

CSC485/2501 A1 Tutorial 2

Samarendra Chandan Bindu Dash



UNIVERSITY OF
TORONTO

Assignment 1

Starter code available on **MarkUs**

Due on **Thursday Oct. 13th**, at **17:00**.

You should work on your code through the **teach.cs** server. The handout has information on how to use GPUs on the server.

Assignment 1

Part 1: Implement your own transition-based dependency parser

Part 2: Implement your own graph-based dependency parser

Assignment 1

Part 1: Implement your own transition-based dependency parser

Part 2: Implement your own graph-based dependency parser

Part 1: Transition-based parser

Tutorial overview

Dependency parsing example with a transition-based parser

Gap degree example

Neural dependency parser with `PyTorch` pointers

Dependency parser

Dependency parser: when given a sentence, it outputs a dependency parse tree.

Three things to keep track of:

1. A **stack of words** being processed.
2. A **buffer of words** to be eventually pushed onto the stack.
3. A **list of predicted dependencies** (i.e. arcs).

Transition-based Parser

There are two variants of the transition-based parser algorithm

1. **Arc-Eager:** In this algorithm, if the top two elements in the stack have a dependency relation then we immediately execute an ARC operation. (This was probably taught to you in class)
2. **Arc-Standard:** In this algorithm, we don't immediately execute ARC operation even when the top two elements (say A and B) in the stack have a dependency relation. (This is simpler to implement)

[1] [An Efficient Algorithm for Projective Dependency Parsing](#) (Nivre, IWPT 2003)

[2] [Incrementality in Deterministic Dependency Parsing](#) (Nivre, 2004)

Transition-based Parser

Three possible operations:

1. **SHIFT**: removes the first word from the buffer and pushes it onto the stack.
2. **LEFT-ARC**: marks the second-from-top item (i.e., second-most recently added word) on the stack as a dependent of the first item and removes the second item from the stack.
3. **RIGHT-ARC**: marks the top item (i.e., most recently added word) on the stack as a dependent of the second item and removes the first item from the stack.

SHIFT Operation

Removes the first word from the buffer and pushes it onto the stack.

- Step T:
 - **Stack**: [ROOT, Stacy, ran]; **Buffer**: [5k, today]
- Step T+1:
 - **Stack**: [ROOT, Stacy, ran, 5k]; **Buffer**: [today]
 - **Action**: SHIFT

LEFT-ARC Operation

Marks the second-from-top item (i.e., second-most recently added word) on the stack as a dependent of the first item and removes the second item from the stack.

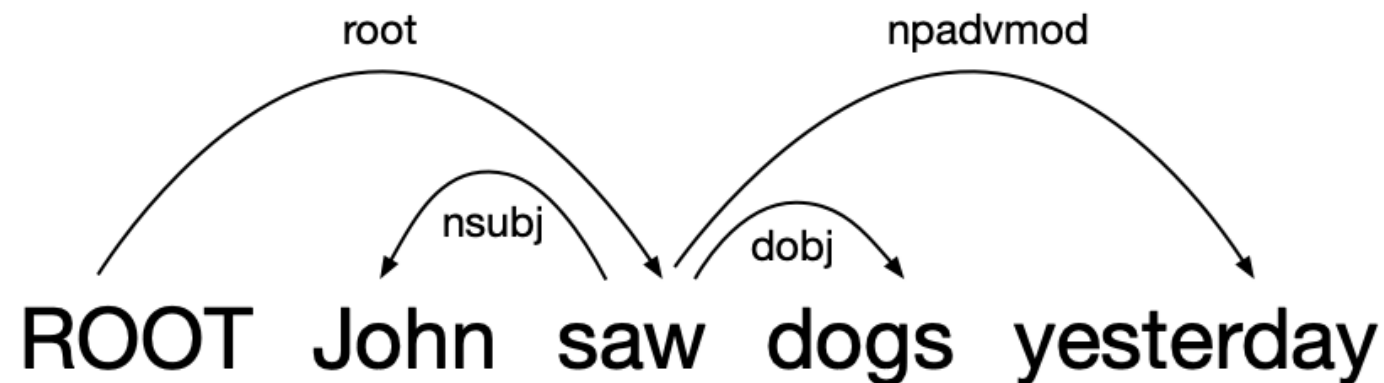
- Step T:
 - **Stack**: [ROOT, Stacy, ran]; **Buffer**: [5k, today]
- Step T+1:
 - **Stack**: [ROOT, ran]; **Buffer**: [5k, today]
 - **New Dependency**: ran $\rightarrow$ Stacy, nsubj
 - **Action**: LEFT-ARC

RIGHT-ARC Operation

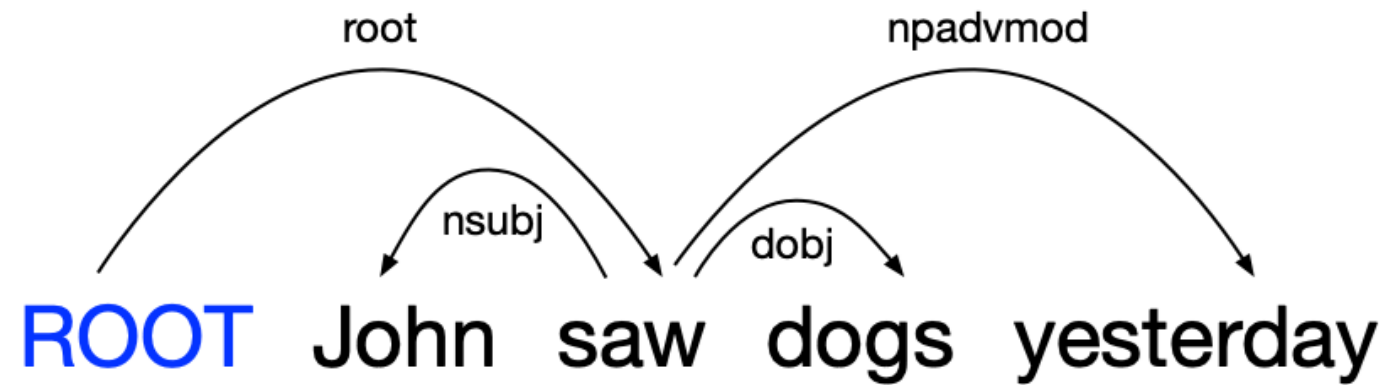
Marks the top item (i.e., most recently added word) on the stack as a dependent of the second item and removes the first item from the stack.

- Step T:
 - **Stack**: [ROOT, ran, 5k]; **Buffer**: [today]
- Step T+1:
 - **Stack**: [ROOT, ran]; **Buffer**: [today]
 - **New Dependency**: ran -> 5k, dobj
 - **Action**: RIGHT-ARC

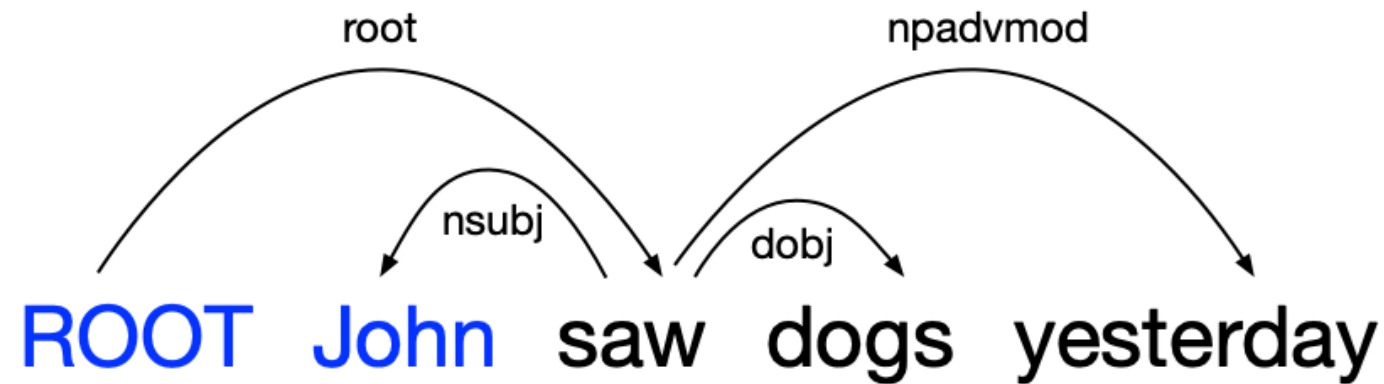
Dependency Parsing Example



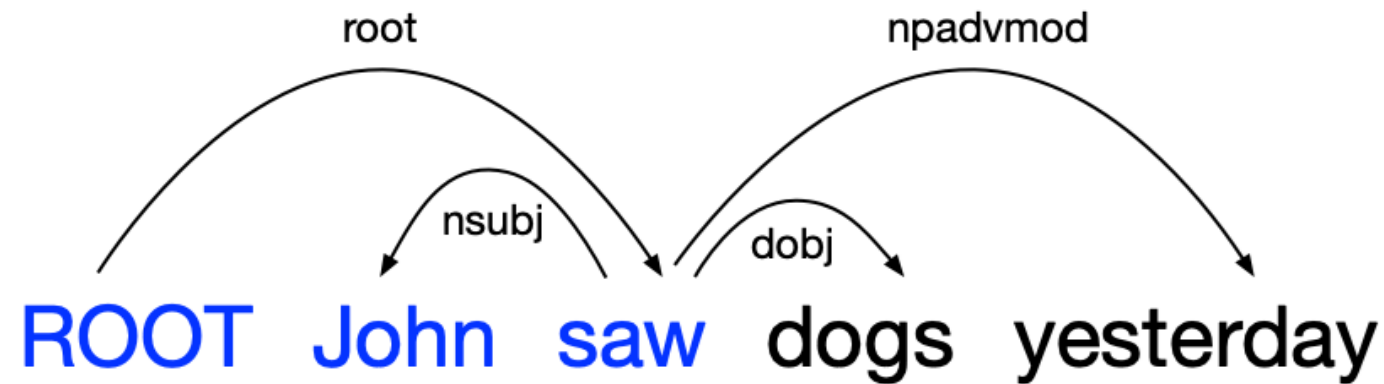
- Given a dependency tree, figure out the intermediate parsing steps.
- Check the top of your stack to see whether it is appropriate to create an arc.
- After creating an arc, record it, and then remove the dependent word from the stack.



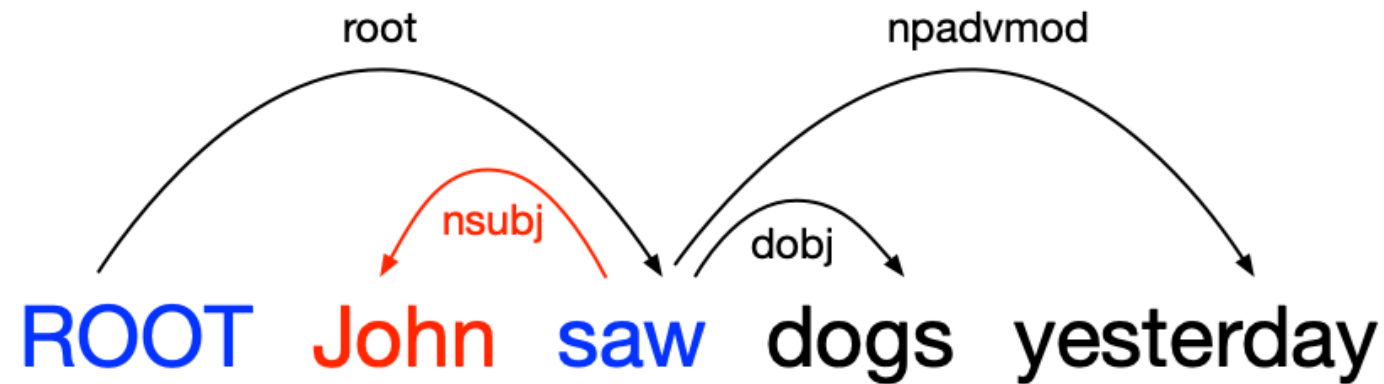
- Step 0:
 - **Stack:** [ROOT]; **Buffer:** [John, saw, dogs, yesterday]



- Step 0:
 - **Stack:** [ROOT]; **Buffer:** [John, saw, dogs, yesterday]
- Step 1:
 - **Stack:** [ROOT, John]; **Buffer:** [saw, dogs, yesterday]
 - **New Dependency:** None
 - **Action:** SHIFT

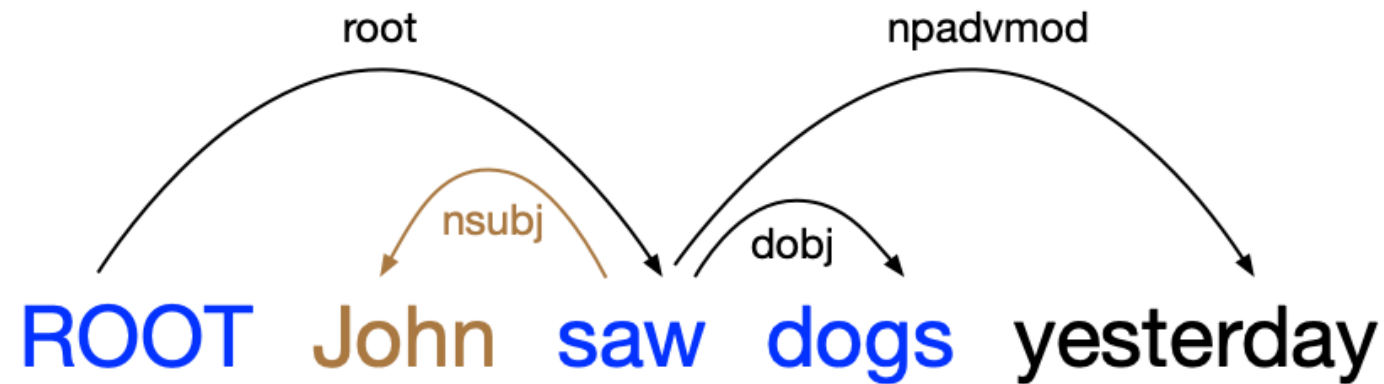


- From Step 1:
 - **Stack:** [ROOT, John]; **Buffer:** [saw, dogs, yesterday]
- Step 2:
 - **Stack:** [ROOT, John, saw]; **Buffer:** [dogs, yesterday]
 - **New Dependency:** None
 - **Action:** SHIFT

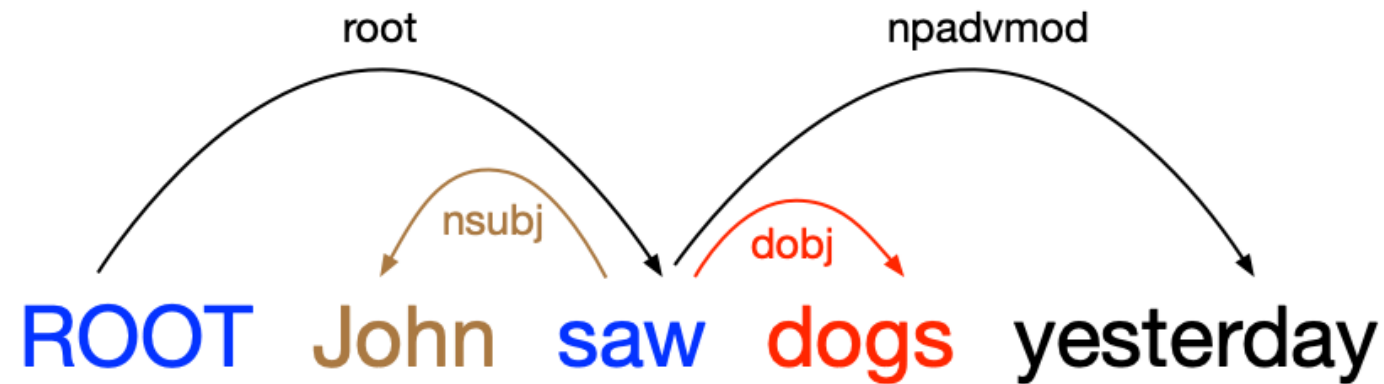


- From Step 2:
 - **Stack**: [ROOT, John, saw]; **Buffer**: [dogs, yesterday]
- Step 3:
 - **Stack**: [ROOT, saw]; **Buffer**: [dogs, yesterday]
 - **New Dependency**: saw → John, nsubj
 - **Action**: LEFT-ARC

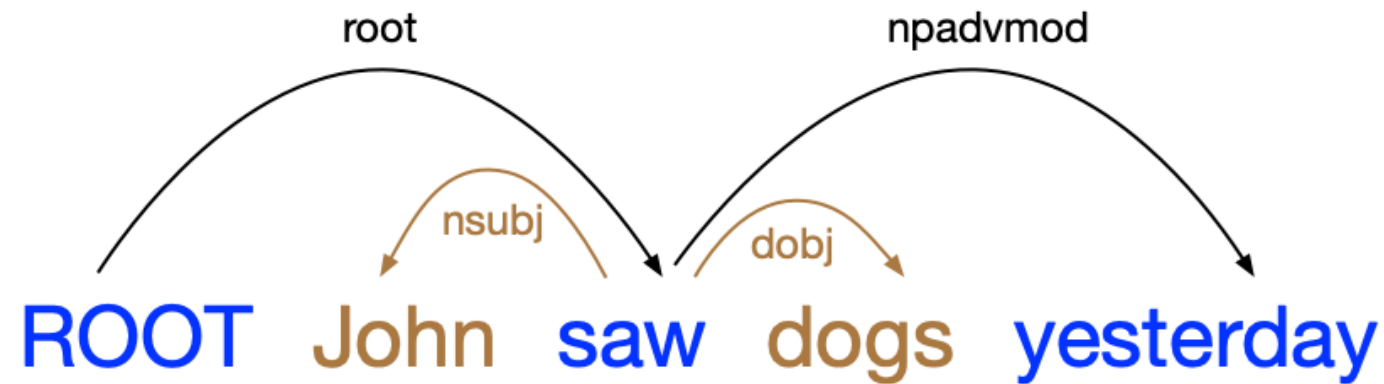
For this assignment:
Choose LEFT-ARC over SHIFT
when both are valid and generate the
same tree.



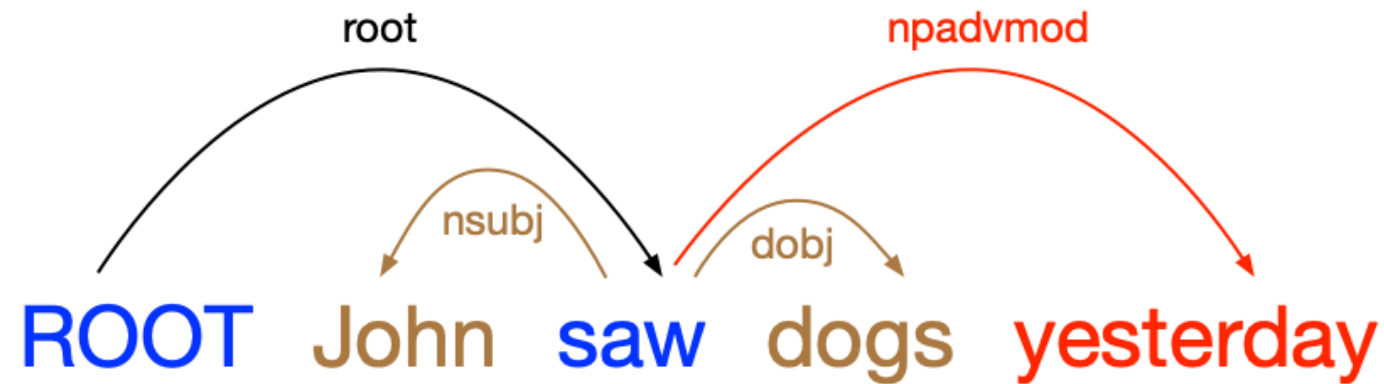
- From Step 3:
 - **Stack**: [ROOT, saw]; **Buffer**: [dogs, yesterday]
- Step 4:
 - **Stack**: [ROOT, saw, dogs]; **Buffer**: [yesterday]
 - **New Dependency**: None
 - **Action**: SHIFT



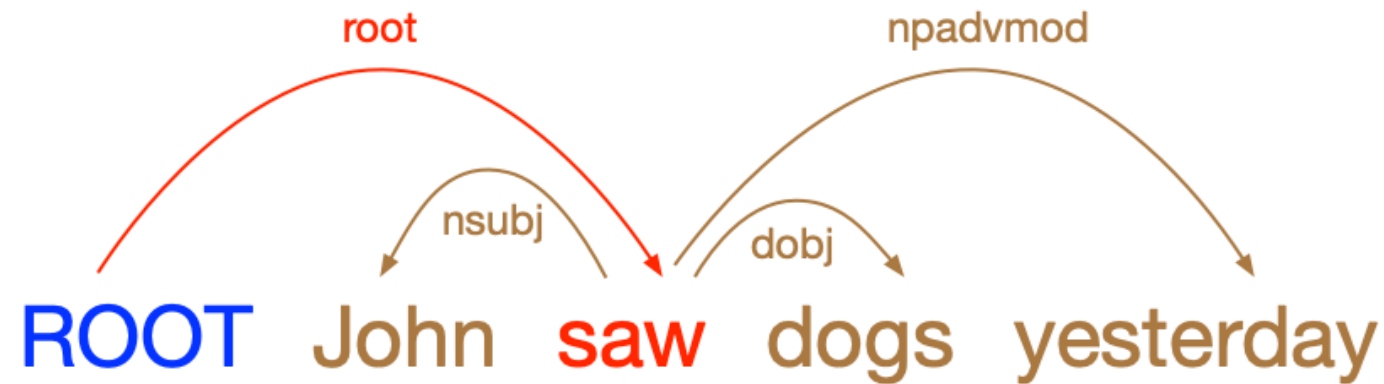
- From Step 4:
 - **Stack**: [ROOT, saw, dogs]; **Buffer**: [yesterday]
- Step 5:
 - **Stack**: [ROOT, saw]; **Buffer**: [yesterday]
 - **New Dependency**: saw -> dogs, dobj
 - **Action**: RIGHT-ARC



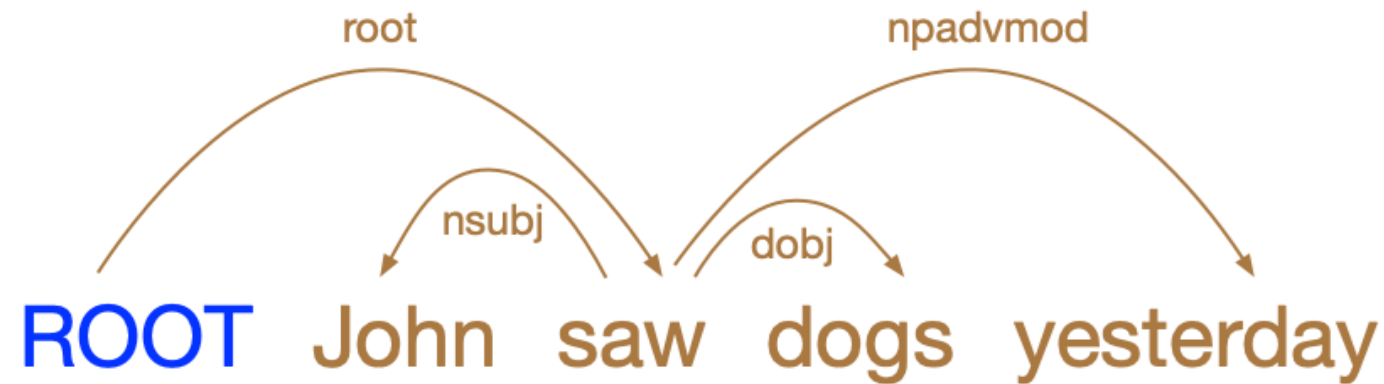
- From Step 5:
 - **Stack:** [ROOT, saw]; **Buffer:** [yesterday]
- Step 6:
 - **Stack:** [ROOT, saw, yesterday]; **Buffer:** []
 - **New Dependency:** None
 - **Action:** SHIFT



- From Step 6:
 - **Stack**: [ROOT, saw, yesterday]; **Buffer**: []
- Step 7:
 - **Stack**: [ROOT, saw]; **Buffer**: []
 - **New Dependency**: saw -> yesterday, npadvmod
 - **Action**: RIGHT-ARC



- From Step 7:
 - **Stack:** [ROOT, saw]; **Buffer:** []
- Step 8:
 - **Stack:** [ROOT]; **Buffer:** []
 - **New Dependency:** ROOT -> saw, root
 - **Action:** RIGHT-ARC



- We've figured out all the parsing steps!
- Similar exercise in the assignment.
- How to do this algorithmically? What are the conditions?

QUIZ: 25 SEPT.

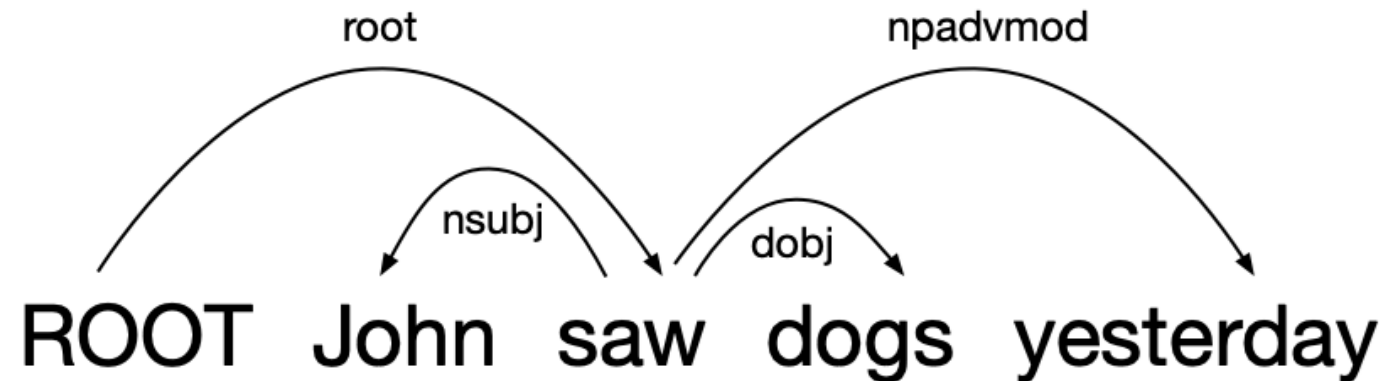
In the sentence, *Mary had a little lamb, little* attaches to its head using Nivre's _ rule.

(We are using Arc-Eager Algorithm here)

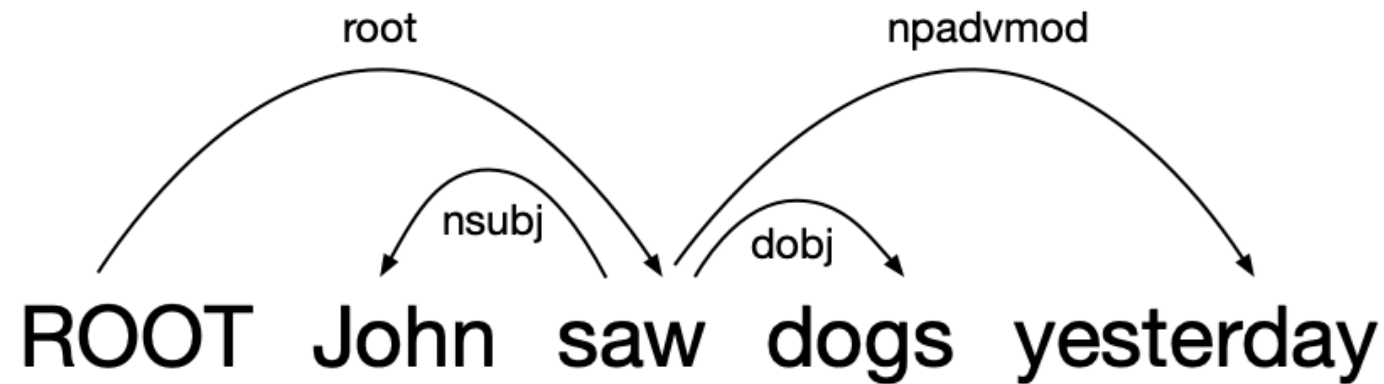
- a) Shift
- b) Reduce
- c) Left-Arc
- d) Right-Arc

Deadline: 5PM Today

Gap Degree Example

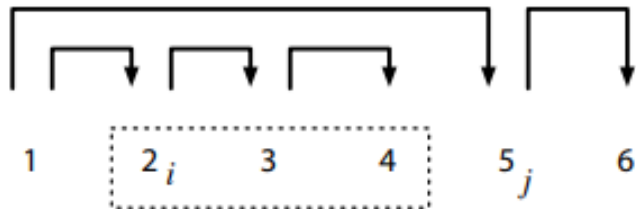


- The **gap degree of a word** in a dependency tree is the least k for which the subsequence consisting of the word and its descendants (both direct and indirect) is entirely comprised of $k + 1$ maximally contiguous substrings.
- The gap degree of a word is the *number of gaps* in the subsequence formed by the word and all its descendants, regardless of the *size* of the gaps.
- The **gap degree of a dependency tree** is the greatest gap degree of any word in the tree.

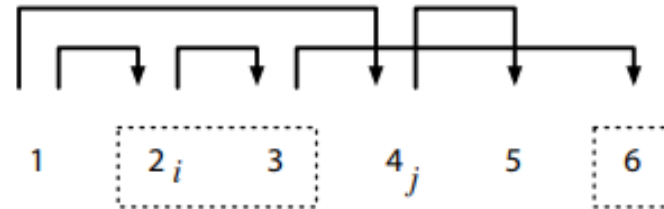


- For each word, check the substring consisting itself and all its descendants:
 - ROOT: ROOT John saw dogs yesterday
 - John: John
 - saw: John saw dogs yesterday
 - dogs: dogs
 - yesterday: yesterday

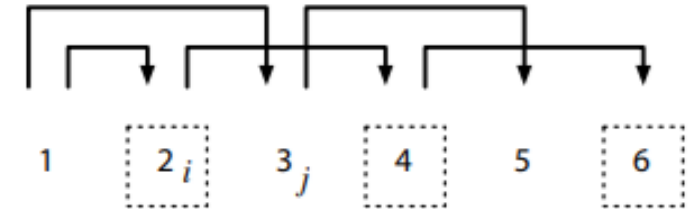
All substrings are
contiguous
 $k=0$



Gap degree = 0

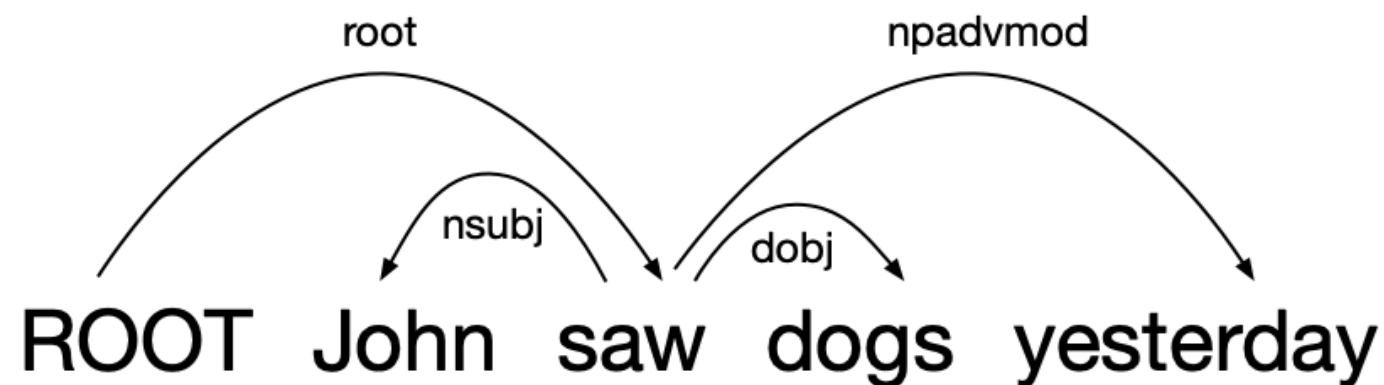


Gap degree = 1



Gap degree = 2

Neural Dependency Parser



- Now assume we don't have the dependency tree.

Neural Dependency Parser

ROOT John saw dogs yesterday

- Now assume we don't have the dependency tree.
- When do we need to make decisions when parsing?

Neural Dependency Parser

ROOT John saw dogs yesterday

- Suppose we have the following partial parse:
 - **Stack**: [ROOT, John, saw]; **Buffer**: [dogs, yesterday]
- Now we need to decide which transition to do next:
 - a) **SHIFT**: Shift dogs onto the stack
 - b) **LEFT-ARC**: create the arc: saw → john
 - c) **RIGHT-ARC**: create the arc john → saw

Neural Dependency Parser

ROOT John saw dogs yesterday

- Use a neural network to make a prediction at each parse step.
- Implement this in PyTorch, read the docs and refer back to the tutorial if you're not familiar:
 - <https://pytorch.org/docs/stable/index.html>

Neural Dependency Parser

ROOT John saw dogs yesterday

- **Input:** Word level features (e.g. word embeddings) for each word in the sentence.
- One linear (fully-connected) **hidden layer**.
- A **softmax layer** to obtain a probability distribution over transitions.

Neural Dependency Parser

ROOT John saw dogs yesterday

- **Input:** Word level features (e.g. word embeddings) for each word in the sentence.
 - `torch.nn.Embedding(size, shape)`
 - `torch.nn.Embedding.from_pretrained(...)`
 - Make sure you DON'T freeze the pre-trained embeddings!!

Neural Dependency Parser

ROOT John saw dogs yesterday

- One linear (fully-connected) **hidden layer**.
 - `hidden_layer = torch.nn.Linear(input_size, output_size)`
 - To apply: `hidden_layer(features)`

You can also checkout `torch.nn.relu(...)` and `torch.nn.dropout(...)`

Neural Dependency Parser

ROOT John saw dogs yesterday

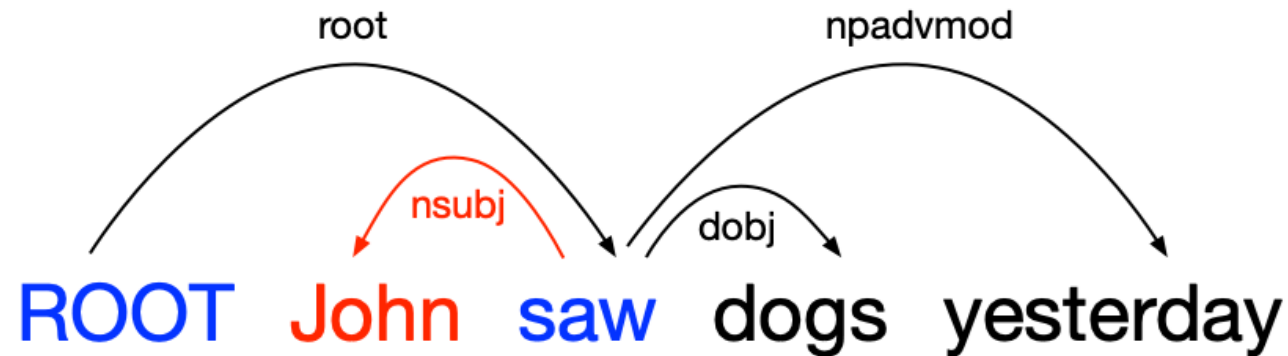
- A **softmax layer** to obtain a probability distribution over transitions.
 - `torch.nn.CrossEntropyLoss`
 - `torch.nn.functional.cross_entropy`

Neural Dependency Parser



- Suppose our neural network gives us an answer:
 - a) **SHIFT**: Shift dogs onto the stack
 - b) **LEFT-ARC**: create the arc: saw $\rightarrow$ john
 - c) **RIGHT-ARC**: create the arc john $\rightarrow$ saw
- How can we tell whether we have made the right choice?

Neural Dependency Parser



- How can we tell whether we have made the right choice?
 - Implement an “**oracle**” that peaks into the parsed tree and tells us the correct transition to make.
- Think about the first example we did in this tutorial.
 - How to make the process automatic?
 - What conditions need to be met to make a particular transition?

[3] [A Fast and Accurate Dependency Parser using Neural Networks](#) (Chen & Manning, EMNLP 2014)

So...

One pitfall of the transition-based parser is that it can only handle projective parse trees (you can try to think about why this is)

Next time, we will look at graph-based dependency parsing, which accounts for non-projective trees.