# ANLP Lecture 15
# Dependency Syntax and Parsing

Shay Cohen
(based on slides by Sharon Goldwater
and Nathan Schneider)

18 October, 2019

## Last class

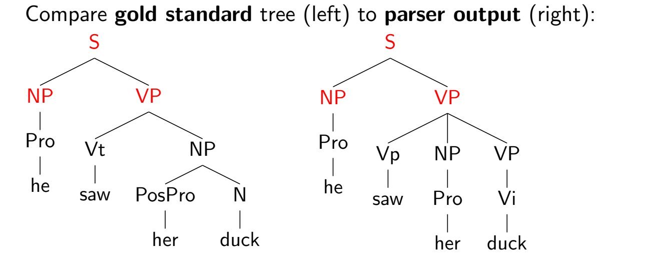
- Probabilistic context-free grammars
- Probabilistic CYK
- Best-first parsing
- Problems with PCFGs (model makes too strong independence assumptions)

## A warm-up question

We described the generative story for PCFGs - pick a rule at random and terminate when choosing a terminal symbol. Does this process have to terminate?

## Evaluating parse accuracy

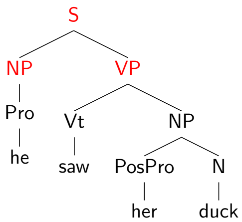
Compare **gold standard** tree (left) to **parser output** (right):

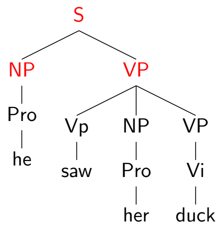



- Output constituent is counted correct if there is a gold constituent that spans the same sentence positions.
- Harsher measure: also require the constituent labels to match.
- Pre-terminals don't count as constituents.

## 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)

## Parsing: where are we now?

- We discussed the basics of probabilistic parsing and you should now have a good idea of the issues involved.
- State-of-the-art parsers address these issues in other ways. For comparison, parsing F-scores on WSJ corpus are:
  - vanilla PCFG: < 80%[1]
  - lexicalizing + cat-splitting: 89.5% (Charniak, 2000)
  - Best current parsers get about 94%
- We'll say a little bit about recent methods later, but most details in sem 2.

[1] Charniak (1996) reports 81% but using gold POS tags as input.

## Parsing: where are we now?

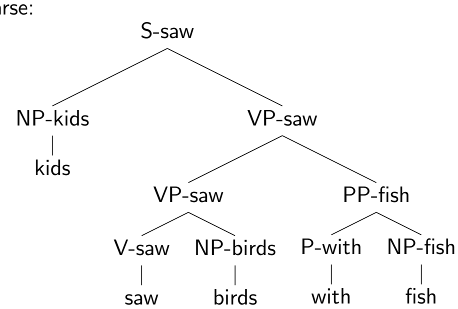
Parsing is not just WSJ. Lots of situations are much harder!

- 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:

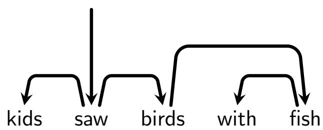



**Dependency syntax** focuses on the head-dependent relationships.

## 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:

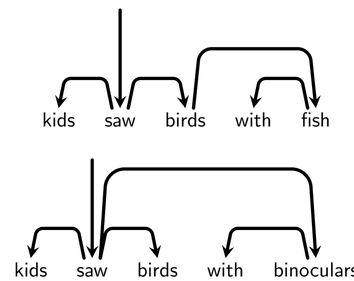



- Here, edges point from heads to their dependents.

## Dependency trees

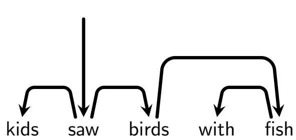
A valid dependency tree for a sentence requires:

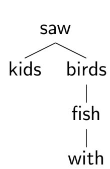
- A single distinguished **root** word.
- All other words have exactly one incoming edge.
- A unique path from the root to each other word.




## It really is a tree!

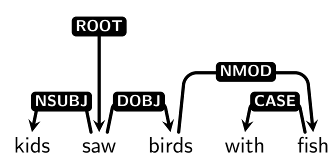
- The usual way to show dependency trees is with edges over ordered sentences.
- But the edge structure (without word order) can also be shown as a more obvious tree:




## Labelled dependencies

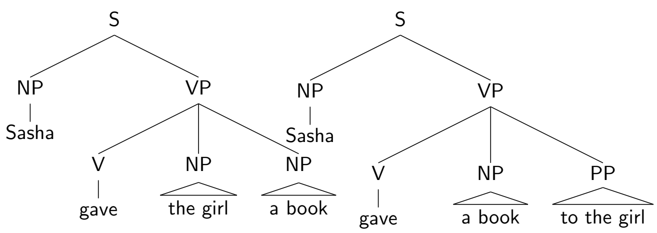
It is often useful to distinguish different kinds of head $\rightarrow$ modifier relations, by labelling edges:




- 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.

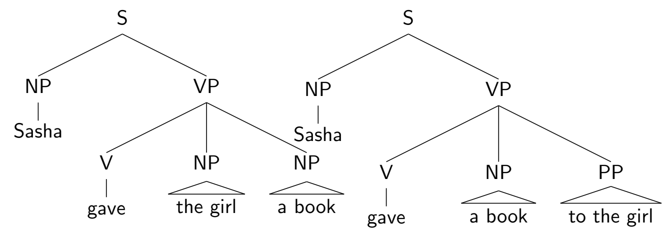
## Why dependencies??

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

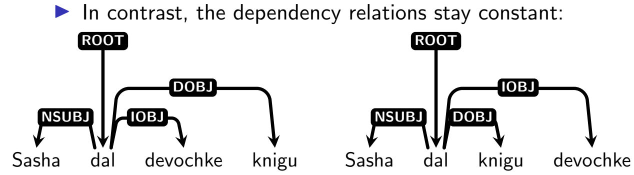
- Russian uses morphology to mark relations between words:
  - knigu means book (kniga) as a direct object.
  - devochke means girl (devochka) as indirect object (to the girl).
- So we can have the same word orders as English:
  - Sasha dal devochke knigu
  - Sasha dal knigu devochke
- But also many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

## Phrase structure vs dependencies

- In languages with **free word order**, phrase structure (constituency) grammars don't make as much sense.
  - E.g., we would need both $\texttt{S} \rightarrow \texttt{NP VP}$ and $\texttt{S} \rightarrow \texttt{VP NP}$, etc. Not very informative about what's really going on.

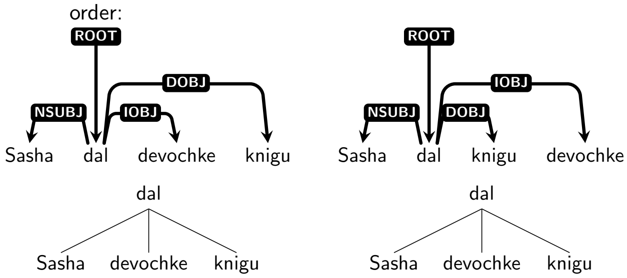
## Phrase structure vs dependencies

- In languages with **free word order**, phrase structure (constituency) grammars don't make as much sense.
  - E.g., we would need both $\texttt{S} \rightarrow \texttt{NP VP}$ and $\texttt{S} \rightarrow \texttt{VP NP}$, etc. Not very informative about what's really going on.
- In contrast, the dependency relations stay constant:




## Phrase structure vs dependencies

- Even more obvious if we just look at the trees without word order:




## Pros and cons

- Sensible framework for free word order languages.
- Identifies syntactic relations directly. (using CFG, how would you identify the subject of a sentence?)
- Dependency pairs/chains can make good features in classifiers, for information extraction, etc.
- Parsers can be very fast (coming up...)

But

- The assumption of asymmetric binary relations isn't always right... e.g., how to parse dogs and cats?

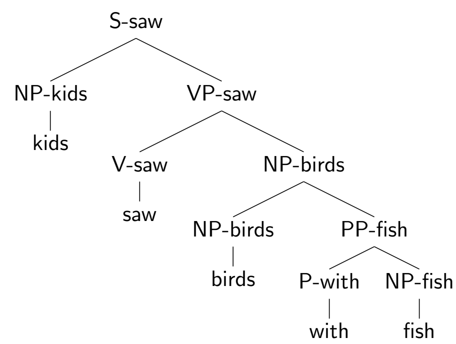
## How do we annotate dependencies?

Two options:

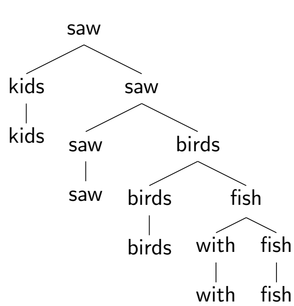
1. Annotate dependencies directly.
2. Convert phrase structure annotations to dependencies. (Convenient if we already have a phrase structure treebank.)

Next slides show how to convert, assuming we have head-finding rules for our phrase structure trees.

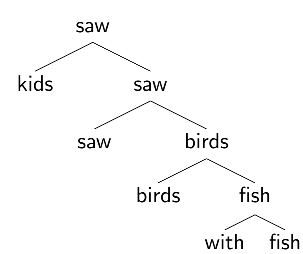
## Lexicalized Constituency Parse



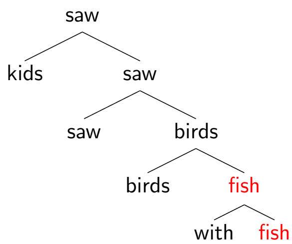

## . . . remove the phrasal categories. . .



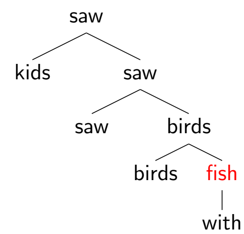

## . . . remove the (duplicated) terminals. . .



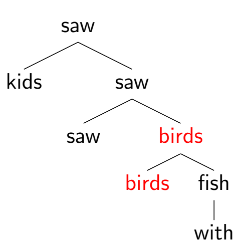
. . . and collapse chains of duplicates. . .



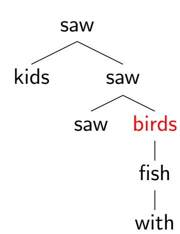

. . . and collapse chains of duplicates. . .



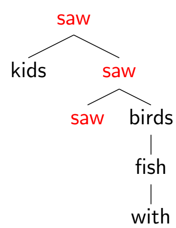

. . . and collapse chains of duplicates. . .



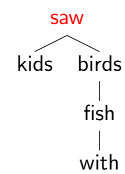

. . . and collapse chains of duplicates. . .



## . . . and collapse chains of duplicates. . .

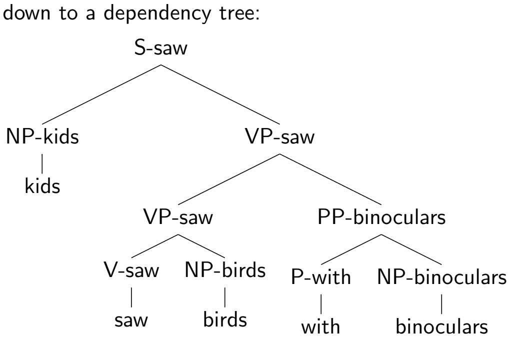



## . . . done!




## Constituency Tree → Dependency Tree

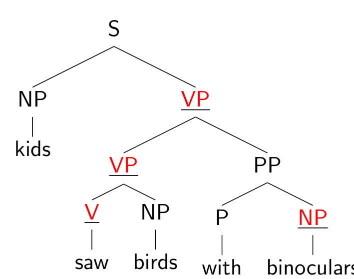
We saw how the **lexical head** of the phrase can be used to collapse down to a dependency tree:




- But how can we find each phrase's head in the first place?

## Head Rules

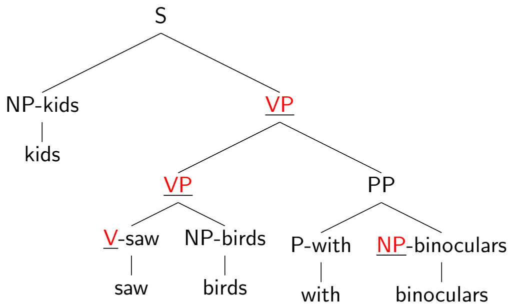
The standard solution is to use **head rules**: for every non-unary (P)CFG production, designate one RHS nonterminal as containing the head. $\texttt{S} \rightarrow \texttt{NP}\ \underline{\texttt{VP}}$, $\texttt{VP} \rightarrow \underline{\texttt{VP}}\ \texttt{PP}$, $\texttt{PP} \rightarrow \texttt{P}\ \underline{\texttt{NP}}$ (content head), etc.




- Heuristics to scale this to large grammars: e.g., within an `NP`, last immediate `N` child is the head.

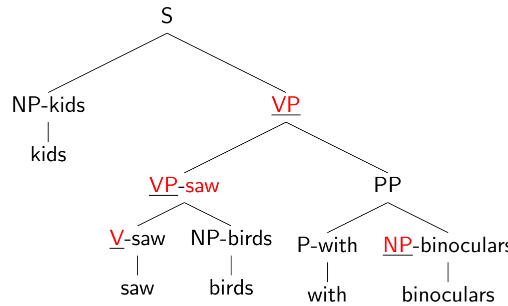
## Head Rules

Then, propagate heads up the tree:



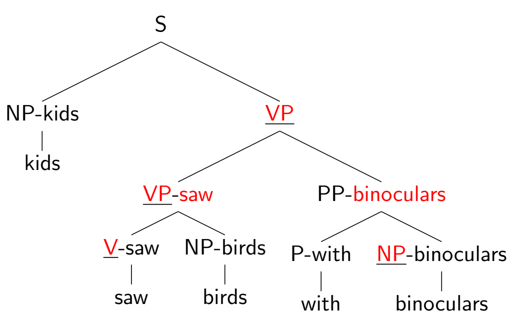

## Head Rules

Then, propagate heads up the tree:



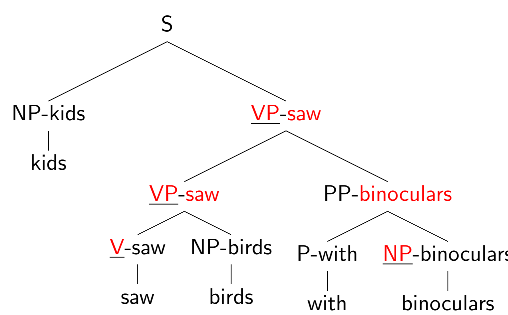

## Head Rules

Then, propagate heads up the tree:



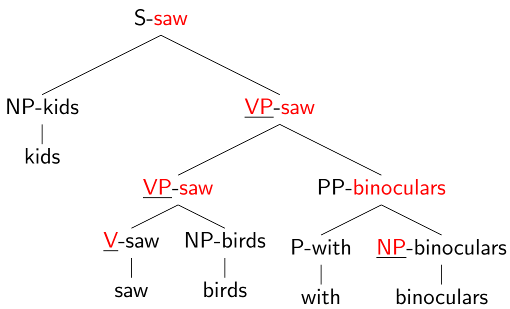

## Head Rules

Then, propagate heads up the tree:



## Head Rules

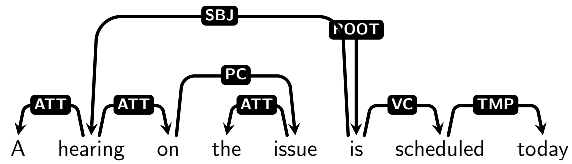
Then, propagate heads up the tree:




## Projectivity

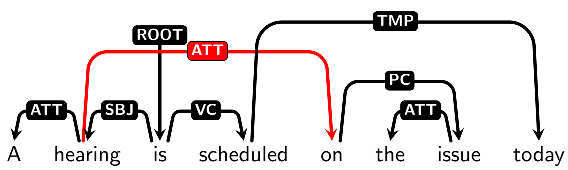
If we convert constituency parses to dependencies, all the resulting trees will be **projective**.

- Every subtree (node and all its descendants) occupies a *contiguous span* of the sentence.
- = the parse can be drawn over the sentence w/ no crossing edges.




## Nonprojectivity

But some sentences are **nonprojective**.




- We'll only get these annotations right if we directly annotate the sentences (or correct the converted parses).
- Nonprojectivity is rare in English, but common in many languages.
- Nonprojectivity presents problems for parsing algorithms.

## Dependency Parsing

Some of the algorithms you have seen for PCFGs can be adapted to dependency parsing.

- **CKY** can be adapted, though efficiency is a concern: obvious approach is $O(Gn^5)$; Eisner algorithm brings it down to $O(Gn^3)$
  - N. Smith's slides explaining the Eisner algorithm: `http://courses.cs.washington.edu/courses/cse517/16wi/slides/an-dep-slides.pdf`
- **Shift-reduce**: more efficient, doesn't even require a grammar!

## Recall: shift-reduce parser with CFG

- Same example grammar and sentence.
- Operations:
  - Reduce (R)
  - Shift (S)
  - Backtrack to step $n$ (B$n$)
- Note that at 9 and 11 we skipped over backtracking to 7 and 5 respectively as there were actually no choices to be made at those points.

| Step | Op. | Stack | Input |
|---|---|---|---|
| 0 | | | the dog bit |
| 1 | S | `the` | dog bit |
| 2 | R | `DT` | dog bit |
| 3 | S | `DT dog` | bit |
| 4 | R | `DT V` | bit |
| 5 | R | `DT VP` | bit |
| 6 | S | `DT VP bit` | |
| 7 | R | `DT VP V` | |
| 8 | R | `DT VP VP` | |
| 9 | B6 | `DT VP bit` | |
| 10 | R | `DT VP NN` | |
| 11 | B4 | `DT V` | bit |
| 12 | S | `DT V bit` | |
| 13 | R | `DT V V` | |
| 14 | R | `DT V VP` | |
| 15 | B3 | `DT dog` | bit |
| 16 | R | `DT NN` | bit |
| 17 | R | `NP` | bit |
| ... | | | |

## Transition-based Dependency Parsing

The **arc-standard** approach parses input sentence $w_1 \ldots w_N$ using two types of **reduce** actions (three actions altogether):

- **Shift:** Read next word $w_i$ from input and push onto the stack.
- **LeftArc:** Assign head-dependent relation $s_2 \leftarrow s_1$; pop $s_2$
- **RightArc:** Assign head-dependent relation $s_2 \rightarrow s_1$; pop $s_1$

where $s_1$ and $s_2$ are the top and second item on the stack, respectively. (So, $s_2$ *preceded* $s_1$ in the input sentence.)

## Example

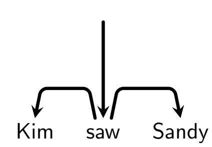
Parsing Kim saw Sandy:

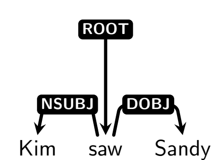
| Step | ←bot. Stacktop→ | Word List | Action | Relations |
|---|---|---|---|---|
| 0 | [root] | [Kim,saw,Sandy] | Shift | |
| 1 | [root,Kim] | [saw,Sandy] | Shift | |
| 2 | [root,Kim,saw] | [Sandy] | LeftArc | Kim←saw |
| 3 | [root,saw] | [Sandy] | Shift | |
| 4 | [root,saw,Sandy] | [] | RightArc | saw→Sandy |
| 5 | [root,saw] | [] | RightArc | root→saw |
| 6 | [root] | [] | (done) | |

- Here, top two words on stack are also always adjacent in sentence. Not true in general! (See longer example in JM3.)

## Labelled dependency parsing

- These parsing actions produce **unlabelled** dependencies (left).
- For **labelled** dependencies (right), just use more actions: LeftArc(NSUBJ), RightArc(NSUBJ), LeftArc(DOBJ), ...



## Differences to constituency parsing

- Shift-reduce parser for CFG: not all sequences of actions lead to valid parses. Choose incorrect action $\rightarrow$ may need to backtrack.
- Here, all valid action sequences lead to valid parses.
  - Invalid actions: can't apply LeftArc with root as dependent; can't apply RightArc with root as head unless input is empty.
  - Other actions may lead to **incorrect** parses, but still **valid**.
- So, parser doesn't backtrack. Instead, tries to greedily predict the correct action at each step.
  - Therefore, dependency parsers can be very fast (linear time).
  - But need a good way to predict correct actions (next lecture).

## Notions of validity

- In constituency parsing, valid parse = grammatical parse.
  - That is, we first define a grammar, then use it for parsing.
- In dependency parsing, we don't normally define a grammar.
  - Valid parses are those with the properties on slide 4.

## Summary: Transition-based Parsing

- **arc-standard** approach is based on simple shift-reduce idea.
- Can do labelled or unlabelled parsing, but need to train a **classifier** to predict next action, as we'll see next time.
- Greedy algorithm means time complexity is linear in sentence length.
- Only finds **projective** trees (without special extensions)
- Pioneering system: Nivre's MaltParser.

## Alternative: Graph-based Parsing

- Global algorithm: From the fully connected directed graph of all possible edges, choose the best ones that form a tree.
- **Edge-factored** models: Classifier assigns a nonnegative score to each possible edge; **maximum spanning tree** algorithm finds the spanning tree with highest total score in $O(n^2)$ time.
- Pioneering work: McDonald's MSTParser
- Can be formulated as constraint-satisfaction with **integer linear programming** (Martins's TurboParser)
- Details in JM3, Ch 16.5 (optional).

## Graph-based vs. Transition-based vs. Conversion-based

- TB: Features in scoring function can look at any part of the stack; no optimality guarantees for search; linear-time; (classically) projective only
- GB: Features in scoring function limited by factorization; optimal search within that model; quadratic-time; no projectivity constraint
- CB: In terms of accuracy, sometimes best to first constituency-parse, then convert to dependencies (e.g., Stanford Parser). Slower than direct methods.

## Choosing a Parser: Criteria

- Target representation: constituency or dependency?
- Efficiency? In practice, both runtime and memory use.
- Incrementality: parse the whole sentence at once, or obtain partial left-to-right analyses/expectations?
- Retrainable system?
- Accuracy?

## Summary

- Constituency syntax: hierarchically nested phrases with categories like `NP`.
- Dependency syntax: trees whose edges connect words in the sentence. Edges often labeled with relations like `nsubj`.
- Can convert constituency to dependency parse using head rules.
- For projective trees, transition-based parsing is very fast and can be very accurate.
- Google "online dependency parser".
  Try out the Stanford parser and SEMAFOR!