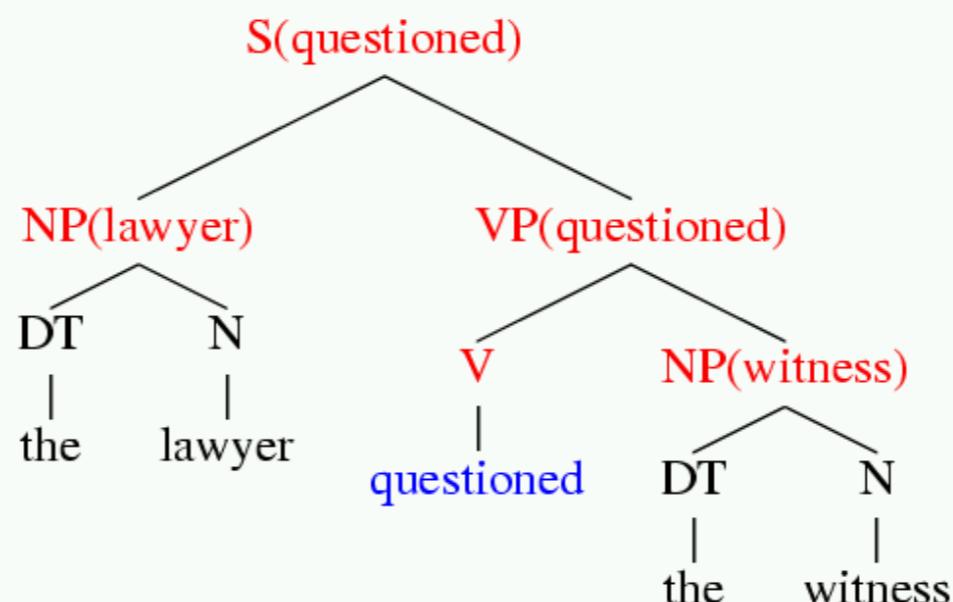
Michael Collins (2003, COLT)

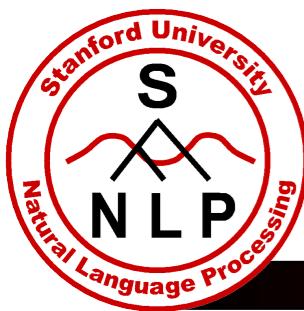
Independence Assumptions

- PCFGs



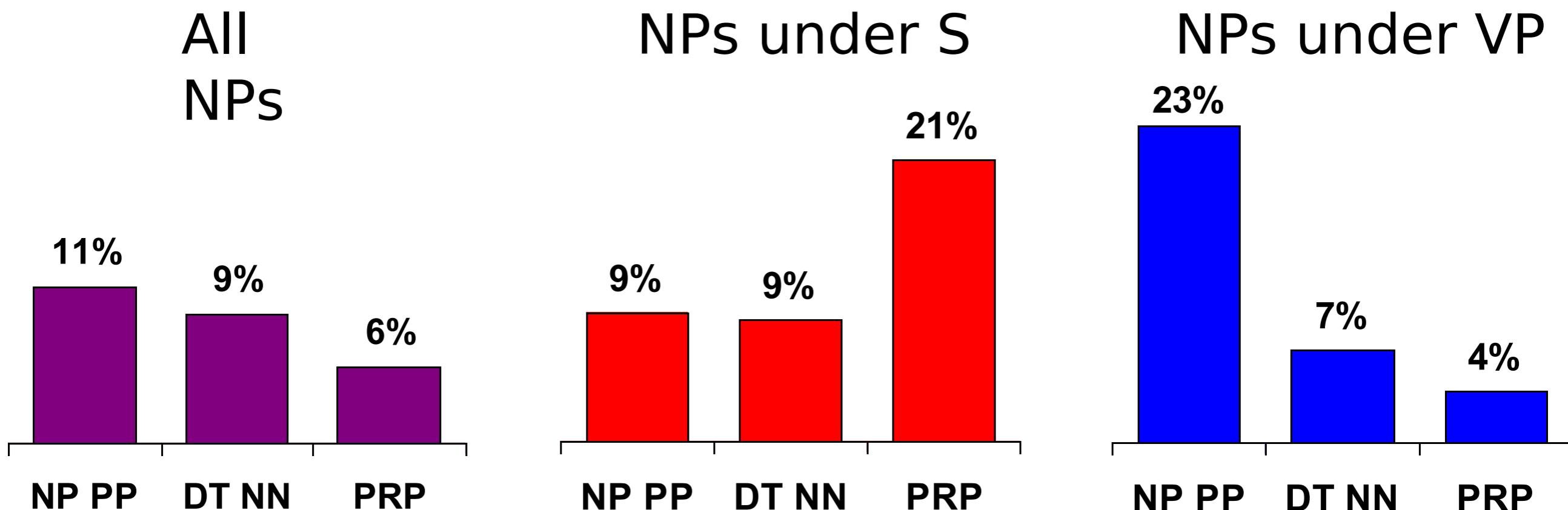
- Lexicalized PCFGs



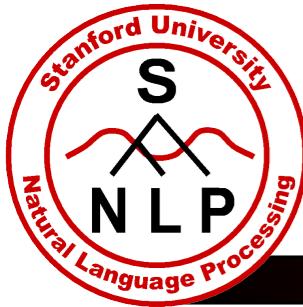


Non-Independence I

- Independence assumptions are often too strong.



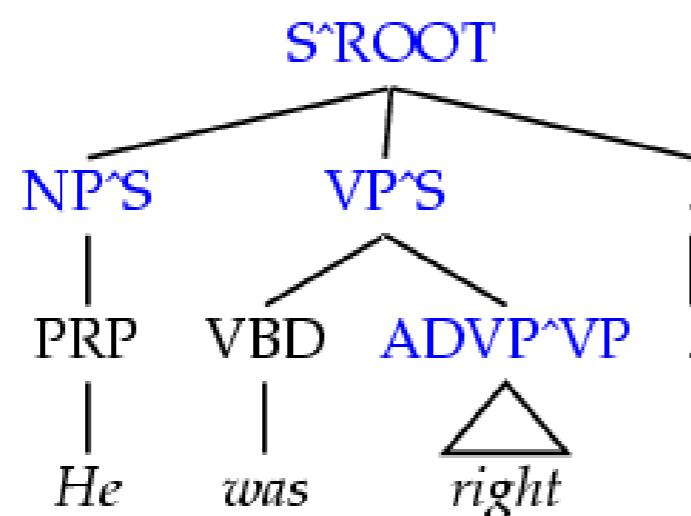
- Example: the expansion of an NP is highly dependent on the parent of the NP (i.e., subjects vs. objects).



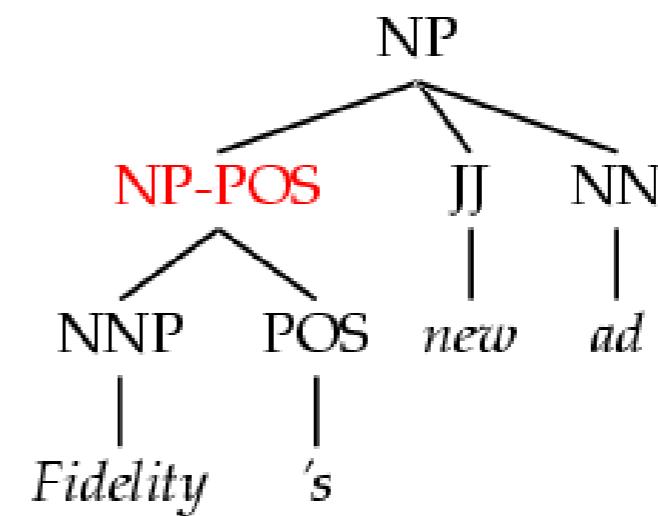
Breaking Up the Symbols

- We can relax independence assumptions by encoding dependencies into the PCFG symbols:

Parent annotation
[Johnson 98]



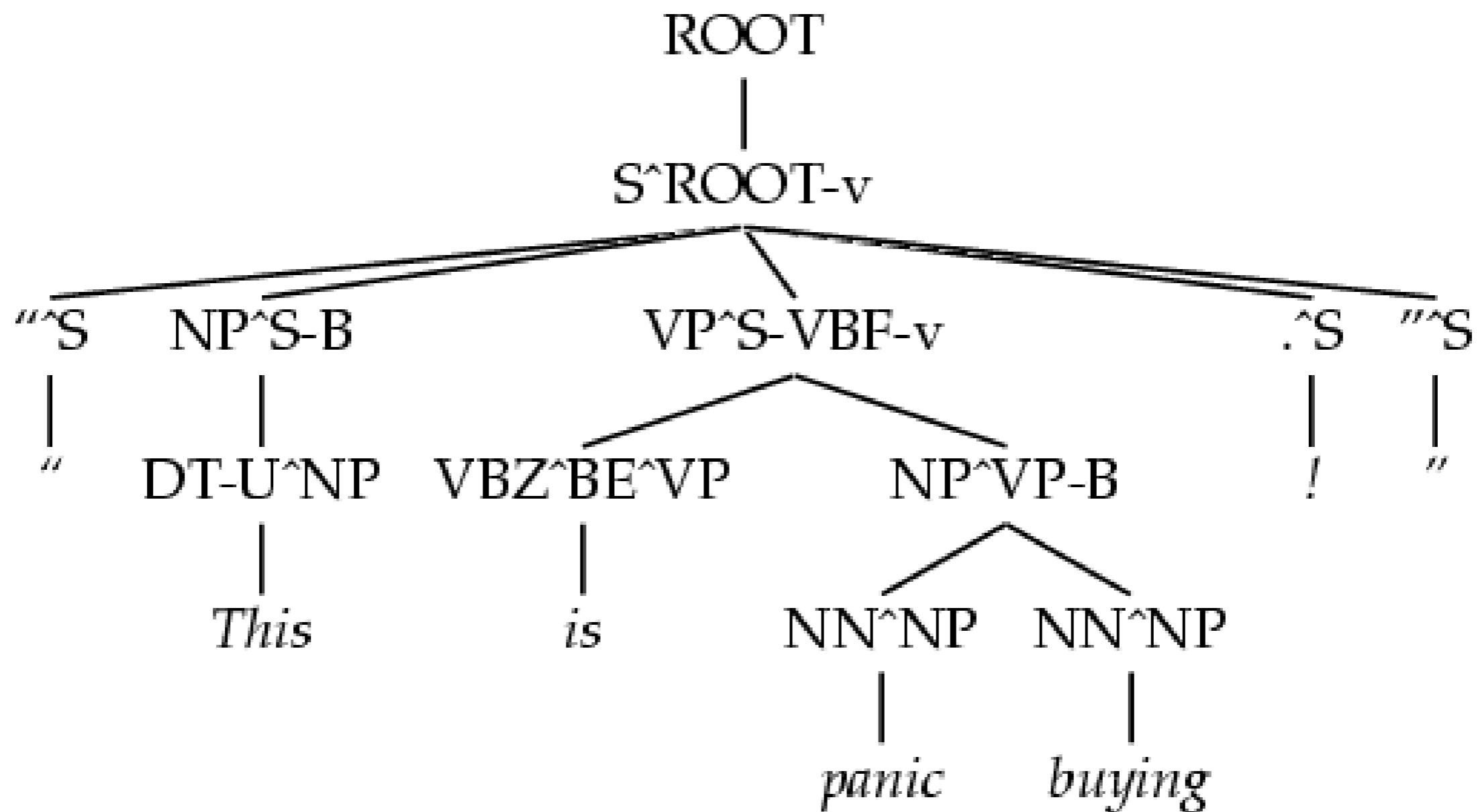
Marking
possessive NPs



- What are the most useful features to encode?



A Fully Annotated Tree





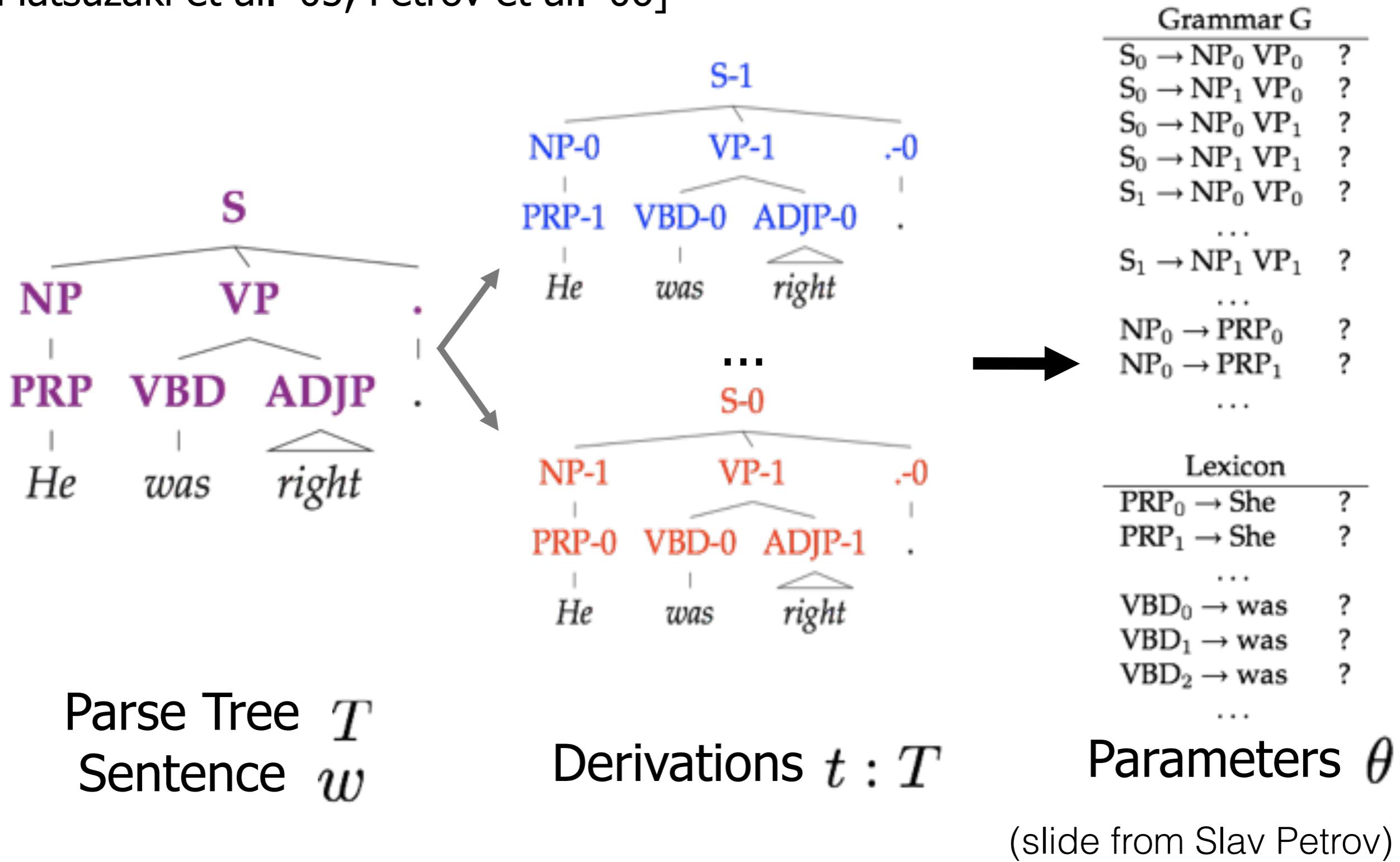
Final Test Set Results

Parser	LP	LR	F1	CB	0 CB
Magerman 95	84.9	84.6	84.7	1.26	56.6
Collins 96	86.3	85.8	86.0	1.14	59.9
Klein & M 03	86.9	85.7	86.3	1.10	60.3
Charniak 97	87.4	87.5	87.4	1.00	62.1
Collins 99	88.7	88.6	88.6	0.90	67.1

- Beats “first generation” lexicalized parsers.

Automatic Annotation (latent variable grammars)

[Matsuzaki et al. '05, Petrov et al. '06]



Constituency Parsing

- Decompose tree into parts (grammar rules)
- Assign score to each rule
- Find best scoring tree using CKY
- Maybe do some tricks for better scoring of rules

Dependency Parsing

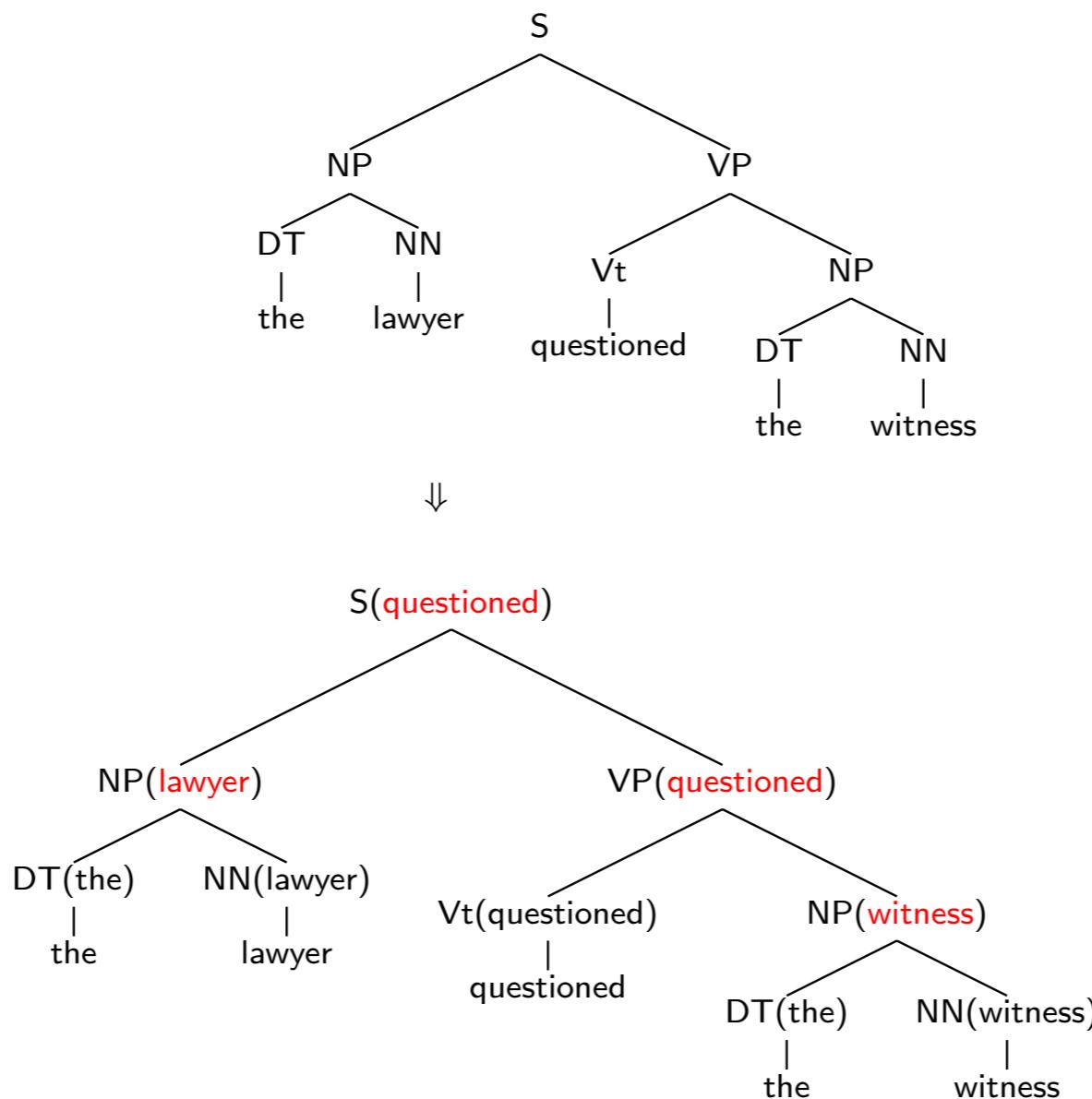
Dependency Representation

Head Words

Each constituent has one word which captures its “essence”.

- ▶ (S John **saw** the young boy with the large hat)
- ▶ (VP **saw** the young boy with the large hat)
- ▶ (NP the young **boy** with the large hat)
- ▶ (NP the large **hat**)
- ▶ (PP **with** the large hat)
 - ▶ **hat** is the “semantic head”
 - ▶ **with** is the “functional head”
 - ▶ (it is common to choose the functional head)

Adding Headwords to Trees

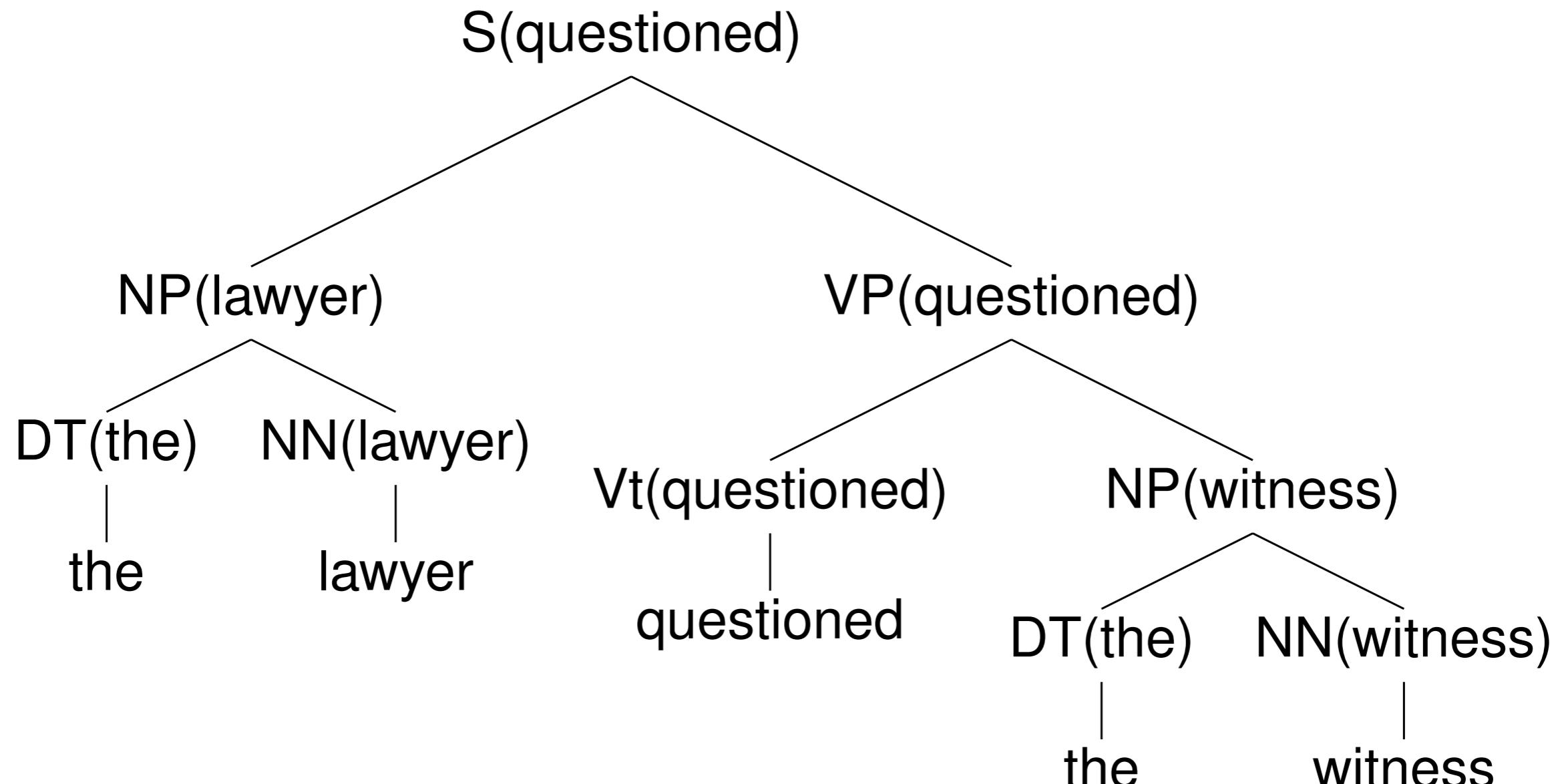


Dependency Representation

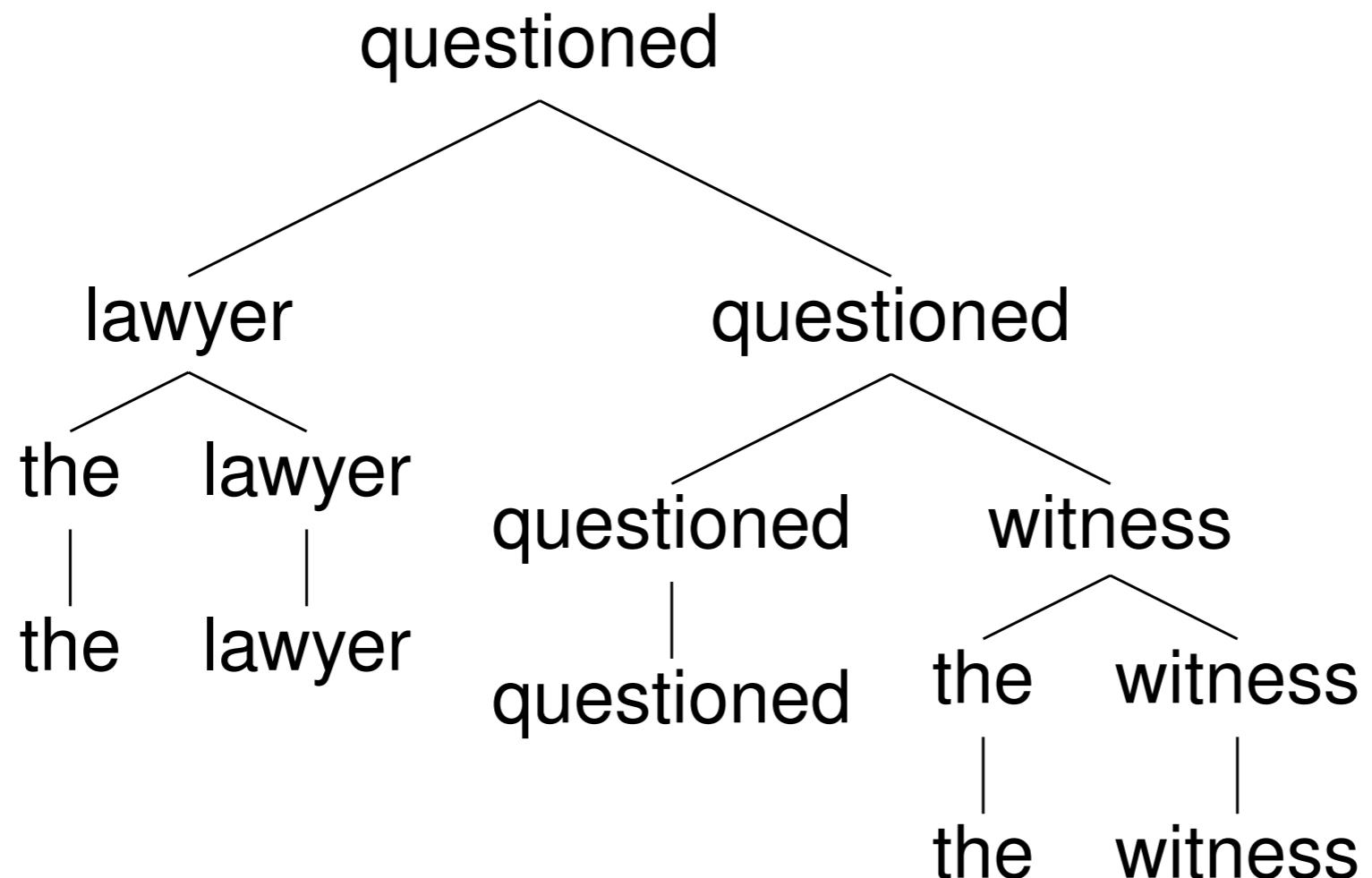
If we take the head-annotated trees and “forget” about the constituents, we get a representation called “dependency structure”.

Dependency structure capture the relation between words in a sentence.

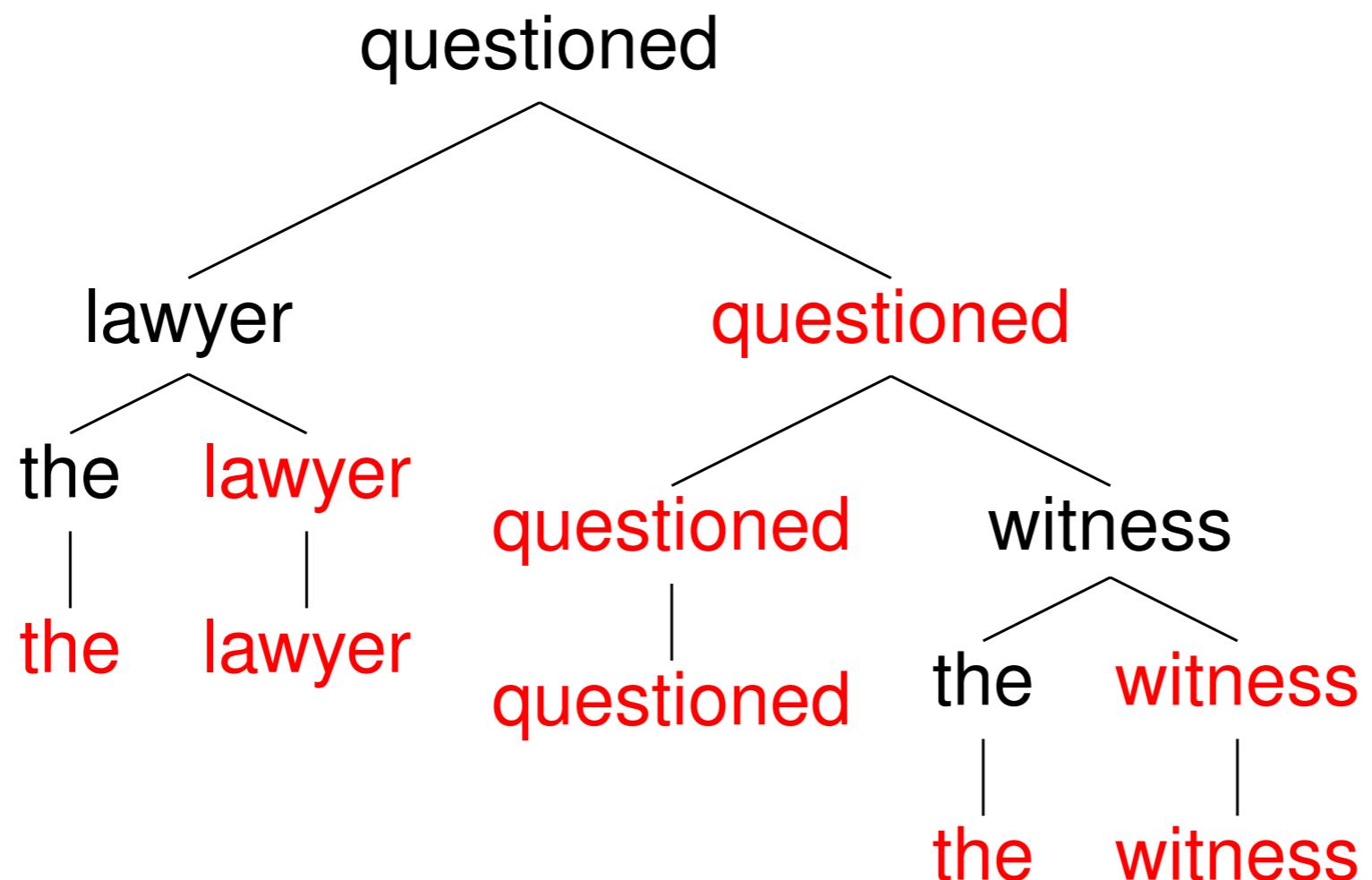
Dependency Representation



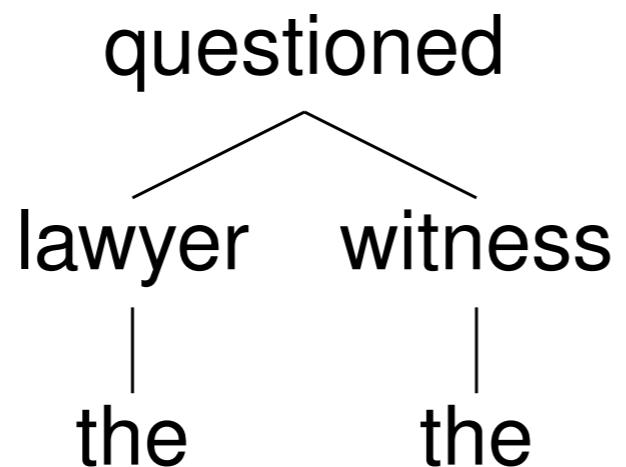
Dependency Representation



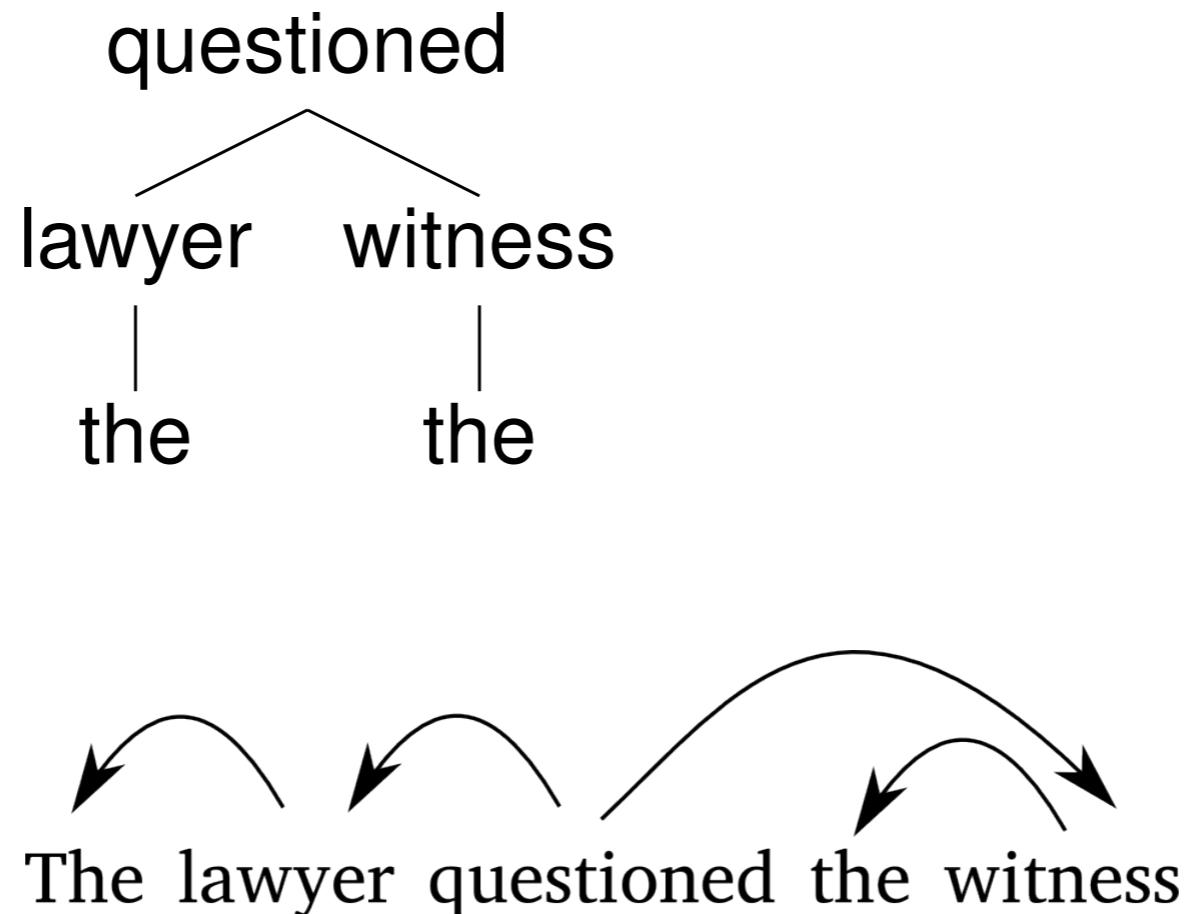
Dependency Representation



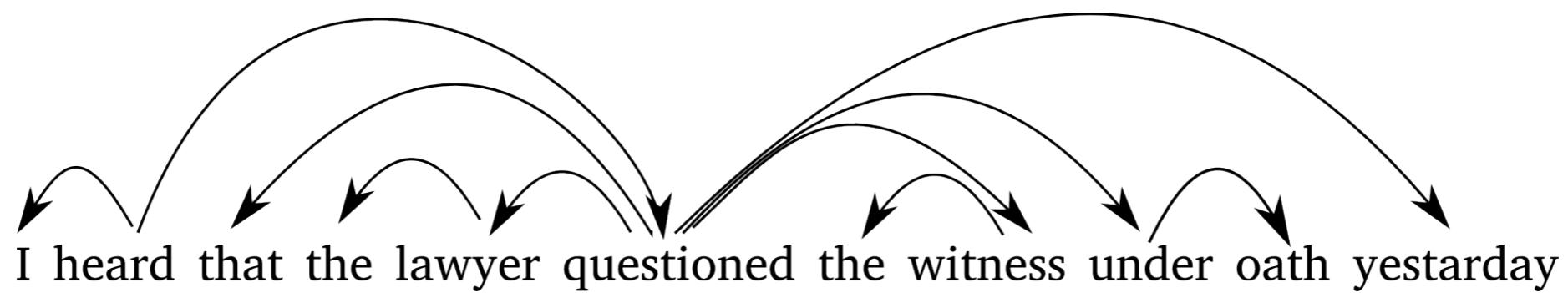
Dependency Representation



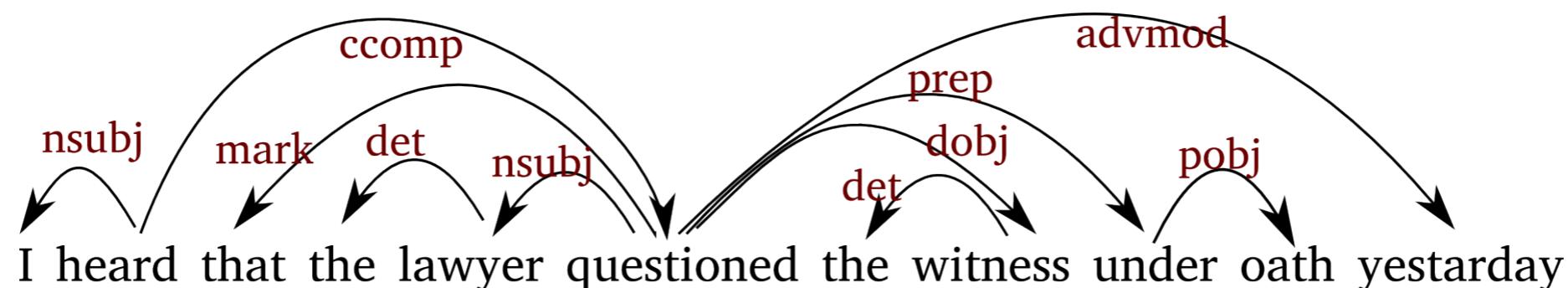
Dependency Representation



Dependency Representation

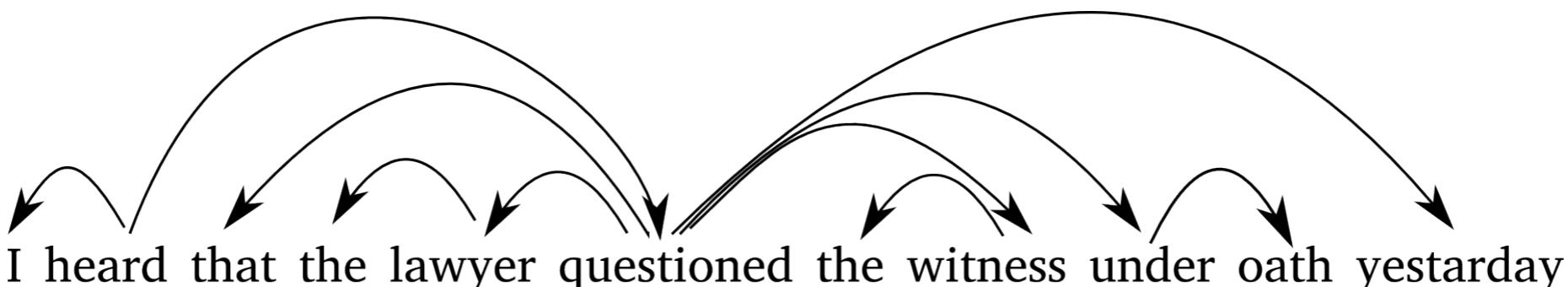


Dependency Representation



Dependency Representation Projectivity

Projective Tree (no crossing arcs):



Non-Projective Tree (crossing arcs):



Dependency Representations

There are many different dependency representations

- ▶ Different choice of heads.
- ▶ Different set of labels.
- ▶ Each language usually has its own treebank, with own choices

Universal Dependencies

- ▶ A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.

Universal Dependencies

- ▶ A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.
- ▶ Abstract over linguistic differences.
- ▶ Same set of parts-of-speech and morphology features.
- ▶ Same dependency relations.
- ▶ Same choice of heads.

Three main approaches to Dependency Parsing

Conversion

- ▶ Parse to constituency structure.
- ▶ Extract dependencies from the trees.

Global Optimization (Graph based)

- ▶ **Define** a scoring function over `<sentence,tree>` pairs.
- ▶ **Search** for best-scoring structure.
- ▶ Simpler scoring ⇒ easier search.
- ▶ (Similar to how we do tagging, constituency parsing.)

Greedy decoding (Transition based)

- ▶ Start with an unparsed sentence.
- ▶ Apply **locally-optimal** actions until sentence is parsed.

Graph-based Parsing

The Parsing Problem

$$\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x)$$

search over all possible parses
and find the one with the highest score

The Parsing Problem

$$\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x)$$

search over all possible parses
and find the one with the highest score

Training objective

$$\text{score}(y, x) > \text{score}(y', x) \quad \forall y \neq y'$$

The Parsing Problem

$$\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x)$$

search over all possible parses
and find the one with the highest score

Challenge: very hard search problem

The Parsing Problem

$$\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x)$$

search over all possible parses
and find the one with the highest score

Challenge: very hard search problem

Solution: decompose score:

$$\text{score}(y, x) = \sum_{p \in y} \text{score}(p)$$

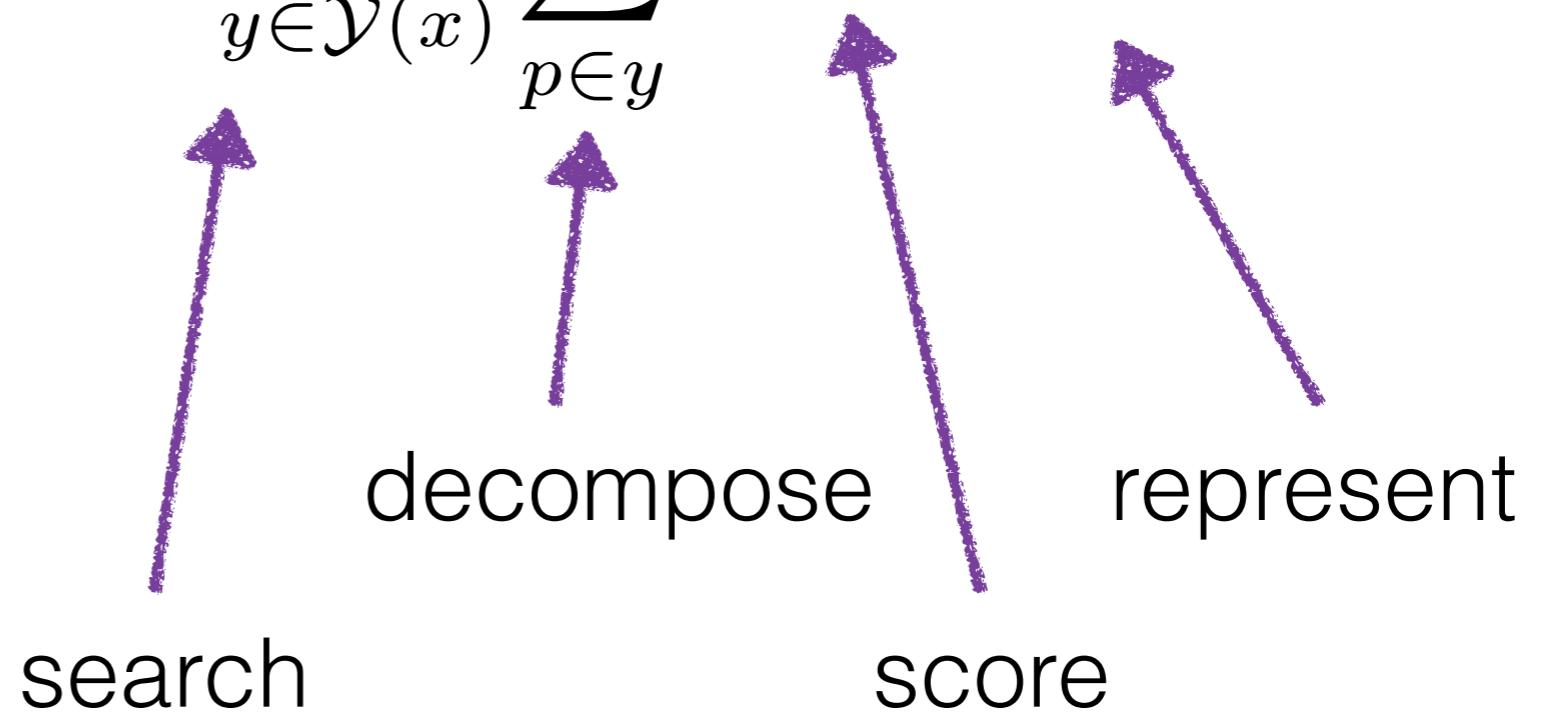
Structured Prediction Recipe

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

- Decompose structure to local factors.
- Assign a score to each factor.
- Structure score = sum of local scores.
- Look for highest scoring structure.

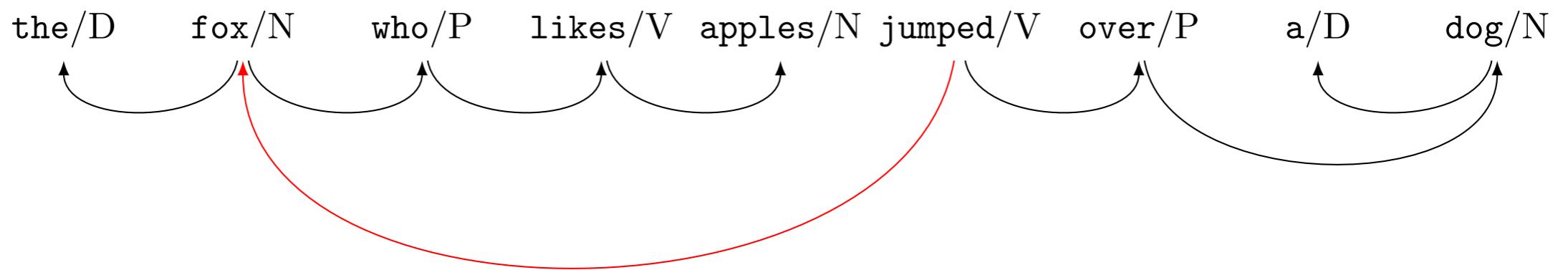
Structured Prediction Recipe

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$



First-order parser ("MST")

- **Goal:** predict a parse tree.
- **Parts:** <head, modifier> pairs (arcs).



$$\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{(h,m) \in y} \text{score}(\phi(h, m, x))$$

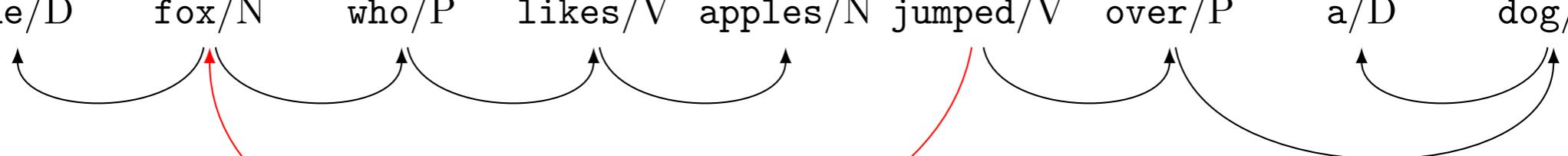
Structured Prediction Recipe

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

search decompose score represent

Decomposing

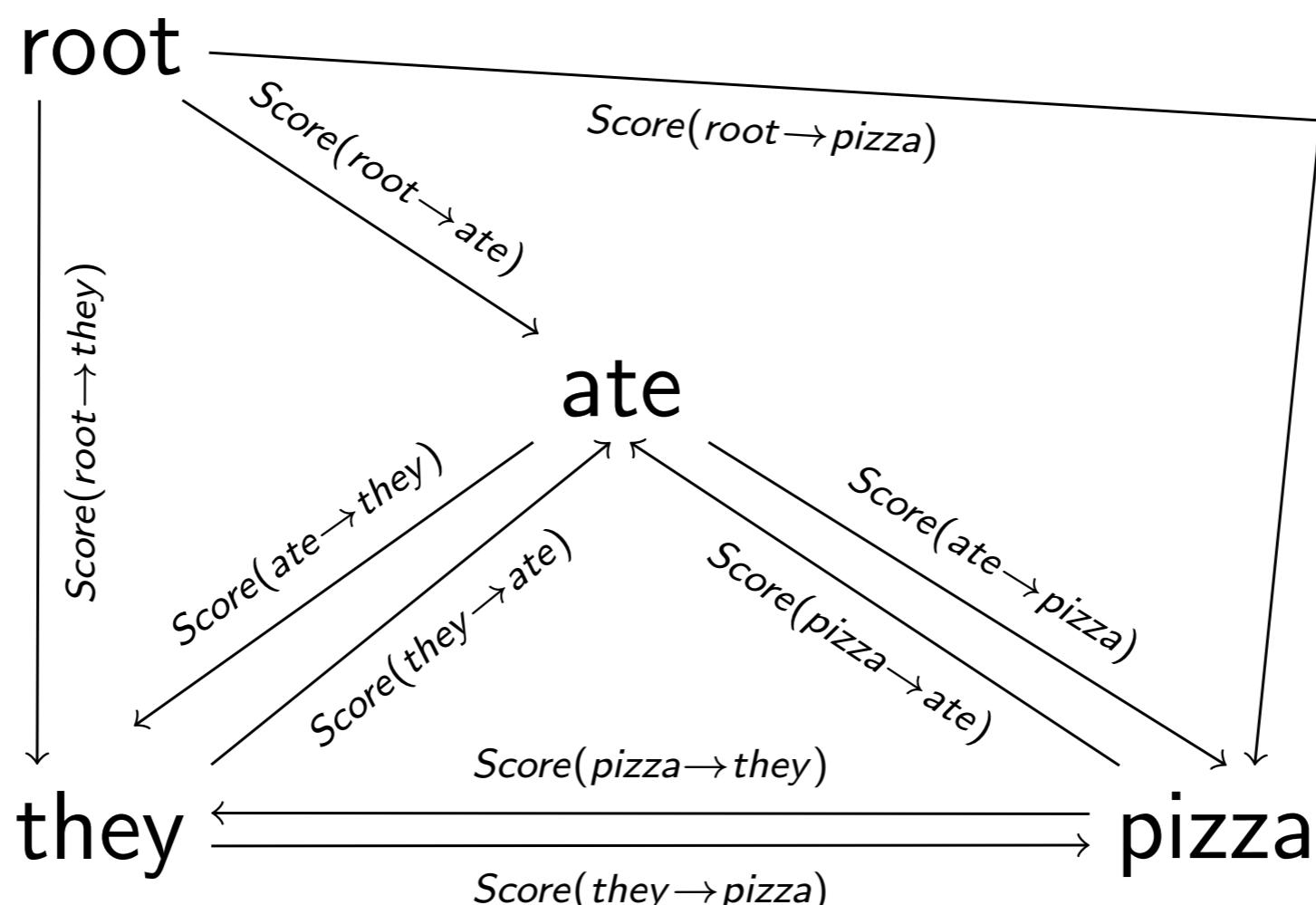
$\text{score}(\text{ the/D } \text{ fox/N } \text{ who/P } \text{ likes/V } \text{ apples/N } \text{ jumped/V } \text{ over/P } \text{ a/D } \text{ dog/N }) =$



=
=score(fox, the)
+score(fox, who)
+score(who, likes)
+score(likes, apples)
+score(jumped, fox)
+score(jumped, over)
+score(over, dog)
+score(dog, a)

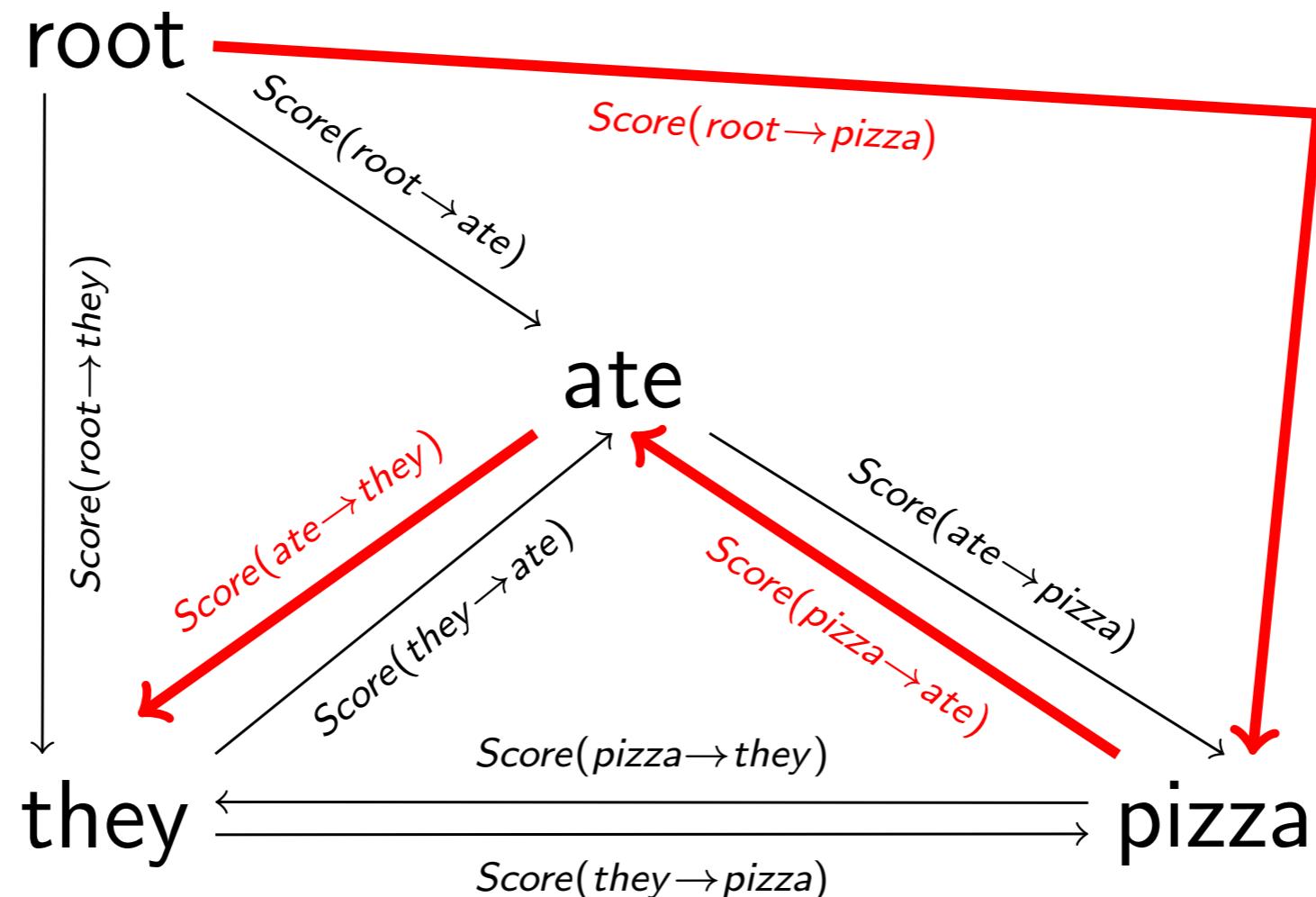
Graph-based Parsing (Inference)

Input Sentence: "They ate pizza"



score each possible arc (n^2)

Graph-based Parsing (Inference)



Spanning tree with maximal score

Structured Prediction Recipe

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

search decompose represent

score

Finding the Best Tree

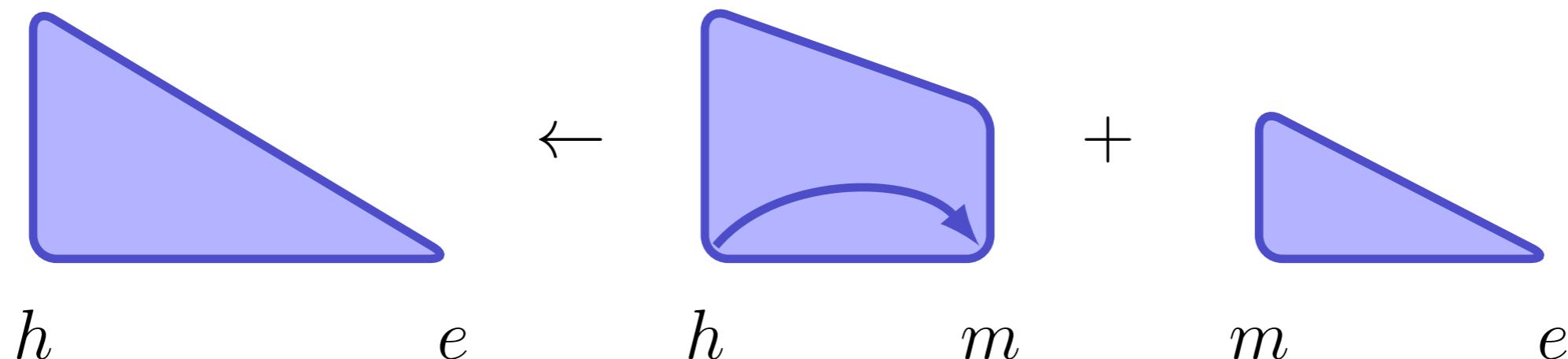
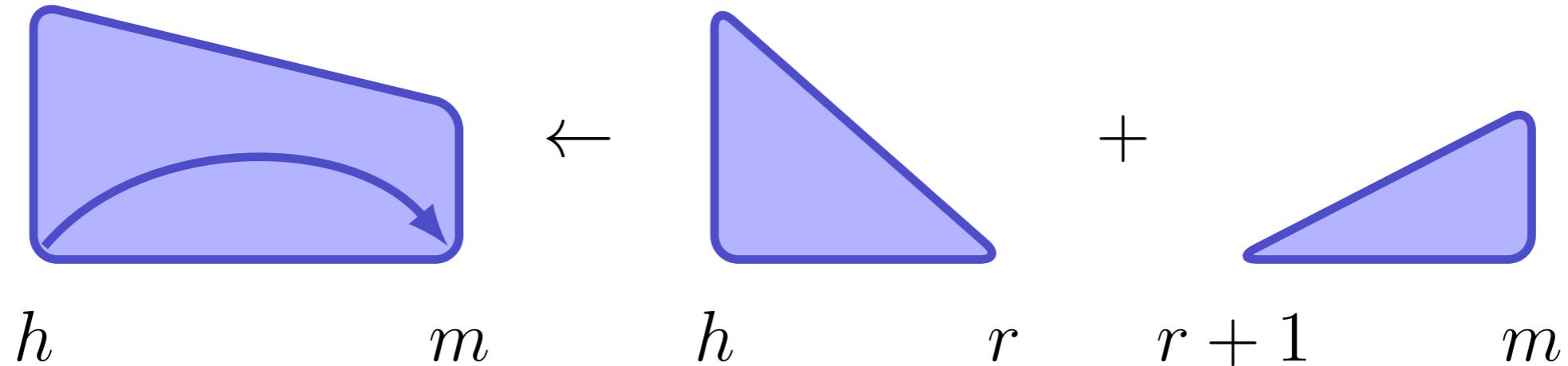
- Projective Parsing:
 - **The Eisner Algorithm** (Dynamic Programming)
- Non-projective parsing:
 - **Directed Spanning Tree**
(Chu Liu Edmunds, Tarjan)

Eisner Parsing Algorithm

Eisner Parsing Algorithm

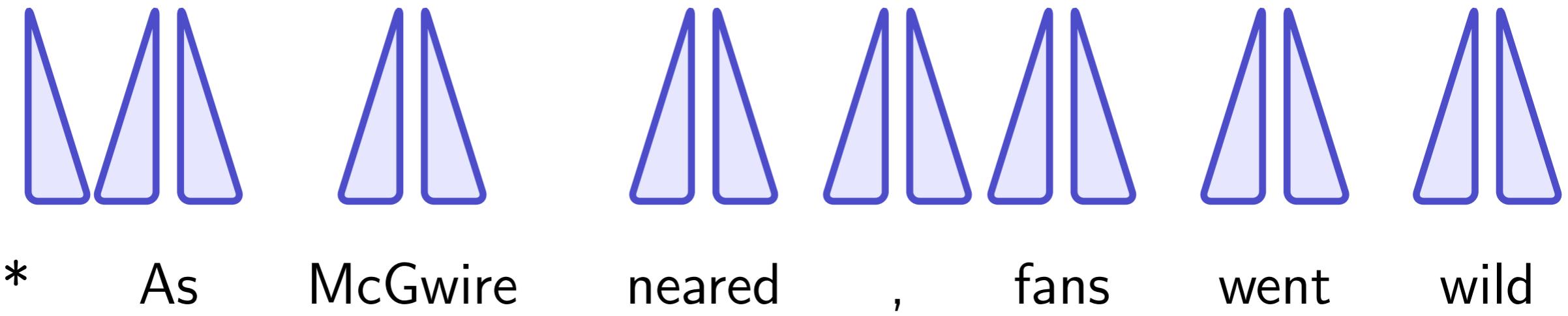
- Finding the best **projective** tree.
- We can use the lexicalized version of CKY
 - ... but this will give use n^5 algorithm.
- Jason Eisner (and Giorgio Satta) reduced this to n^3 using a more specialized algorithm.
 - Main idea: decouple scoring of left and right modifiers.

Dependency Parsing Algorithm - First-order Model



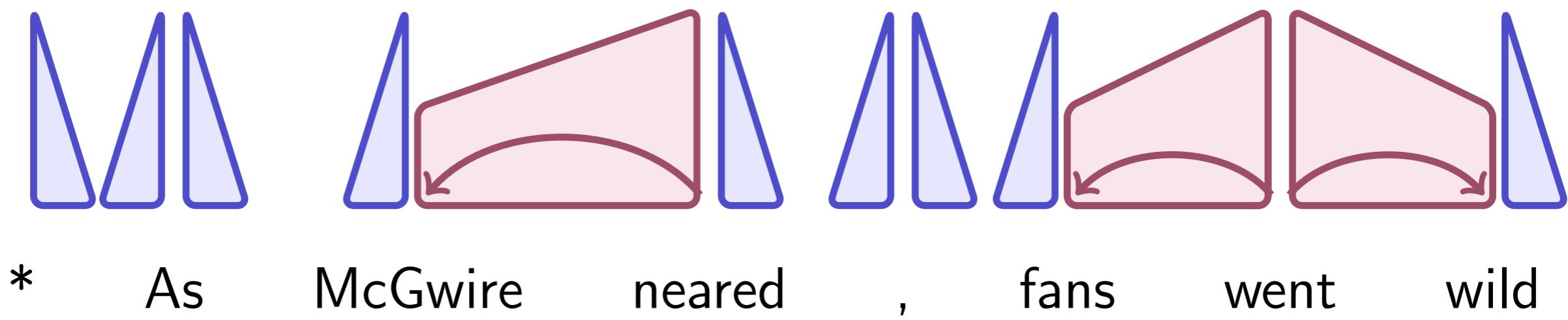
(slide from Alexander Rush)

Base Case



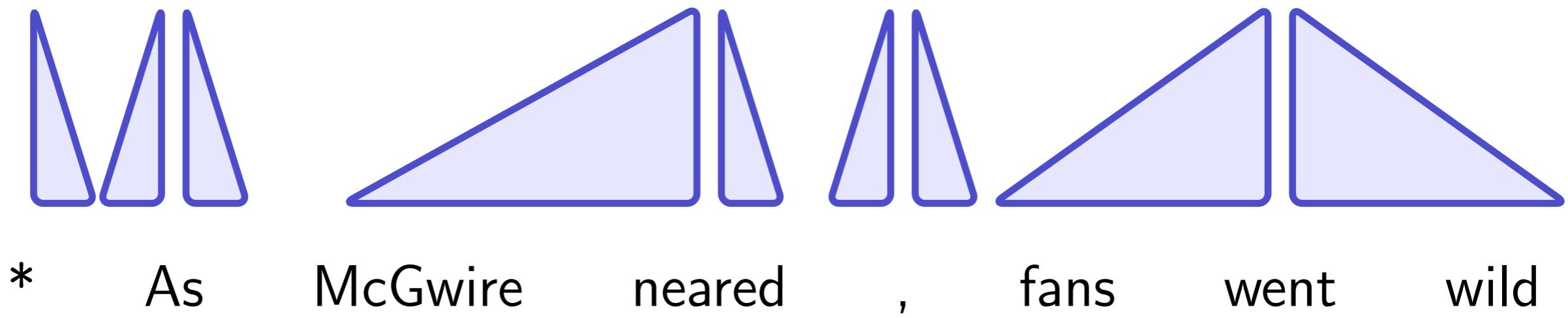
(slide from Alexander Rush)

Parsing



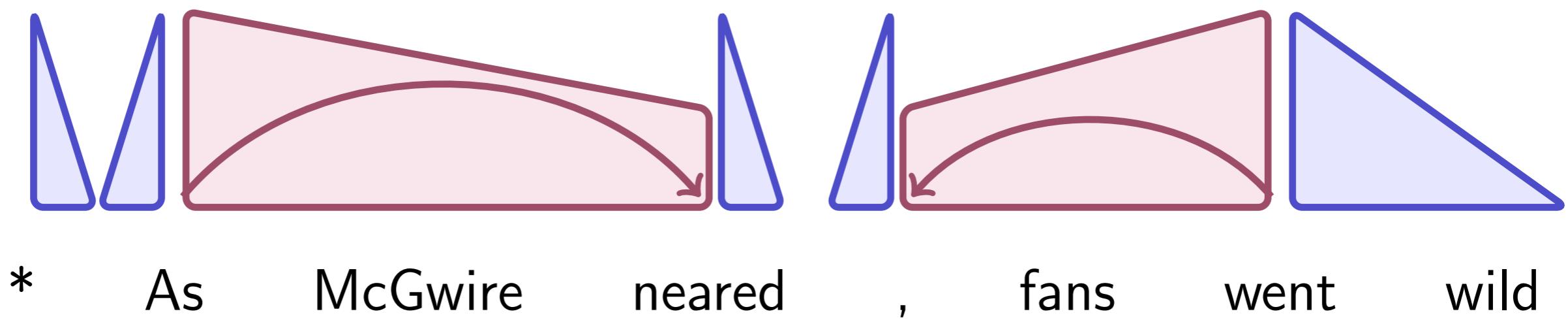
(slide from Alexander Rush)

Parsing



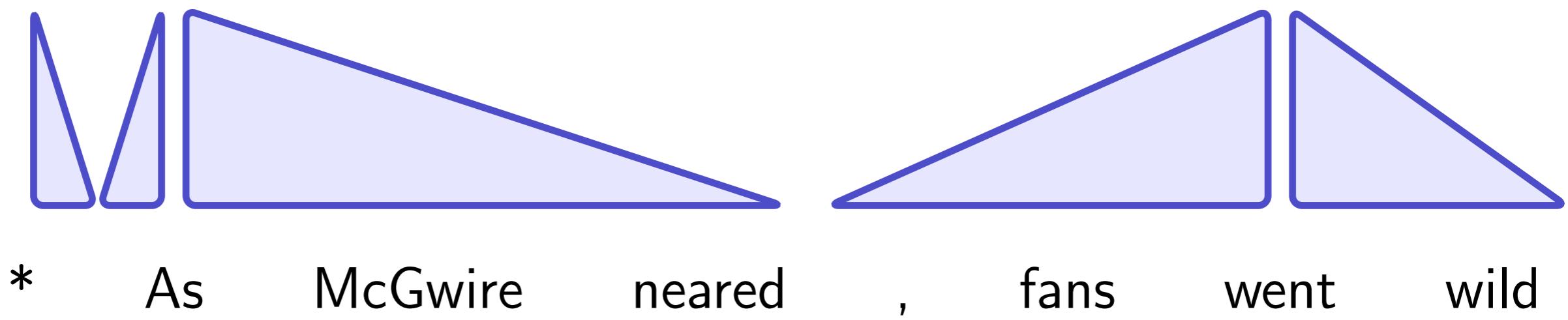
(slide from Alexander Rush)

Parsing



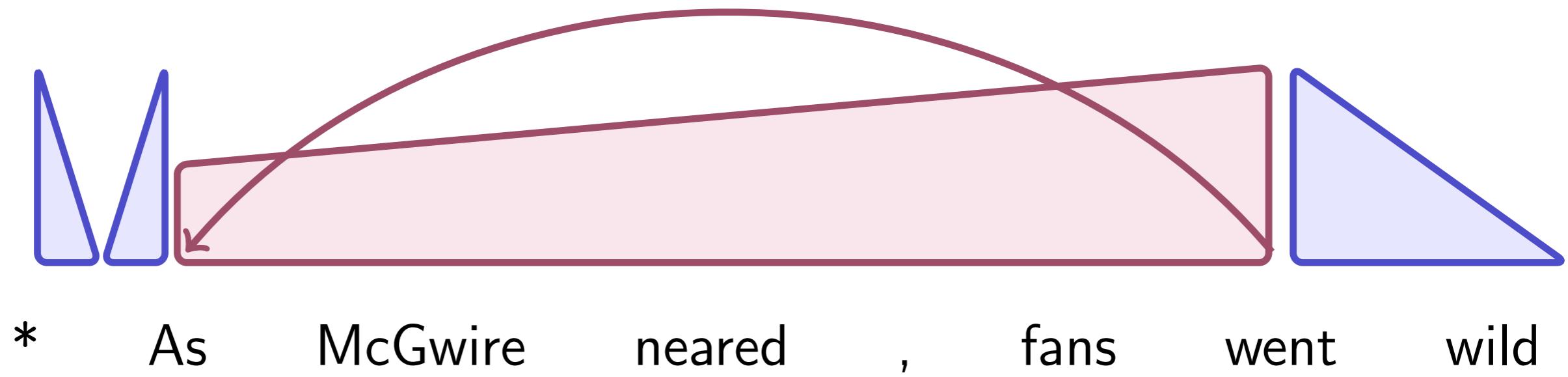
(slide from Alexander Rush)

Parsing



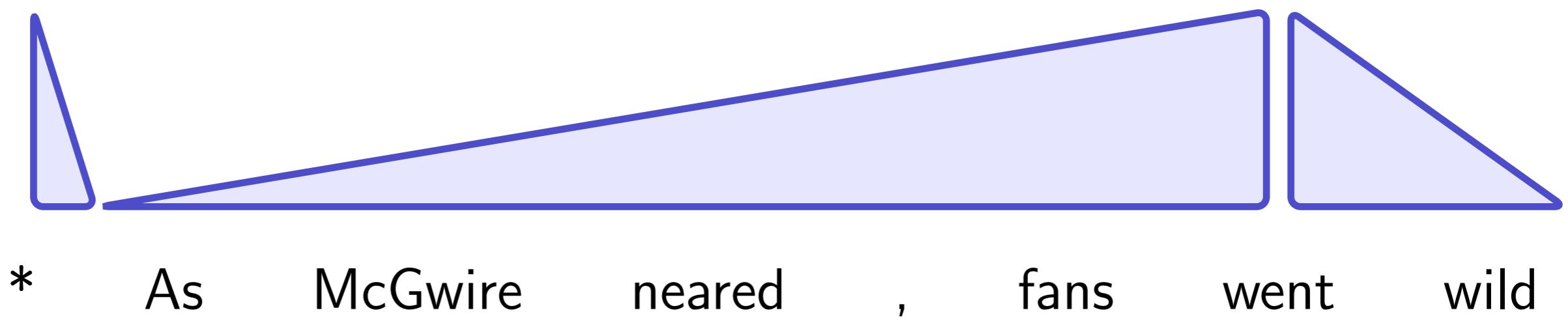
(slide from Alexander Rush)

Parsing



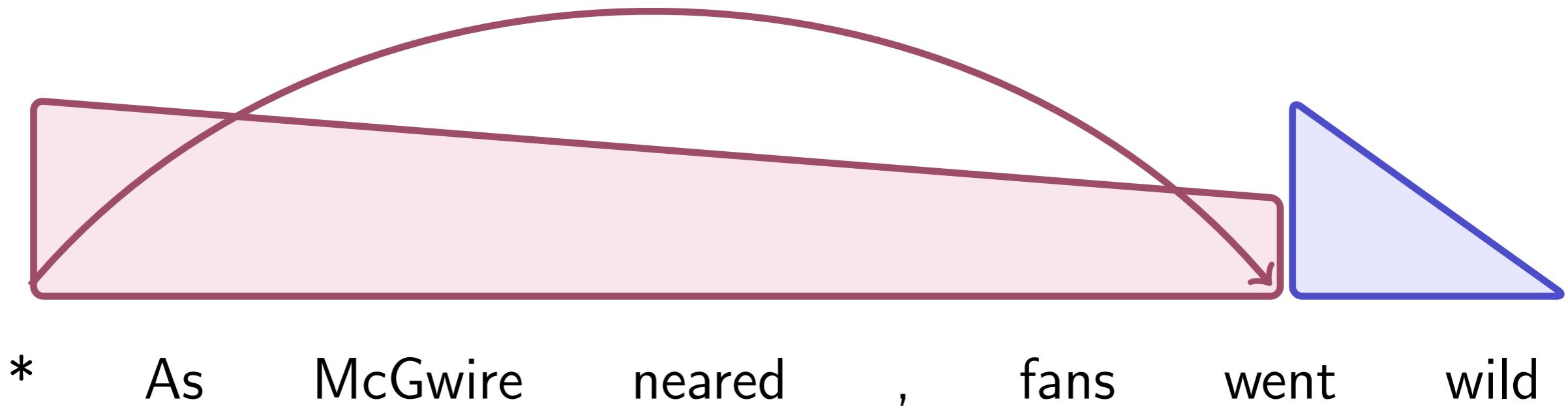
(slide from Alexander Rush)

Parsing



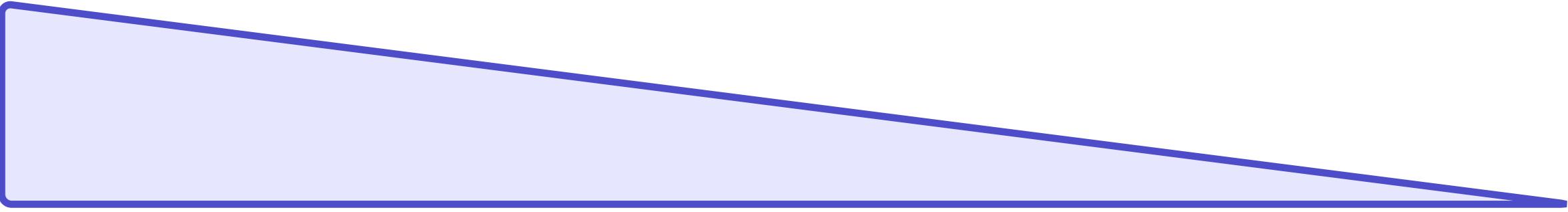
(slide from Alexander Rush)

Parsing



(slide from Alexander Rush)

Parsing



* As McGwire neared , fans went wild

(slide from Alexander Rush)

Eisner's algorithm

- Start with two triangles for each word.
- Combine two small triangles into a larger trapezoid, and add arc between the words.
- Combine trapezoid and triangle into larger triangle.
- Need an order that computes all smaller parts before larger ones.

Algorithm Key

- ▶ L; left-facing item
- ▶ R; right-facing item
- ▶ C; completed item (triangle)
- ▶ I; incomplete item (trapezoid)

Algorithm

Initialize:

for i in $0 \dots n$ **do**

$$\pi[C, L, i, i] = 0$$

$$\pi[C, R, i, i] = 0$$

$$\pi[I, L, i, i] = 0$$

$$\pi[I, R, i, i] = 0$$

Inner Loop:

for k in $1 \dots n$ **do**

for s in $0 \dots n$ **do**

$$t \leftarrow k + s$$

if $t \geq n$ **then** break

$$\pi[I, L, s, t] = \max_{r \in s \dots t-1} \pi[C, R, s, r] + \pi[C, L, r+1, t]$$

$$\pi[I, R, s, t] = \max_{r \in s \dots t-1} \pi[C, R, s, r] + \pi[C, L, r+1, t]$$

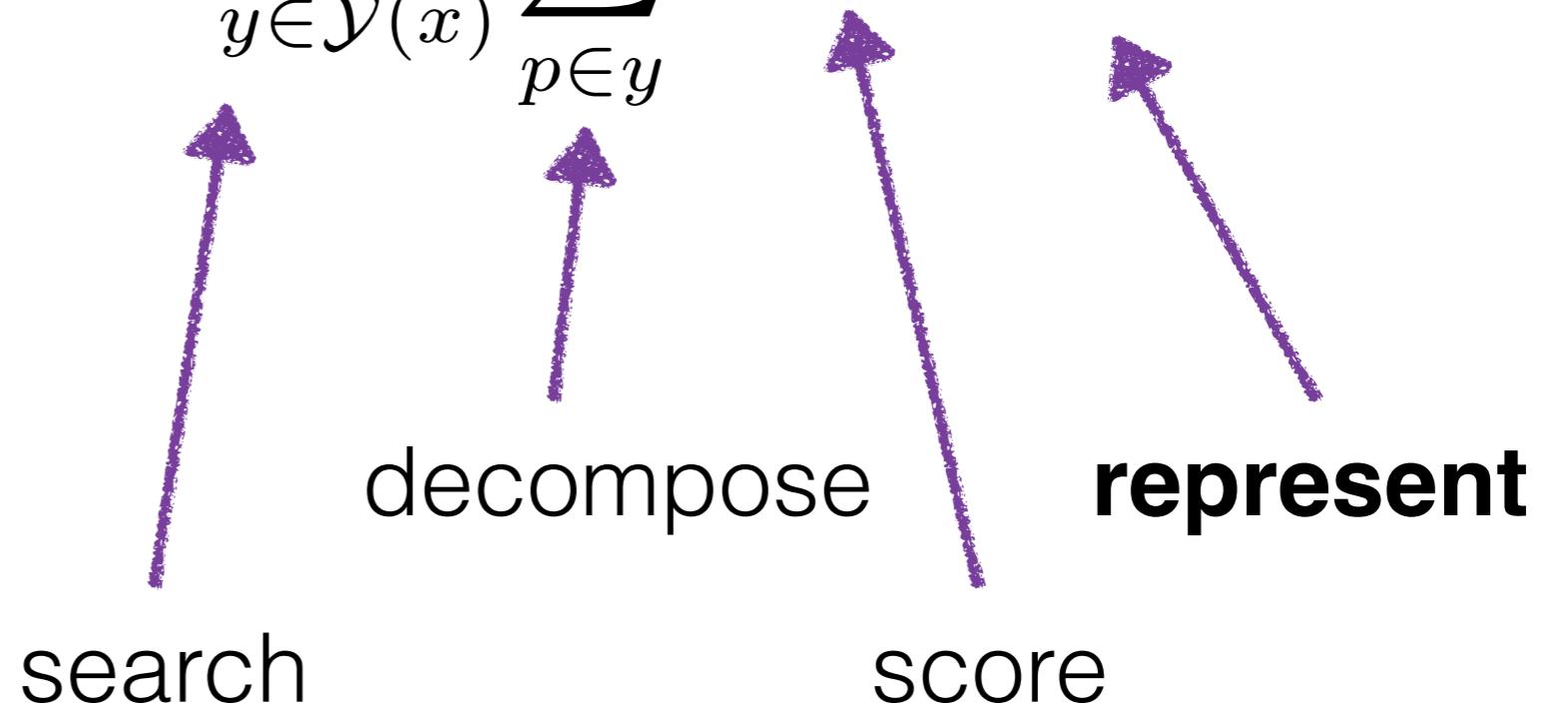
$$\pi[C, L, s, t] = \max_{r \in s \dots t-1} \pi[C, L, s, r] + \pi[I, L, r, t]$$

$$\pi[C, R, s, t] = \max_{r \in s+1 \dots t} \pi[I, R, s, r] + \pi[C, R, r, t]$$

return $\pi[C, R, 0, n]$

Structured Prediction Recipe

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$



Representing

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

- *feature function* extracts useful signals from parts.
- most work traditionally goes into this component.

Linear features

$\phi(saw, with)$



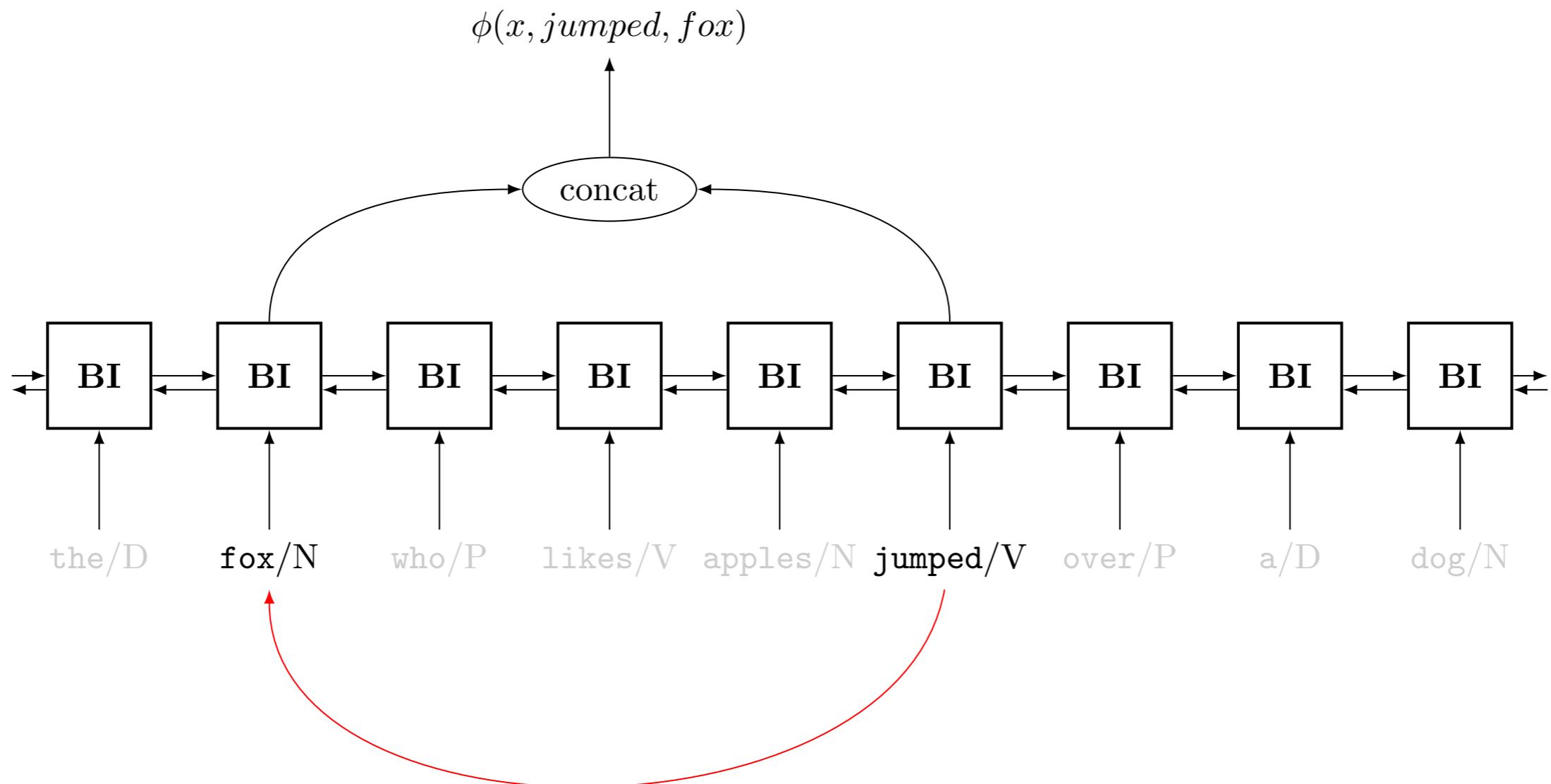
- Identities of the words w_i and w_j and the label l_k
- Part-of-speech tags of the words w_i and w_j and the label l_k
- Part-of-speech of words surrounding and between w_i and w_j
- Number of words between w_i and w_j , and their orientation
- Combinations of the above

Linear features

[went]	[VBD]	[As]	[ADP]	[went]
[VERB]	[As]	[IN]	[went, VBD]	[As, ADP]
[went, As]	[VBD, ADP]	[went, VERB]	[As, IN]	[went, As]
[VERB, IN]	[VBD, As, ADP]	[went, As, ADP]	[went, VBD, ADP]	[went, VBD, As]
[ADJ, *, ADP]	[VBD, *, ADP]	[VBD, ADJ, ADP]	[VBD, ADJ, *]	[NNS, *, ADP]
[NNS, VBD, ADP]	[NNS, VBD, *]	[ADJ, ADP, NNP]	[VBD, ADP, NNP]	[VBD, ADJ, NNP]
[NNS, ADP, NNP]	[NNS, VBD, NNP]	[went, left, 5]	[VBD, left, 5]	[As, left, 5]
[ADP, left, 5]	[VERB, As, IN]	[went, As, IN]	[went, VERB, IN]	[went, VERB, As]
[JJ, *, IN]	[VERB, *, IN]	[VERB, JJ, IN]	[VERB, JJ, *]	[NOUN, *, IN]
[NOUN, VERB, IN]	[NOUN, VERB, *]	[JJ, IN, NOUN]	[VERB, IN, NOUN]	[VERB, JJ, NOUN]
[NOUN, IN, NOUN]	[NOUN, VERB, NOUN]	[went, left, 5]	[VERB, left, 5]	[As, left, 5]
[IN, left, 5]	[went, VBD, As, ADP]	[VBD, ADJ, *, ADP]	[NNS, VBD, *, ADP]	[VBD, ADJ, ADP, NNP]
[NNS, VBD, ADP, NNP]	[went, VBD, left, 5]	[As, ADP, left, 5]	[went, As, left, 5]	[VBD, ADP, left, 5]
[went, VERB, As, IN]	[VERB, JJ, *, IN]	[NOUN, VERB, *, IN]	[VERB, JJ, IN, NOUN]	[NOUN, VERB, IN, NOUN]
[went, VERB, left, 5]	[As, IN, left, 5]	[went, As, left, 5]	[VERB, IN, left, 5]	[VBD, As, ADP, left, 5]
[went, As, ADP, left, 5]	[went, VBD, ADP, left, 5]	[went, VBD, As, left, 5]	[ADJ, *, ADP, left, 5]	[VBD, *, ADP, left, 5]
[VBD, ADJ, ADP, left, 5]	[VBD, ADJ, *, left, 5]	[NNS, *, ADP, left, 5]	[NNS, VBD, ADP, left, 5]	[NNS, VBD, *, left, 5]
[ADJ, ADP, NNP, left, 5]	[VBD, ADP, NNP, left, 5]	[VBD, ADJ, NNP, left, 5]	[NNS, ADP, NNP, left, 5]	[NNS, VBD, NNP, left, 5]
[VERB, As, IN, left, 5]	[went, As, IN, left, 5]	[went, VERB, IN, left, 5]	[went, VERB, As, left, 5]	[JJ, *, IN, left, 5]
[VERB, *, IN, left, 5]	[VERB, JJ, IN, left, 5]	[VERB, JJ, *, left, 5]	[NOUN, *, IN, left, 5]	[NOUN, VERB, IN, left, 5]

(slide from Slav Petrov)

Neural Features (bi-LSTM)



$$\phi(x, h, m) = [BIRNN(x, h); BIRNN(x, m)]$$

Structured Prediction Recipe

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

search decompose **score** represent

Scoring

$$\text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p))$$

- *a model (linear or not) assigns a score based on features*

$$\text{score}(\phi(h, m)) = \mathbf{w} \cdot \phi(h, m)$$

Scoring

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

- a model (linear or not) assigns a score based on features

Linear: $score(\phi(h, m)) = \mathbf{w} \cdot \phi(h, m)$

Non-linear: $score(\phi(h, m)) = \mathbf{w} \cdot \tanh(\mathbf{U} \cdot \phi(h, m))$

Scoring

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} score(\phi(p))$$

- a model (linear or not) assigns a score based on features

Linear: $score(\phi(h, m)) = \mathbf{w} \cdot \phi(h, m)$

Non-linear: $score(\phi(h, m)) = \mathbf{w} \cdot \tanh(\mathbf{U} \cdot \phi(h, m))$

Bi-linear: $score(\phi(h, m)) = \phi(h) \mathbf{W} \phi(m)$

Training (linear, structured perceptron)

- For each gold pair (x, y) :

- Predict \hat{y} based on x .

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{h, m \in y} \mathbf{w} \cdot \phi(h, m)$$

- if $\hat{y} \neq y$, update:

$$\mathbf{w} \leftarrow \mathbf{w} + \sum_{h, m \in y} \phi(h, m) - \sum_{h, m \in \hat{y}} \phi(h, m)$$

Training (linear, structured perceptron)

- For each gold pair (x, y) :

- Predict \hat{y} based on x .

$$predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{h, m \in y} \mathbf{w} \cdot \phi(h, m)$$

- if $\hat{y} \neq y$, update:

$$\mathbf{w} \leftarrow \mathbf{w} + \sum_{h, m \in y} \phi(h, m) - \sum_{h, m \in \hat{y}} \phi(h, m)$$

gold features

predicted features

At Parsing Time

- Assign a score to each head,modifier pair.
- Use Eisner/CLE to find best scoring tree.

Transition-based parsing

Transition-based (greedy) parsing

1. Start with an unparsed sentence.
2. Apply **locally-optimal** actions until sentence is parsed.

Transition-based (greedy) parsing

1. Start with an unparsed sentence.
2. Apply **locally-optimal** actions until sentence is parsed.
3. Use whatever features you want.
4. Surprisingly accurate.
5. Can be extremely fast.

Intro to Transition-based Dependency Parsing

An abstract machine composed of a **stack** and a **buffer**.

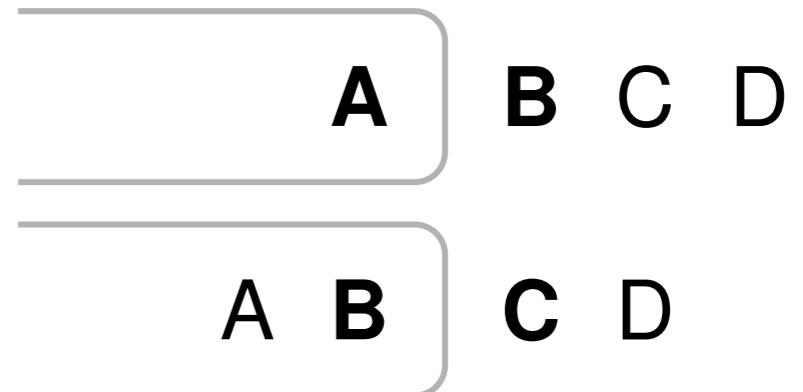
Machine is initialized with the words of a sentence.

A set of actions process the words by moving them from buffer to stack, removing them from the stack, or adding links between them.

A specific set of actions define a transition system.

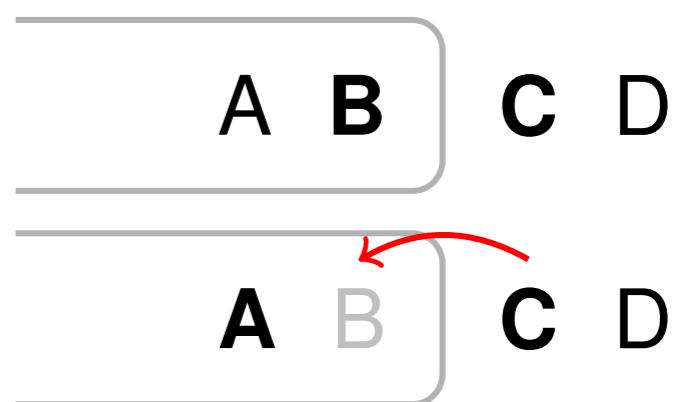
The Arc-Eager Transition System

- ▶ **SHIFT** move first word from buffer to stack.
(pre: Buffer not empty.)



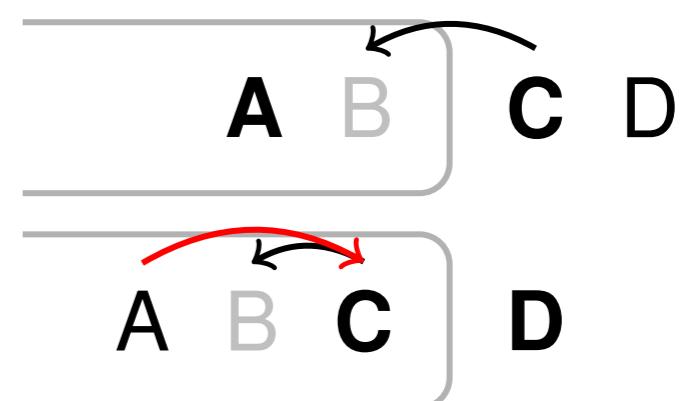
The Arc-Eager Transition System

- ▶ **SHIFT** move first word from buffer to stack.
(pre: Buffer not empty.)
- ▶ **LEFTARC**_{label} make first word in buffer head of top of stack, pop the stack.
(pre: Stack not empty. Top of stack does not have a parent.)



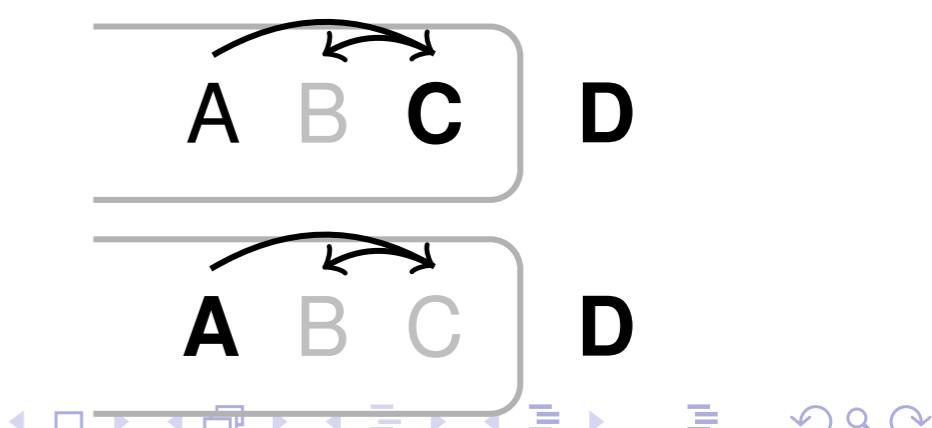
The Arc-Eager Transition System

- ▶ **SHIFT** move first word from buffer to stack.
(pre: Buffer not empty.)
- ▶ **LEFTARC**_{label} make first word in buffer head of top of stack, pop the stack.
(pre: Stack not empty. Top of stack does not have a parent.)
- ▶ **RIGHTARC**_{label} make top of stack head of first in buffer, move first in buffer to stack.
(pre: Buffer not empty.)

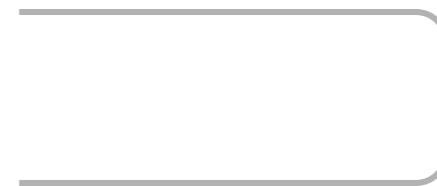


The Arc-Eager Transition System

- ▶ **SHIFT** move first word from buffer to stack.
(pre: Buffer not empty.)
- ▶ **LEFTARC**_{label} make first word in buffer head of top of stack, pop the stack.
(pre: Stack not empty. Top of stack does not have a parent.)
- ▶ **RIGHTARC**_{label} make top of stack head of first in buffer, move first in buffer to stack.
(pre: Buffer not empty.)
- ▶ **REDUCE** pop the stack
(pre: Stack not empty. Top of stack has a parent.)



Parsing Example



She ate pizza with pleasure

Parsing Example

She **ate** pizza with pleasure



The word 'ate' is bolded, and the words 'ate' and 'pizza' are enclosed in a light gray rounded rectangular box. A black curved arrow is positioned above this box, pointing from the left towards the right, indicating a dependency or a specific parse path for those two words.

Parsing Example

She ate pizza with pleasure



Parsing Example

She ate pizza with pleasure

The diagram illustrates a dependency parse for the sentence "She ate pizza with pleasure". The words are arranged horizontally. Arrows connect the words to show their grammatical relationships: "She" points to "ate", "ate" points to "pizza", "with" points to "pleasure", and both "pizza" and "with" point to "pleasure". The word "pleasure" is enclosed in a light gray rounded rectangle, indicating it is the head of the phrase.

Parsing Example

She ate pizza with pleasure

The diagram illustrates the dependencies between the words in the sentence "She ate pizza with pleasure". The words are arranged horizontally. Arrows connect the word "ate" to both "pizza" and "with", and another arrow connects "with" to "pleasure". The word "pleasure" is enclosed in a light gray rounded rectangle, suggesting it is a noun phrase or a semantic role. A horizontal line is drawn under the word "pleasure".

Parsing Example

She ate pizza with pleasure

The diagram illustrates a dependency parse for the sentence "She ate pizza with pleasure". The words are arranged horizontally: "She", "ate", "pizza", "with", and "pleasure". Three curved arrows originate from the word "ate": one points left to "She", another points right to "pizza", and a third points right to "pleasure". The word "with" has a curved arrow pointing right to "pleasure". All five words are contained within a light gray rounded rectangular box.

Parsing Example

She ate pizza with pleasure



What do we know about the arc-eager transition system?

- ▶ Every sequence of actions result in a valid projective structure.
- ▶ Every projective tree is derivable by (at least one) sequence of actions.
- ▶ Given a tree, finding a sequence of actions for deriving it. ("oracle")

we know these things also for the
arc-standard, arc-hybrid and other transition systems

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She

ate pizza with pleasure ↵ ↺ ↻ ↻ ↻

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She

ate pizza with pleasure ↵ ↺ ↻ ↻ ↻

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



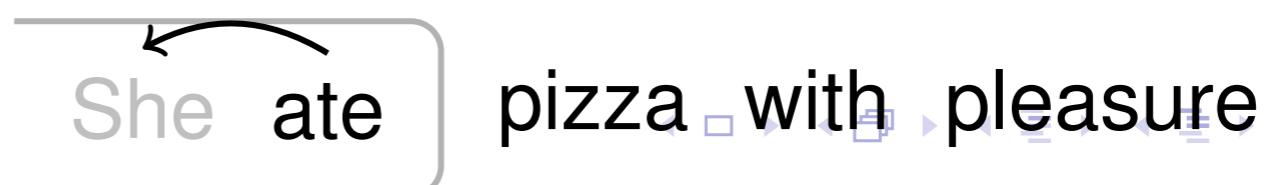
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



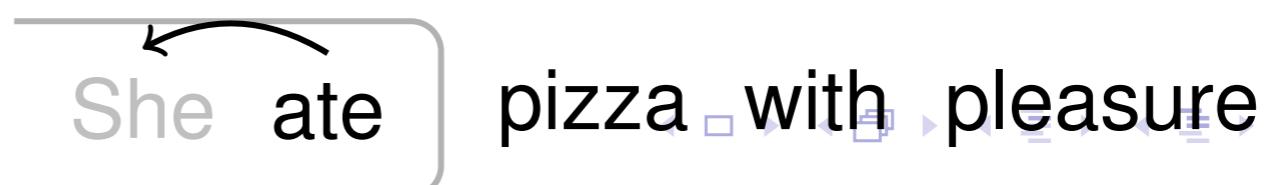
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH **RIGHT** RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH **RIGHT** RE RIGHT RIGHT RE RE RE

She ate pizza

with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza

with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE **RIGHT** RIGHT RE RE RE



with pleasure

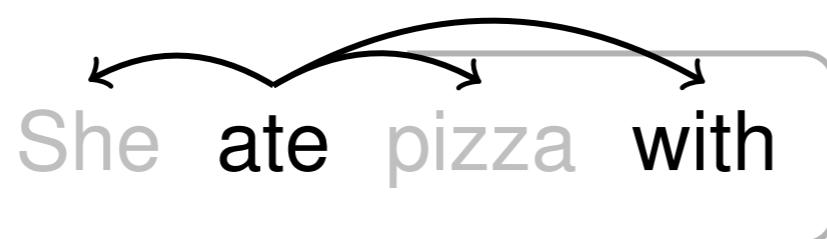
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE **RIGHT** RIGHT RE RE RE



pleasure

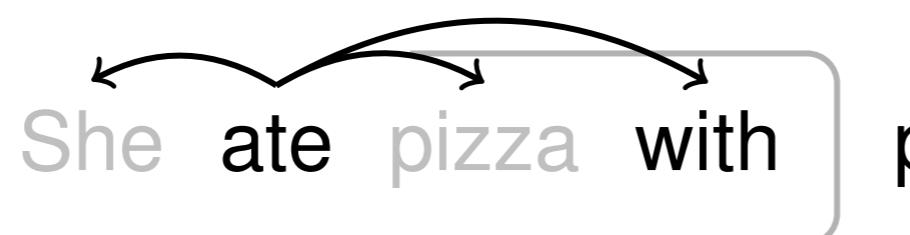
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT **RIGHT** RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT **RIGHT** RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE



This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing without an oracle

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

This knowledge is quite powerful

Parsing without an oracle

```
start with weight vector  $w$ 
configuration  $\leftarrow$  initialize(sentence)
while not configuration.IsFinal() do
    action  $\leftarrow$  predict( $w$ ,  $\phi(\text{configuration})$ )
    configuration  $\leftarrow$  configuration.apply(action)
return configuration.tree
```

This knowledge is quite powerful

Parsing without an oracle

summarize the configuration
as a feature vector

```
start with weight vector  $w$ 
configuration  $\leftarrow$  initialize(sentence)
while not configuration.IsFinal() do
    action  $\leftarrow$  predict( $w$ ,  $\phi(\text{configuration})$ )
    configuration  $\leftarrow$  configuration.apply(action)
return configuration.tree
```

This knowledge is quite powerful

Parsing without an oracle

summarize the configuration
as a feature vector

```
start with weight vector  $w$ 
configuration  $\leftarrow$  initialize(sentence)
while not configuration.IsFinal() do
    action  $\leftarrow$  predict( $w$ ,  $\phi(\text{configuration})$ )
    configuration  $\leftarrow$  configuration.apply(action)
return configuration.tree
```

predict the action based on the features

This knowledge is quite powerful

Parsing without an oracle

summarize the configuration
as a feature vector

start with weight vector w

configuration \leftarrow initialize(sentence)

while not configuration.IsFinal() **do**

action \leftarrow predict(w , $\phi(\text{configuration})$)

configuration \leftarrow configuration.apply(action)

return configuration.tree

predict the action based on the features

need to learn the correct weights

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
```

This knowledge is quite powerful

Learning a parser

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
```

```
    configuration ← configuration.apply(action)
```

This knowledge is quite powerful

Learning a parser

$w \leftarrow 0$

for sentence,tree pair in corpus **do**

 sequence \leftarrow oracle(sentence, tree)

 configuration \leftarrow initialize(sentence)

while not configuration.IsFinal() **do**

 action \leftarrow sequence.next()

 features $\leftarrow \phi(\text{configuration})$

 predicted \leftarrow predict(w , $\phi(\text{configuration})$)

if predicted \neq action **then**

$w.\text{update}(\phi(\text{configuration}), \text{action}, \text{predicted})$

 configuration \leftarrow configuration.apply(action)

return w

This knowledge is quite powerful

Parsing time

```
configuration ← initialize(sentence)
while not configuration.isFinal() do
    action ← predict( $w$ ,  $\phi(\text{configuration})$ )
    configuration ← configuration.apply(action)
return configuration.tree
```

In short

- ▶ Summarize configuration by a set of features.
- ▶ Learn the best action to take at each configuration.
- ▶ Hope this generalizes well.

Transition Based Parsing

- ▶ A different approach.
- ▶ Very common.
- ▶ Can be as accurate as first-order graph-based parsing.
 - ▶ Higher-order graph-based are still better.
- ▶ Easy to implement.
- ▶ Very fast. ($O(n)$)
- ▶ Can be improved further:
 - ▶ Easy-first
 - ▶ Dynamic oracle
 - ▶ Beam Search

Summary

- Syntax (hierarchical structure)
- Grammars, PCFG, CKY Algorithm
 - Head Words
- Constituency to dependency with head-words
- Dependency Parsing
 - Graph Based
 - Transition Based

Parsers

- Phrase Based
 - Berkeley Parser
 - Stanford Parser
- Dependency
 - spaCy
 - Stanford Parser
- Dependency + research
 - BiST parser (Kiperwasser and Goldberg)
 - Stack LSTM parser (Dyer et al)