Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det				
1 very		Adv	AP		
2 heavy			A,AP		
3 orange					
4 book					

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det				
1 very		Adv	AP		
2 heavy			A,AP		
3 orange				Nom,A,AP	
4 book					

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det				
1 very		Adv	AP		
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det				
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	
4 book					Nom

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	
3 orange				Nom,A,AP	Nom
4 book					Nom

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	
1 very		Adv	AP	Nom	
2 heavy			A,AP	Nom	Nom
3 orange				Nom,A,AP	Nom
4 book					Nom

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	
1 very		Adv	AP	Nom	Nom
2 heavy			A,AP	Nom	Nom
3 orange				Nom,A,AP	Nom
4 book					Nom

Shay Cohen, Accelerated Natural Language Processing (Slides)

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	NP
1 very		Adv	AP	Nom	Nom
2 heavy			A,AP	Nom	Nom
3 orange				Nom,A,AP	Nom
4 book					Nom

Shay Cohen, Accelerated Natural Language Processing (Slides)

CYK: The general algorithm

function CKY-Parse(*words*, *grammar*) **returns** *table* **for**

$j \leftarrow$ **from** 1 **to** $\textsc{Length}(words)$ **do**

$\quad table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$

$\quad$ **for** $i \leftarrow$ **from** $j-2$ **downto** 0 **do**

$\qquad$ **for** $k \leftarrow i+1$ **to** $j-1$ **do**

$$table[i,j] \leftarrow table[i,j] \cup \{A \mid A \rightarrow BC \in grammar, B \in table[i,k], C \in table[k,j]\}$$

CYK: The general algorithm

function CKY-Parse(*words*, *grammar*) **returns** *table* **for**

$j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do** | loop over the columns

$table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$ | fill bottom cell

for $i \leftarrow$ **from** $j-2$ **downto** 0 **do** | fill row i in column j

for $k \leftarrow i+1$ **to** $j-1$ **do** | loop over split locations between i and j

$$table[i,j] \leftarrow table[i,j] \cup \{A \mid A \rightarrow BC \in grammar, B \in table[i,k], C \in table[k,j]\}$$

Check the grammar for rules that link the constituent in $[i,k]$ with those in $[k,j]$. For each rule found store LHS in cell $[i,j]$.

A succinct representation of CKY

We have a Boolean table called Chart, such that $\mathrm{Chart}[A, i, j]$ is true if there is a sub-phrase according the grammar that dominates words i through words j

Build this chart recursively, similarly to the Viterbi algorithm:

For $j > i + 1$:

$$\mathrm{Chart}[A, i, j] = \bigvee_{k=i+1}^{j-1} \bigvee_{A \to B\ C} Chart[B, i, k] \wedge Chart[C, k, j]$$

Seed the chart, for $i + 1 = j$:
$\mathrm{Chart}[A, i, i + 1] = \mathrm{True}$ if there exists a rule $A \to w_{i+1}$ where w_{i+1} is the $(i + 1)$th word in the string

From CYK Recognizer to CYK Parser

- So far, we just have a chart recognizer, a way of determining whether a string belongs to the given language.
- Changing this to a parser requires recording which existing constituents were combined to make each new constituent.
- This requires another field to record the one or more ways in which a constituent spanning (i,j) can be made from constituents spanning (i,k) and (k,j). (More clearly displayed in graph representation, see next lecture.)
- In any case, for a fixed grammar, the CYK algorithm runs in time $O(n^3)$ on an input string of n tokens.
- The algorithm identifies all possible parses.

Shay Cohen, Accelerated Natural Language Processing (Slides)

CYK-style parse charts

Even without converting a grammar to CNF, we can draw *CYK-style* parse charts:

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	NP
1	very		OptAdv	OptAP	Nom	Nom
2	heavy			A,OptAP	Nom	Nom
3	orange				N,Nom,A,AP	Nom
4	book					N,Nom

(We haven't attempted to show ϵ-phrases here. Could in principle use cells below the main diagonal for this . . .)

However, CYK-style parsing will have run-time worse than $O(n^3)$ if e.g. the grammar has rules $A \rightarrow BCD$.

Dynamic Programming as a problem-solving technique

- Given a problem, systematically fill a table of solutions to sub-problems: this is called memoization.
- Once solutions to all sub-problems have been accumulated, solve the overall problem by composing them.
- For parsing, the sub-problems are analyses of sub-strings and correspond to constituents that have been found.
- Sub-trees are stored in a chart (aka well-formed substring table), which is a record of all the substructures that have ever been built during the parse.

Solves **re-parsing problem**: sub-trees are looked up, not re-parsed!
Solves **ambiguity problem**: chart implicitly stores all parses!

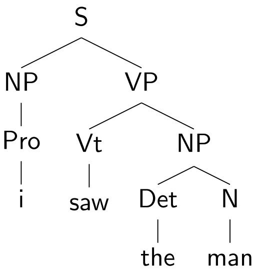
Content

Treebank grammars

- The big idea: instead of paying linguists to write a grammar, pay them to annotate real sentences with parse trees.
- This way, we implicitly get a grammar (for CFG: read the rules off the trees).
- **And** we get probabilities for those rules (using any of our favorite estimation techniques).
- We can use these probabilities to improve disambiguation and even speed up parsing.

Treebank grammars

For example, if we have this tree in our corpus:



Then we add rules

`S → NP VP`
`NP → Pro`
`Pro → i`
`VP → Vt NP`
`Vt → saw`
`NP → Det N`
`Art → the`
`N → man`

With more trees, we can start to count rules and estimate their probabilities.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Example: The Penn Treebank

- The first large-scale parse annotation project, begun in 1989.
- Original corpus of syntactic parses: Wall Street Journal text
  - About 40,000 annotated sentences (1m words)
  - Standard phrasal categories like S, NP, VP, PP.
- Now many other data sets (e.g. transcribed speech), and different kinds of annotation; also inspired treebanks in many other languages.

Other language treebanks

- Many annotated with **dependency grammar** rather than CFG (see next lecture).
- Some require paid licenses, others are free.
- Just a few examples:
  - Danish Dependency Treebank
  - Alpino Treebank (Dutch)
  - Bosque Treebank (Portuguese)
  - Talbanken (Swedish)
  - Prague Dependency Treebank (Czech)
  - TIGER corpus, Tuebingen Treebank, NEGRA corpus (German)
  - Penn Chinese Treebank
  - Penn Arabic Treebank
  - Tuebingen Treebank of Spoken Japanese, Kyoto Text Corpus

Creating a treebank PCFG

A **probabilistic context-free grammar** (PCFG) is a CFG where each rule $\mathtt{A} \rightarrow \alpha$ (where α is a symbol sequence) is assigned a probability $P(\alpha|A)$.

- The sum over all expansions of $\mathtt{A}$ must equal 1: $\sum_{\alpha'} P(\alpha'|A) = 1$.
- Easiest way to create a PCFG from a treebank: MLE
  - Count all occurrences of $\mathtt{A} \rightarrow \alpha$ in treebank.
  - Divide by the count of all rules whose LHS is $\mathtt{A}$ to get $P(\alpha|A)$
- But as usual many rules have very low frequencies, so MLE isn't good enough and we need to smooth.

The generative model

Like *n*-gram models and HMMs, PCFGs are a **generative model**.
Assumes sentences are generated as follows:

- Start with root category S.
- Choose an expansion α for S with probability $P(\alpha|S)$.
- Then recurse on each symbol in α.
- Continue until all symbols are terminals (nothing left to expand).

The probability of a parse

- Under this model, the probability of a parse t is simply the product of all rules in the parse:

$$P(t) = \prod_{A \to \alpha \in t} p(A \to \alpha \mid A)$$

Statistical disambiguation example

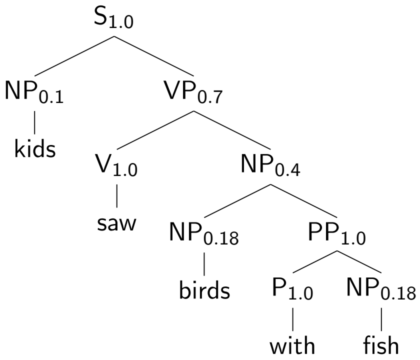
How can parse probabilities help disambiguate PP attachment?

- Let's use the following PCFG, inspired by Manning & Schuetze (1999):

S $\rightarrow$ NP VP	1.0	NP $\rightarrow$ NP PP	0.4
PP $\rightarrow$ P NP	1.0	NP $\rightarrow$ kids	0.1
VP $\rightarrow$ V NP	0.7	NP $\rightarrow$ birds	0.18
VP $\rightarrow$ VP PP	0.3	NP $\rightarrow$ saw	0.04
P $\rightarrow$ with	1.0	NP $\rightarrow$ fish	0.18
V $\rightarrow$ saw	1.0	NP $\rightarrow$ binoculars	0.1

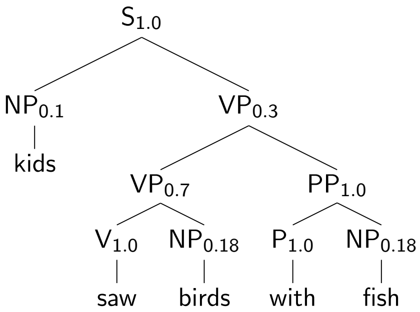
- We want to parse kids saw birds with fish.

Probability of parse 1



- $P(t_1) = 1.0 \cdot 0.1 \cdot 0.7 \cdot 1.0 \cdot 0.4 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0009072$

Probability of parse 2



- $P(t_2) = 1.0 \cdot 0.1 \cdot 0.3 \cdot 0.7 \cdot 1.0 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0006804$
- which is less than $P(t_1) = 0.0009072$, so t_1 is preferred. Yay!

The probability of a sentence

- Since t implicitly includes the words $\vec{w}$, we have $P(t) = P(t, \vec{w})$.
- So, we also have a **language model**. Sentence probability is obtained by summing over $T(\vec{w})$, the set of valid parses of $\vec{w}$:

$$P(\vec{w}) = \sum_{t \in T(\vec{w})} P(t, \vec{w}) = \sum_{t \in T(\vec{w})} P(t)$$

- In our example, $P(\text{kids saw birds with fish}) = 0.0006804 + 0.0009072$.

How to find the best parse?

First, remember standard CKY algorithm.

- Fills in cells in well-formed substring table (chart) by combining previously computed child cells.

	1	2	3	4
0	Pro, NP			S
1		Vt,Vp,N		VP
2			Pro, PosPro, D	NP
3				N,Vi
	$_o$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

Probabilistic CKY

It is straightforward to extend CKY parsing to the probabilistic case.

- Goal: return the highest probability parse of the sentence.
  - When we find an A spanning (i, j), store its probability along with its label and backpointers in cell (i, j)
  - If we later find an A with the same span but higher probability, replace the probability for A in cell (i, j), and update the backpointers to the new children.
- Analogous to Viterbi: we iterate over all possible child pairs (rather than previous states) and store the probability and backpointers for the best one.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Probabilistic CKY

We also have analogues to the other HMM algorithms.

- The **inside algorithm** computes the probability of the sentence (analogous to forward algorithm)
  - Same as above, but instead of storing the *best* parse for A, store the *sum* of all parses.
- The **inside-outside algorithm** algorithm is a form of EM that learns grammar rule probs from unannotated sentences (analogous to forward-backward).

Best-first probabilistic parsing

- So far, we've been assuming **exhaustive** parsing: return all possible parses.
- But treebank grammars are huge!!
  - Exhaustive parsing of WSJ sentences up to 40 words long adds on average over 1m items to chart per sentence.[1]
  - Can be hundreds of possible parses, but most have extremely low probability: do we really care about finding these?
- **Best-first** parsing can help.

[1] Charniak, Goldwater, and Johnson, WVLC 1998.

Best-first probabilistic parsing

Use probabilities of subtrees to decide which ones to build up further.

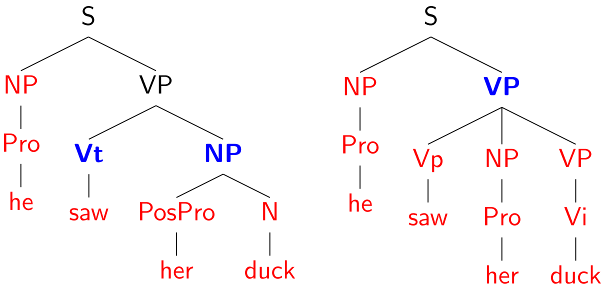
- Each time we find a new constituent, we give it a **score** ("figure of merit") and add it to an **agenda**[2], which is ordered by score.
- Then we pop the next item off the agenda, add it to the chart, and see which new constituents we can make using it.
- We add those to the agenda, and iterate.

Notice we are no longer filling the chart in any fixed order.
Many variations on this idea, often limiting the size of the agenda by **pruning** out low-scoring edges (**beam search**).

[2] aka a priority queue

Best-first intuition

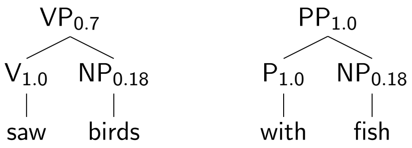
Suppose red constituents are in chart already; **blue** are on agenda.



If the `VP` in right-hand tree scores high enough, we'll pop that next, add it to chart, then find the `S`. So, we could complete the whole parse before even finding the alternative `VP`.

How do we score constituents?

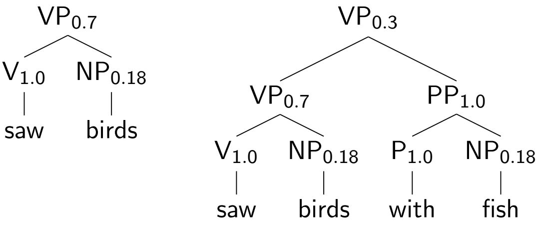
Perhaps according to the probability of the subtree they span? So, P(left example)=(0.7)(0.18) and P(right example)=0.18.



Shay Cohen, Accelerated Natural Language Processing (Slides)

How do we score constituents?

But what about comparing different sized constituents?



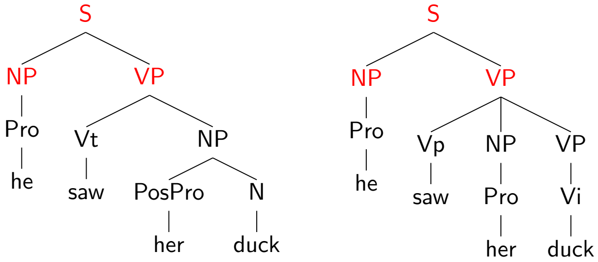
Shay Cohen, Accelerated Natural Language Processing (Slides)

A better figure of merit

- If we use raw probabilities for the score, **smaller** constituents will almost always have higher scores.
  - Meaning we pop all the small constituents off the agenda before the larger ones.
  - Which would be very much like exhaustive bottom-up parsing!
- Instead, we can divide by the **number of words** in the constituent.
  - Very much like we did when comparing language models (recall **per-word** cross-entropy)!
- This works much better, though still not guaranteed to find the best parse first. Other improvements are possible.

Evaluating parse accuracy

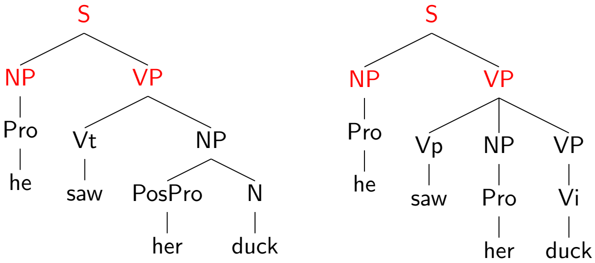
Compare **gold standard** tree (left) to **parser output** (right):



- Output constituent is counted correct if there is a gold constituent that spans the same sentence positions.
- Harsher measure: also require the constituent labels to match.
- Pre-terminals don't count as constituents.

Evaluating parse accuracy

Compare **gold standard** tree (left) to **parser output** (right):



- **Precision**: (# correct constituents)/(# in parser output) = 3/5
- **Recall**: (# correct constituents)/(# in gold standard) = 3/4
- **F-score**: balances precision/recall: 2pr/(p+r)

Shay Cohen, Accelerated Natural Language Processing (Slides)

Content

Parsing: where are we now?

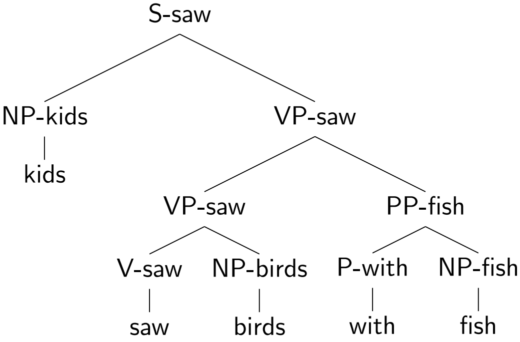
Parsing is not just WSJ. Lots of situations are much harder!

- Other languages, esp with free word order (up next) or little annotated data.
- Other domains, esp with jargon (e.g., biomedical) or non-standard language (e.g., social media text).

In fact, due to increasing focus on multilingual NLP, constituency syntax/parsing (English-centric) is losing ground to **dependency parsing**...

Lexicalization, again

We saw that adding **lexical head** of the phrase can help choose the right parse:



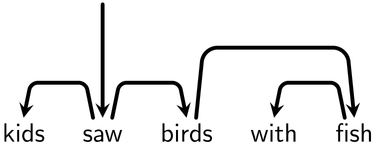
Dependency syntax focuses on the head-dependent relationships.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Dependency syntax

An alternative approach to sentence structure.

- A fully lexicalized formalism: no phrasal categories.
- Assumes *binary, asymmetric* grammatical relations between words: **head-dependent** relations, shown as directed edges:

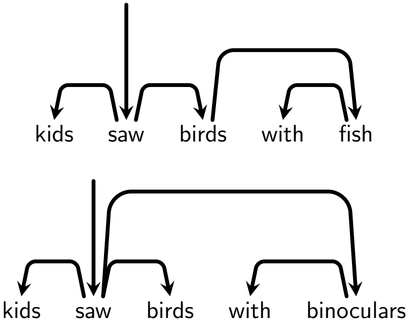


- Here, edges point from heads to their dependents.

Dependency trees

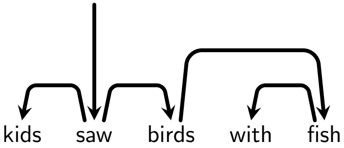
A valid dependency tree for a sentence requires:

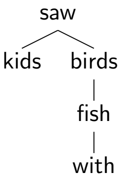
- A single distinguished **root** word.
- All other words have exactly one incoming edge.
- A unique path from the root to each other word.



It really is a tree!

- The usual way to show dependency trees is with edges over ordered sentences.
- But the edge structure (without word order) can also be shown as a more obvious tree:

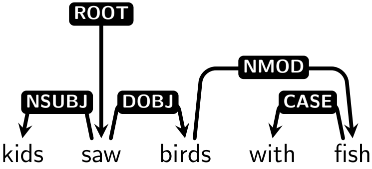




Shay Cohen, Accelerated Natural Language Processing (Slides)

Labelled dependencies

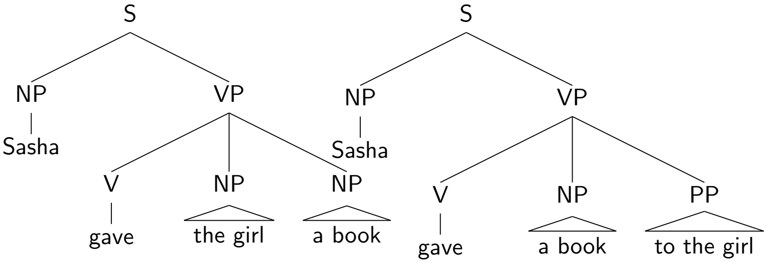
It is often useful to distinguish different kinds of head $\rightarrow$ modifier relations, by labelling edges:



- Historically, different treebanks/languages used different labels.
- Now, the **Universal Dependencies** project aims to standardize labels and annotation conventions, bringing together annotated corpora from over 50 languages.
- Labels in this example (and in textbook) are from UD.

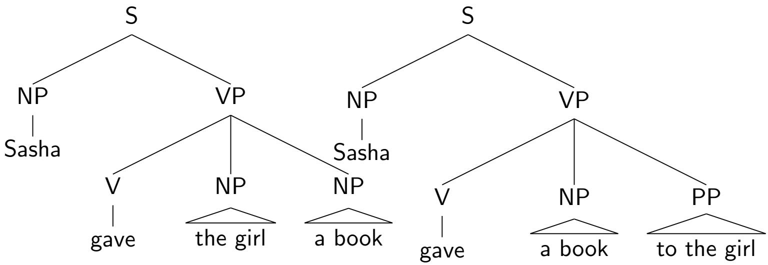
Why dependencies??

Consider these sentences. Two ways to say the same thing:



Why dependencies??

Consider these sentences. Two ways to say the same thing:



- We only need a few phrase structure rules:
  S → NP VP

  VP → V NP NP

  VP → V NP PP

  plus rules for NP and PP.

Equivalent sentences in Russian

- Russian uses morphology to mark relations between words:
  - knigu means book (kniga) as a direct object.
  - devochke means girl (devochka) as indirect object (to the girl).
- So we can have the same word orders as English:
  - Sasha dal devochke knigu
  - Sasha dal knigu devochke

Equivalent sentences in Russian

- Russian uses morphology to mark relations between words:
  - knigu means book (kniga) as a direct object.
  - devochke means girl (devochka) as indirect object (to the girl).
- So we can have the same word orders as English:
  - Sasha dal devochke knigu
  - Sasha dal knigu devochke
- But also many others!
  - Sasha devochke dal knigu
  - Devochke dal Sasha knigu
  - Knigu dal Sasha devochke