COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

Dependency Parsing (2)

CMSC 470

Marine Carpuat

Fig credits: Joakim Nivre, Dan Jurafsky & James Martin

Transition-based Dependency Parser

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)         ; choose a transition operator to apply
    state ← APPLY(t, state)   ; apply it, creating a new state
  return state
```

Figure 14.6 A generic transition-based dependency parser

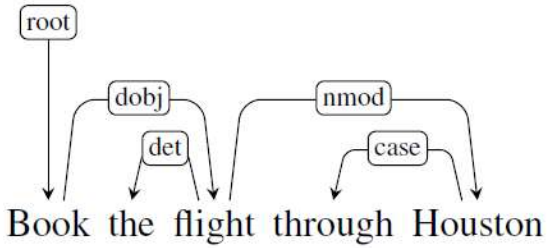
Properties of this algorithm:
- Linear in sentence length
- A greedy algorithm
- Output quality depends on oracle

Where do we get an oracle?

- Multiclass classification problem
  - Input: current parsing state (e.g., current and previous configurations)
  - Output: one transition among all possible transitions
  - Q: size of output space?
    - 3 for unlabeled parsing (LEFT-ARC, RIGHT-ARC, SHIFT)
    - 1 + 2L for labeled parsing where L is the number of labeled dependencies
- Any supervised classifier can be used
  - E.g., perceptron, neural network
- Open questions
  - What are good features for this task?
  - Where do we get training examples?

Generating Training Examples

- What we have in a treebank



- What we need to train an oracle
  - Pairs of configurations and predicted parsing action

Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

Generating training examples

- Approach: simulate parsing to generate reference tree
- Given
  - A current config with stack S, dependency relations Rc
  - A reference parse (V,Rp)
- Do

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t. (S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**

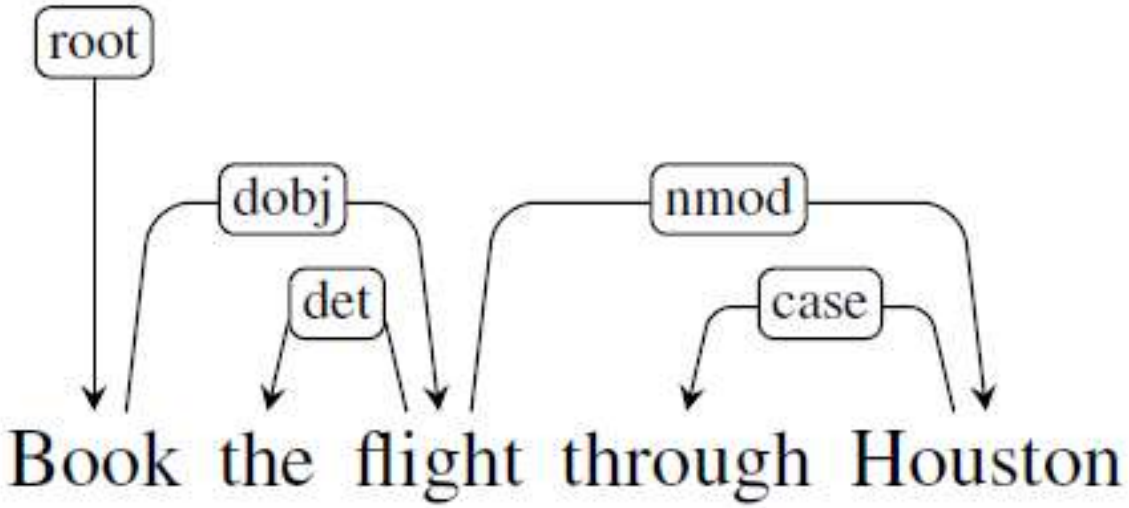
Additional condition on RightArc makes sure a word is not removed from stack before its been attached to all its dependent

Let’s try it out

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t. (S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**



How can we define classifier features?

- What makes a good feature?
  - Captures useful correlations between patterns in input and predicted class
  - Avoid sparsity, encourage generalization

- Here input is parser configuration
  - consists of stack, buffer, current set of relations

- Typical features
  - Features focus on top level of stack
  - Use word forms, POS, and their location in stack and buffer
  - Use dependency relations found so far

Features example

- Given configuration

Stack	Word buffer	Relations
[root, canceled, flights]	[to Houston]	(canceled → United) (flights → morning) (flights → the)

- Example of useful features

$$\langle s_1.w = flights, op = shift \rangle$$
$$\langle s_2.w = canceled, op = shift \rangle$$
$$\langle s_1.t = NNS, op = shift \rangle$$
$$\langle s_2.t = VBD, op = shift \rangle$$
$$\langle b_1.w = to, op = shift \rangle$$
$$\langle b_1.t = TO, op = shift \rangle$$
$$\langle s_1.wt = flightsNNS, op = shift \rangle$$

$$\langle s_1 t.s_2 t = NNSVBD, op = shift \rangle$$

Features example

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Figure 14.9 Standard feature templates for training transition-based dependency parsers. In the template specifications s_n refers to a location on the stack, b_n refers to a location in the word buffer, w refers to the wordform of the input, and t refers to the part of speech of the input.

Research highlight: Dependency parsing with stack-LSTMs

- From Dyer et al. 2015: http://www.aclweb.org/anthology/P15-1033

- Idea
  - Instead of hand-crafted feature
  - Predict next transition using recurrent neural networks to learn representation of stack, buffer, sequence of transitions

Research highlight: Dependency parsing with stack-LSTMs

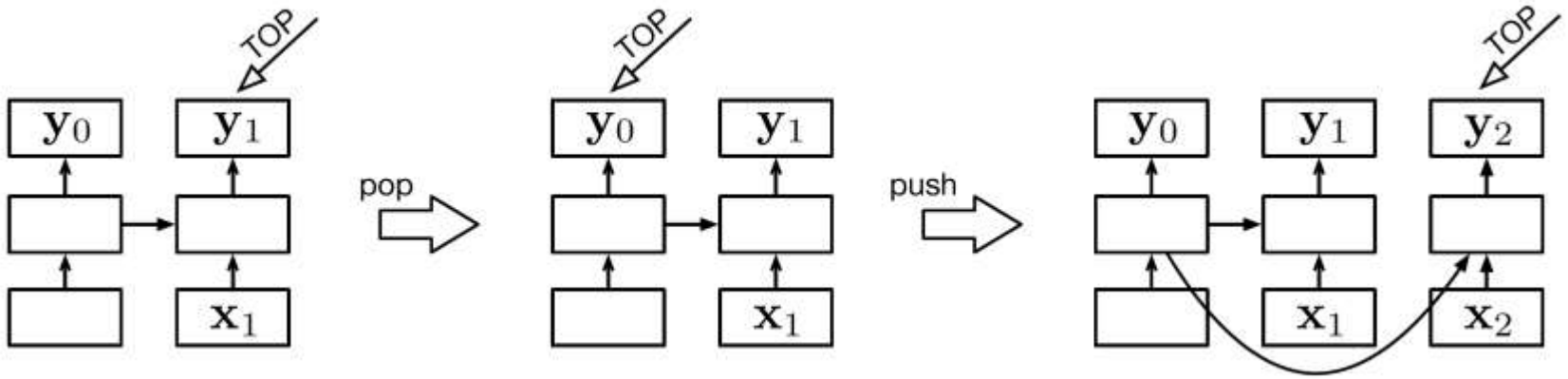


Figure 1: A stack LSTM extends a conventional left-to-right LSTM with the addition of a stack pointer (notated as TOP in the figure). This figure shows three configurations: a stack with a single element (left), the result of a pop operation to this (middle), and then the result of applying a push operation (right). The boxes in the lowest rows represent stack contents, which are the inputs to the LSTM, the upper rows are the outputs of the LSTM (in this paper, only the output pointed to by TOP is ever accessed), and the middle rows are the memory cells (the $\mathbf{c}_t$'s and $\mathbf{h}_t$'s) and gates. Arrows represent function applications (usually affine transformations followed by a nonlinearity), refer to §2.1 for specifics.

Research highlight: Dependency parsing with stack-LSTMs

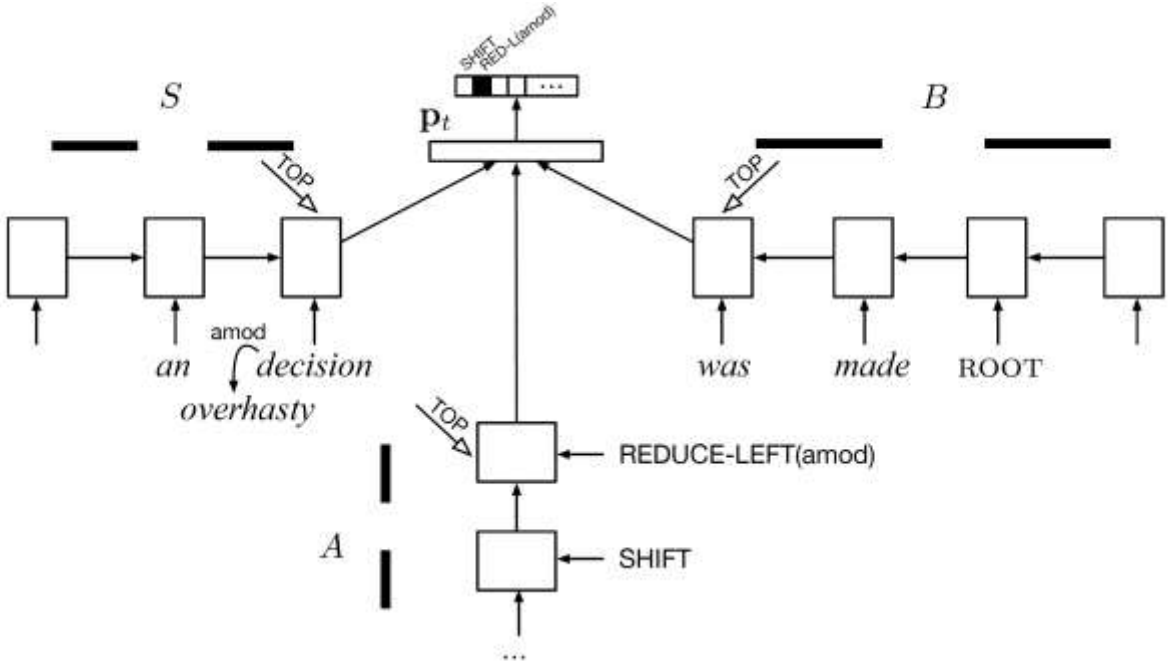
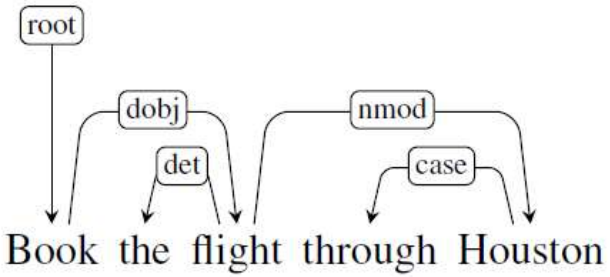


Figure 2: Parser state computation encountered while parsing the sentence "*an overhasty decision was made*." Here S designates the stack of partially constructed dependency subtrees and its LSTM encoding; B is the buffer of words remaining to be processed and its LSTM encoding; and A is the stack representing the history of actions taken by the parser. These are linearly transformed, passed through a ReLU nonlinearity to produce the parser state embedding $\mathbf{p}_t$. An affine transformation of this embedding is passed to a softmax layer to give a distribution over parsing decisions that can be taken.

An Alternative to the Arc-Standard Transition System

A weakness of arc-standard parsing

Right dependents cannot be attached to their head
until all their dependents have been attached



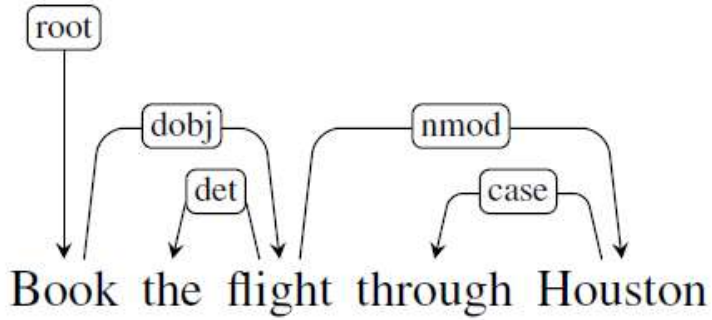
Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

Arc Eager Parsing

- LEFT-ARC
  - Create head-dependent rel. between word at front of buffer and word at top of stack
  - pop the stack
- RIGHT-ARC
  - Create head-dependent rel. between word on top of stack and word at front of buffer
  - Shift buffer head to stack
- SHIFT
  - Remove word at head of input buffer
  - Push it on the stack
- REDUCE
  - Pop the stack

Arc Eager Parsing Example



Step	Stack	Word List	Action	Relation Added
0	[root]	[book, the, flight, through, houston]	RIGHTARC	(root → book)
1	[root, book]	[the, flight, through, houston]	SHIFT	
2	[root, book, the]	[flight, through, houston]	LEFTARC	(the ← flight)
3	[root, book]	[flight, through, houston]	RIGHTARC	(book → flight)
4	[root, book, flight]	[through, houston]	SHIFT	
5	[root, book, flight, through]	[houston]	LEFTARC	(through ← houston)
6	[root, book, flight]	[houston]	RIGHTARC	(flight → houston)
7	[root, book, flight, houston]	[]	REDUCE	
8	[root, book, flight]	[]	REDUCE	
9	[root, book]	[]	REDUCE	
10	[root]	[]	Done	

Figure 14.10 A processing trace of *Book the flight through Houston* using the arc-eager transition operators.

Properties of transition-based parsing algorithms

Trees & Forests

- A dependency tree is a graph satisfying the following conditions
  - Root
  - Single head
  - No cycles
  - Connectedness

- A dependency forest is a dependency graph satisfying
  - Root
  - Single head
  - No cycles
  - but **not** Connectedness

Properties of this transition-based parsing algorithm

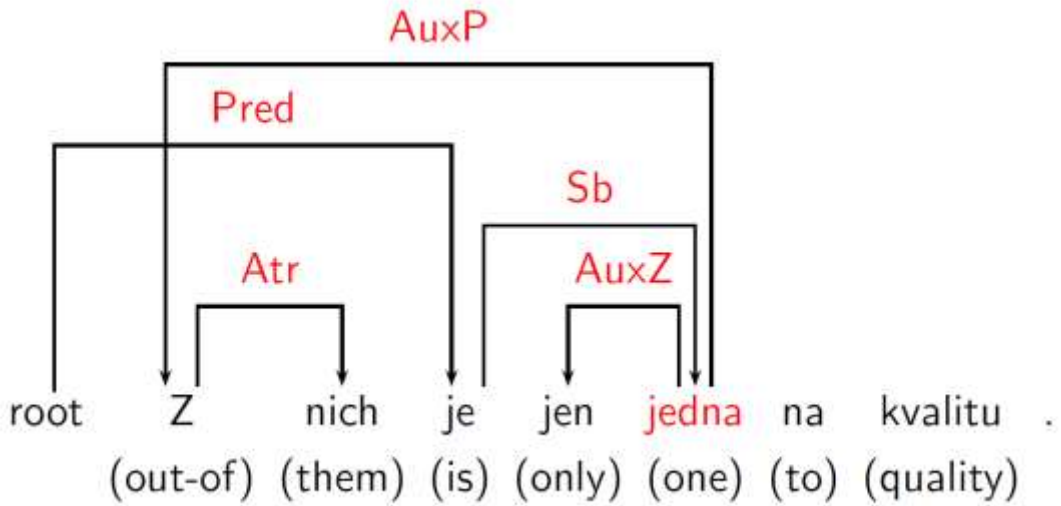
- Correctness
  - For every complete transition sequence, the resulting graph is a projective dependency forest (soundness)
  - For every projective dependency forest G, there is a transition sequence that generates G (completeness)

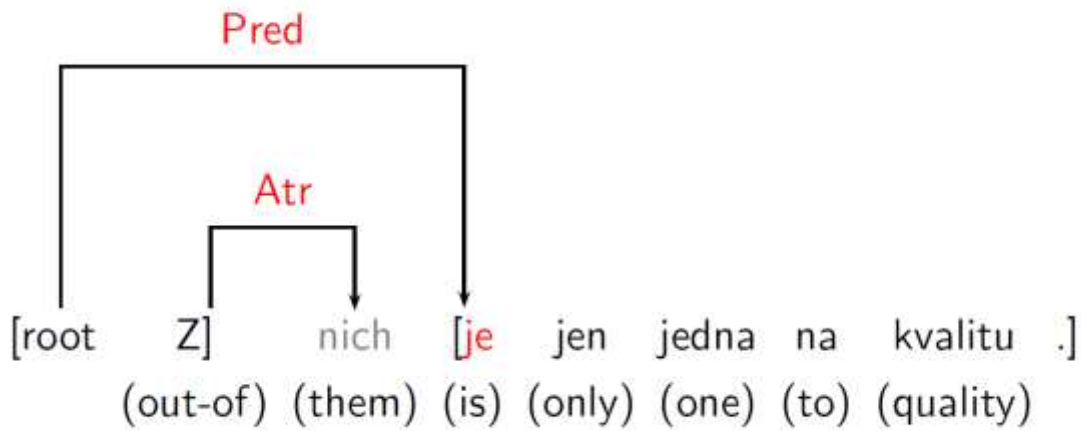
- Trick: forest can be turned into tree by adding links to ROOT_0

Projectivity

- **Arc** from head to dependent is **projective**
  - If there is a path from head to every word between head and dependent
- **Dependency tree** is **projective**
  - If all arcs are projective
  - Or equivalently, if it can be drawn with no crossing edges
- Projective trees make computation easier
- But most theoretical frameworks do not assume projectivity
  - Need to capture long-distance dependencies, free word order

Arc-standard parsing can’t produce non-projective trees




Pred
Atr
[root Z] nich [je jen jedna na kvalitu .]
(out-of) (them) (is) (only) (one) (to) (quality)

How frequent are non-projective structures?

- Statistics from CoNLL shared task
  - NPD = non projective dependencies
  - NPS = non projective sentences

Language	%NPD	%NPS
Dutch	5.4	36.4
German	2.3	27.8
Czech	1.9	23.2
Slovene	1.9	22.2
Portuguese	1.3	18.9
Danish	1.0	15.6

How to deal with non-projectivity? (1) change the transition system

Transition		Precondition
NP-Left$_r$	$(\sigma\|w_i\|w_k, w_j\|\beta, A) \Rightarrow (\sigma\|w_k, w_j\|\beta, A \cup \{(w_j, r, w_i)\})$	$i \neq 0$
NP-Right$_r$	$(\sigma\|w_i\|w_k, w_j\|\beta, A) \Rightarrow (\sigma\|w_i, w_k\|\beta, A \cup \{(w_i, r, w_j)\})$	

- Intuition:
  - Add new transitions
    - That apply to 2nd word of the stack
    - Top word of stack is treated as context

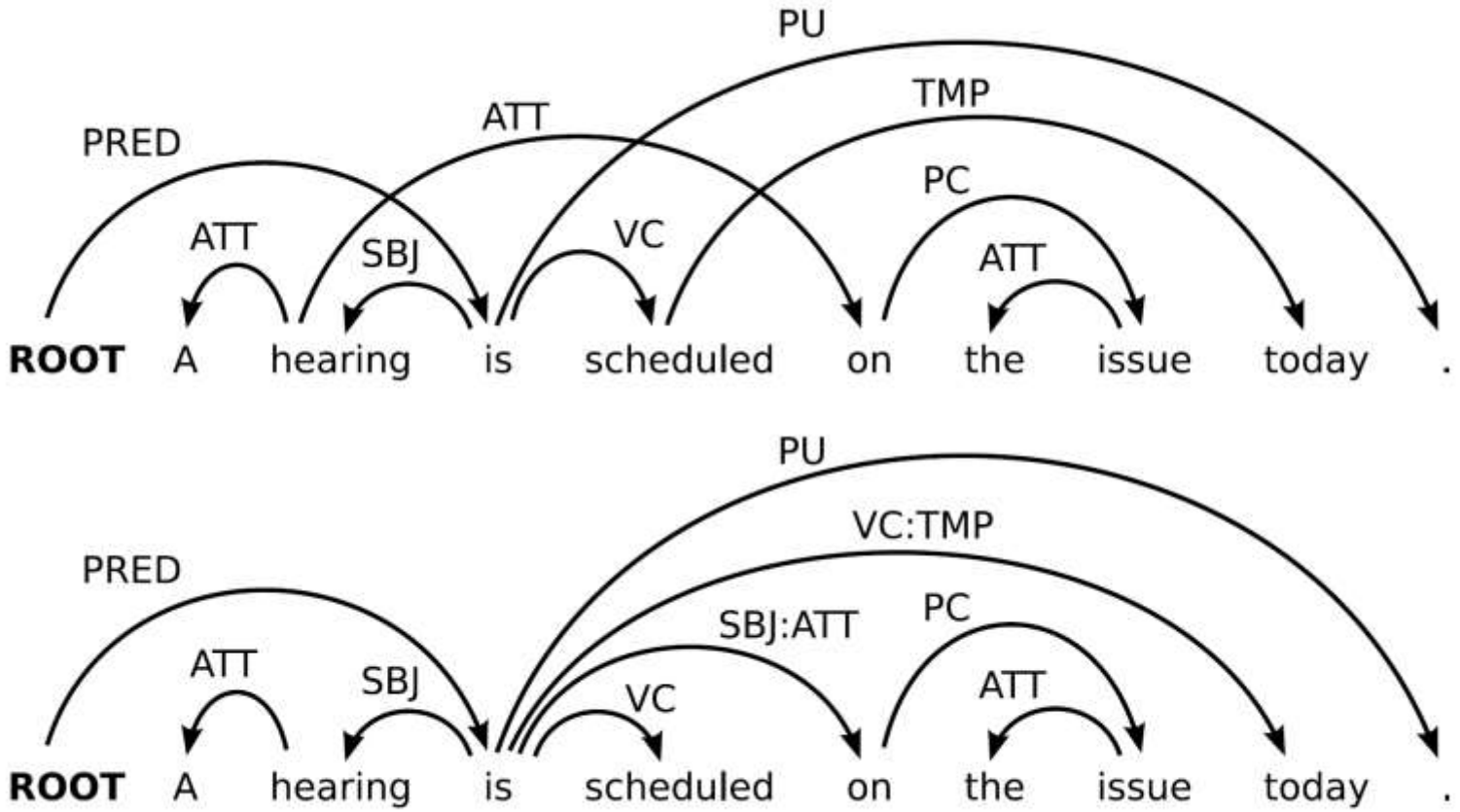
[Attardi 2006]

How to deal with non-projectivity? (2) pseudo-projective parsing

Solution:

- “projectivize” a non-projective tree by creating new projective arcs
- That can be transformed back into non-projective arcs in a post-processing step

How to deal with non-projectivity? (2) pseudo-projective parsing



Dependency Parsing: what you should know

- Transition-based dependency parsing
  - Shift-reduce parsing
  - Transition systems: arc standard, arc eager
  - Oracle algorithm: how to obtain a transition sequence given a tree
  - How to construct a multiclass classifier to predict parsing actions
  - What transition-based parsers can and cannot do
  - That transition-based parsers provide a flexible framework that allows many extensions
    - such as RNNs vs feature engineering, non-projectivity (but I don't expect you to memorize these algorithms)
- Next: Graph-based dependency parsing