Announcement

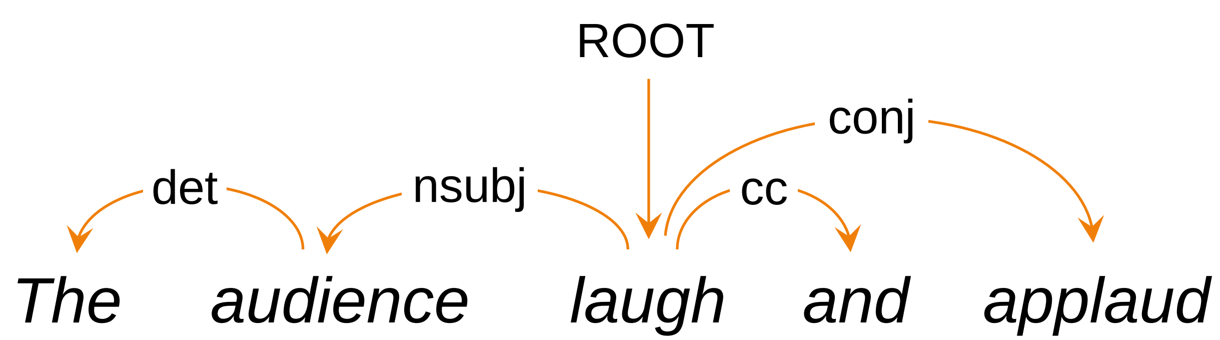
- Homework 5
  - Available in Blackboard -> Homework
  - Due: Apr. 20, 11:59pm

Dependency Parsing

SLP3 Ch 14; INLP Ch 11

Dependency Parse

- A dependency parse is a directed tree where
  - the nodes are the words in a sentence
    - ROOT: a special root node
  - The links between the words represent their dependency relations
    - Typically drawn as a directed arc from head to dependent
    - Dependency arcs may be typed (labeled)



Dependency Relations

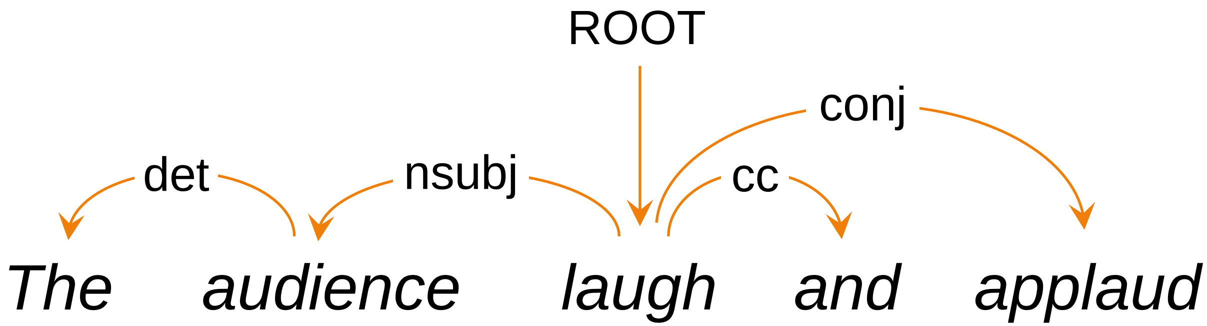
Argument Dependencies	Description
nsubj	nominal subject
csubj	clausal subject
dobj	direct object
iobj	indirect object
pobj	object of preposition
Modifier Dependencies	**Description**
tmod	temporal modifier
appos	appositional modifier
det	determiner
prep	prepositional modifier

Dependency Parsing

- Advantages
  - Deals well with free word order languages where the constituent structure is quite fluid
    - Ex: Czech, Turkish
  - Dependency parses of sentences having the same meaning are more similar across languages than constituency parses
  - Dependency structure often captures the syntactic relations needed by downstream applications
  - Parsing can be faster than CFG-bases parsers
- Disadvantages
  - There is little agreement about what constitutes a dependency relation.
    - In contrast, there are widely agreed-upon tests for constituency.
  - Dependency maps less cleanly to formal semantic representations than constituency

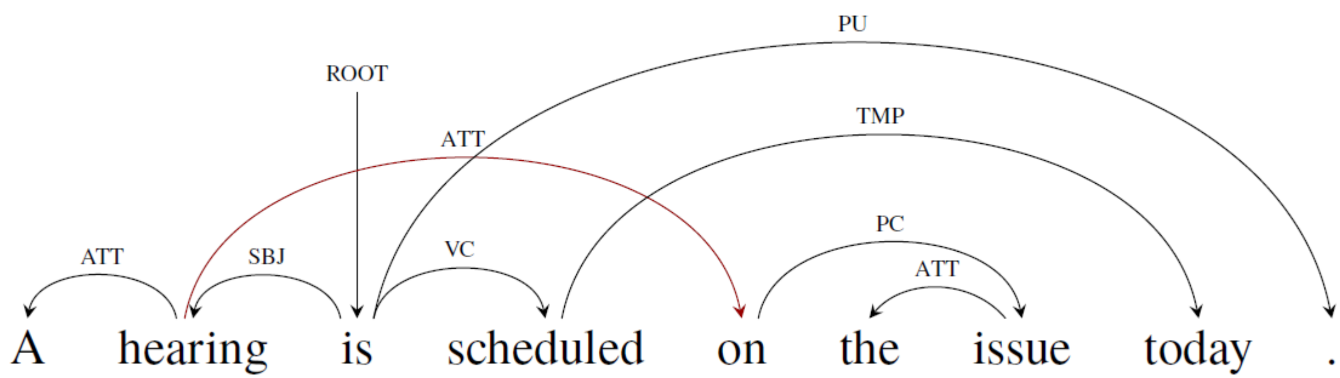
Projectivity

- Projective parse: There are no crossing dependency arcs when the words are laid out in their linear order, with all arcs above the words



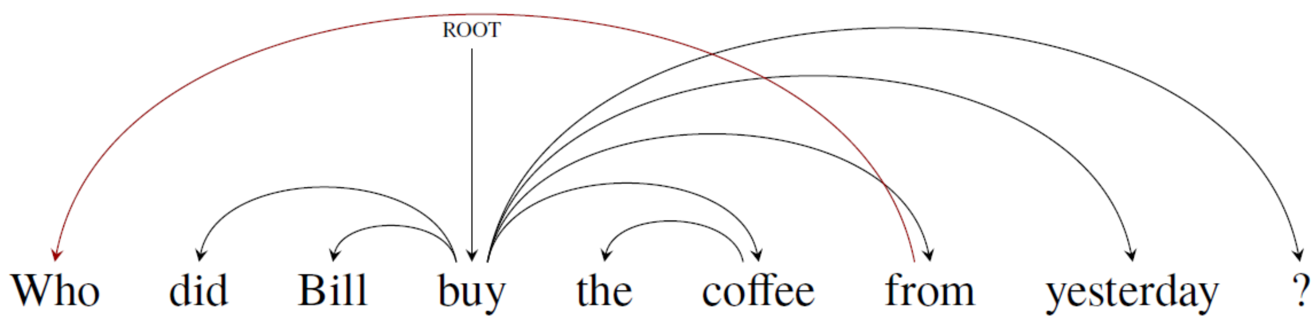
Projectivity

- Projective parse: There are no crossing dependency arcs when the words are laid out in their linear order, with all arcs above the words
- Most syntactic structures are projective, but some are non-projective.



Projectivity

- Projective parse: There are no crossing dependency arcs when the words are laid out in their linear order, with all arcs above the words
- Most syntactic structures are projective, but some are non-projective.

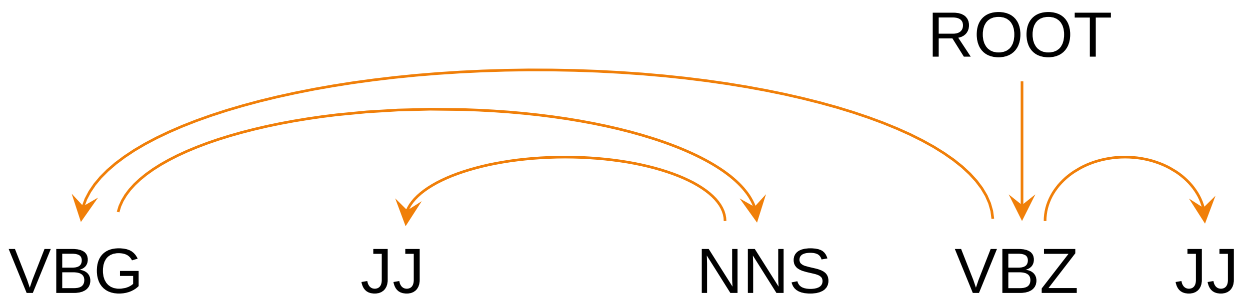


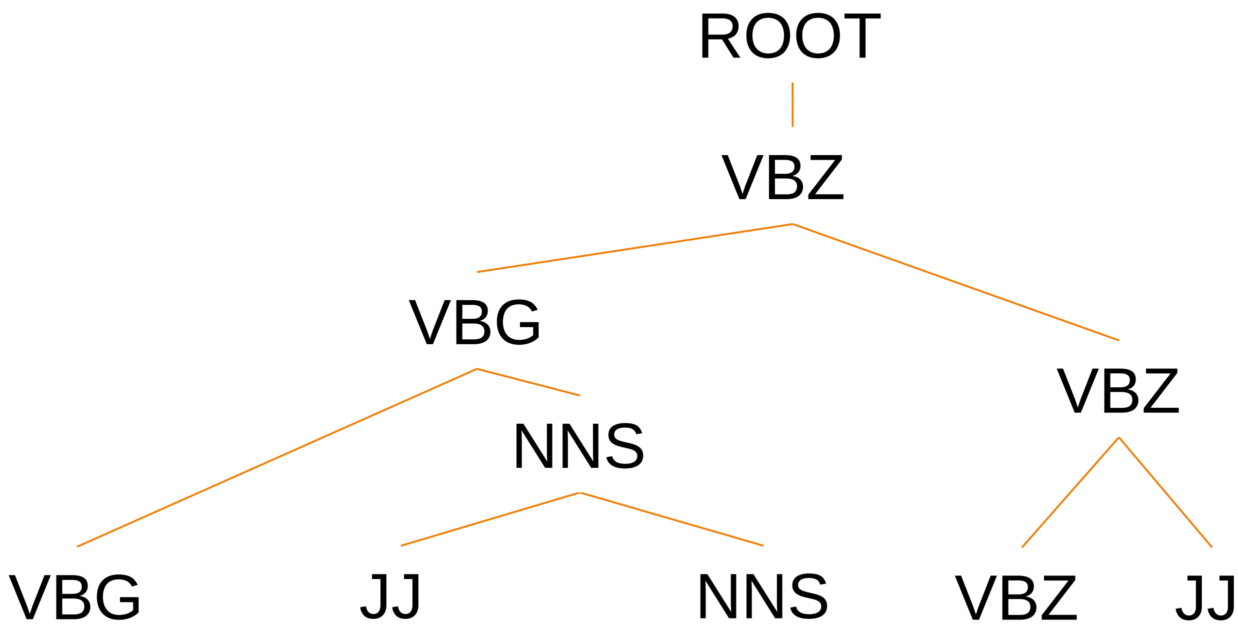
DG to CFG

- Projective dependency parsing can be modeled with a CFG
  - Hence, a projective dependency parse tree can be converted to a constituency parse tree

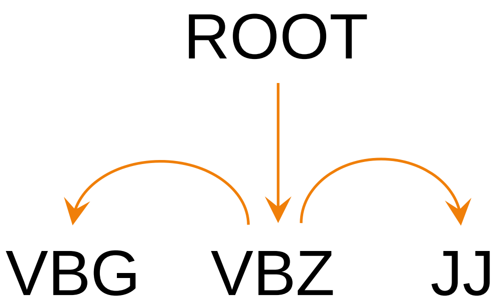
$a \rightarrow b$	$A \rightarrow AB$
$a \rightarrow c$	$A \rightarrow AC$
$d \leftarrow a$	$A \rightarrow DA$
$e \leftarrow a$	$A \rightarrow EA$
	$A \rightarrow a$
	$B \rightarrow b$
	$C \rightarrow c$
	$D \rightarrow d$
	$E \rightarrow e$

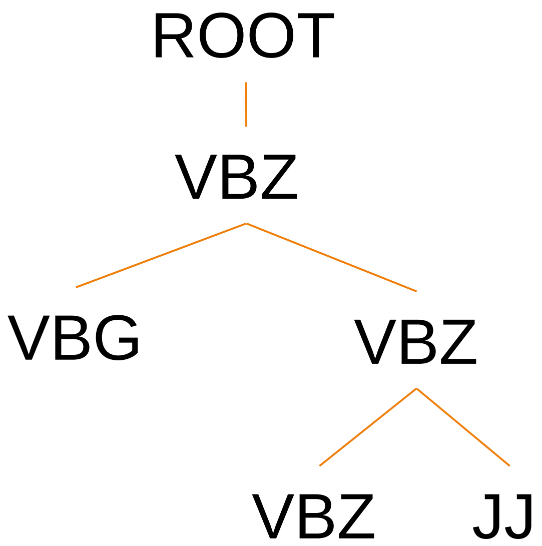
DG to CFG

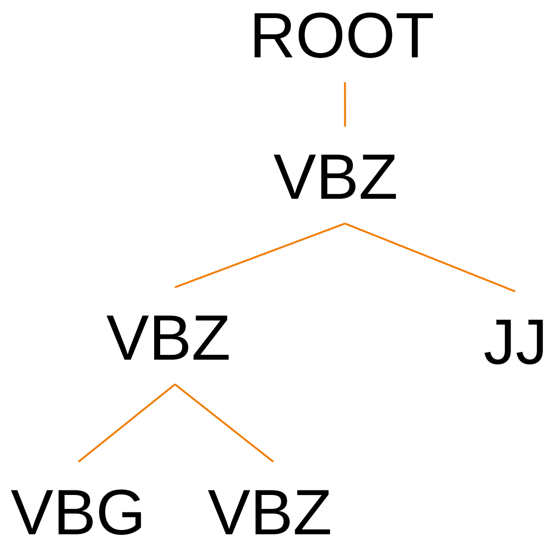




DG to CFG







DG to CFG

- Differentiate left-branching and right-branching nonterminals to avoid ambiguity

$a \rightarrow b$

$a \rightarrow c$

$d \leftarrow a$

$e \leftarrow a$

$A \rightarrow AB$

$A \rightarrow AC$

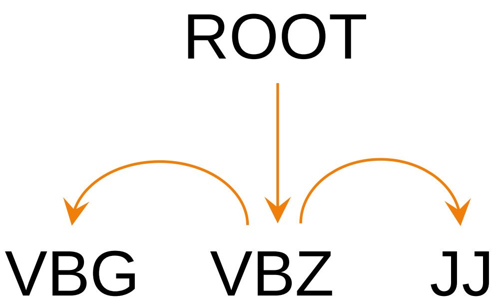
$A' \rightarrow DA'$

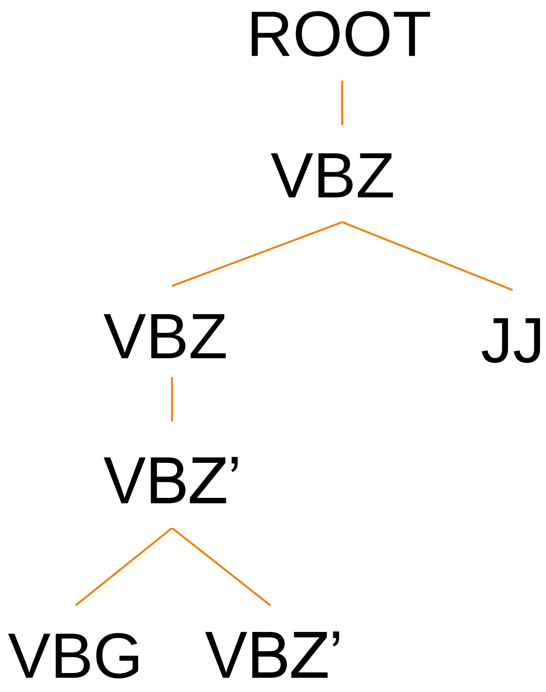
$A' \rightarrow EA'$

$A \rightarrow A'$ …

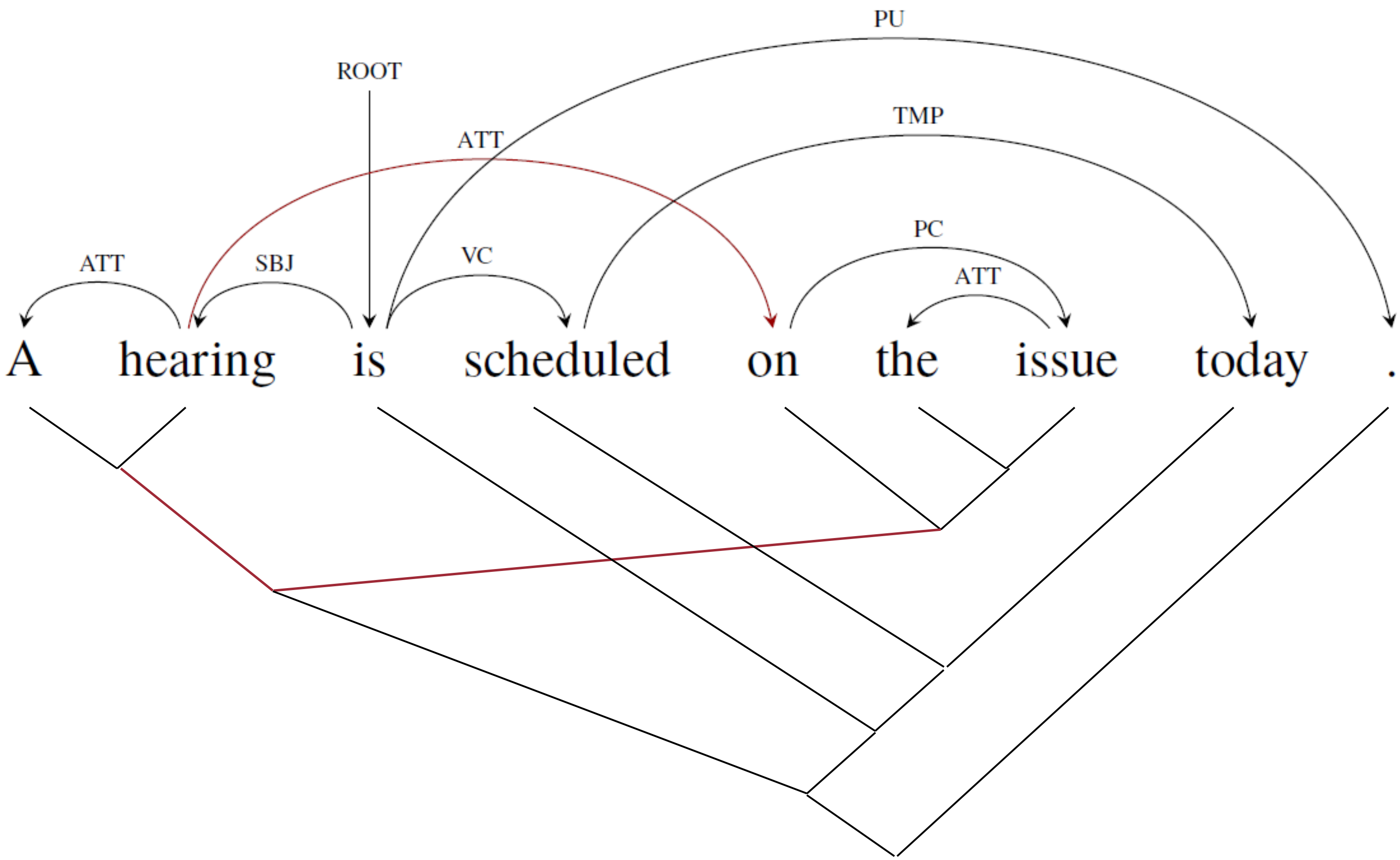
$A' \rightarrow a$ …

DG to CFG



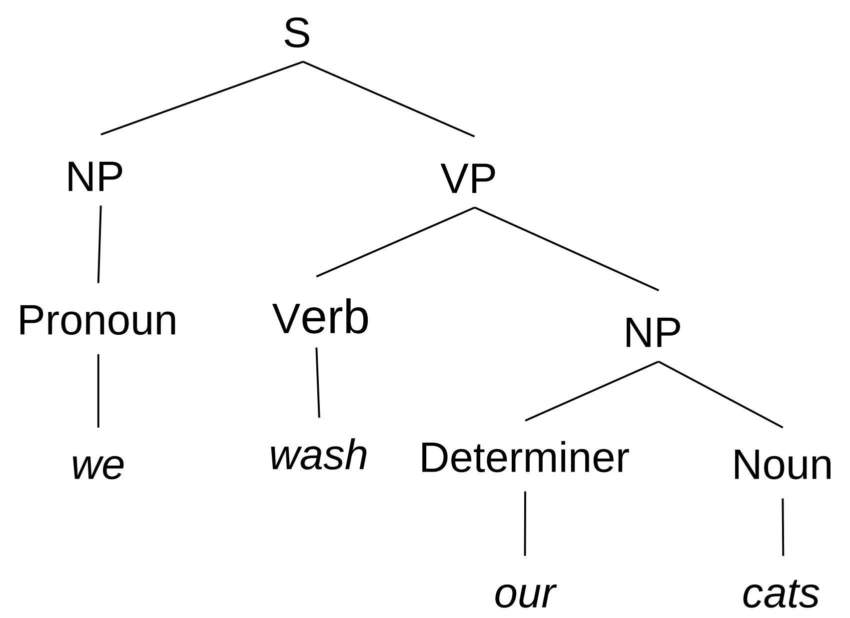


Non-projectivity & Discontinuous Constituent



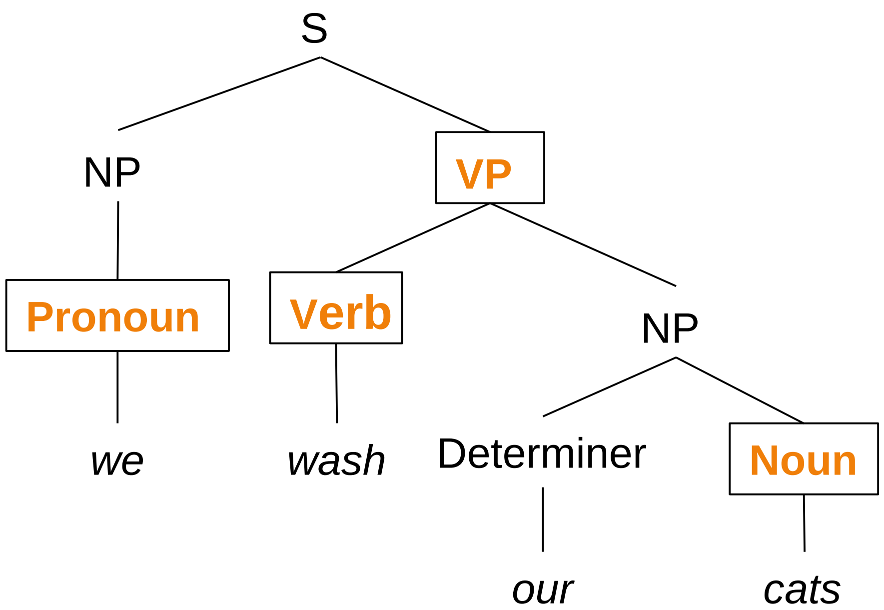
Constituency to dependency

- From a constituent tree to a dependency tree
  - Constituent tree



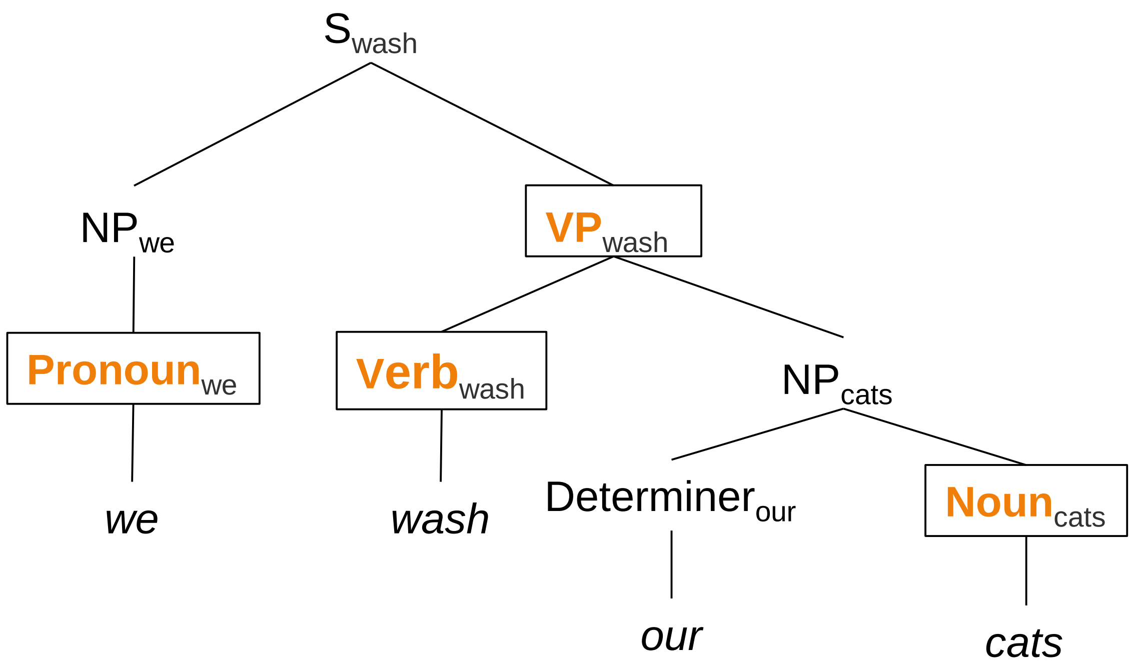
Constituency to dependency

- From a constituent tree to a dependency tree
  - Constituent tree with heads



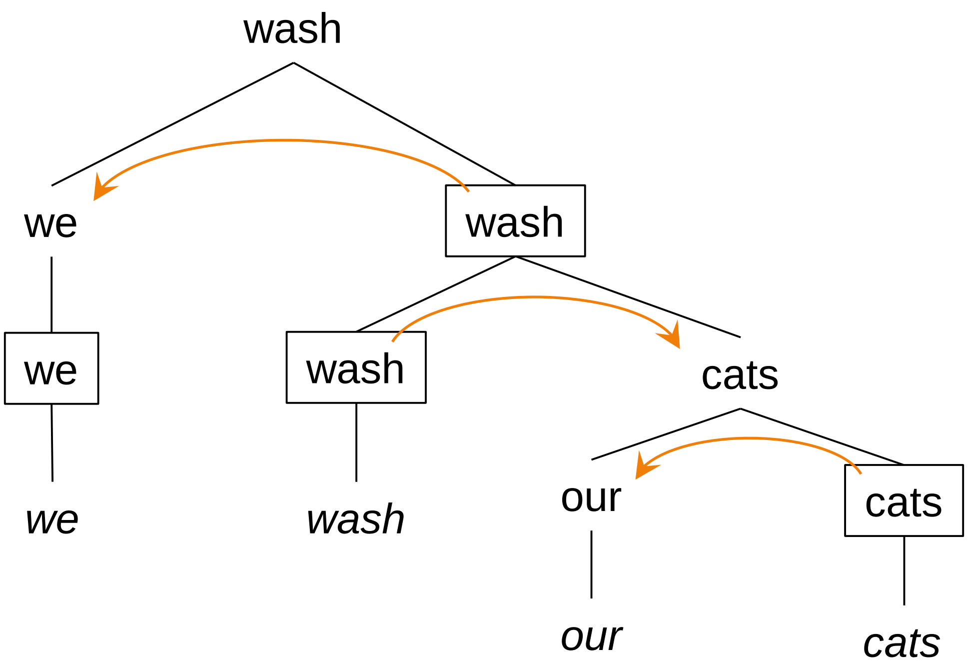
Constituency to dependency

- From a constituent tree to a dependency tree
  - Constituent tree with heads, lexicalized



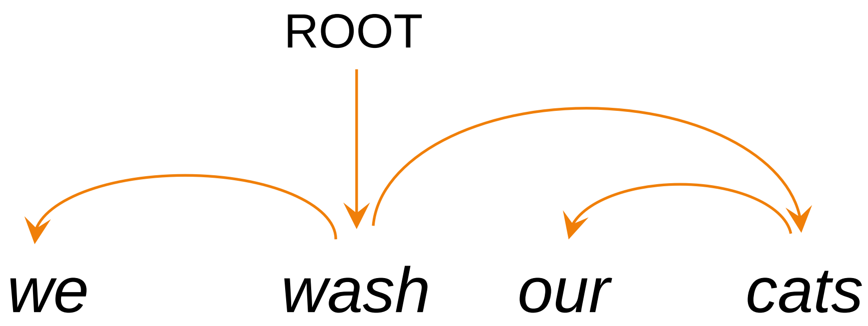
Constituency to dependency

- From a constituent tree to a dependency tree
  - Constituent tree with heads, lexicalized



Constituency to dependency

- From a constituent tree to a dependency tree
  - Dependency tree



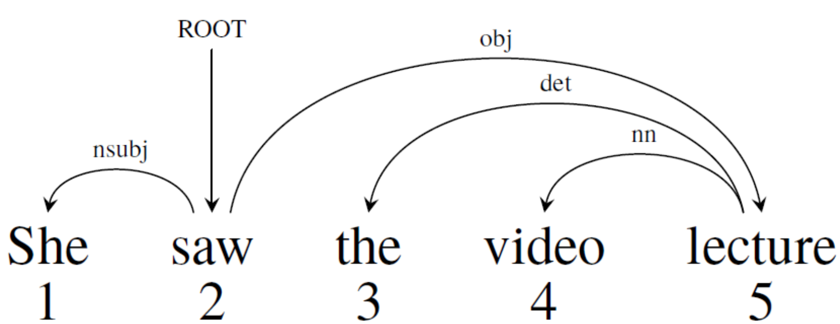
Parsing

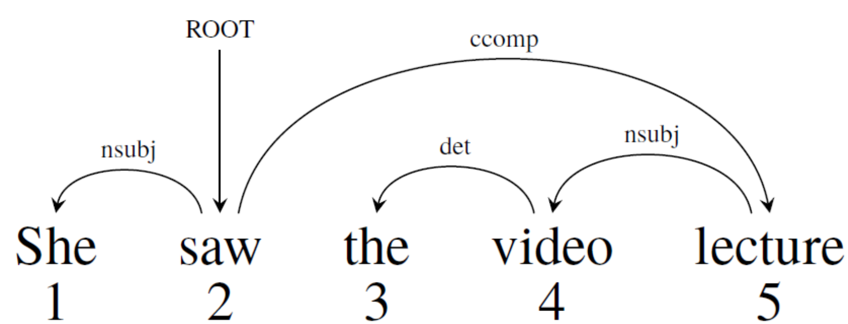
- Parsing: taking a string and returning the (best) parse tree for that string
- Algorithms
  - Graph-based parsing: MST, Eisner, etc.
    - can find the global optimum
  - Transition-based parsing: arc-standard, arc-eager, arc-hybrid
    - local optimum, fast
  - Parsing as sequence labeling
  - Headed-span-based parsing

Evaluating dependency parsing

Gold		
1	2	nsubj
2	0	root
3	5	det
4	5	nn
5	2	obj

Parsed		
1	2	nsubj
2	0	root
3	4	det
4	5	nsubj
5	2	ccomp





$$\text{Acc} = \frac{\#\ correct\ deps}{\#\ of\ deps},\quad \text{UAS} = 4/5 = 80\%,\quad \text{LAS} = 2/5 = 40\%$$

Resource

- Universal Dependencies (UD) Treebanks
  - As of 2022, nearly 200 treebanks in over 100 languages

Current UD Languages

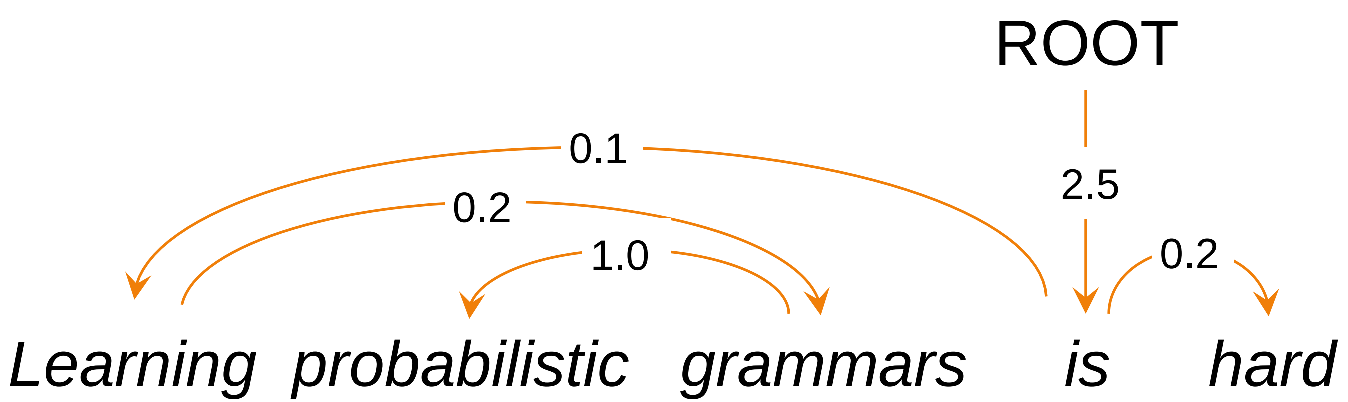
Information about language families (and genera for families with multiple branches) is mostly taken from WALS Online (IE = Indo-European).

Afrikaans	1	49K		IE, Germanic
Akkadian	2	25K		Afro-Asiatic, Semitic
Akuntsu	1	<1K		Tupian, Tupari
Albanian	1	<1K	W	IE, Albanian
Amharic	1	10K		Afro-Asiatic, Semitic
Ancient Greek	2	416K		IE, Greek
Apurina	1	<1K		Arawakan
Arabic	3	1,042K	W	Afro-Asiatic, Semitic
Armenian	2	55K		IE, Armenian
Assyrian	1	<1K		Afro-Asiatic, Semitic
Bambara	1	13K		Mande
Basque	1	121K		Basque
Beja	1	1K		Afro-Asiatic, Cushitic
Belarusian	1	305K	W	IE, Slavic
Bengali	1	<1K		IE, Indic

Graph-Based Dependency Parsing

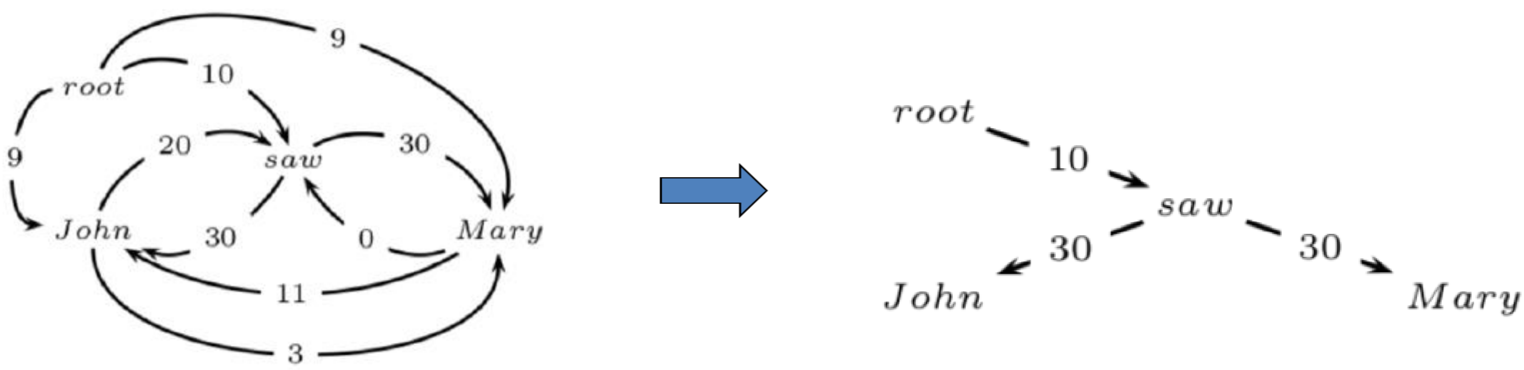
First-order graph-based dependency parsing

- Parse tree scoring
  - Each arc has a score. The tree score is the sum of arc scores.
  - An arc score is often computed from features of the two words
    - Possible features: neighboring words, their POS tags, contextual word embeddings



First-order graph-based dependency parsing

- Parse tree scoring
- Parsing
  - Maximum spanning tree (more precisely, spanning arborescence)
  - What about dependency labels?
    - For each arc, we may simply predict its label from features of the two words



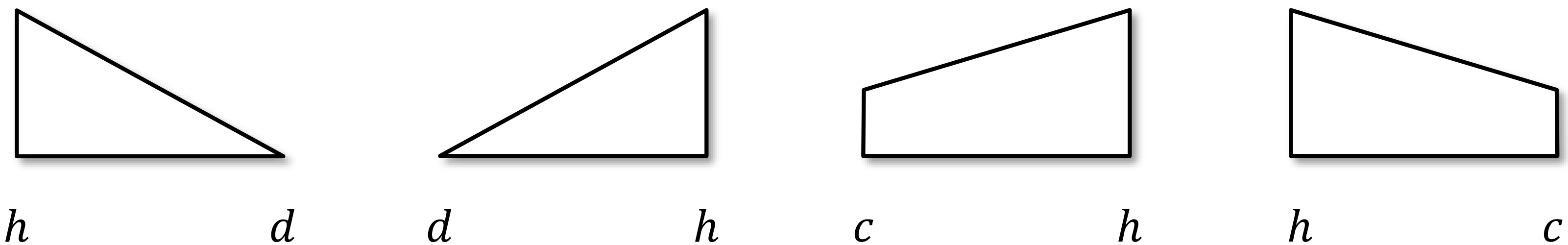
CYK

- Remember: projective dependency parsing can be modeled with a CFG
- We can run CYK for parsing
- But runtime is $O(n^5)$!
  - CYK is $O(n^3 \cdot |G|)$
  - There are n^2 rules for a sentence:

$$\forall w_1, w_2\text{: } N_{w_1} \rightarrow N_{w_1} N_{w_2} \text{ or } N_{w_2} \rightarrow N_{w_1} N_{w_2}$$

Eisner Algorithm

Items:



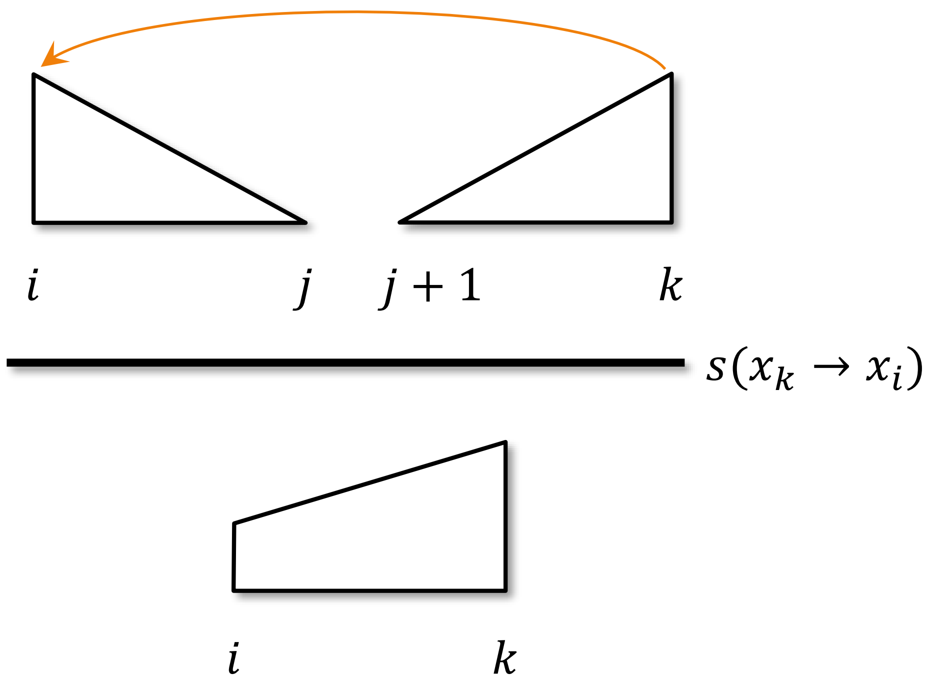
- Triangles: a partial tree whose root is x_h and no words except x_h expect more children.
- Trapezoids: x_c is a child of x_h and x_c still expects children on its side.
- In all cases, the words in between are descendants of x_h.

Eisner Algorithm – DP base case

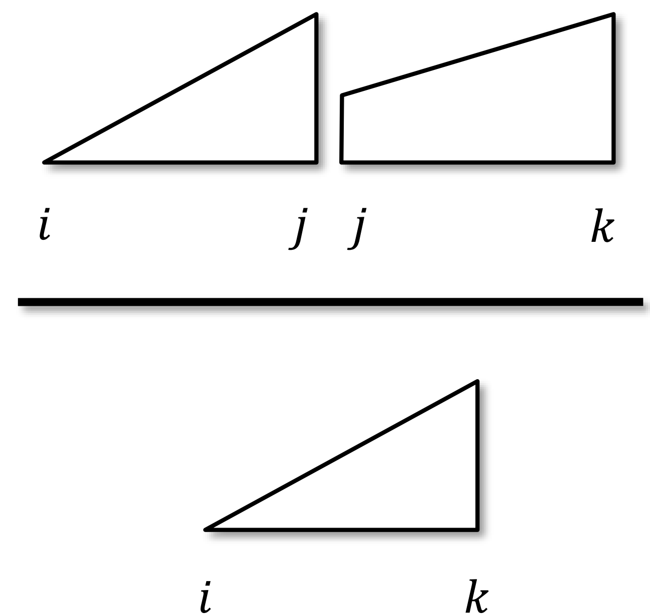
Initialization:

Eisner Algorithm – DP recursion

Attach a left dependent:

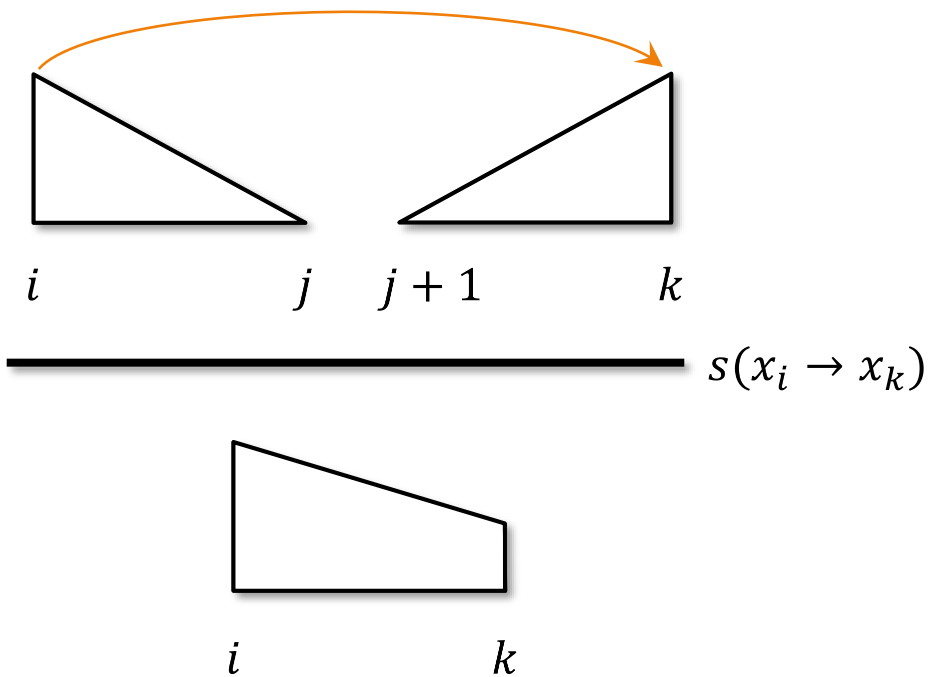


Complete a left child:

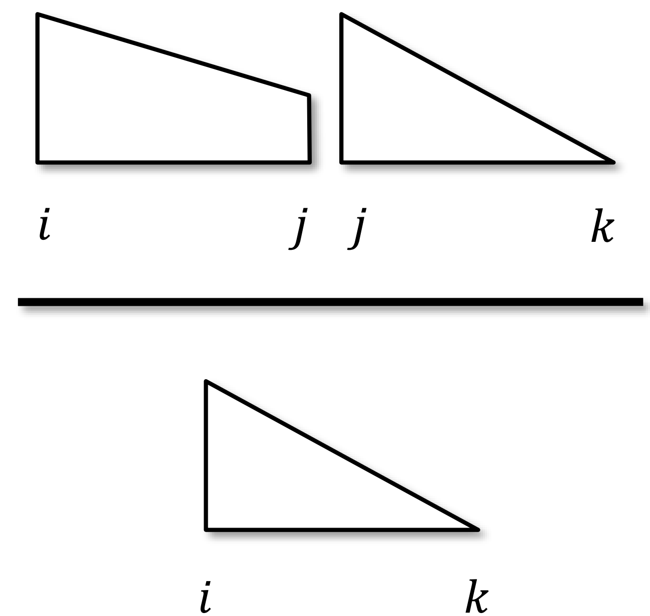


Eisner Algorithm – DP recursion

Attach a right dependent:

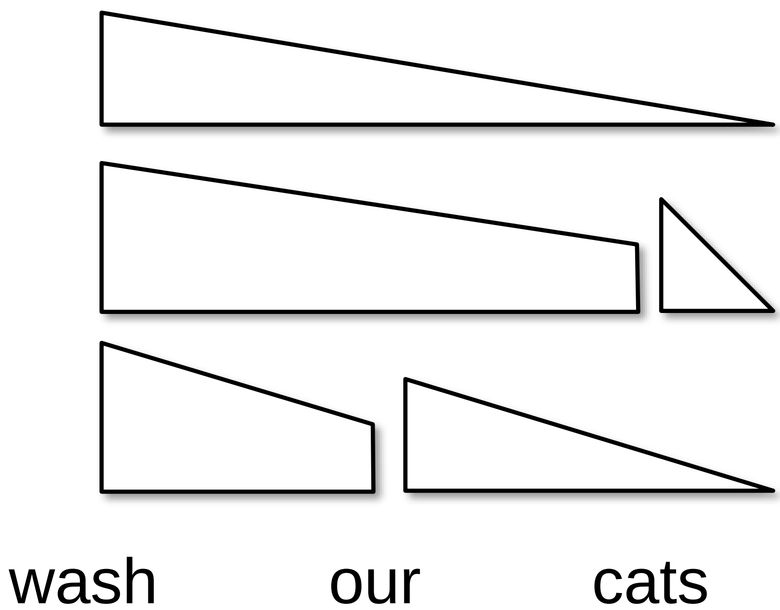


Complete a right child:



Eisner Algorithm – DP recursion

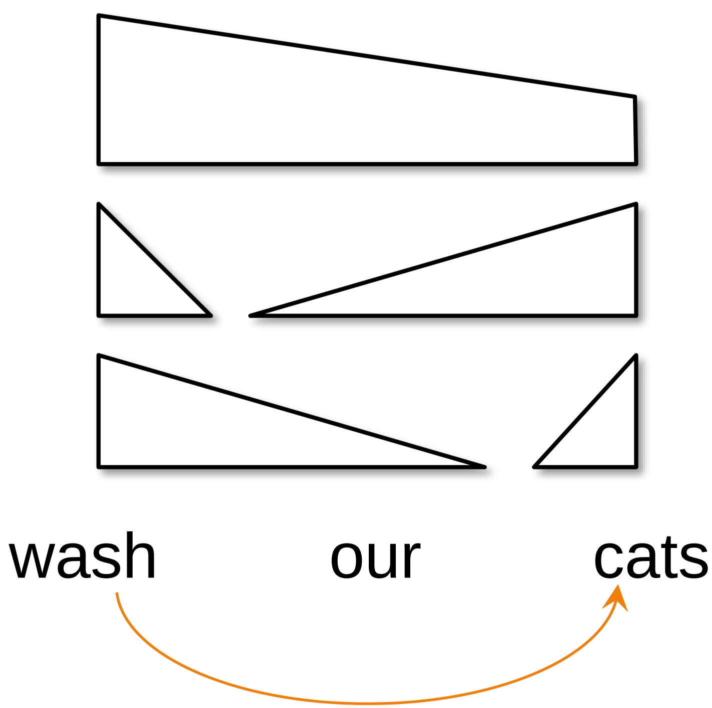
- Ambiguity
  - An item may have multiple valid compositions
  - Pick the highest scoring one (and record its composition)



$$C_{2\to 4} = \max\begin{pmatrix} I_{2\to 4} + C_{4\to 4}, \\ I_{2\to 3} + C_{3\to 4} \end{pmatrix}$$

Eisner Algorithm – DP recursion

- Ambiguity
  - An item may have multiple valid compositions
  - Pick the highest scoring one (and record its composition)

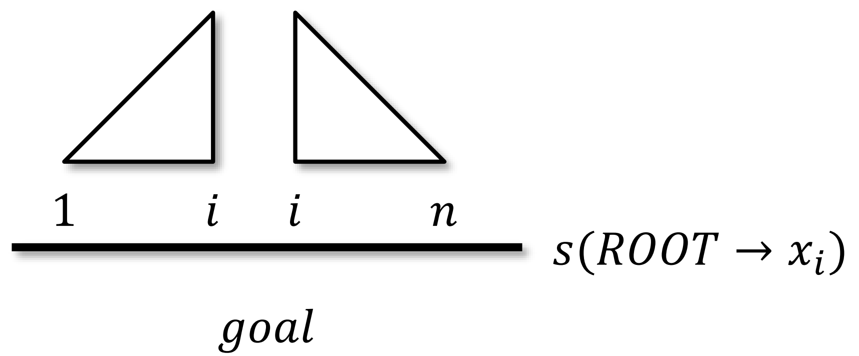


$$I_{2\to 4} = \max\begin{pmatrix} C_{2\to 2} + C_{3\leftarrow 4}, \\ C_{2\to 3} + C_{4\leftarrow 4} \end{pmatrix}$$

$$+s(x_2 \to x_4)$$

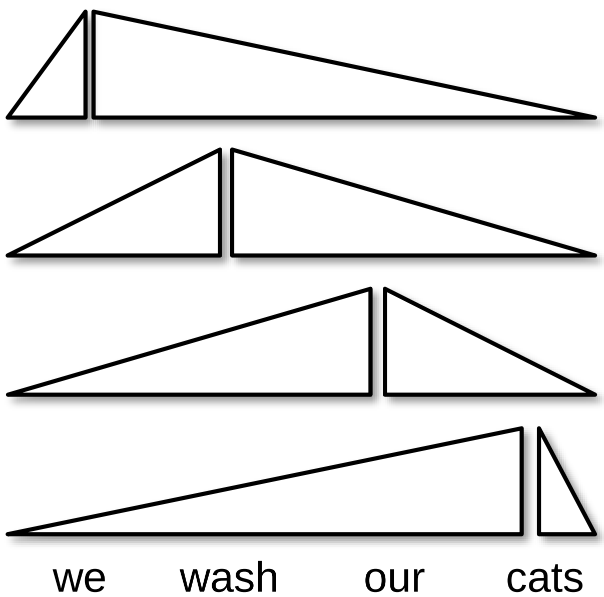
Eisner Algorithm – DP goal state

Goal:



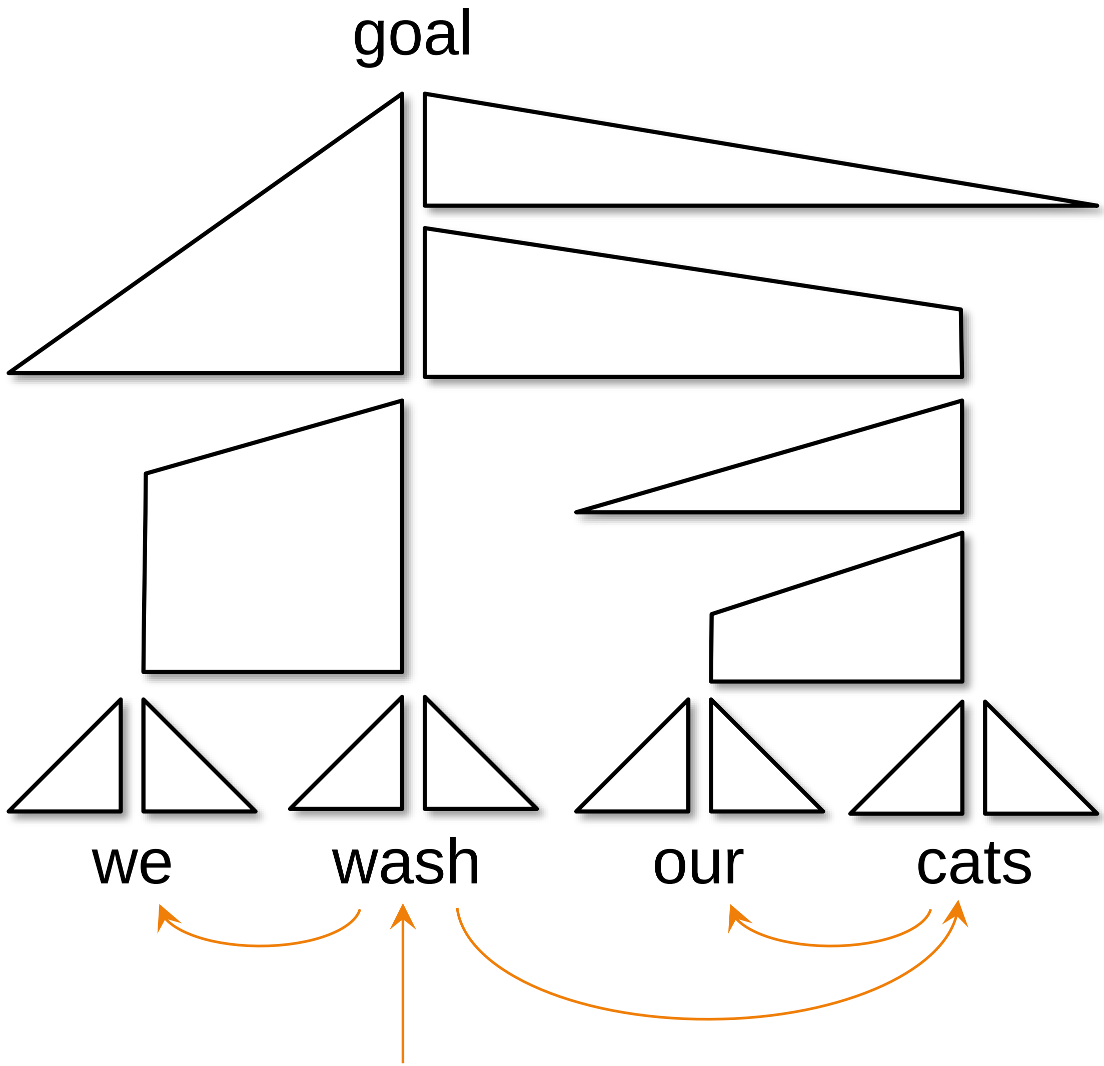
Eisner Algorithm – DP goal state

- Ambiguity
  - There may be multiple goal states
  - Pick the highest scoring one (and record its composition)



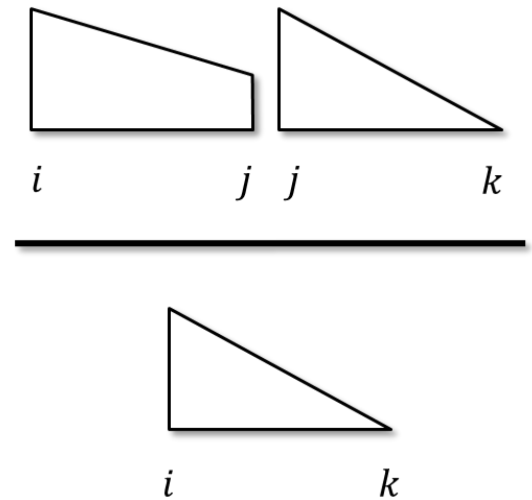
$$s_{tree} = \max_i (C_{1 \leftarrow i} + C_{i \rightarrow 4} + s(ROOT \rightarrow x_i))$$

Eisner Algorithm – obtain the parse tree

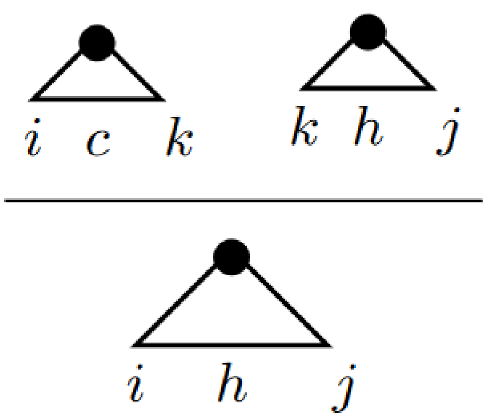


Eisner Algorithm

- Time complexity
  - $O(n^2)$ items
  - Each item has $O(n)$ possible compositions
  - The run time is $O(n^3)$



- Comparison with CYK
  - CYK has $O(n^3)$ items, each has $O(n^2)$ possible compositions, hence the run time is $O(n^5)$
  - Eisner does head-splitting to eliminate $O(n^2)$ time!

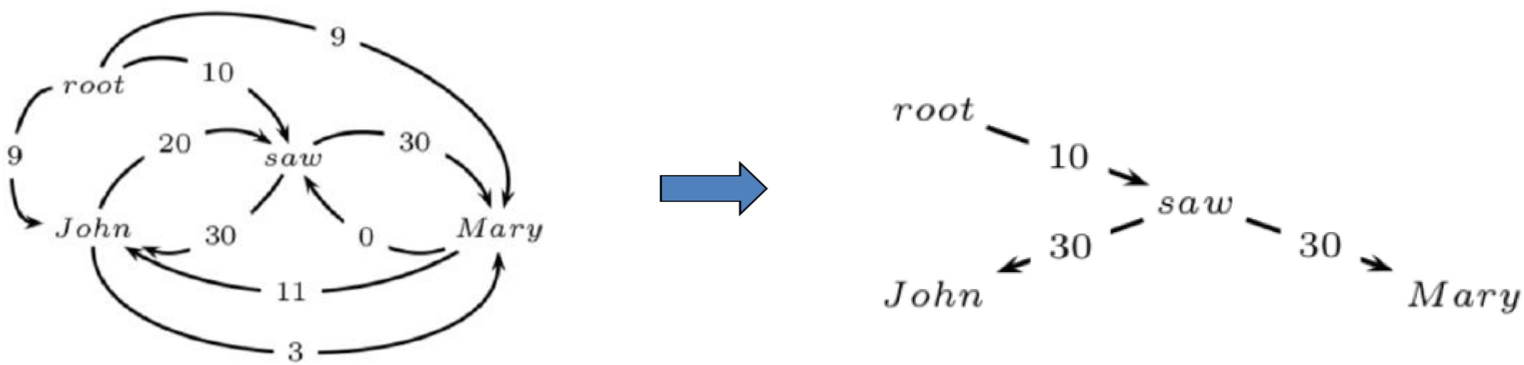


Non-projective dependency parsing

- Eisner algorithm only finds projective dependency parses
- We need a different algorithm for finding non-projective dependency parses
  - MST parser based on the Chu-Liu-Edmonds algorithm

Chu-Liu-Edmonds: Preliminary

- Maximum spanning tree (arborescence)
- Every non-root node needs exactly one incoming edge.
- In fact, every connected component that doesn't contain root needs exactly one incoming edge.



Chu-Liu-Edmonds

- Two stages:
  - **Contraction**
  - **Expansion**

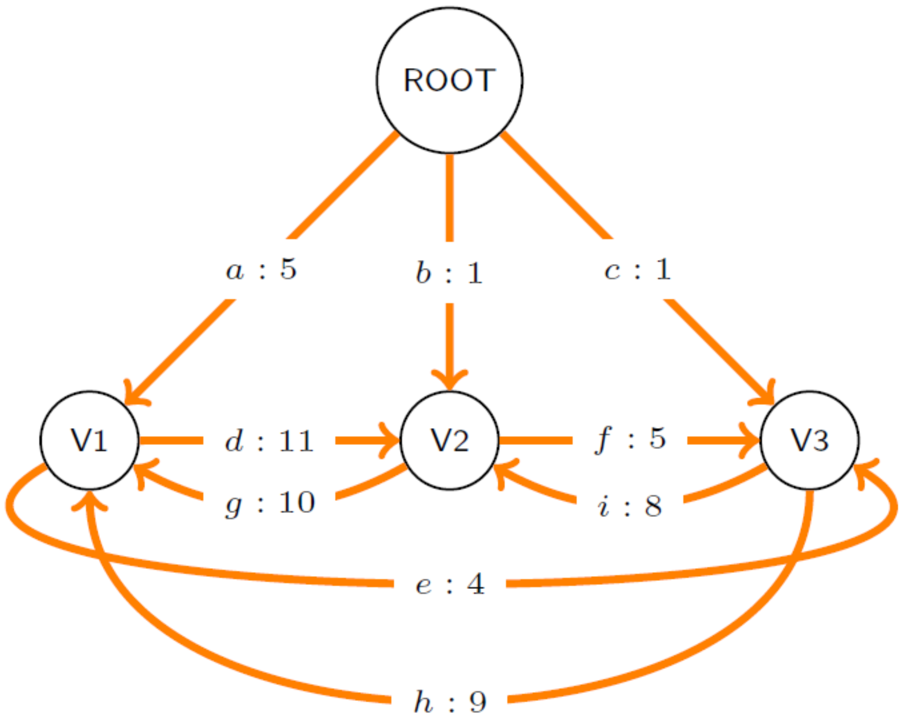
Chu-Liu-Edmonds: Contraction

1. For each non-$ROOT$ node v, pick its highest scoring incoming edge
   - Set `bestInEdge`$[v]$ to be the edge
2. If this forms an arborescence (i.e., no cycle)
   - We are done! Go to the next stage.
3. Otherwise, a cycle C is formed
   - To build an arborescence, we need to:
     - Find an incoming edge for C
     - Kick out some edge in C
   - Choosing the incoming edge for C determines which edge to kick out.

Chu-Liu-Edmonds: Contraction

3. Otherwise, a cycle C is formed
   - Choosing the incoming edge for C determines which edge to kick out.
   - contract the nodes in C into a new node v_C and adjust related edges:
     - Edges outgoing from any node in C now get source v_C
     - Edges incoming to any node in C now get destination v_C
     - For each edge e incoming to $v \in C$ from outside of C:
       - $e.\mathtt{kicksOut} \leftarrow \mathtt{bestInEdge}[v]$
       - $e.\mathtt{score} \leftarrow e.\mathtt{score} - e.\mathtt{kicksOut}.\mathtt{score}$
         - This is the change in total score if we choose e and kick out the original incoming edge to v
4. Now we get a new graph. Go back to step 1 and repeat.

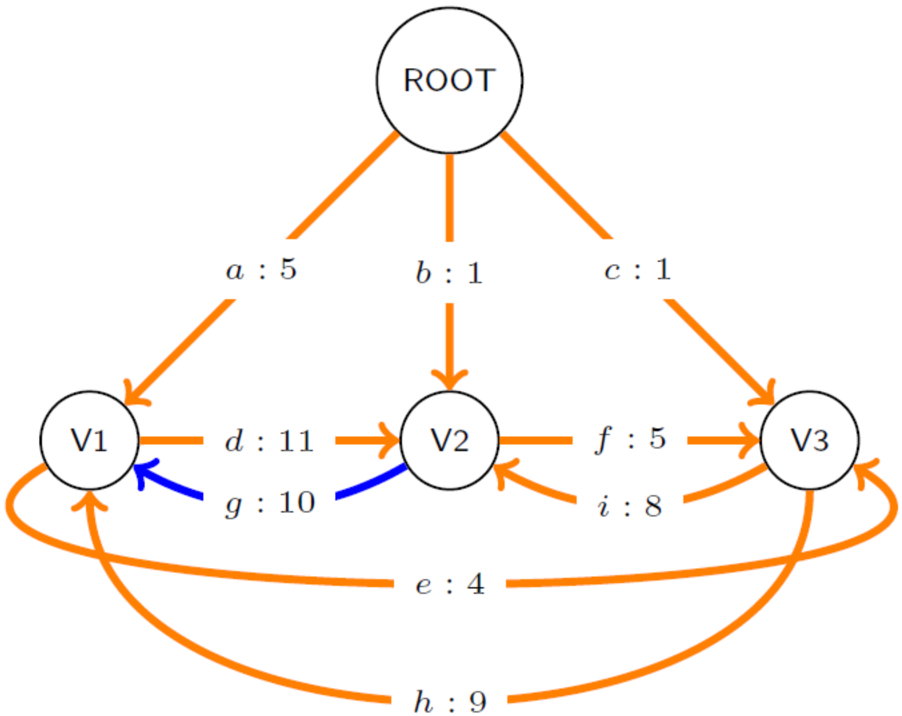
Contraction Example



	bestInEdge
V1	
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

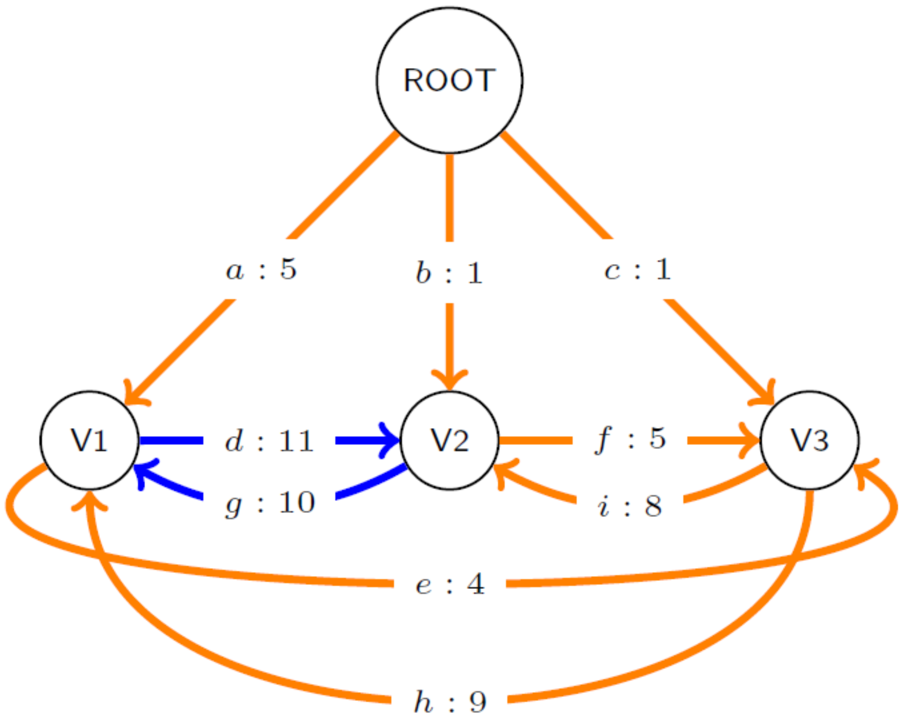
Contraction Example



	bestInEdge
V1	g
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

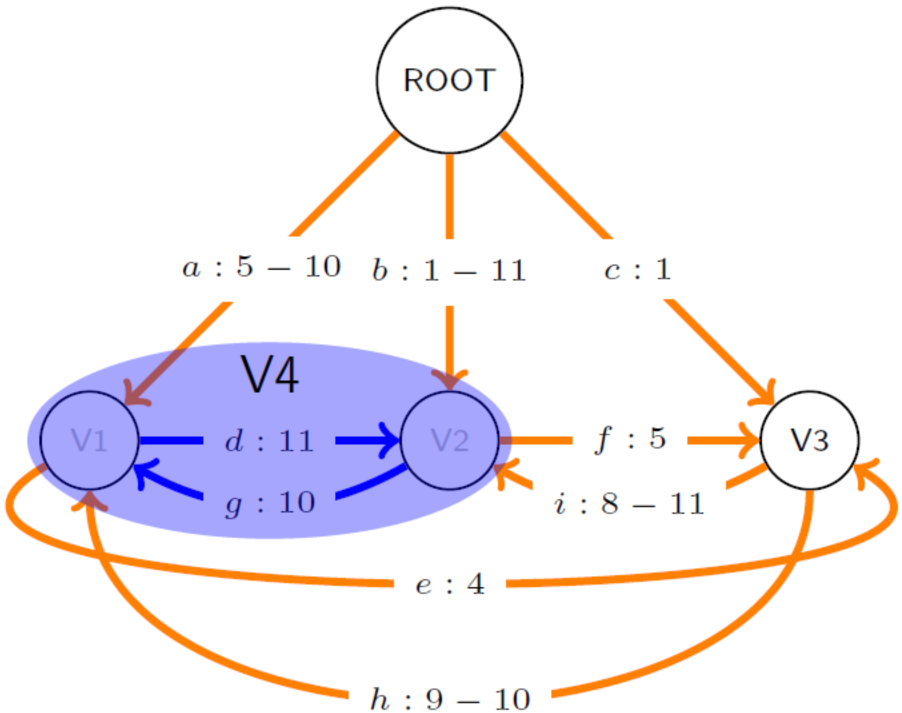
Contraction Example



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

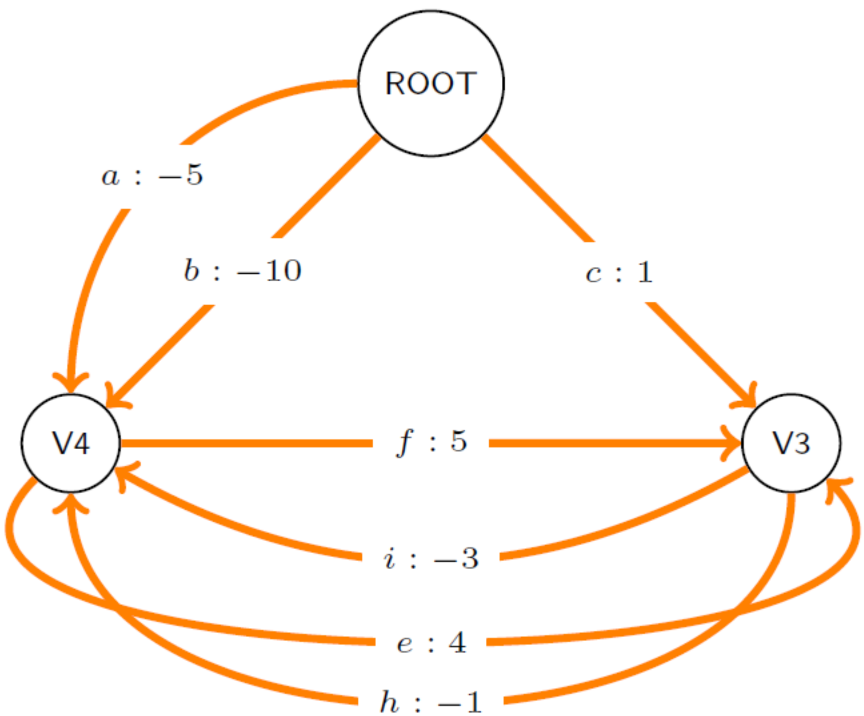
Contraction Example



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

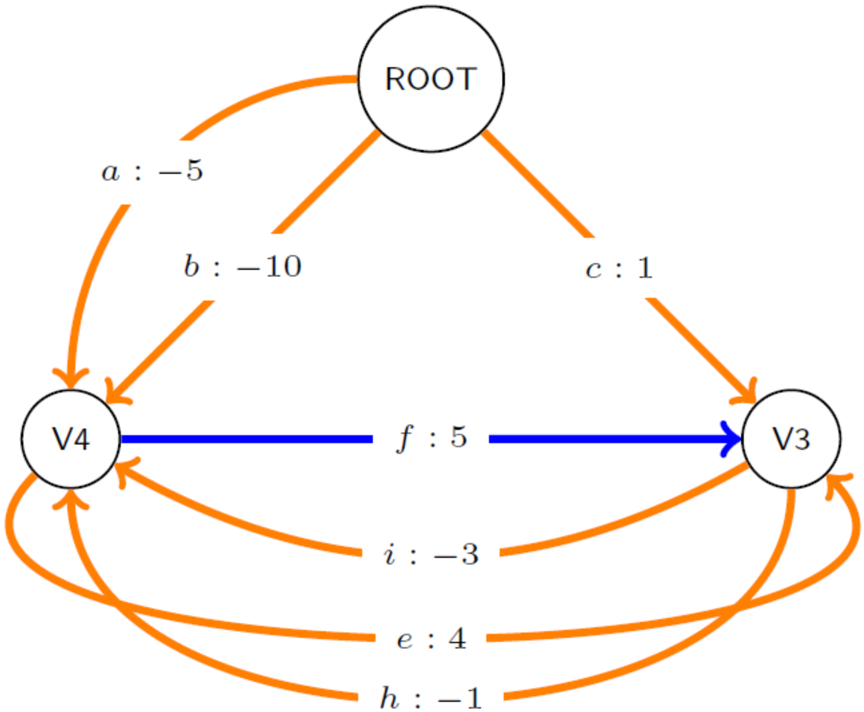
Contraction Example



	bestInEdge
V1	g
V2	d
V3	
V4	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

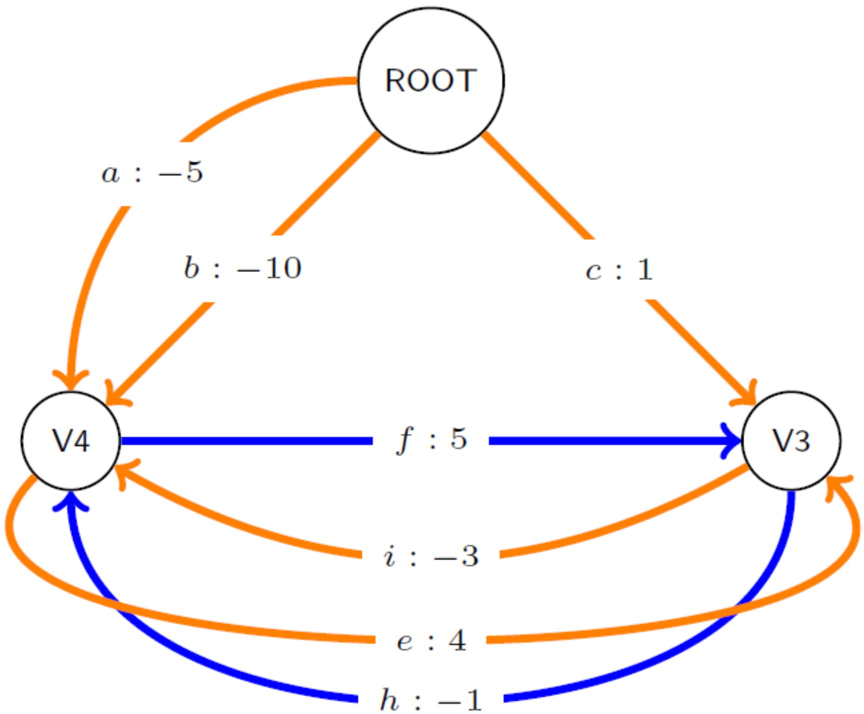
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

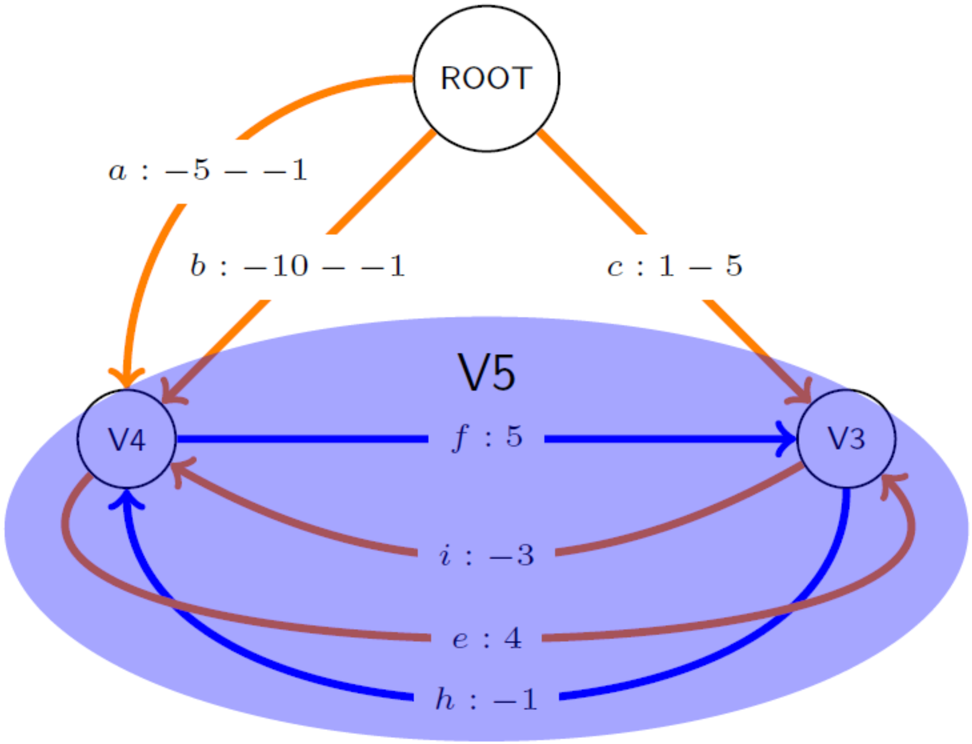
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

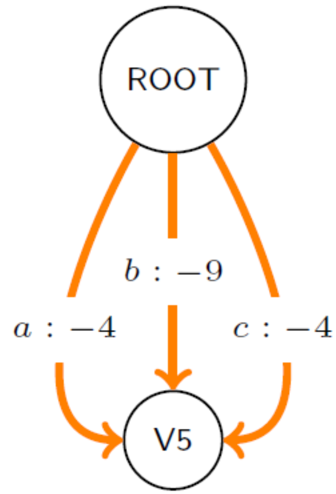
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

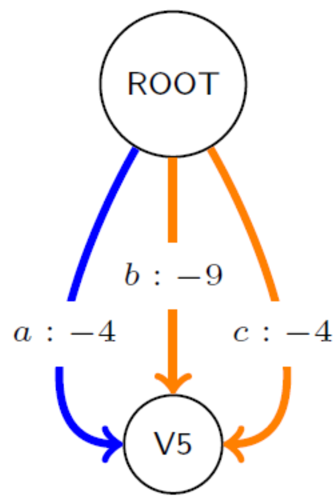
Contraction Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

Contraction Example



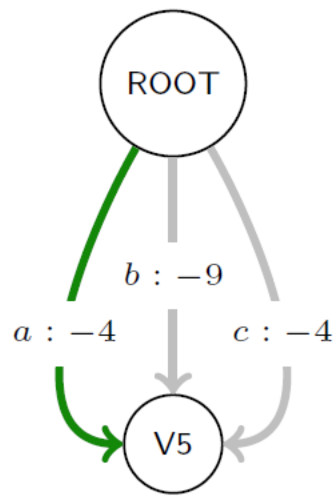
	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

Chu-Liu-Edmonds: Expansion

- After the contraction stage, every contracted node will have exactly one `bestInEdge`. This edge will kick out one edge inside the contracted node, breaking the cycle.
  - Go through each `bestInEdge` e in the reverse order that we added them
  - Lock down e, and remove every edge in `kicksOut`(e) from `bestInEdge`.

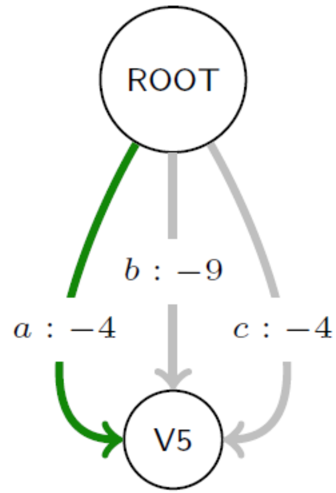
Expansion Example



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

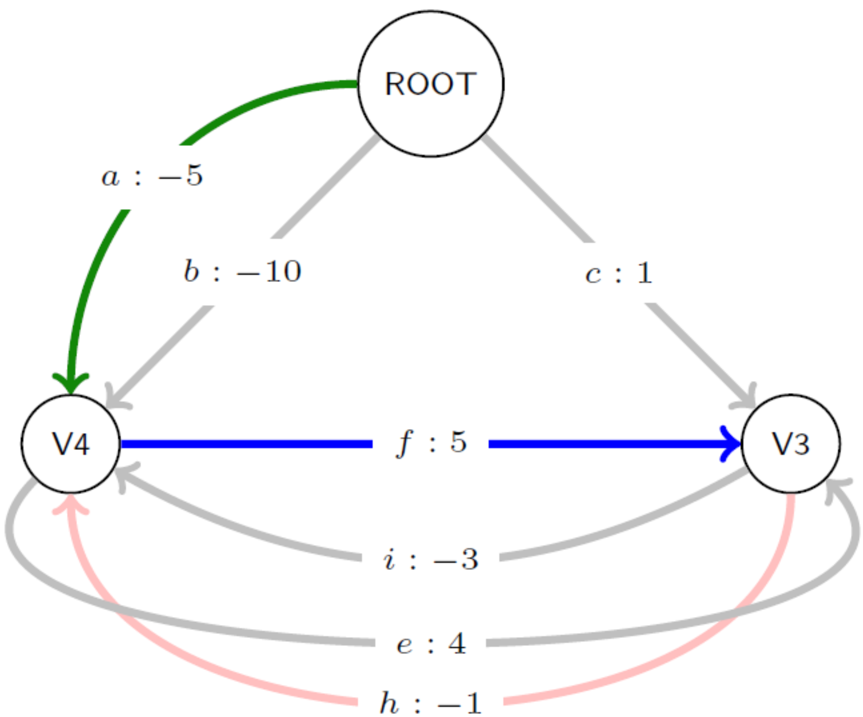
Expansion Example



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

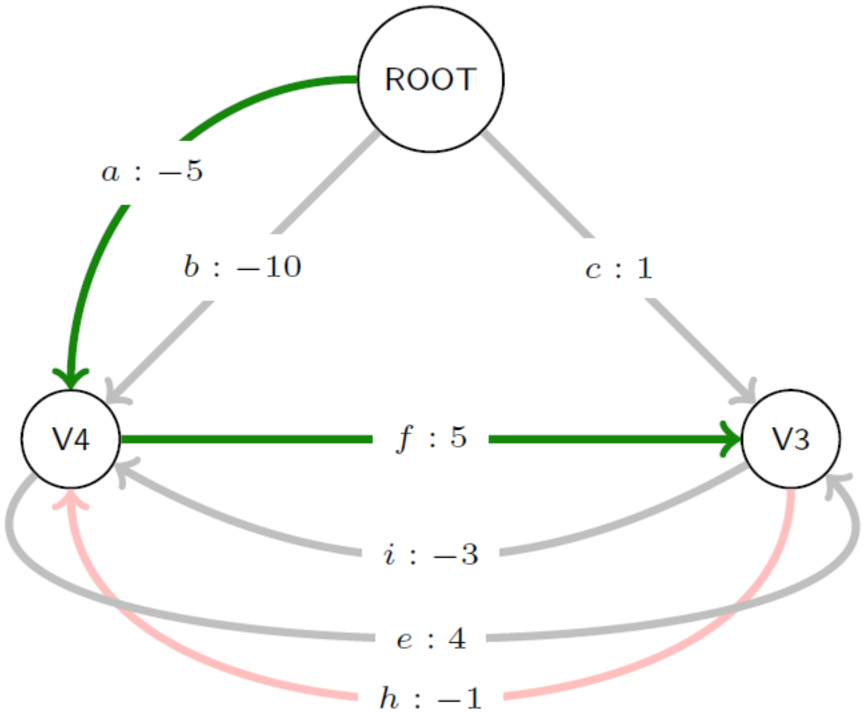
Expansion Example



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

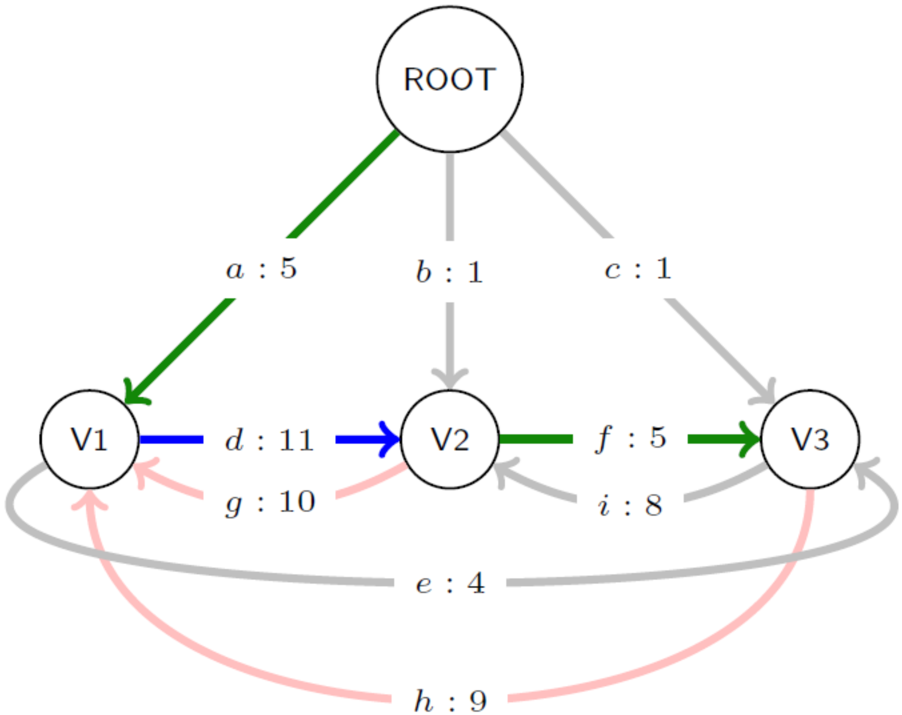
Expansion Example



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

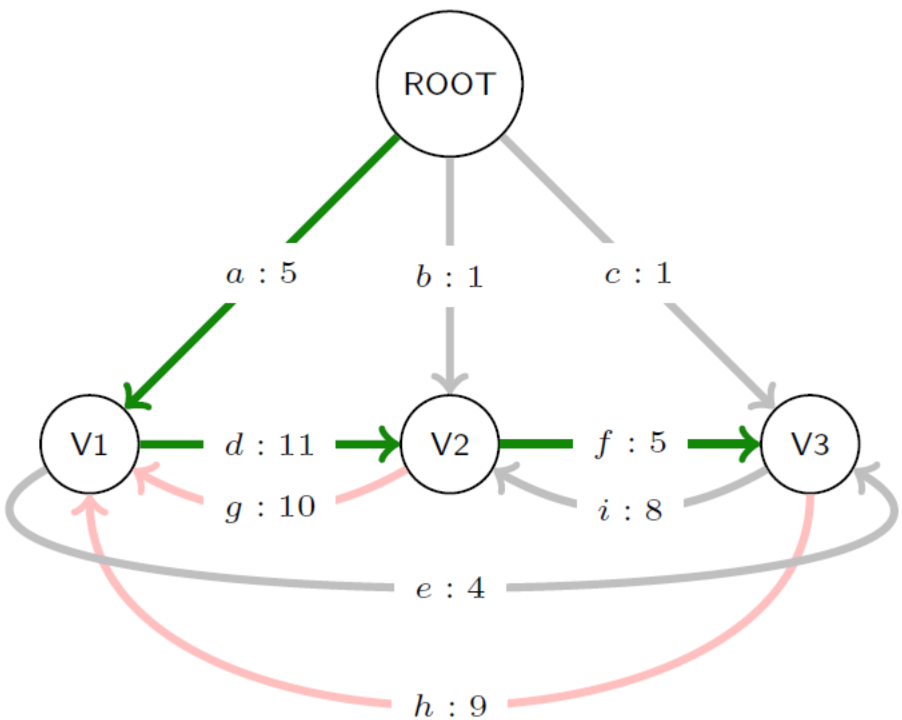
Expansion Example



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

Expansion Example



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

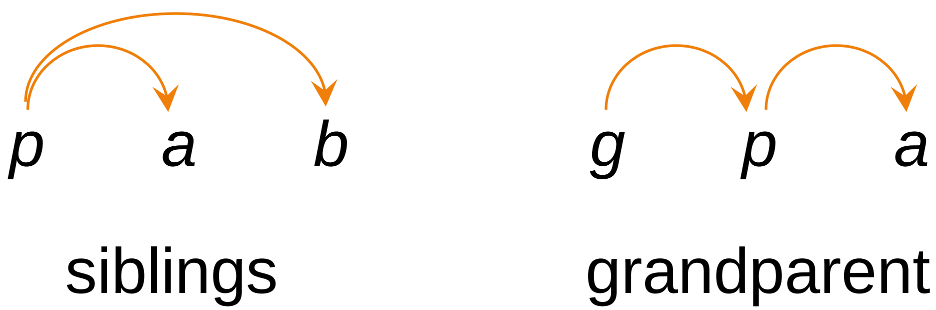
Chu-Liu-Edmonds Algorithm

- Time complexity
  - Simple implementation: $O(n^3)$
  - Fast implementation: $O(n^2 + n \log n)$

- Single-root parsing: only one word can be the root
  - Several extensions of CLE
  - A simple solution: root reweighting
    - Subtract a large constant from ROOT edges, so that any single-root tree has a larger score than any multi-root tree
  - See https://aclanthology.org/2021.emnlp-main.823.pdf

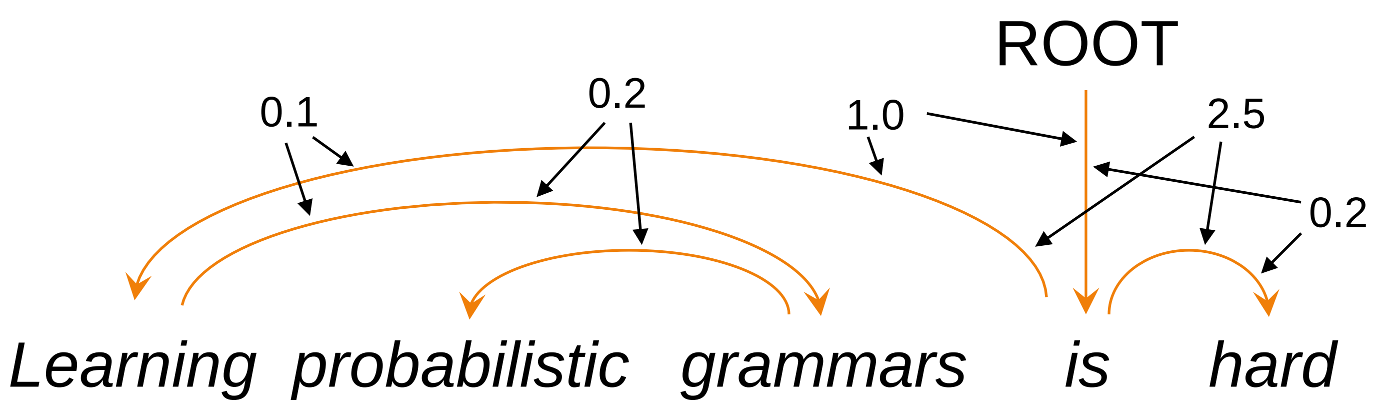
Second-order graph-based dependency parsing

- Parse tree scoring:
  - Each connected pair of arcs has a score. The tree score is the sum of arc-pair scores.



Second-order graph-based dependency parsing

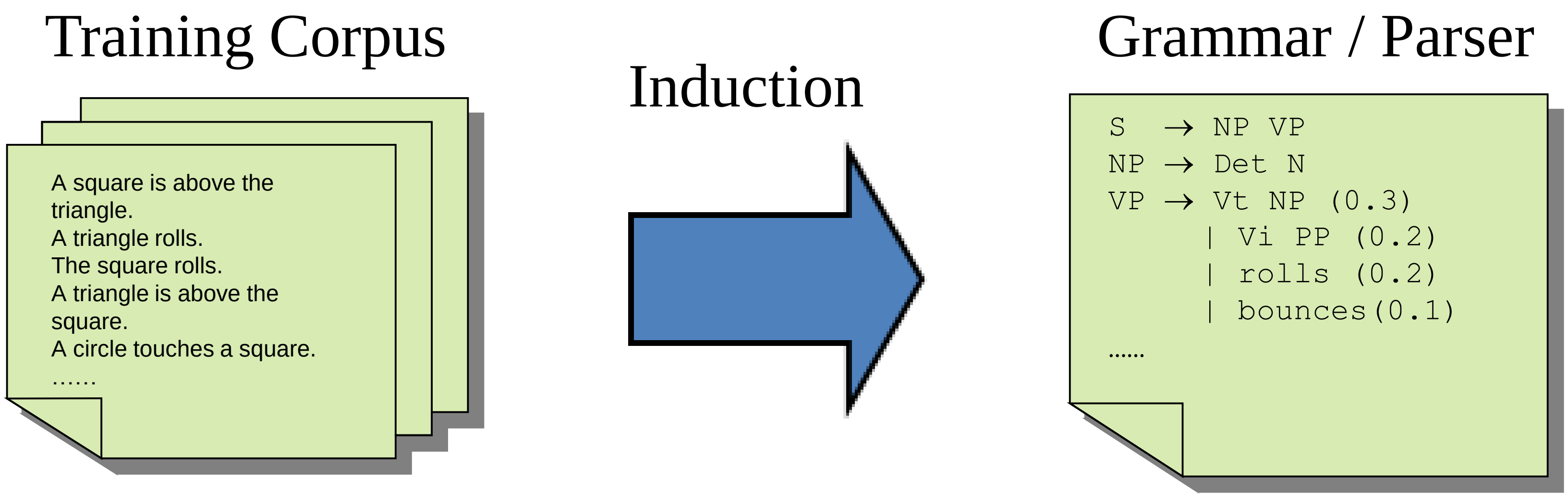
- Parse tree scoring:
  - Each connected pair of arcs has a score. The tree score is the sum of arc-pair scores.



Second-order graph-based dependency parsing

- Parse tree scoring:
  - Each connected pair of arcs has a score. The tree score is the sum of arc-pair scores.
- Parsing:
  - $O(n^4)$ time for projective dependency parsing
  - NP-hard for non-projective dependency parsing
  - Approximate algorithms exist

Learning a grammar from a corpus



- Supervised Methods
  - Rely on a training corpus of sentences annotated with parses (treebank)
- Unsupervised Methods (Grammar Induction)
  - Do not require annotated data

Supervised Learning

- Objective
  - Conditional likelihood: P(gold parse | sentence)

$$P(t|x) = \frac{\exp\left(s(t)\right)}{Z(x)} = \frac{\exp\left(\sum_{r \in (t,x)} s(r)\right)}{Z(x)} = \frac{\prod_{r \in (t,x)} \exp\left(s(r)\right)}{Z(x)}$$

    - Problem: how to compute the partition function
      - Impossible to enumerate parse trees
    - Projective: replace max with sum in Eisner algorithm (similar to inside algorithm)
    - Non-projective: Kirchhoff's matrix tree theorem
      - The *determinant* of the Kirchoff (aka Laplacian) adjacency matrix of directed graph G without row and column r is equal to the sum of scores of all directed spanning trees of G rooted at node r.

Supervised Learning

- Objective
  - Conditional likelihood: P(gold parse | sentence)
  - Margin-based objective
- Optimization
  - Gradient-based methods

Unsupervised Learning

- Generative method
  - Dependency parser as a PCFG
    - Ex: Dependency Model with Valence (DMV)
  - Run EM algorithm or SGD to maximize likelihood P(sentence)

- Discriminative method
  - CRF-Autoencoder
    - Encoder: a graph-based dependency parser
    - Decoder: predict each word from its head
  - Maximize reconstruction probability using SGD

Transition-Based Dependency Parsing

Transition-Based Parsing

- A dependency tree represented as a linear sequence of transitions.
- Transitions: simple actions to be executed on a parser configuration.
- Parser configuration
  - Buffer B: unprocessed words of the input sentence
  - Stack S: parse tree under construction

Transition-Based Parsing

- Initial Configuration
  - Buffer B contains the complete input sentence and stack S only contains ROOT.
- During parsing
  - Apply a classifier to decide which transition to take next.
  - No backtracking.
- Final Configuration
  - Buffer B is empty and stack S contains the entire parse tree.

Transition-Based Parsing

- Transitions: "arc-standard" transition set (Nivre, 2004)
  - SHIFT: move the word at the front of buffer B onto stack S.
  - $\mathrm{RIGHT\text{-}ARC}$: $u = \mathrm{pop}(S);\ v = \mathrm{pop}(S);\ \mathrm{push}(S, v \rightarrow u)$.
  - $\mathrm{LEFT\text{-}ARC}$: $u = \mathrm{pop}(S);\ v = \mathrm{pop}(S);\ \mathrm{push}(S, v \leftarrow u)$.
- For labeled parsing, add labels to the $\mathrm{RIGHT-ARC}$ and $\mathrm{LEFT-ARC}$ transitions.
  - Ex. $\mathrm{LEFT-ARC\text{-}nSubj}$

- There are other transition sets: arc-eager, arc-hybrid, …

Transition-Based Parsing: Example

Stack S:

ROOT

Buffer B:

we
vigorously
wash
our
cats
who
stink

Actions:

Transition-Based Parsing: Example

Stack S:

we
ROOT

Buffer B:

vigorously
wash
our
cats
who
stink

Actions: SHIFT

Transition-Based Parsing: Example

Stack S:

vigorously
we
ROOT

Buffer B:

wash
our
cats
who
stink

Actions: SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:

wash
vigorously
we
ROOT

Buffer B:

our
cats
who
stink

Actions: SHIFT SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:

vigorously wash
we
ROOT

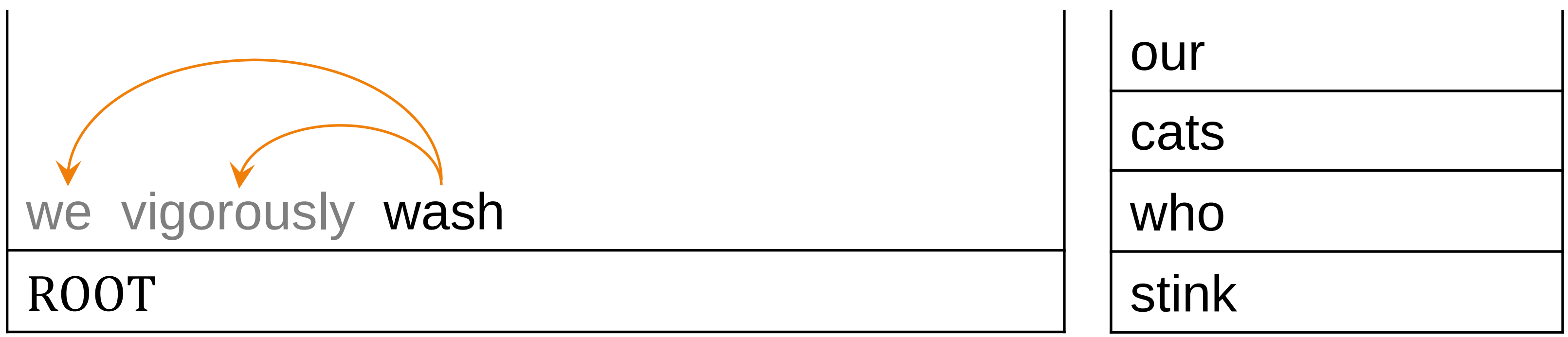
Buffer B:

our
cats
who
stink

Actions: SHIFT SHIFT SHIFT LEFT−ARC

Transition-Based Parsing: Example

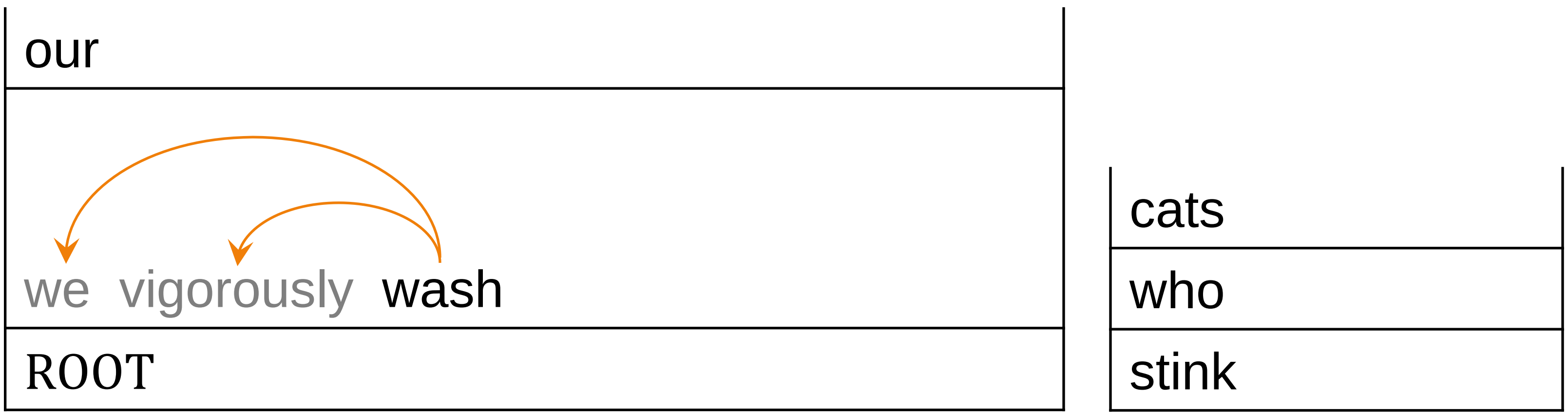
Stack S: Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT−ARC LEFT − ARC

Transition-Based Parsing: Example

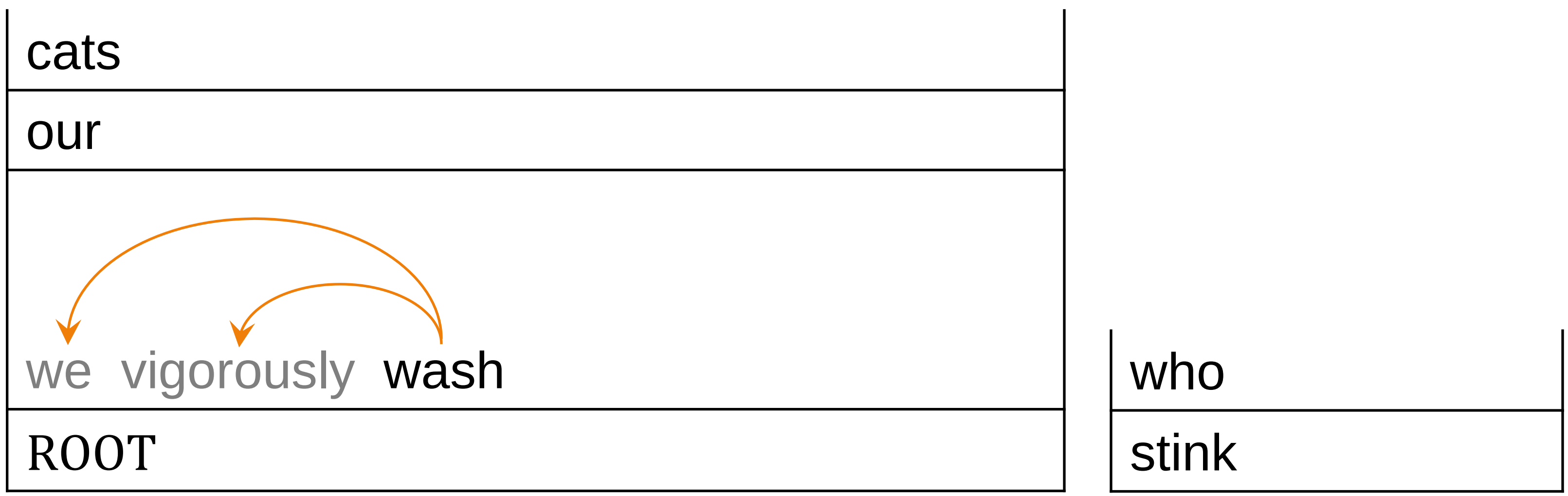
Stack S: Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT−ARC LEFT−ARC SHIFT

Transition-Based Parsing: Example

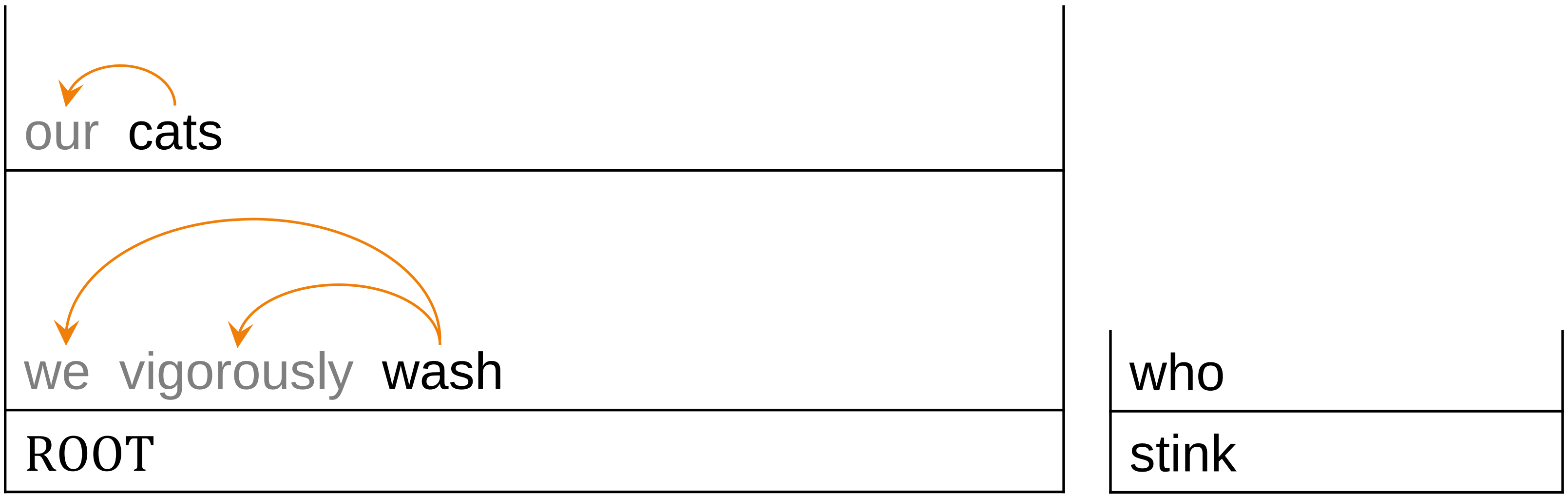
Stack S: Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT–ARC LEFT–ARC SHIFT SHIFT

Transition-Based Parsing: Example

Stack S: Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT–ARC LEFT–ARC SHIFT SHIFT LEFT–ARC

Transition-Based Parsing: Example

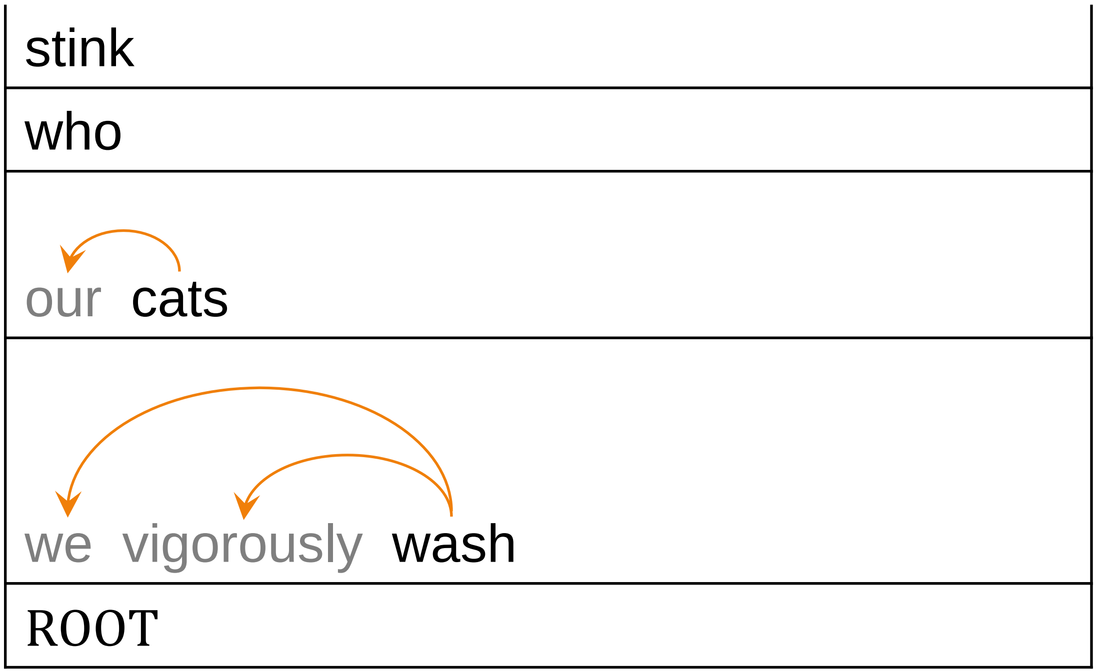
Stack S: Buffer B:

Actions: SHIFT SHIFT SHIFT LEFT–ARC LEFT–ARC SHIFT SHIFT LEFT–ARC SHIFT

Transition-Based Parsing: Example

Stack S:

Buffer B:

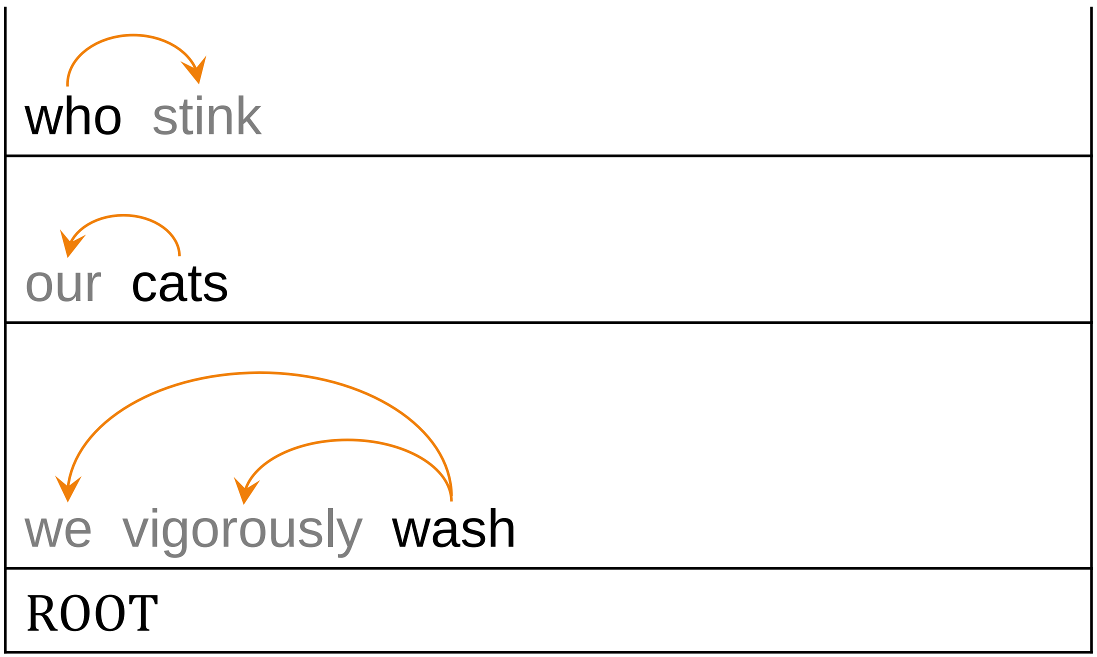


Actions: SHIFT SHIFT SHIFT LEFT−ARC LEFT−ARC SHIFT SHIFT LEFT−ARC SHIFT SHIFT

Transition-Based Parsing: Example

Stack S:

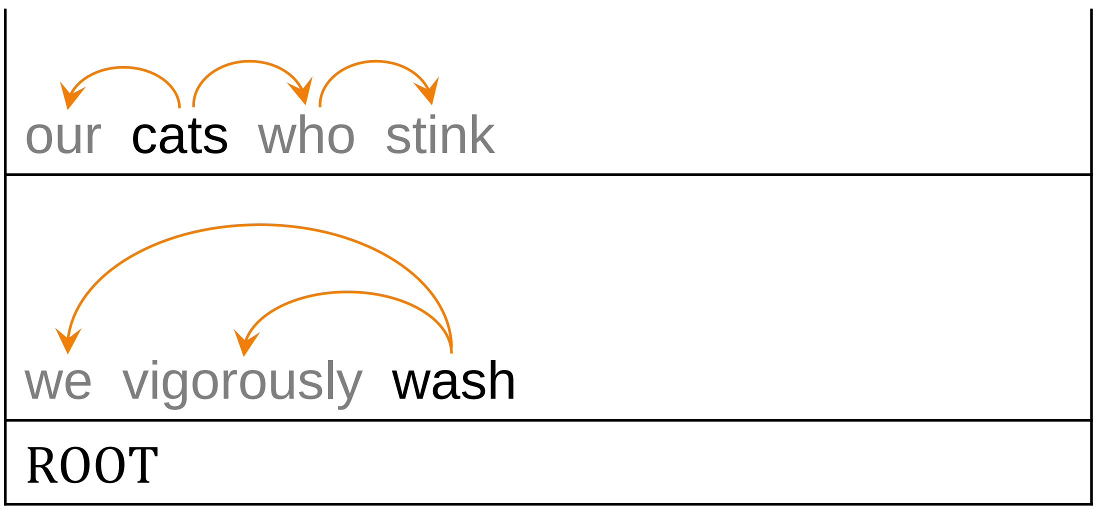
Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT–ARC LEFT–ARC SHIFT SHIFT LEFT–ARC SHIFT SHIFT RIGHT–ARC

Transition-Based Parsing: Example

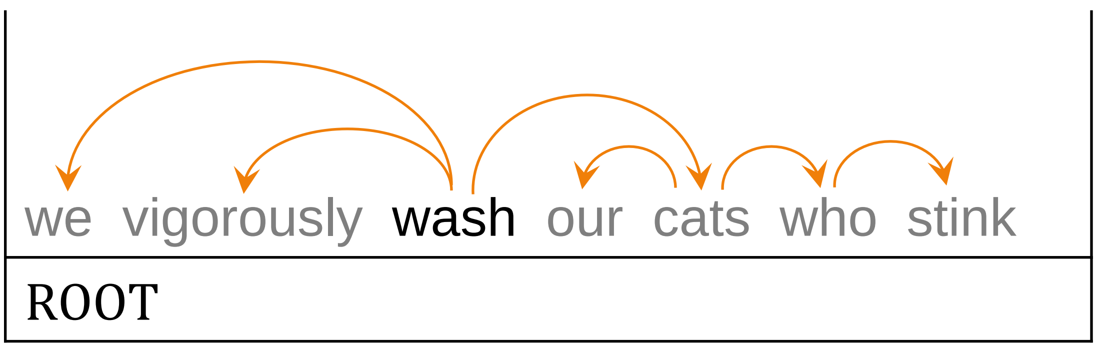
Stack S: Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT−ARC LEFT−ARC SHIFT SHIFT LEFT−ARC SHIFT SHIFT RIGHT−ARC RIGHT−ARC

Transition-Based Parsing: Example

Stack S: Buffer B:

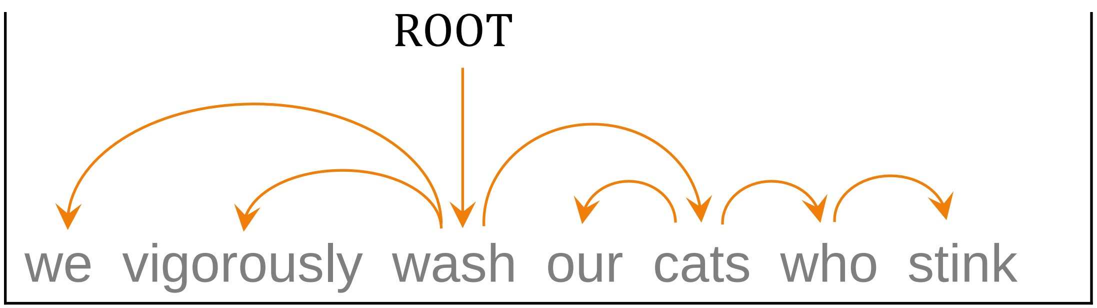


Actions: SHIFT SHIFT SHIFT LEFT−ARC LEFT−ARC SHIFT SHIFT LEFT−ARC SHIFT SHIFT RIGHT−ARC RIGHT−ARC RIGHT−ARC

Transition-Based Parsing: Example

Stack S:

Buffer B:



Actions: SHIFT SHIFT SHIFT LEFT−ARC LEFT−ARC SHIFT SHIFT LEFT−ARC SHIFT SHIFT RIGHT−ARC RIGHT−ARC RIGHT−ARC RIGHT−ARC

Transition-Based Parsing

- See Ch.10 for discussion of classifier design, training and inference
- Time complexity
  - Each word gets SHIFTed once and participates as a child in one ARC.
  - Linear time!

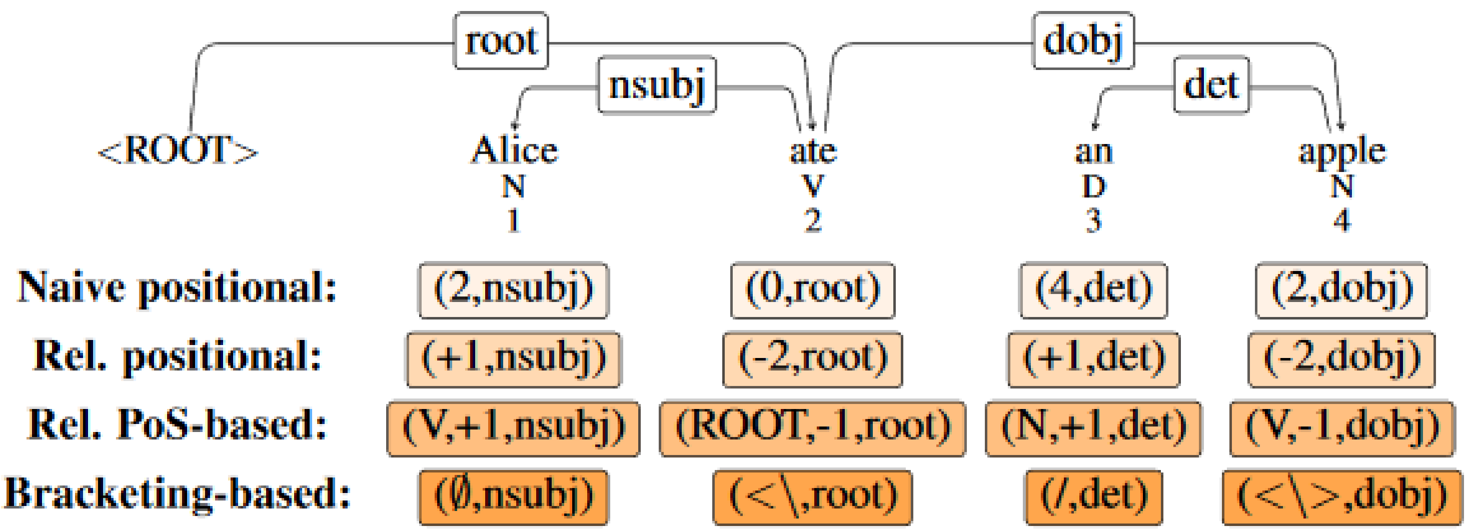
Other Methods

Dependency parsing by head selection

- For each word, learn to predict its head
  - Problem: there may be cycles
  - Surprise: neural models with enough training data can learn to avoid cycles!
- This is similar to first-order graph-based parsing, but without using Eisner or CLE algorithms to avoid cycles
  - Time complexity $O(n^3) \rightarrow O(n^2)$

Dependency parsing as sequence labeling

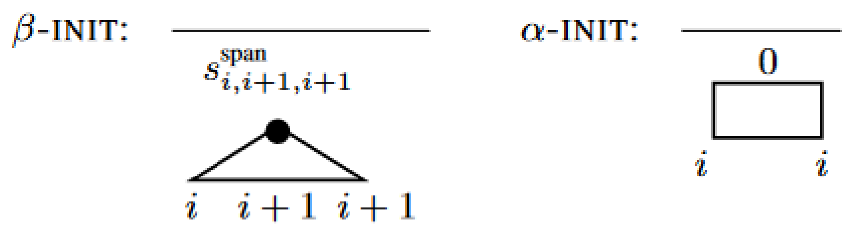
- Cast dependency parsing as a sequence labeling task
  - Advantage: faster parsing speed



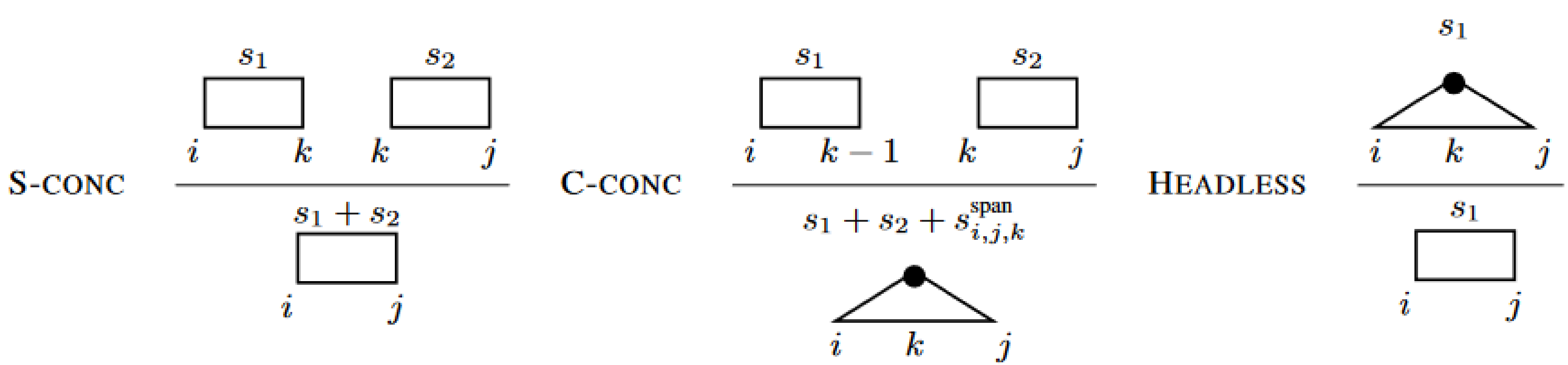
Headed-Span-Based Dependency Parsing

[exclusive, inclusive] span index

Axioms:



Deduction Rules:



Summary

Dependency Parsing

- Dependency Parsing
  - Projectivity
  - Relation to constituency parsing
- Graph-based parsing
  - 1st-order: Eisner, MST
  - Learning
    - Supervised: discriminative methods
    - Unsupervised: EM, CRF-AE
- Transition-based parsing
  - Arc-standard
  - Learning: from configuration to transition