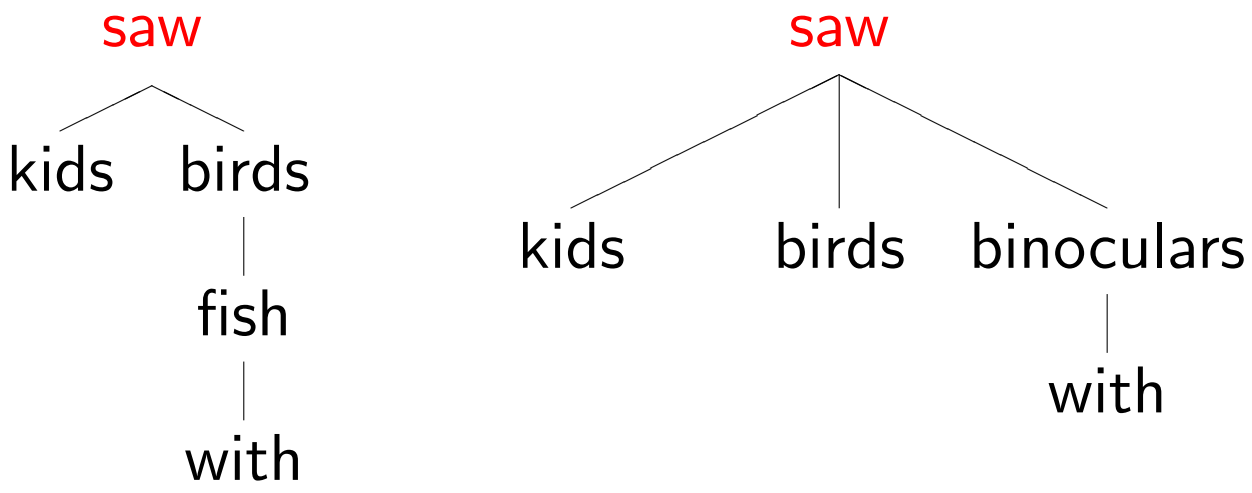
Empirical Methods in Natural Language Processing Lecture 18: Dependency Parsing

(transition-based slides from Harry Eldridge)

3 April 2019

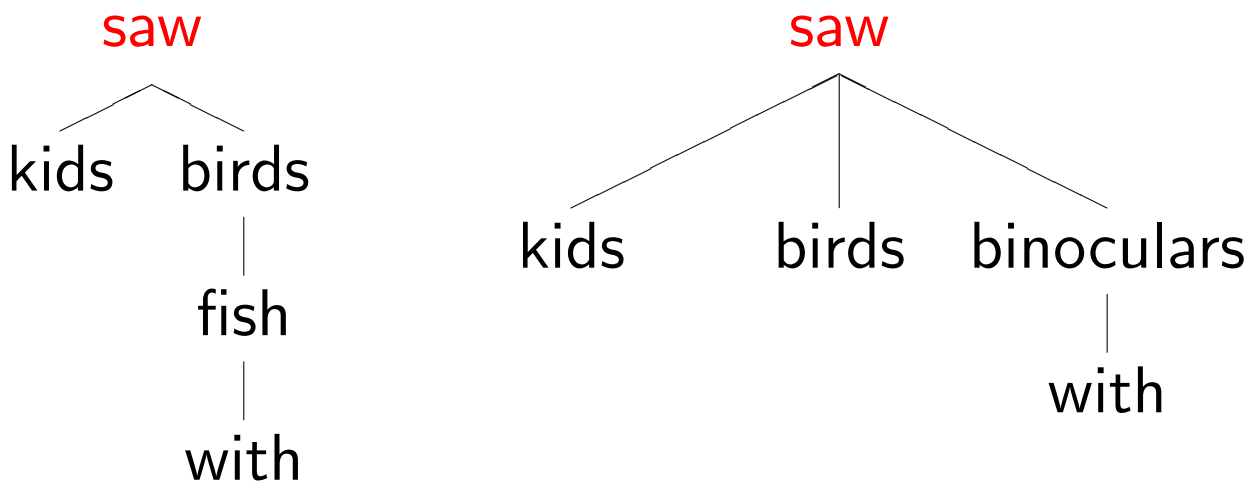
Dependency Parse



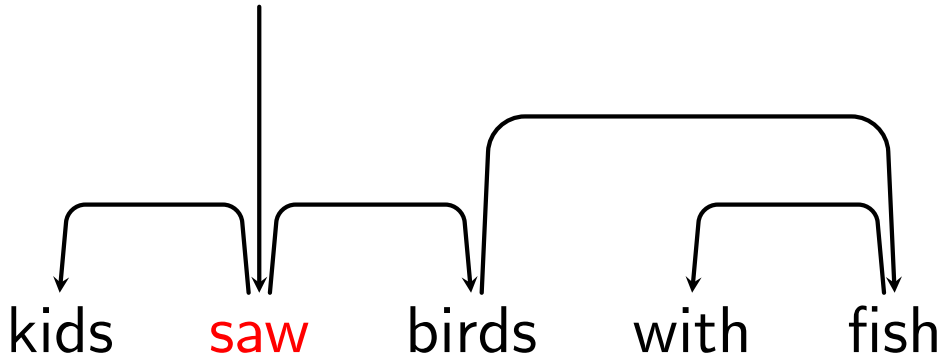
Linguists have long observed that the meanings of words within a sentence depend on one another, mostly in *asymmetric*, *binary* relations.

- Though some constructions don't cleanly fit this pattern: e.g., coordination, relative clauses.

Dependency Parse



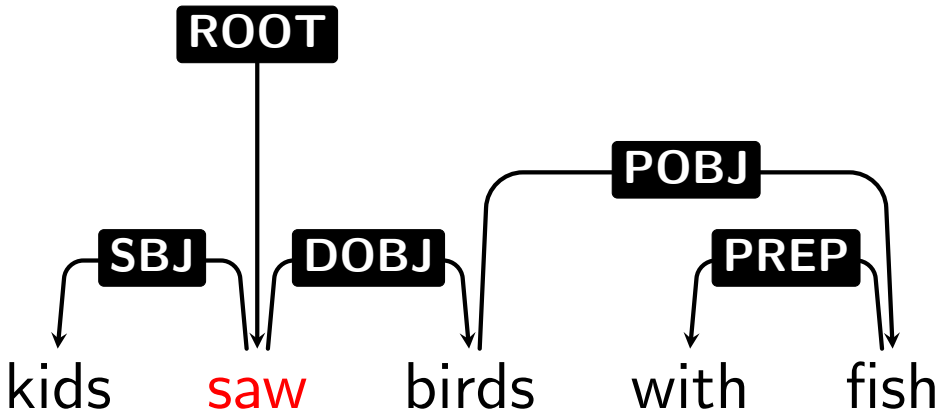
Equivalently, but showing word order (head → modifier):



Because it is a tree, every word has exactly one parent.

Edge Labels

It is often useful to distinguish different kinds of head → modifier **relations**, by labeling edges:

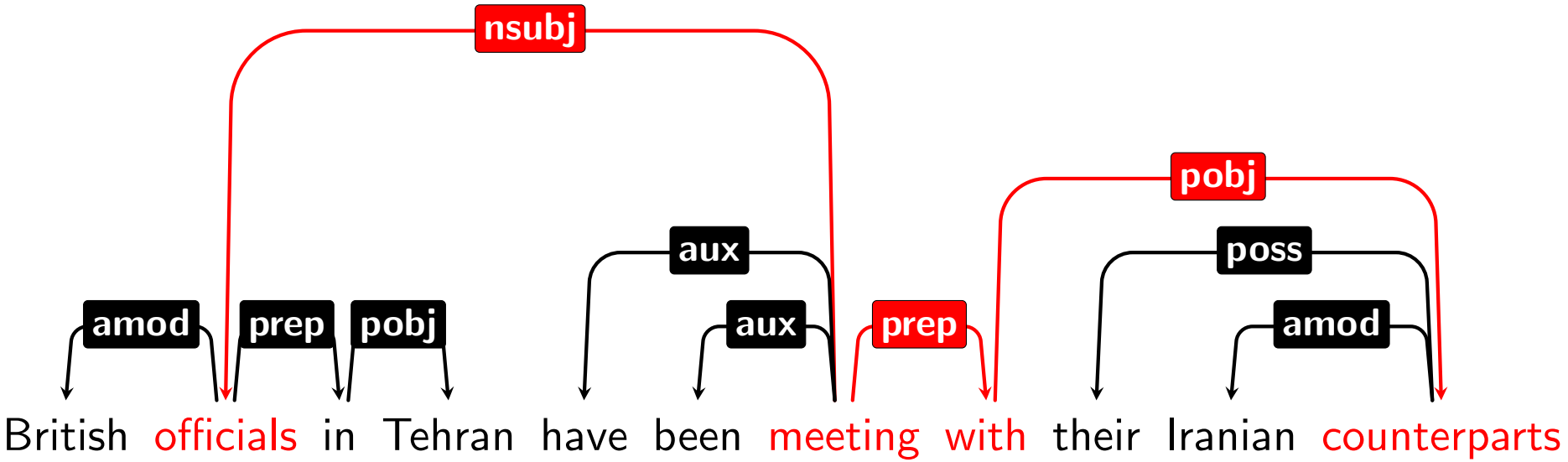


Important relations for English include subject, direct object, determiner, adjective modifier, adverbial modifier, etc. (Different treebanks use somewhat different label sets.)

- How would you identify the subject in a constituency parse?

Dependency Paths

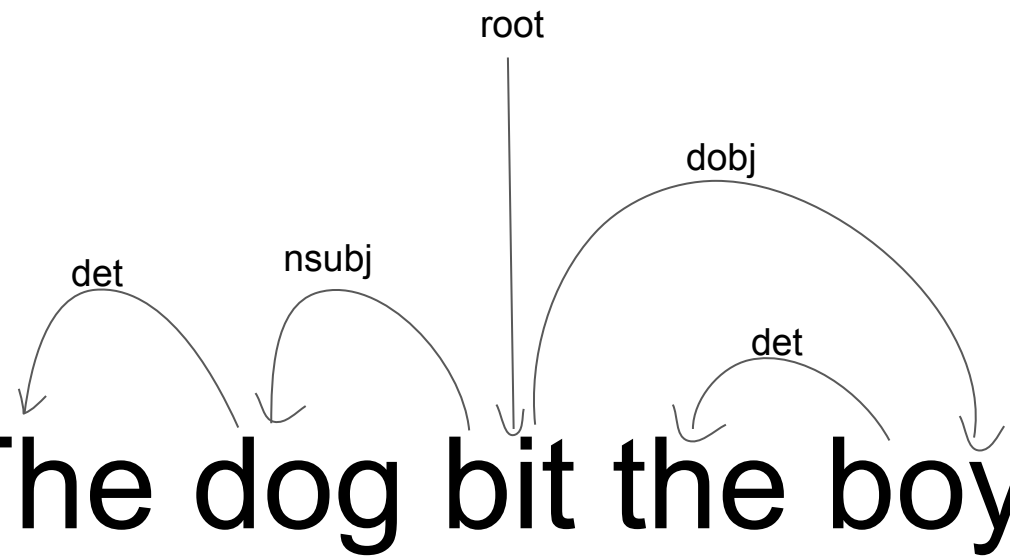
For **information extraction** tasks involving real-world relationships between entities, chains of dependencies can provide good features:



(example from Brendan O'Connor)

A quick dependency parse:

The dog bit the boy


root
dobj
nsubj
det
det
The dog bit the boy

Why is this useful?

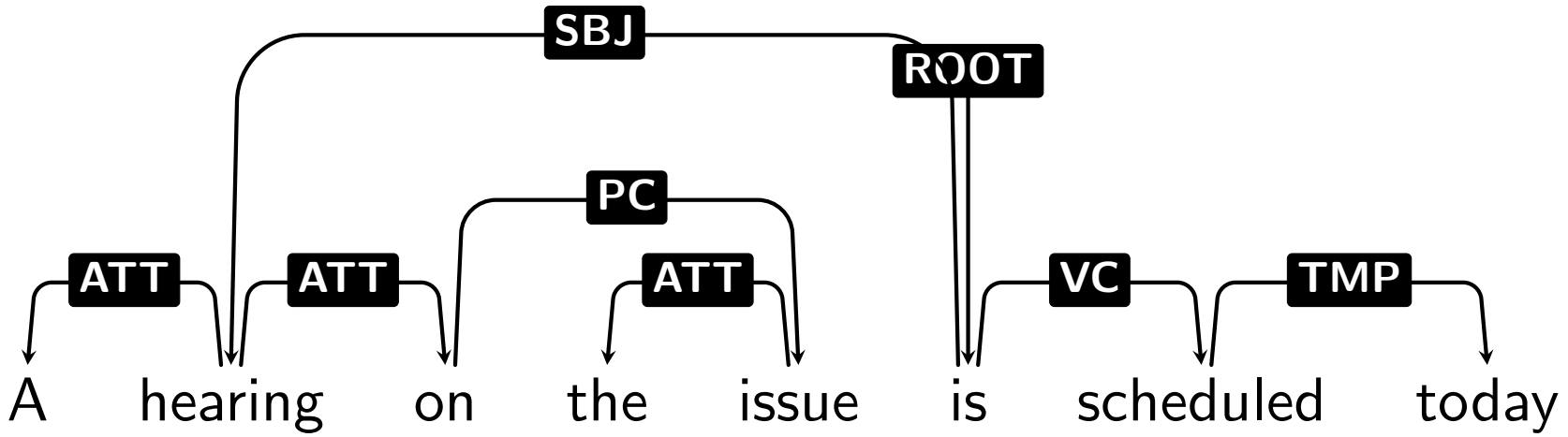
Why is this useful?

- Conveys some level of semantic meaning
- Good for languages with freer word order

Projectivity

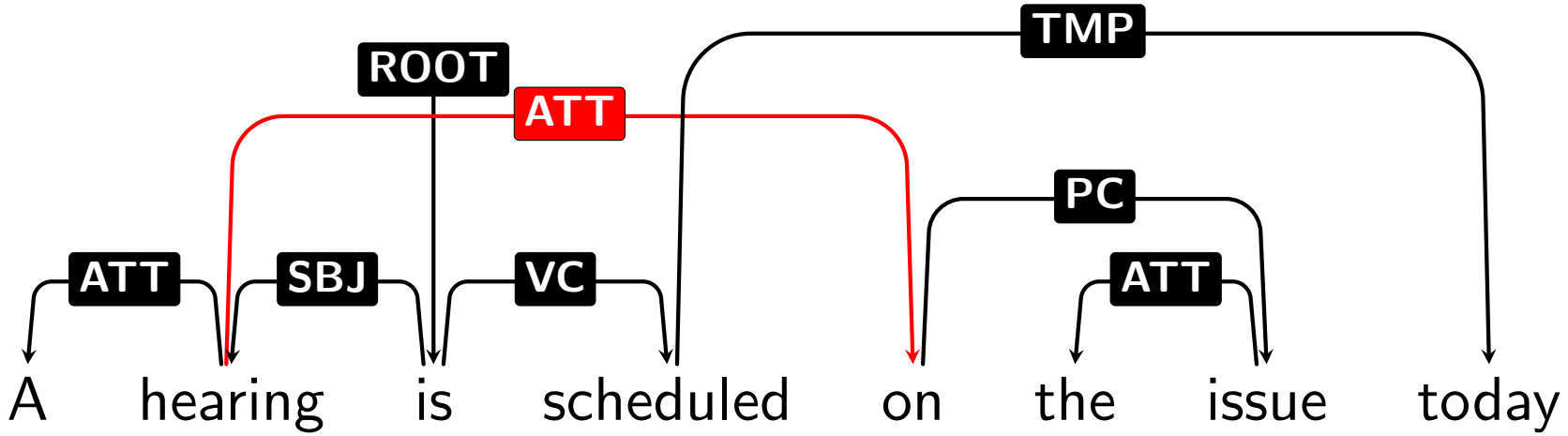
- A sentence's dependency parse is said to be **projective** if every subtree (node and all its descendants) occupies a *contiguous span* of the sentence.

- = The dependency parse can be drawn on top of the sentence without any crossing edges.



Nonprojectivity

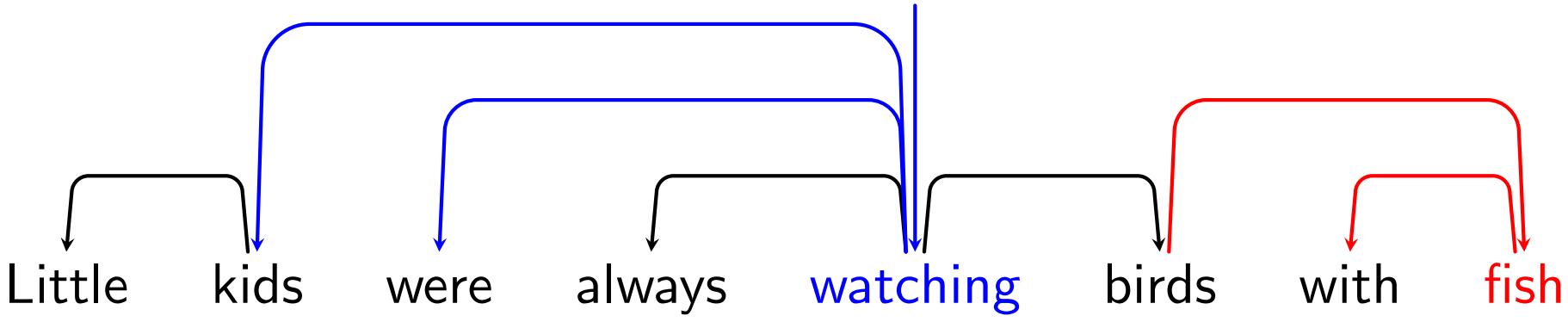
- Other sentences are **nonprojective**:



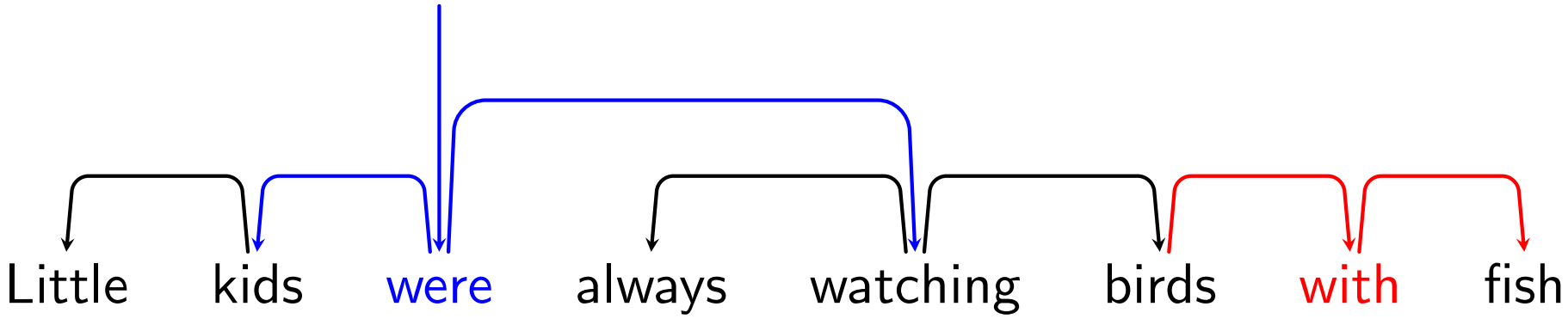
- Nonprojectivity is rare in English, but quite common in many languages.

Content vs. Functional Heads

Some treebanks prefer **content heads**:



Others prefer **functional heads**:



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`

Transition-based Parsing

- Adapts shift-reduce methods: stack and buffer
- Remember: latent structure is just edges between words. Train a **classifier** to predict next action (SHIFT, REDUCE, ATTACH-LEFT, or ATTACH-RIGHT), and proceed left-to-right through the sentence. $O(n)$ time complexity!
- Only finds **projective** trees (without special extensions)
- Pioneering system: Nivre's MaltParser
- See `http://spark-public.s3.amazonaws.com/nlp/slides/Parsing-Dependency.pdf` (Jurafsky & Manning Coursera slides) for details and examples

Transition Based Dependency Parsing

- High level idea
  - Process words from left to right

Transition Based Dependency Parsing

- High level idea
  - Process words from left to right
  - At each stage, decide if two words should be attached

Transition Based Dependency Parsing

- Similar to shift-reduce parsing for programming languages
- 3 components
  - Input buffer (the words of the sentence)
  - Stack (where the words are moved to and manipulated)
  - Dependency relations (the list of relations between words that becomes the dependency parse)
- **Configuration**: some state of the 3 components
- Parsing consists of a sequence of transitions between configurations until all the words have been accounted for
  - The available transitions define the type of approach

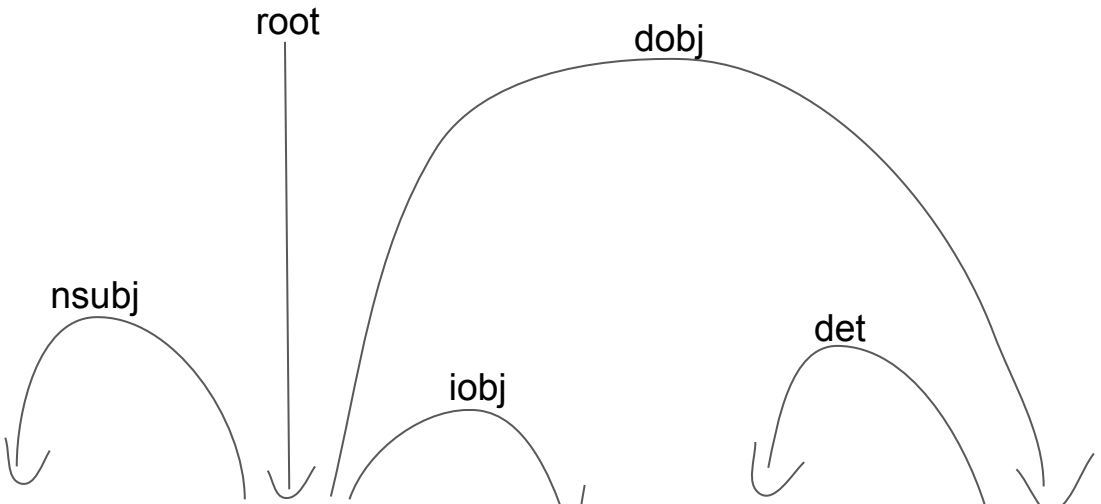
The Arc-Standard Approach

- **LEFTARC**: Assert a head-dependent relation between the word at the top of the stack and the word directly beneath it; remove the lower word from the stack
- **RIGHTARC**: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack
- **SHIFT**: Remove the word from the front of the input buffer and push it onto the stack

Restrictions

- LEFTARC cannot be applied when the root is the second element of the stack (the root cannot be a dependent)
- LEFTARC & RIGHTARC can only be applied if there are 2 or more elements on the stack.

She gave me the book



She gave me the book

STACK
[root]

WORD LIST
[She, gave, me, the, book]

RELATIONS

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	

Operation: SHIFT

STACK

[root]

[root, She]

[root, She, gave]

WORD LIST

[She, gave, me, the, book]

[gave, me, the, book]

[me, the, book]

RELATIONS

Operation: SHIFT

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	(She ← gave)
[root, gave]	[me, the, book]	

Operation: LEFTARC

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	(She ← gave)
[root, gave]	[me, the, book]	(root → gave)
[root]	[me, the, book]	

Operation: RIGHTARC?

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	(She ← gave)
[root, gave]	[me, the, book]	
[root, gave, me]	[the, book]	

Operation: SHIFT!

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	(She ← gave)
[root, gave]	[me, the, book]	
[root, gave, me]	[the, book]	(gave → me)
[root, gave]	[the, book]	

Operation: RIGHTARC

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	(She ← gave)
[root, gave]	[me, the, book]	
[root, gave, me]	[the, book]	(gave → me)
[root, gave]	[the, book]	
[root, gave, the]	[book]	

Operation: SHIFT

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	(She ← gave)
[root, gave]	[me, the, book]	
[root, gave, me]	[the, book]	(gave → me)
[root, gave]	[the, book]	
[root, gave, the]	[book]	
[root, gave, the, book]	[]	

Operation: SHIFT

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	
[root, gave]	[me, the, book]	(She ← gave)
[root, gave, me]	[the, book]	
[root, gave]	[the, book]	(gave → me)
[root, gave, the]	[book]	
[root, gave, the, book]	[]	
[root, gave, book]	[]	(the ← book)

Operation: LEFTARC

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	
[root, gave]	[me, the, book]	(She ← gave)
[root, gave, me]	[the, book]	
[root, gave]	[the, book]	(gave → me)
[root, gave, the]	[book]	
[root, gave, the, book]	[]	
[root, gave, book]	[]	(the ← book)
[root, gave]	[]	(gave → book)

Operation: RIGHTARC

STACK	WORD LIST	RELATIONS
[root]	[She, gave, me, the, book]	
[root, She]	[gave, me, the, book]	
[root, She, gave]	[me, the, book]	
[root, gave]	[me, the, book]	(She ← gave)
[root, gave, me]	[the, book]	
[root, gave]	[the, book]	(gave → me)
[root, gave, the]	[book]	
[root, gave, the, book]	[]	
[root, gave, book]	[]	(the ← book)
[root, gave]	[]	(gave → book)
[root]	[]	(root → gave)

Operation: RIGHTARC

Run time

Run time

- Linear in the size of the sentence

Run time

- Linear in the size of the sentence
- A head decision for each word uniquely defines a tree

How to decide what to do at each step?

How to decide what to do at each step?

- Build an oracle

How to decide what to do at each step?

- Build an oracle with machine learning!
- Need something that maps configurations to transitions

How to decide what to do at each step?

- Build an oracle with machine learning!
- Need something that maps configurations to transitions
- Data comes from Treebanks

How to decide what to do at each step?

- Build an oracle with machine learning!
- Need something that maps configurations to transitions
- Data comes from Treebanks
  - Corpora annotated with gold trees
  - http://universaldependencies.org/

How to decide what to do at each step?

- Build an oracle with machine learning!
- Need something that maps configurations to transitions
- Data comes from Treebanks
  - Corpora annotated with gold trees
  - http://universaldependencies.org/
- Best results have historically come from multinomial logistic regression and SVM models.

How to decide what to do at each step?

- Build an oracle with machine learning!
- Need something that maps configurations to transitions
- Data comes from Treebanks
  - Corpora annotated with gold trees
  - http://universaldependencies.org/
- Best results have historically come from multinomial logistic regression and SVM models.
- Recently, Neural Networks have been performing well

How to decide what to do at each step?

- Build an oracle with machine learning!
- Need something that maps configurations to transitions
- Data comes from Treebanks
  - Corpora annotated with gold trees
  - http://universaldependencies.org/
- Best results have historically come from multinomial logistic regression and SVM models.
- Recently, Neural Networks have been performing well.
  - Naturally lend themselves to the task
    - Forms analysis before reading in the whole sentence
    - Neural networks model a sequence of decisions, which is exactly how the parsing operates

Possible features?

Possible features?

- Some obvious ones, the word currently at the top of the stack, etc.

Possible features?

- Some obvious ones, the word currently at the top of the stack, etc.
- POS tags are also very useful

Possible features?

- Some obvious ones, the word currently at the top of the stack, etc.
- POS tags are also very useful
  - Usually a POS tagged is run and used as input to the dependency parser

Edge Labels

- The example only dealt with connections
- Can modify the oracle to learn and output the transition, as well as the arc label at each step (if RIGHTARC or LEFTARC is called)

Possible Weaknesses?

Possible Weaknesses?

- Can only produce projective parses

Weaknesses of Dependency Parses

Weakness of Dependency Parses

- Head-modifier relation doesn’t always work neatly
- Coordination
  - “Cats **and** dogs ran.”
- Auxiliaries
  - “**Do** you want coffee?”
- Relative clauses
  - “I met the girl **who** started this year”
- Prepositional phrases:
  - “I saw a cow **in** the barn”

Advanced Methods

- Arc Eager transition system

Advanced Methods

- Arc Eager transition system
  - We couldn’t add the arc between root and gave because gave still needed to point to other words
  - In general, the longer a word has to wait to get assigned its head the more opportunities there are for something to go awry

Advanced Methods

- Arc Eager transition system
  - We couldn’t add the arc between root and gave because gave still needed to point to other words
  - In general, the longer a word has to wait to get assigned its head the more opportunities there are for something to go awry
  - Solution: Change the set of operators

New Operators

- LEFTARC: Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack; pop the stack.
- RIGHTARC: Assert a head-dependent relation between the word on the top of the stack and the word at the front of the input buffer; shift the word at the front of the input buffer to the stack.
- SHIFT: Remove the word from the front of the input buffer and push it onto the stack (stays the same).
- REDUCE: Pop the stack.

Advanced Methods

- Arc Eager transition system
  - We couldn’t add the arc between root and gave because gave still needed to point to other words
  - In general, the longer a word has to wait to get assigned its head the more opportunities there are for something to go awry
- Graph based methods
  - Can think of dependency parses as a directed graph with arc labels
  - Other methods use graph based algorithms to find the best dependency parse

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.
  - Edge-factored assumption can be relaxed (higher-order models score larger units; more expensive).
  - Unlabeled parse $\rightarrow$ edge-labeling classifier (pipeline).

- Pioneering work: McDonald's MSTParser

- Can be formulated as constraint-satisfaction with **integer linear programming** (Martins's TurboParser)

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.

Dependency Parsing Evaluation

For training and evaluation, we can automatically convert constituency treebanks (like the Penn Treebank) to dependencies—see below—or we can use dependency treebanks like Universal Dependencies, available in many languages (`http://universaldependencies.org`).

Standard metrics for comparing against a gold standard are:

- **UAS** (unlabeled attachment score): % of words attached correctly (correct head)
- **LAS** (labeled attachment score): % of words attached to the correct head with the correct relation label

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?

Advanced Topic: Relationship between constituency and dependency parses

Constituency parses/grammars can be extended with a notion of **lexical head**, which can

- improve constituency parsing, or
- help convert a constituency parse to a dependency parse

Vanilla PCFGs: no lexical dependencies

Replacing one word with another with the same POS will never result in a different parsing decision, even though it should!

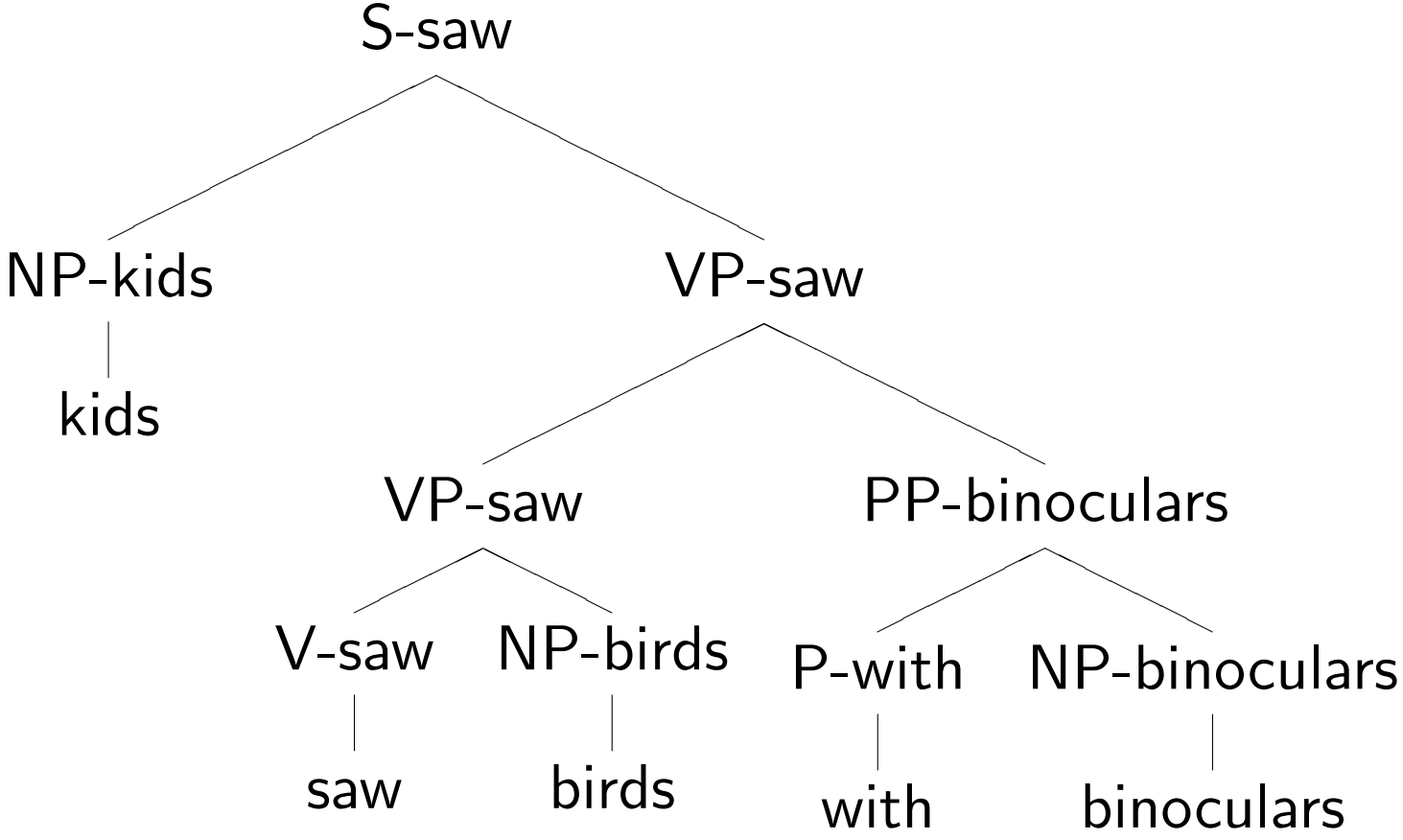
- kids saw birds with fish vs.
  kids saw birds with binoculars

- She stood by the door covered in tears vs.
  She stood by the door covered in ivy

- stray cats and dogs vs.
  Siamese cats and dogs

A way to fix PCFGs: lexicalization

Create new categories, this time by adding the **lexical head** of the phrase (note: N level under NPs not shown for brevity):



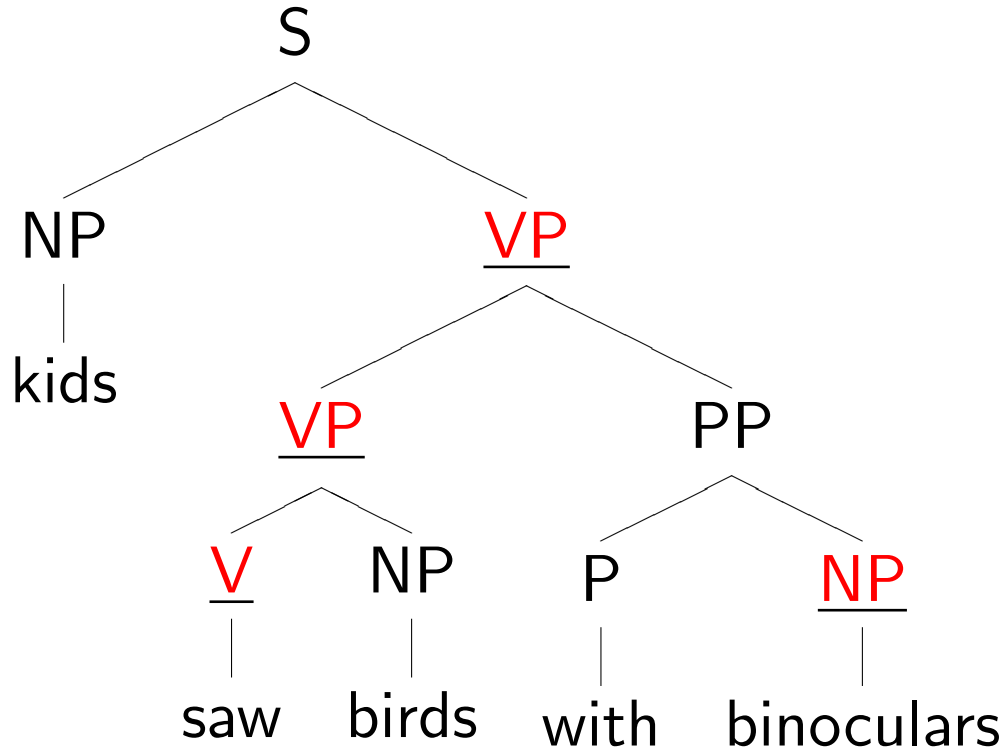
- Now consider:

`VP-saw` → `VP-saw PP-fish` vs. `VP-saw` → `VP-saw PP-binoculars`

How to get lexical heads?

Head Rules

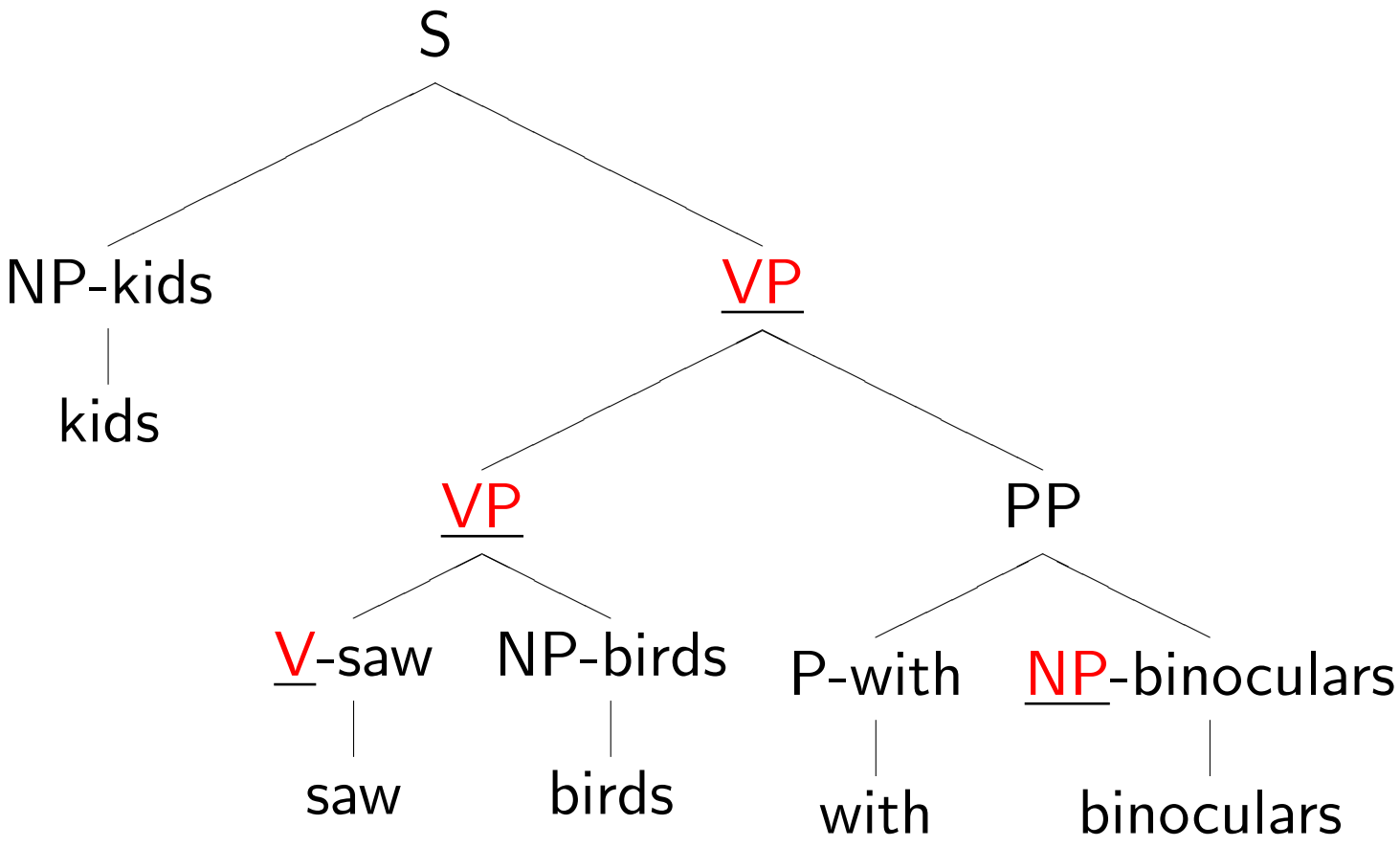
The standard solution is to use **head rules**: for every non-unary (P)CFG production, designate one RHS nonterminal as containing the head. S → NP VP, VP → VP PP, PP → P NP (content head), etc.



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

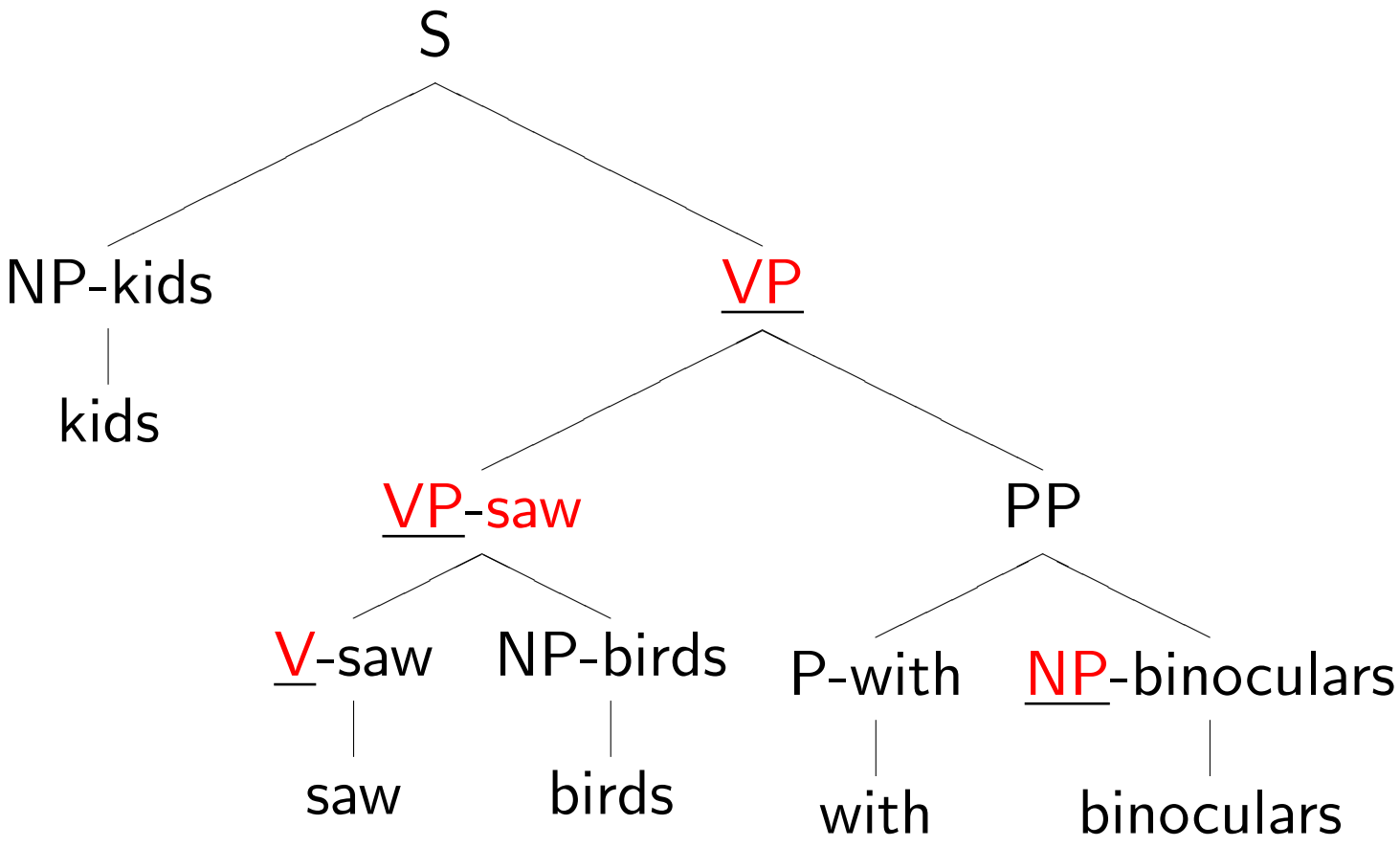
Head Rules

Then, propagate heads up the tree:



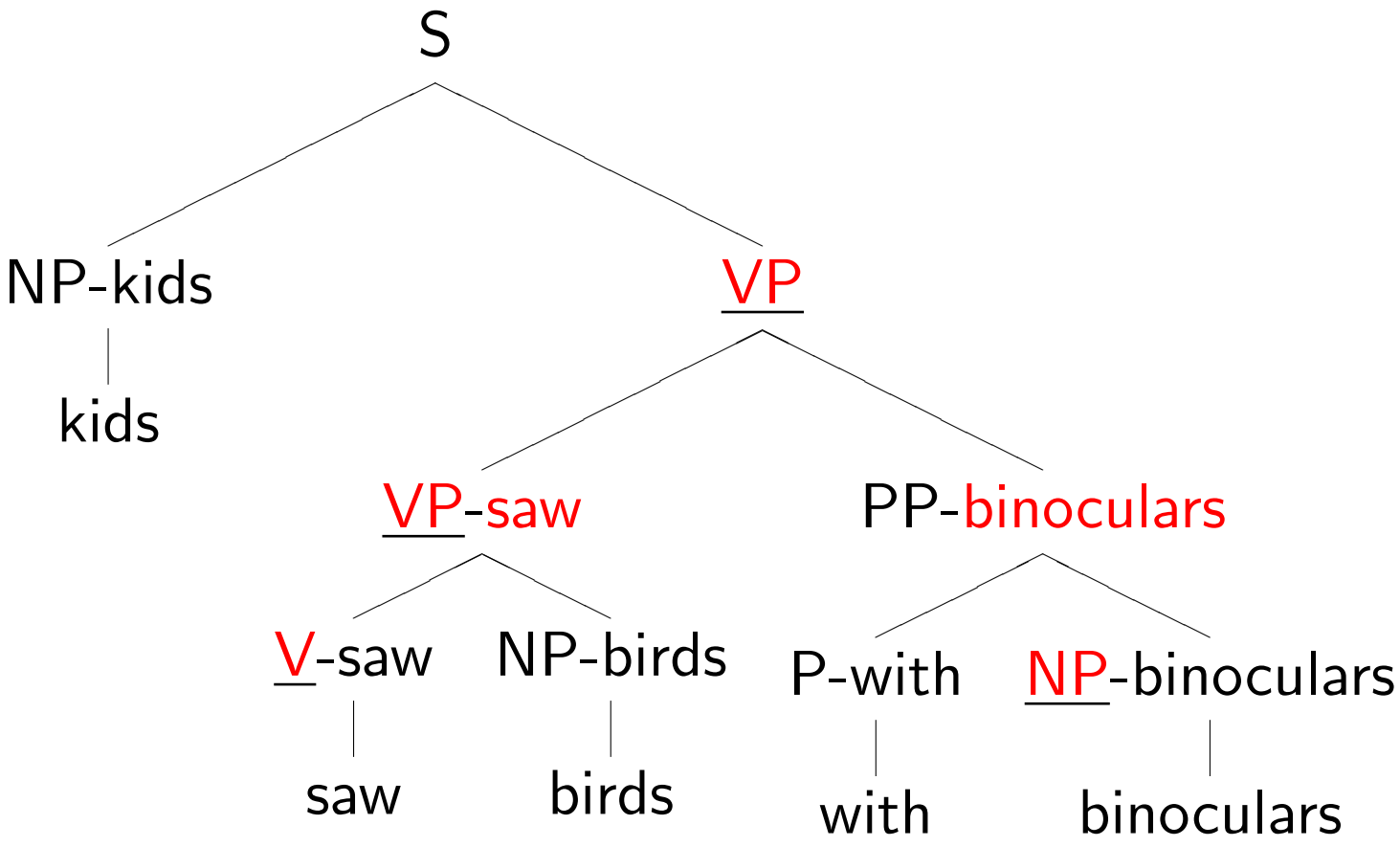
Head Rules

Then, propagate heads up the tree:



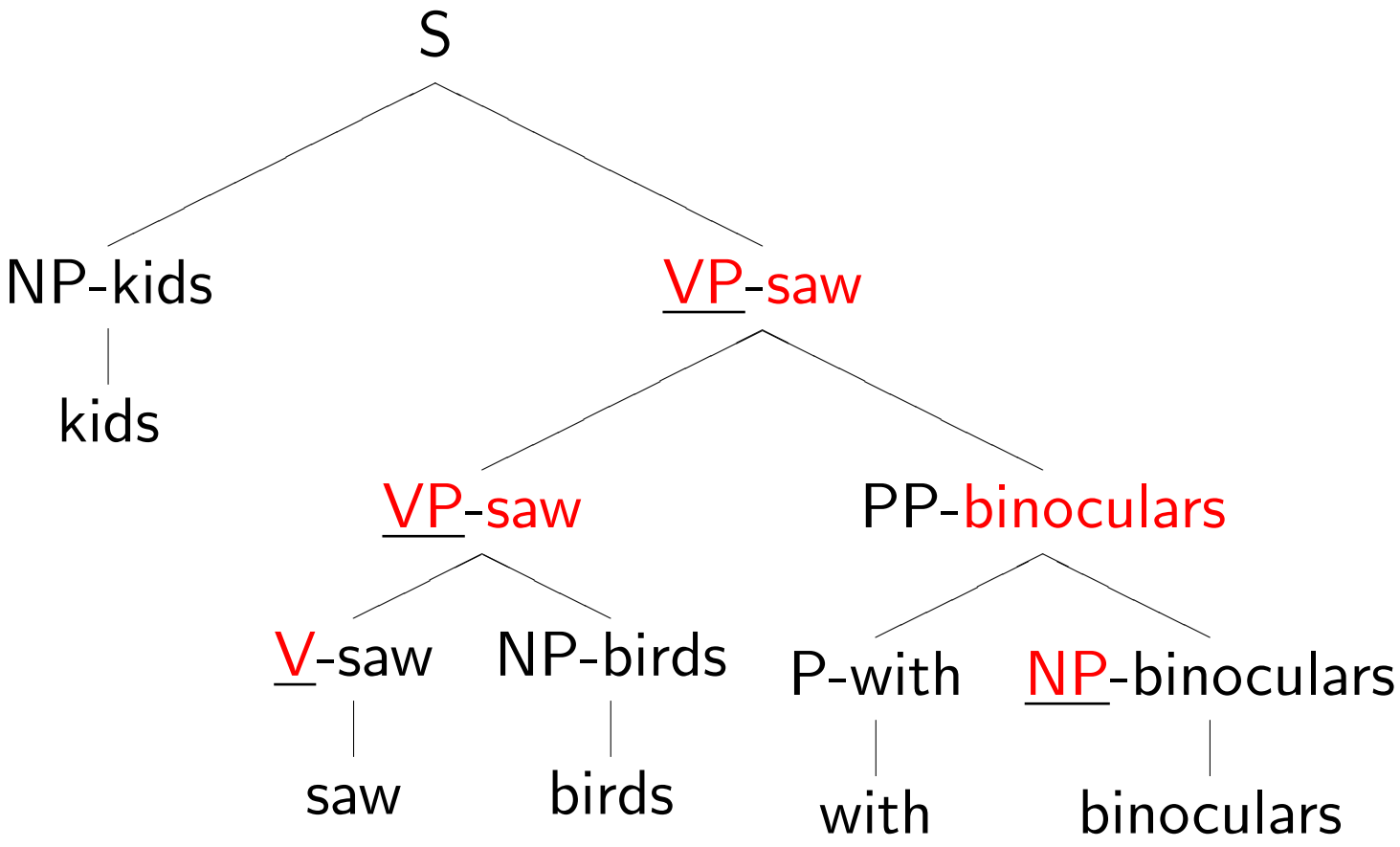
Head Rules

Then, propagate heads up the tree:



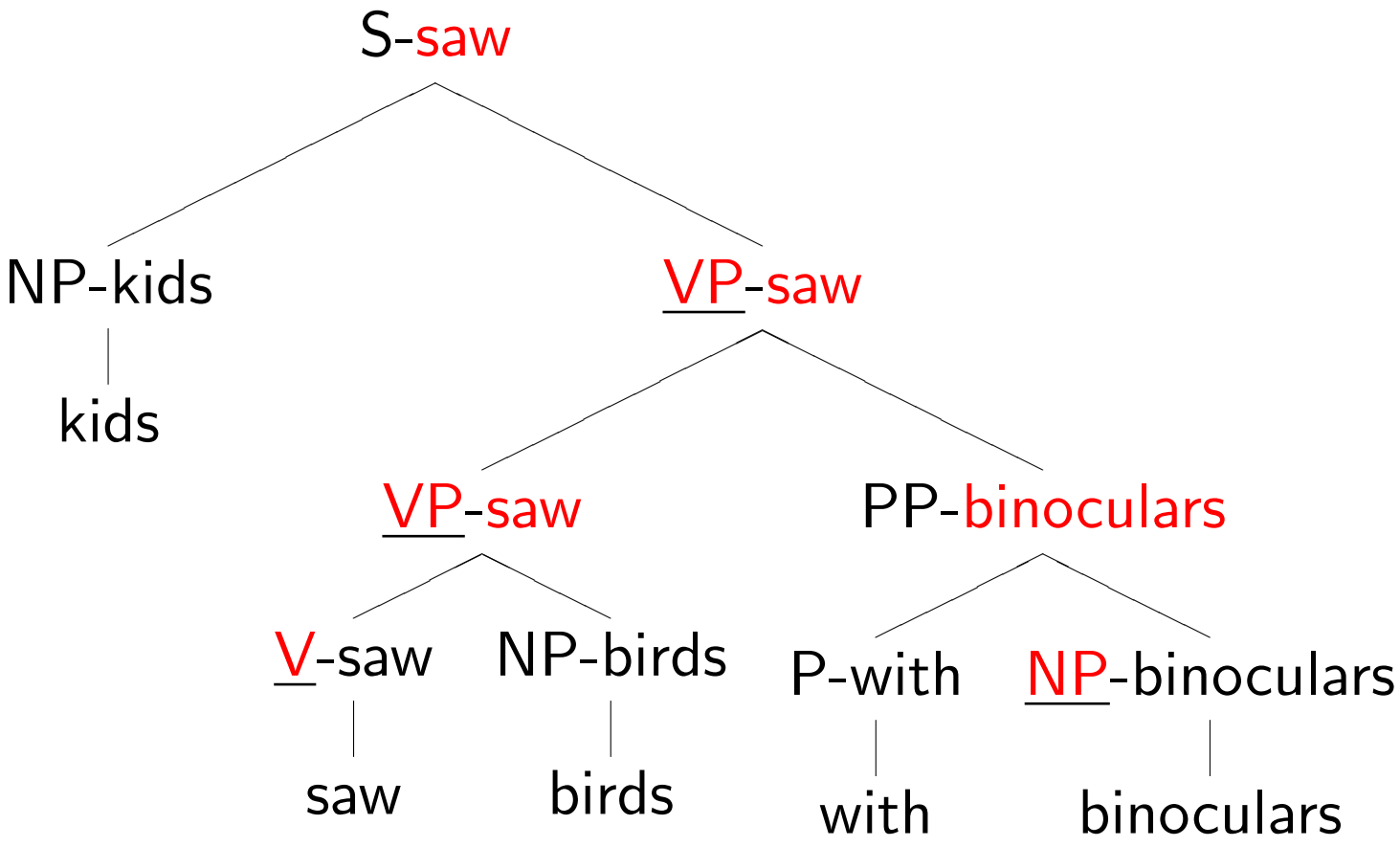
Head Rules

Then, propagate heads up the tree:

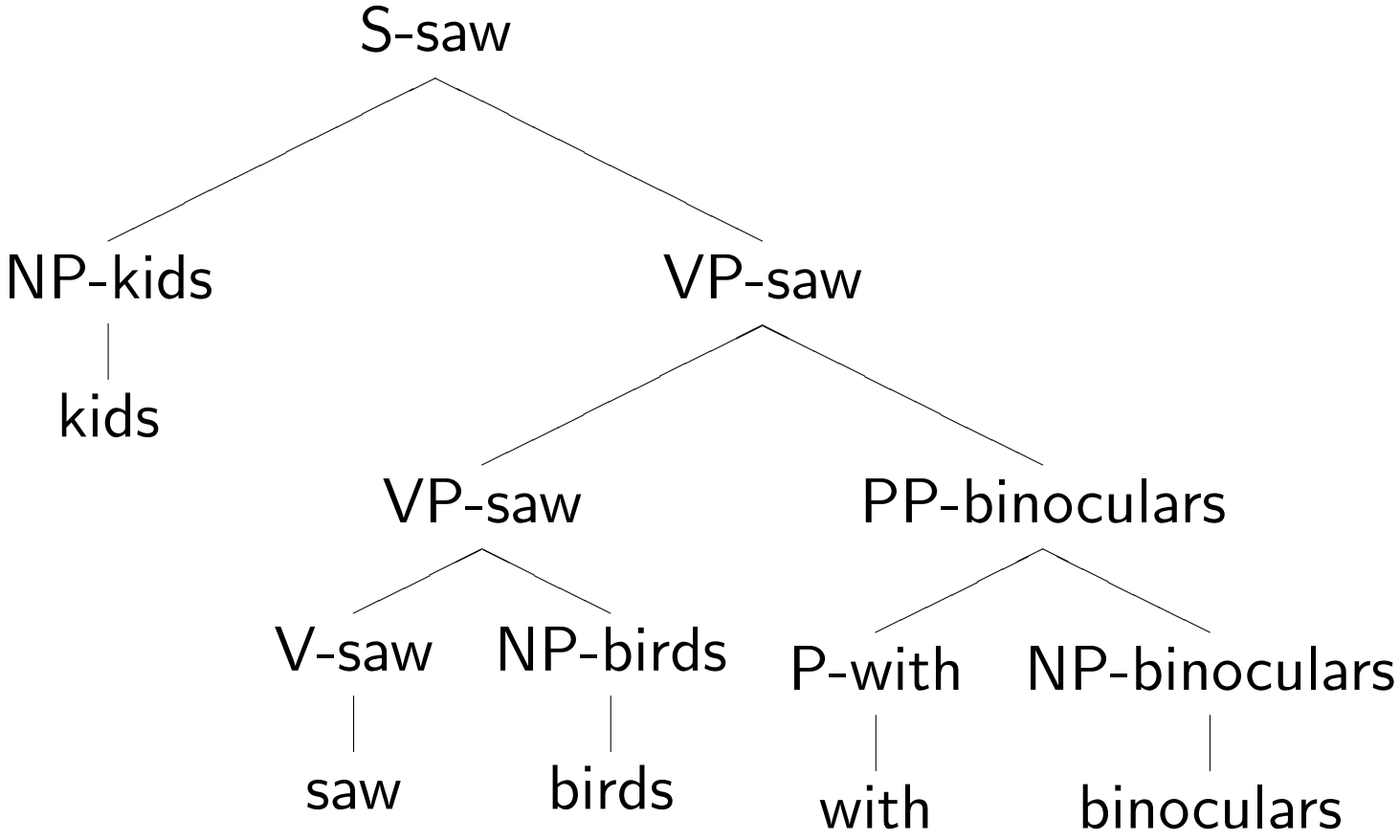


Head Rules

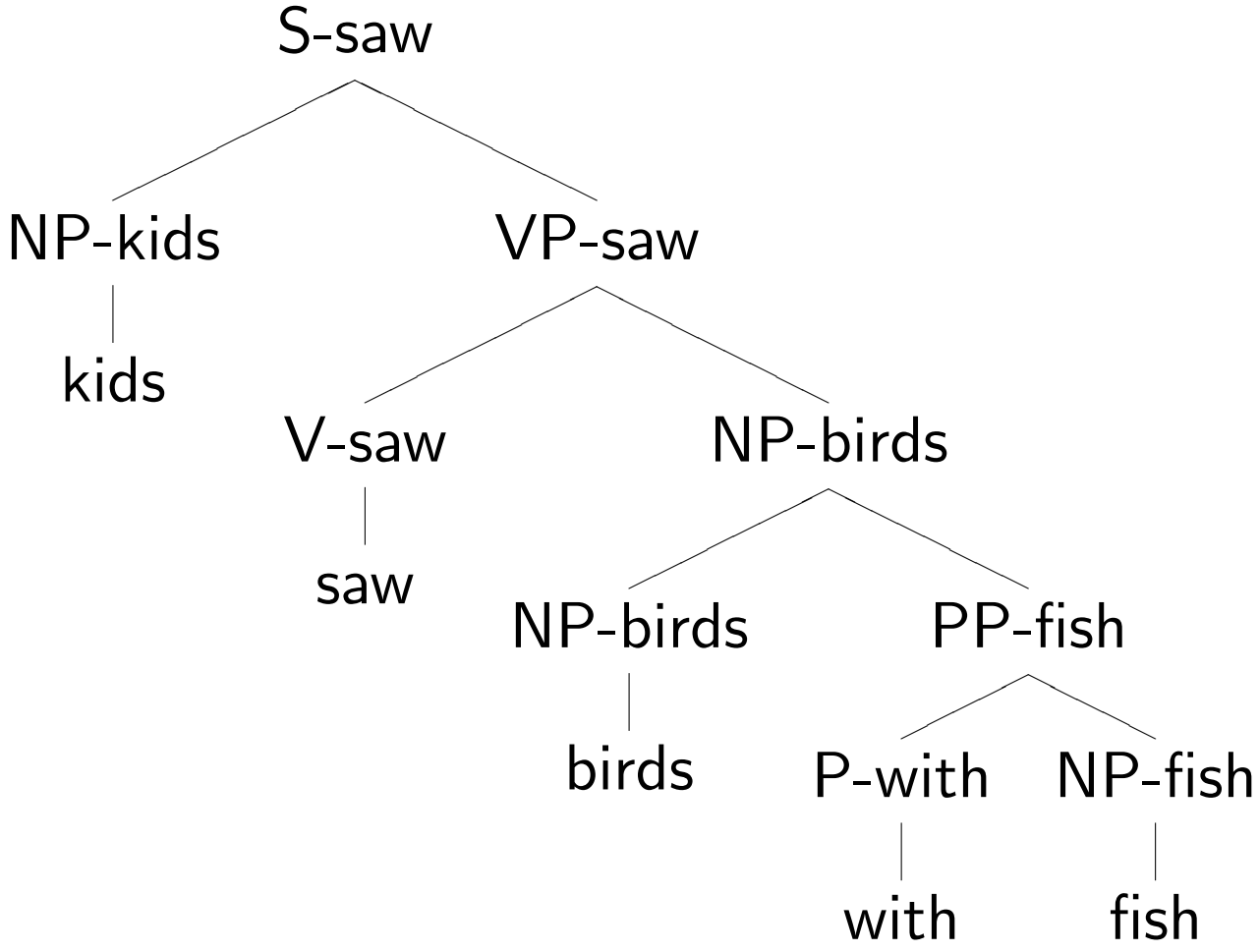
Then, propagate heads up the tree:



Lexicalized Constituency Parse (reading 1)



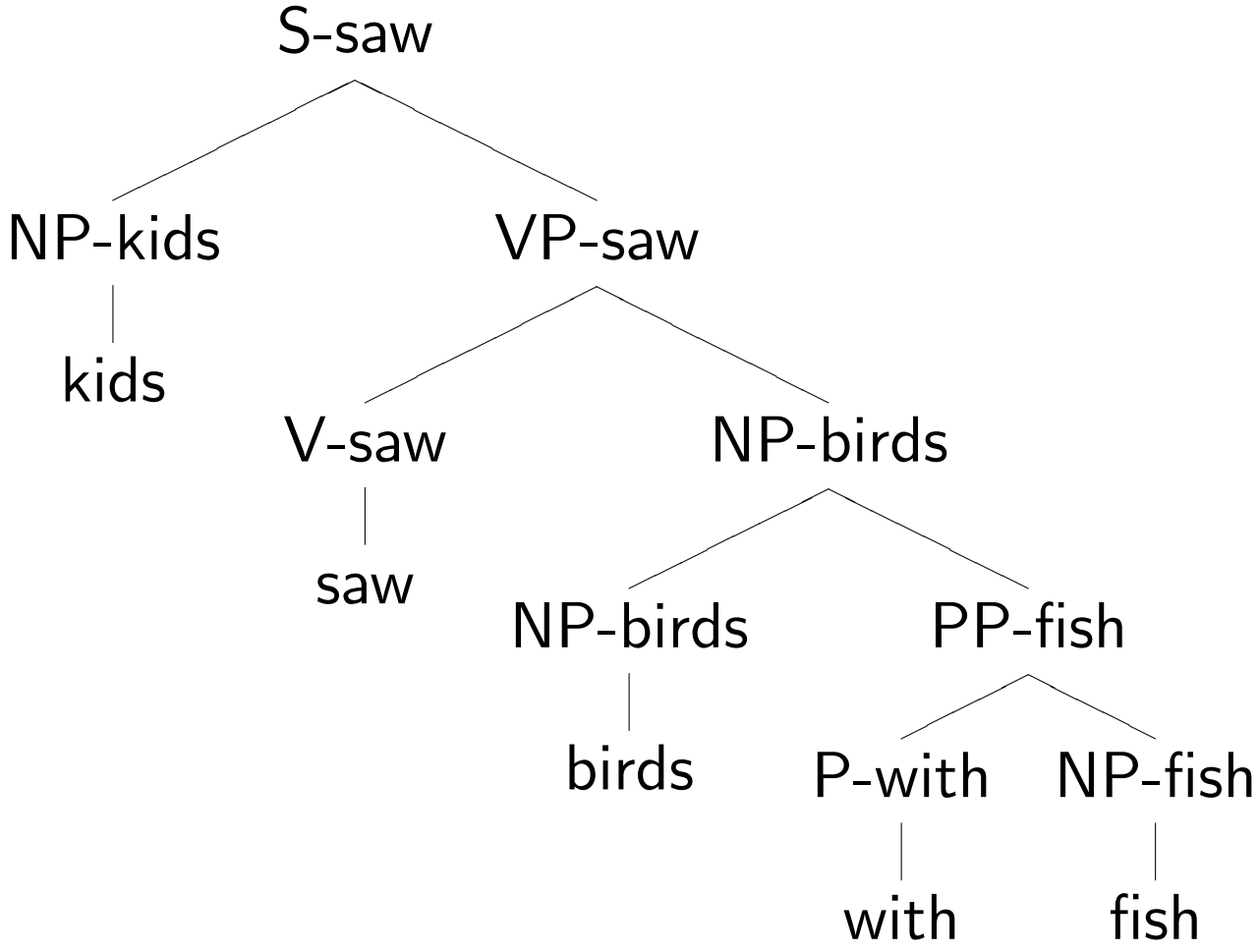
Lexicalized Constituency Parse (reading 2)



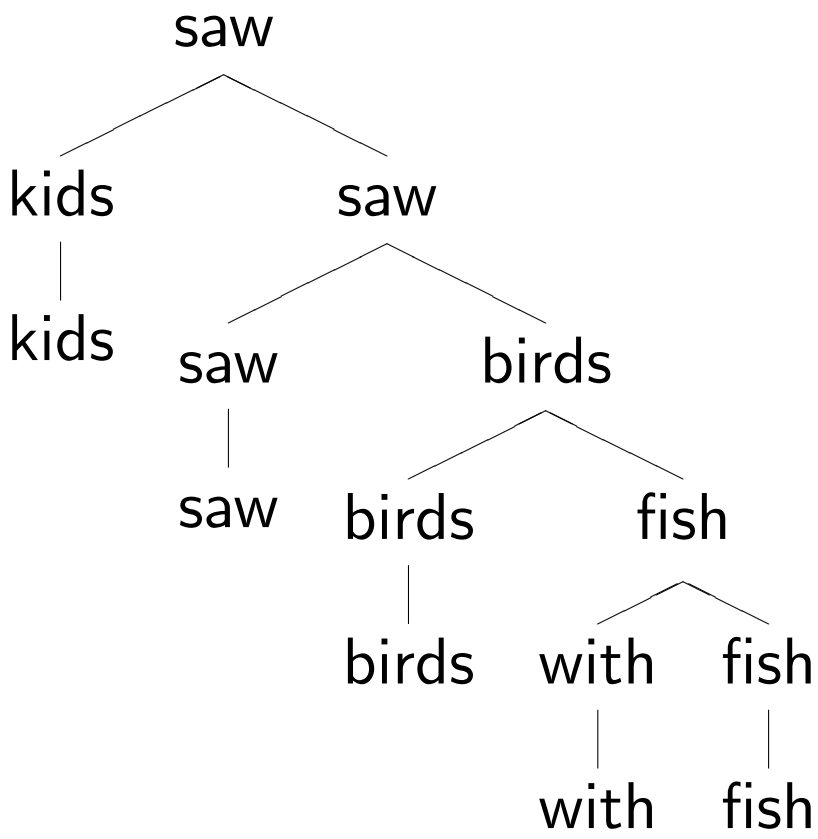
Constituency Tree → Dependency Tree

The lexical heads can then be used to collapse down to an unlabeled dependency tree.

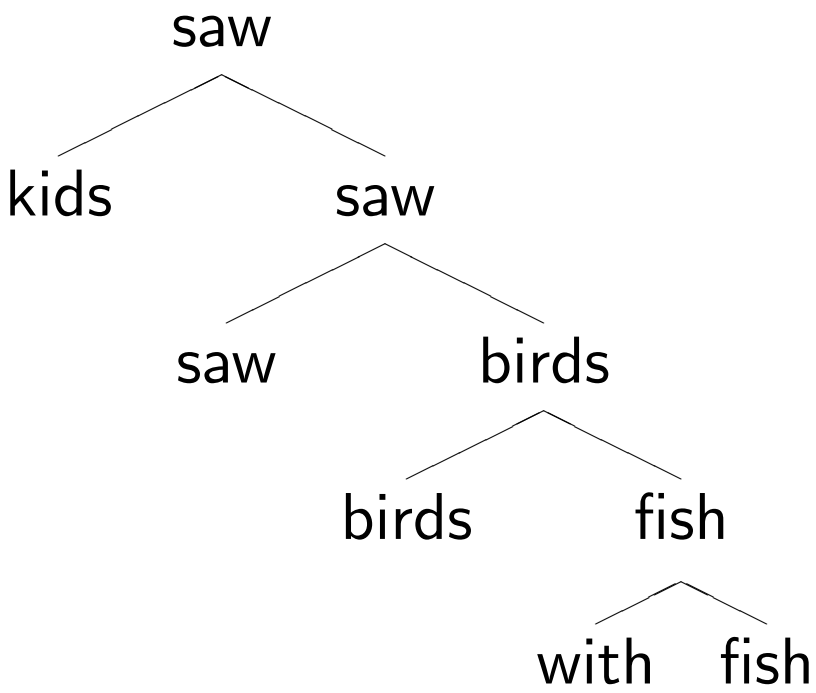
Lexicalized Constituency Parse



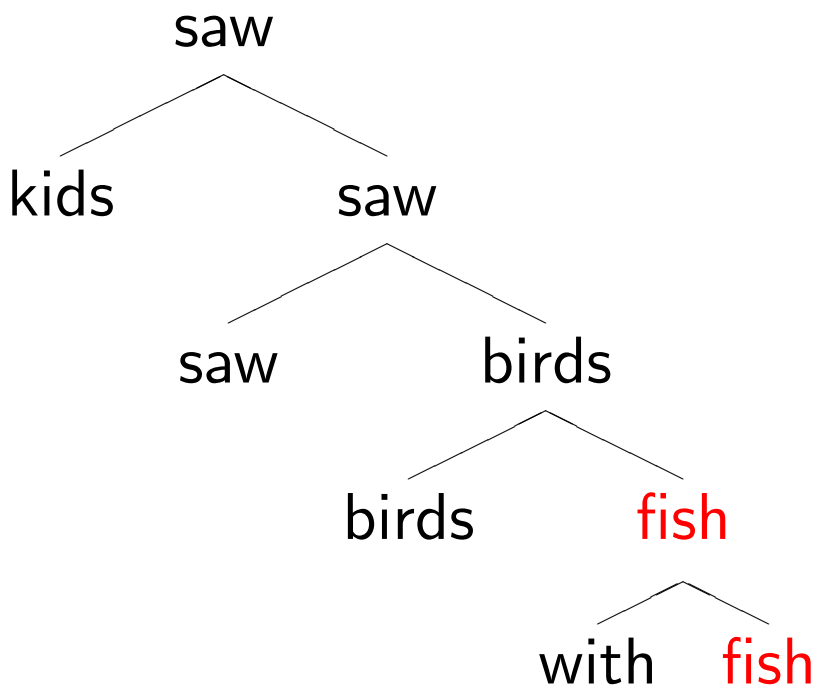
. . . remove the phrasal categories. . .



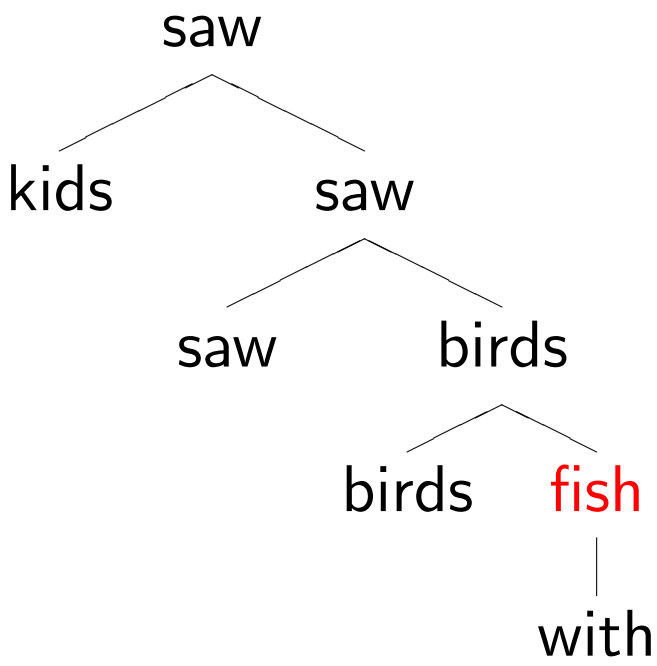
. . . remove the (duplicated) terminals. . .



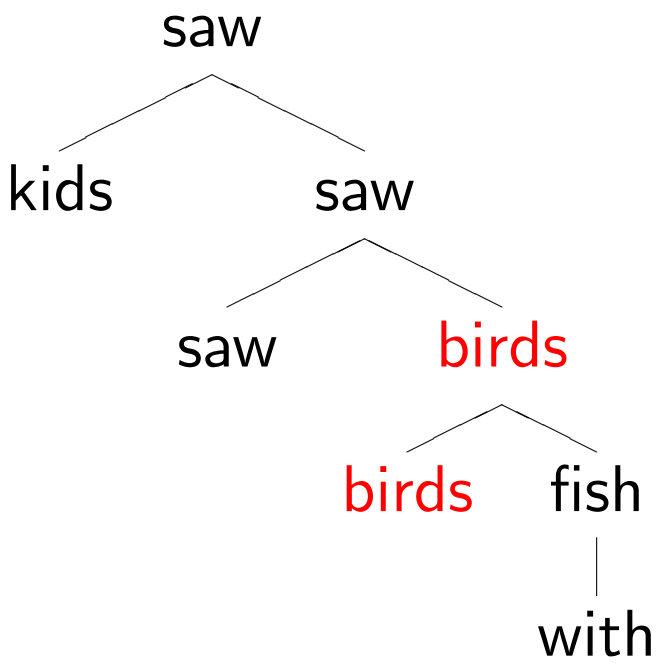
. . . and collapse chains of duplicates. . .



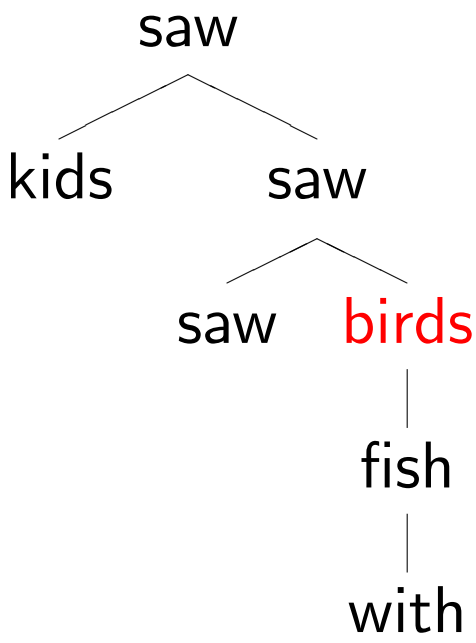
. . . and collapse chains of duplicates. . .



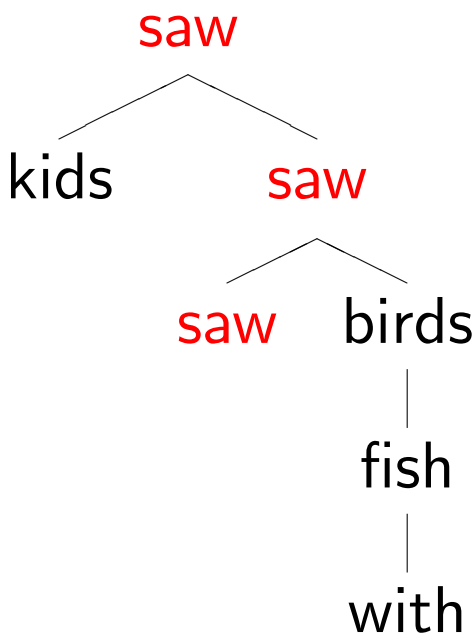
. . . and collapse chains of duplicates. . .



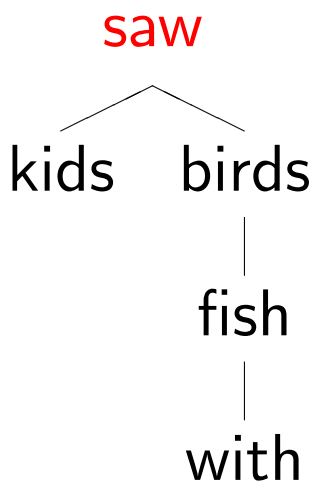
. . . and collapse chains of duplicates. . .



. . . and collapse chains of duplicates. . .



. . . and collapse chains of duplicates. . .



Practicalities of Lexicalized CFG Constituency Parsing

- Leads to huge grammar blowup and very sparse data (bad!)
  - There are fancy techniques to address these issues. . . and they can work pretty well.
  - But: Do we really need phrase structures in the first place? Not always!
- Hence: Sometimes we want to parse directly to dependencies, as with transition-based or graph-based algorithms.

Summary

- While constituency parses give hierarchically nested phrases, dependency parses represent syntax with trees whose edges connect words in the sentence. (No abstract phrase categories like NP.) Edges often labeled with relations like subject.

- Head rules govern how a lexicalized constituency grammar can be extracted from a treebank, and how a constituency parse can be coverted to a dependency parse.

- For English, it is often fastest and most convenient to parse directly to dependencies. Two main paradigms, graph-based and transition-based, with different kinds of models and search algorithms.