CS11-747 Neural Networks for NLP

Transition-based Parsing with Neural Nets

Graham Neubig

Carnegie Mellon University

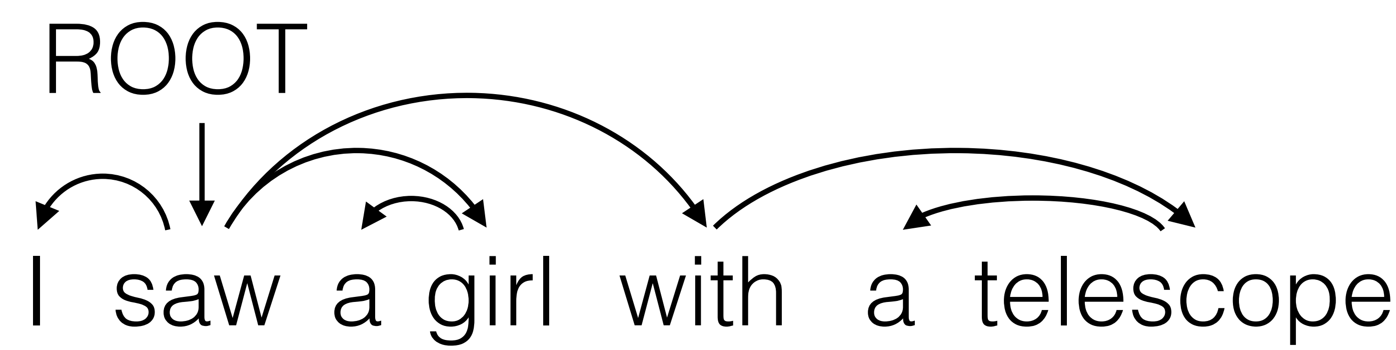
Language Technologies Institute

Site

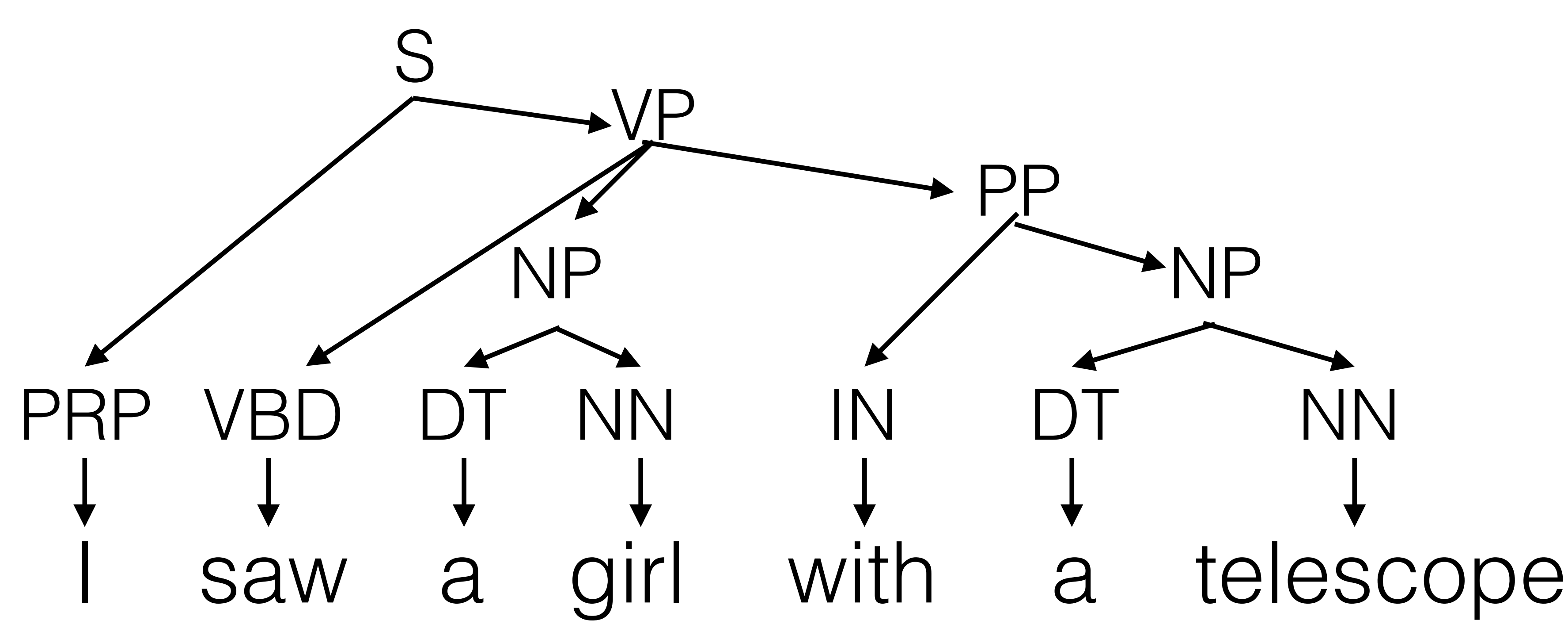
https://phontron.com/class/nn4nlp2019/

Two Types of Linguistic Structure

- **Dependency:** focus on relations between words



- **Phrase structure:** focus on the structure of the sentence



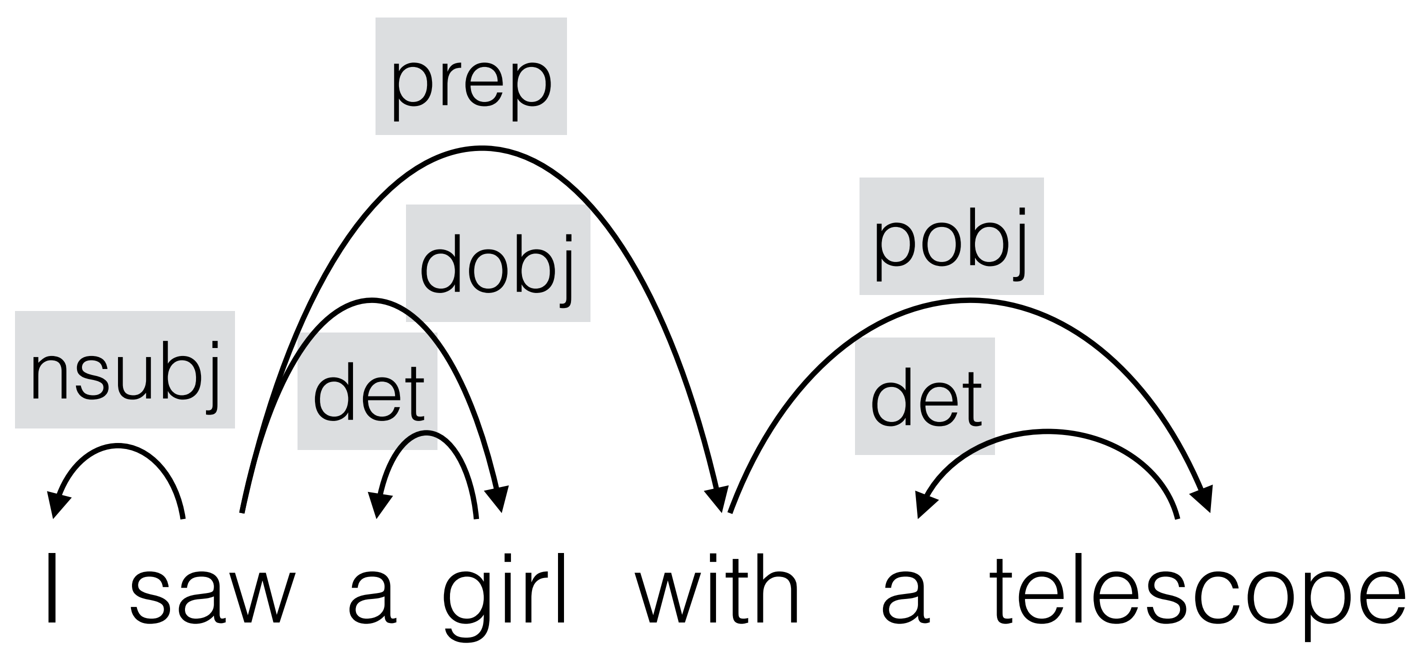
Parsing

- Predicting linguistic structure from input sentence
- **Transition-based models**
  - step through actions one-by-one until we have output
  - like history-based model for POS tagging
- **Graph-based models**
  - calculate probability of each edge/constituent, and perform some sort of dynamic programming
  - like linear CRF model for POS

Shift-reduce Dependency Parsing

Why Dependencies?

- Dependencies are often good for semantic tasks, as related words are close in the tree
- It is also possible to create labeled dependencies, that explicitly show the relationship between words

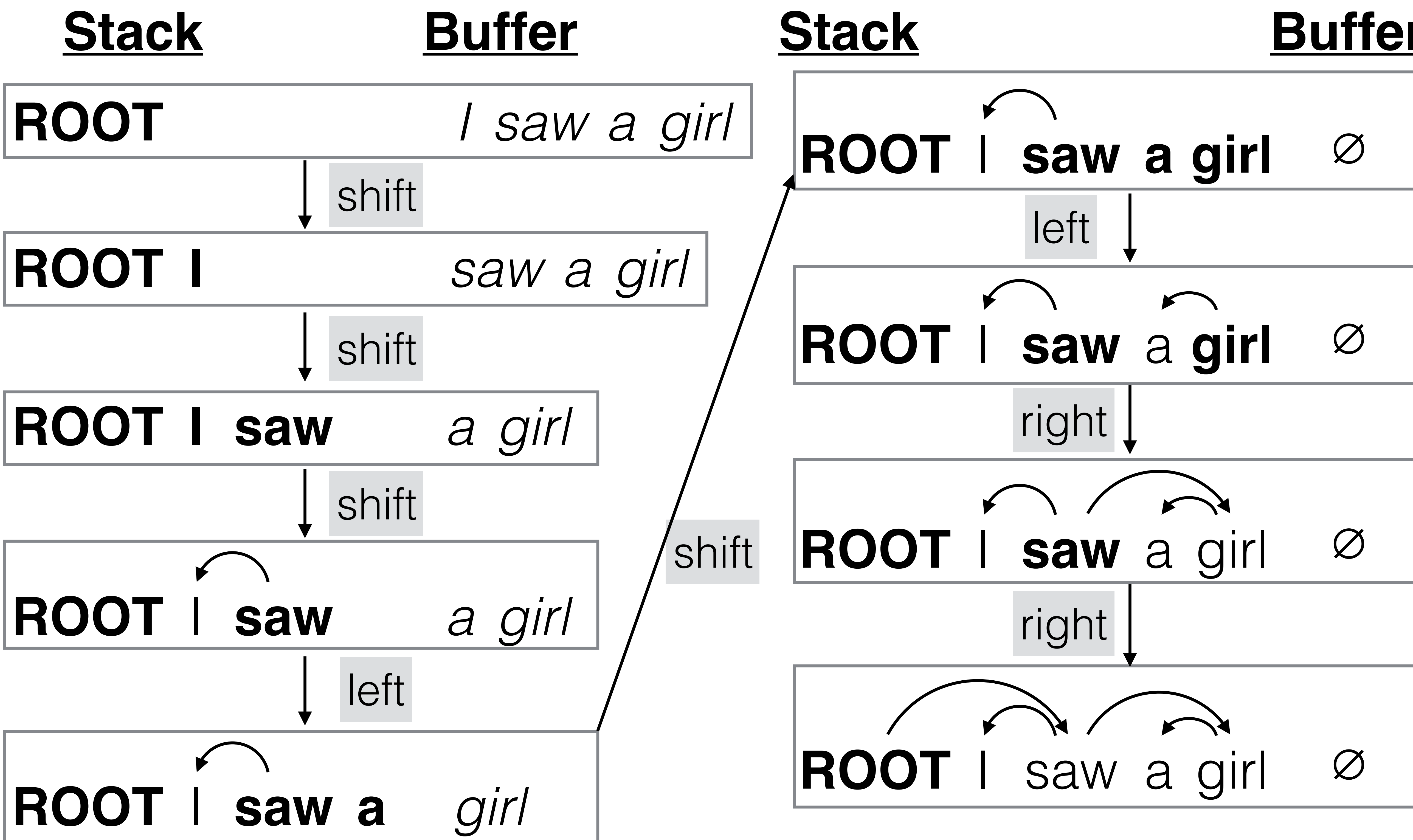


Arc Standard Shift-Reduce Parsing

(Yamada & Matsumoto 2003, Nivre 2003)

- Process words one-by-one left-to-right
- Two data structures
  - **Queue:** of unprocessed words
  - **Stack:** of partially processed words
- At each point choose
  - **shift:** move one word from queue to stack
  - **reduce left:** top word on stack is head of second word
  - **reduce right:** second word on stack is head of top word
- Learn how to choose each action with a classifier

Shift Reduce Example



Classification for Shift-reduce

- Given a **configuration**

<u>Stack</u> **<u>Buffer</u>**

ROOT I saw a *girl*

- Which **action** do we choose?

shift

ROOT I saw a girl ∅

right

ROOT I saw a *girl*

left

ROOT I saw a *girl*

Making Classification Decisions

- Extract features from the configuration
  - what words are on the stack/buffer?
  - what are their POS tags?
  - what are their children?
- Feature combinations are important!
  - Second word on stack is verb **AND** first is noun: “right” action is likely
- Combination features used to be created manually (e.g. Zhang and Nivre 2011), now we can use neural nets!

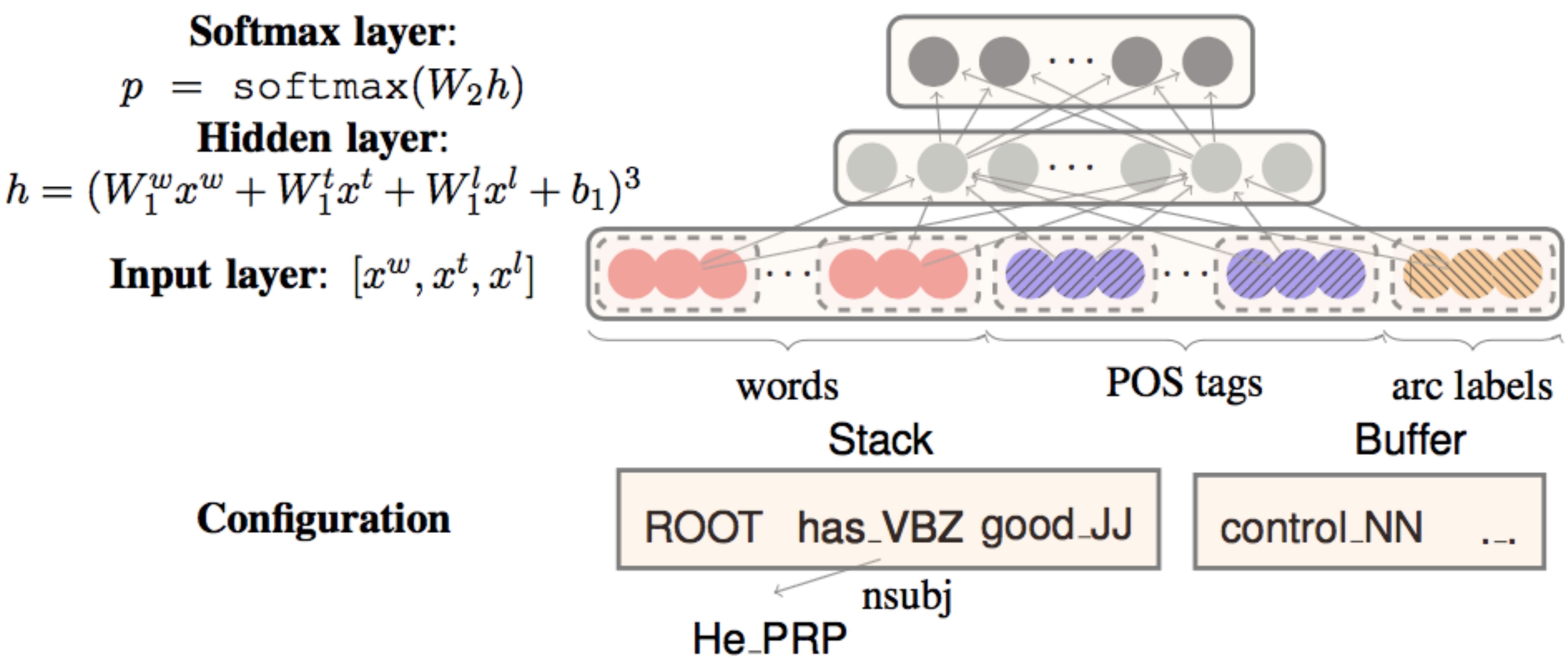
A Feed-forward Neural Model for Shift-reduce Parsing

(Chen and Manning 2014)

A Feed-forward Neural Model for Shift-reduce Parsing

(Chen and Manning 2014)

- Extract non-combined features (embeddings)
- Let the neural net do the feature combination



What Features to Extract?

- The top 3 words on the stack and buffer (6 features)
  s_1, s_2, s_3, b_1, b_2, b_3

- The two leftmost/rightmost children of the top two words on the stack (8 features)
  $lc_1(s_i)$, $lc_2(s_i)$, $rc_1(s_i)$, $rc_2(s_i)$ $i=1,2$

- leftmost and rightmost grandchildren (4 features)
  $lc_1(lc_1(s_i))$, $rc_1(rc_1(s_i))$ $i=1,2$

- POS tags of all of the above (18 features)

- Arc labels of all children/grandchildren (12 features)

Non-linear Function: Cube Function

- Take the cube of the input value vector

$$h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3$$

- Why? Directly extracts feature combinations of up to three (similar to Polynomial Kernel in SVMs)

$$g(w_1 x_1 + \ldots + w_m x_m + b) =$$
$$\sum_{i,j,k} (w_i w_j w_k) x_i x_j x_k + \sum_{i,j} b(w_i w_j) x_i x_j \ldots$$

Result

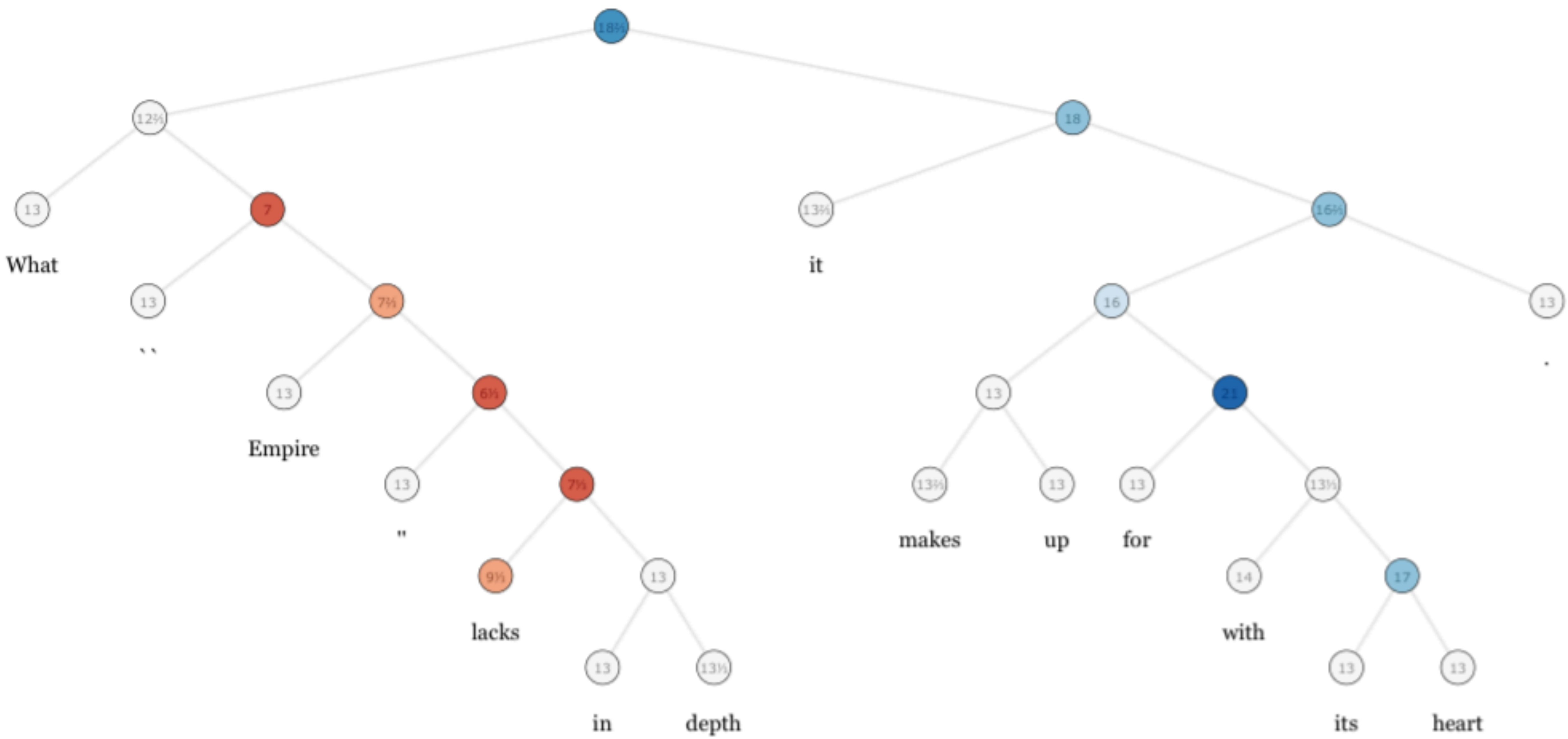
- Faster than most standard dependency parsers (1000 words/second)
    - Use pre-computation trick to cache matrix multiplies of common words
- Strong results, beating most existing transition-based parsers at the time

Let’s Try it Out!

`ff-depparser.py`

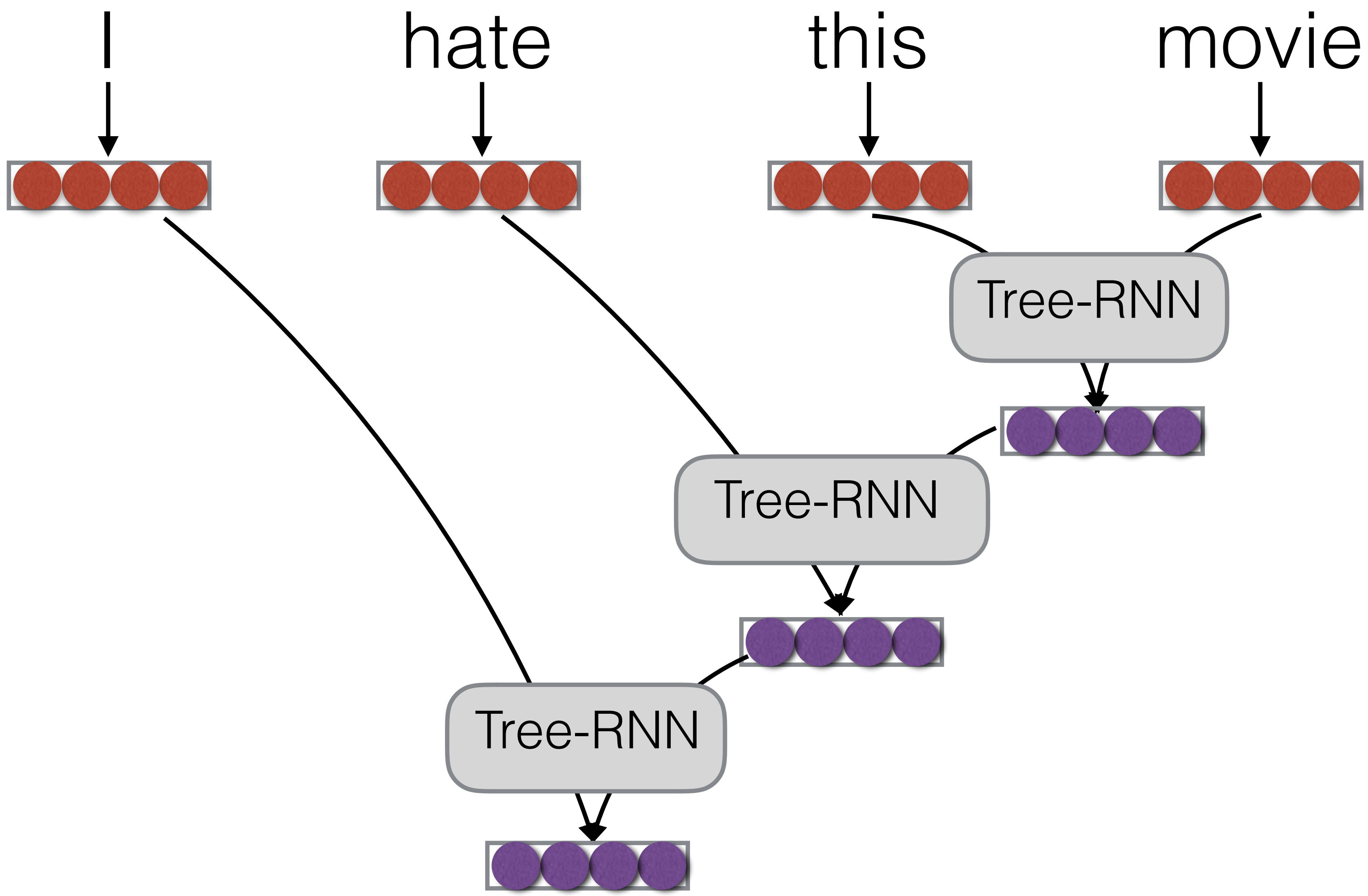
Using Tree Structure in NNs: Syntactic Composition

Why Tree Structure?



Recursive Neural Networks

(Socher et al. 2011)



$$\text{tree-rnn}(\boldsymbol{h}_1, \boldsymbol{h}_2) = \tanh(W[\boldsymbol{h}_1; \boldsymbol{h}_2] + \boldsymbol{b})$$

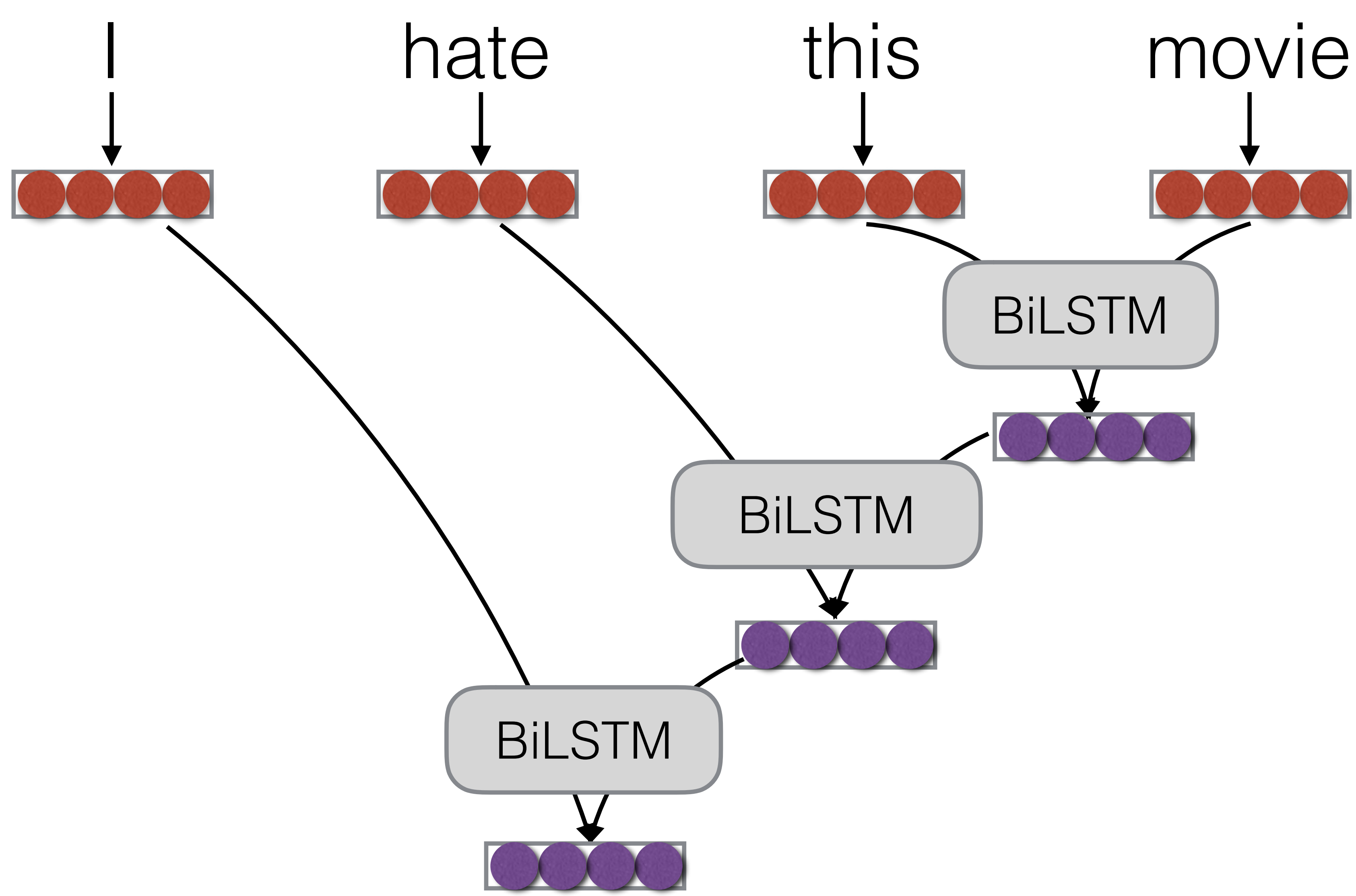
Can also parameterize by constituent type → different composition behavior for NP, VP, etc.

Tree-structured LSTM
(Tai et al. 2015)

- **Child Sum Tree-LSTM**
  - Parameters shared between all children (possibly based on grammatical label, etc.)
  - Forget gate value is different for each child → the network can learn to “ignore” children (e.g. give less weight to non-head nodes)
- **N-ary Tree-LSTM**
  - Different parameters for each child, up to N (like the Tree RNN)

Bi-LSTM Composition
(Dyer et al. 2015)

- Simply read in the constituents with a BiLSTM
- The model can learn its own composition function!



Let's Try it Out!

`tree-lstm.py`

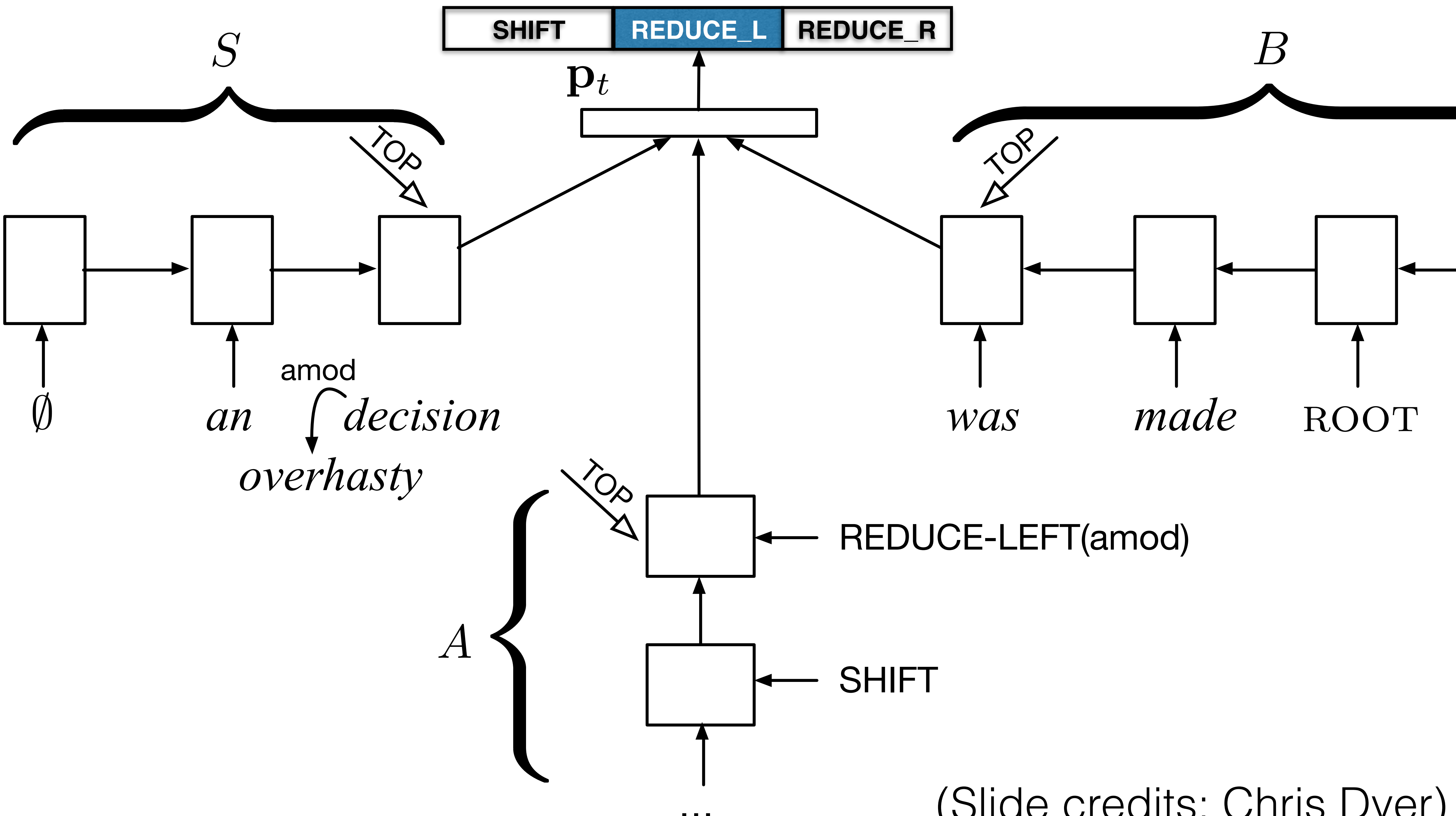
Stack LSTM: Dependency Parsing w/ Less Engineering, Wider Context

(Dyer et al. 2015)

Encoding Parsing Configurations w/ RNNs

- We don’t want to do feature engineering (why leftmost and rightmost grandchildren only?!)
- Can we encode all the information about the parse configuration with an RNN?
- Information we have: stack, buffer, past actions

Encoding Stack Configurations w/ RNNs



(Slide credits: Chris Dyer)

Transition-based parsing
State embeddings

- We can embed words, and can embed tree fragments using syntactic compositon
- The contents of the buffer are just a sequence of embedded words
  - which we periodically "shift" from
- The contents of the stack is just a sequence of embedded trees
  - which we periodically pop from and push to
- Sequences -> use RNNs to get an encoding!
- But running an RNN for each state will be expensive. **Can we do better?**

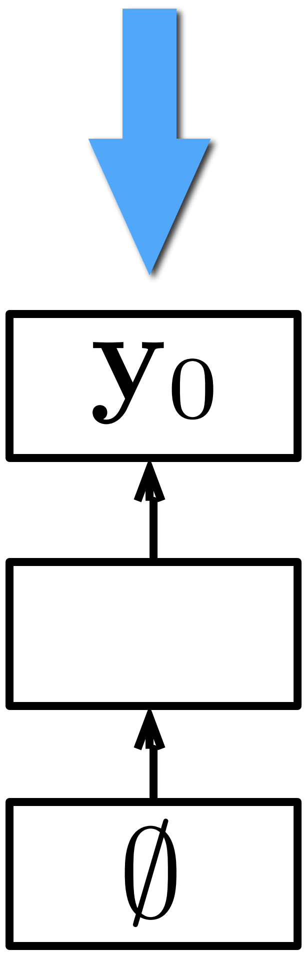
(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs

- Augment RNN with a **stack pointer**
- Three ***constant-time*** operations
  - **`push`** - read input, add to top of stack
  - **`pop`** - move stack pointer back
  - **`embedding`** - return the RNN state at the location of the stack pointer (which summarizes its current contents)

(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs

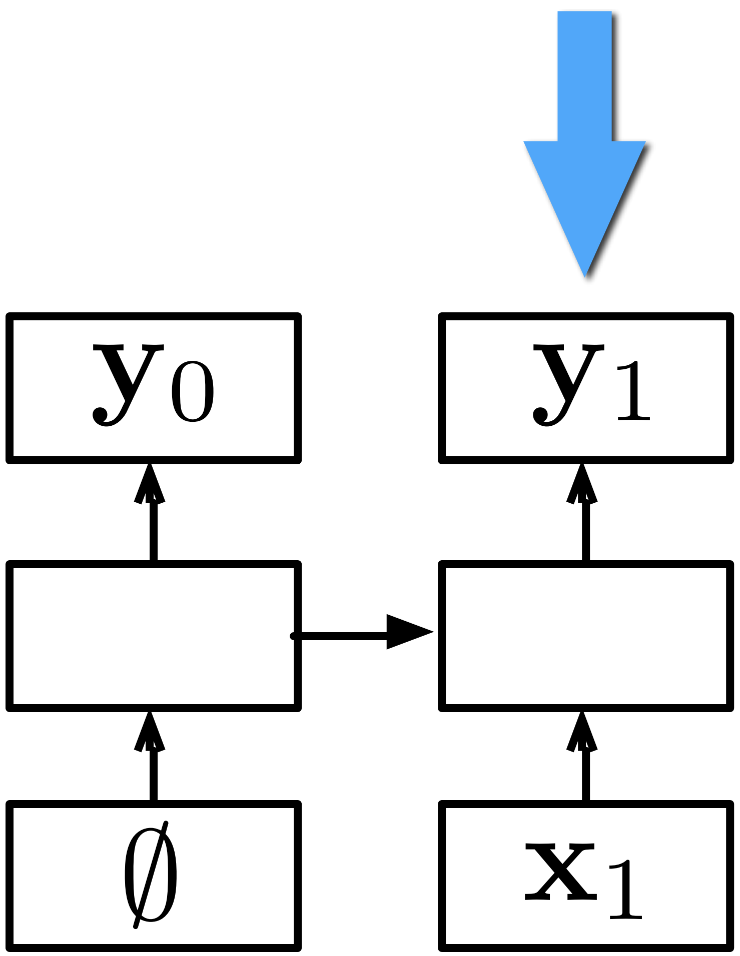


DyNet:

```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs

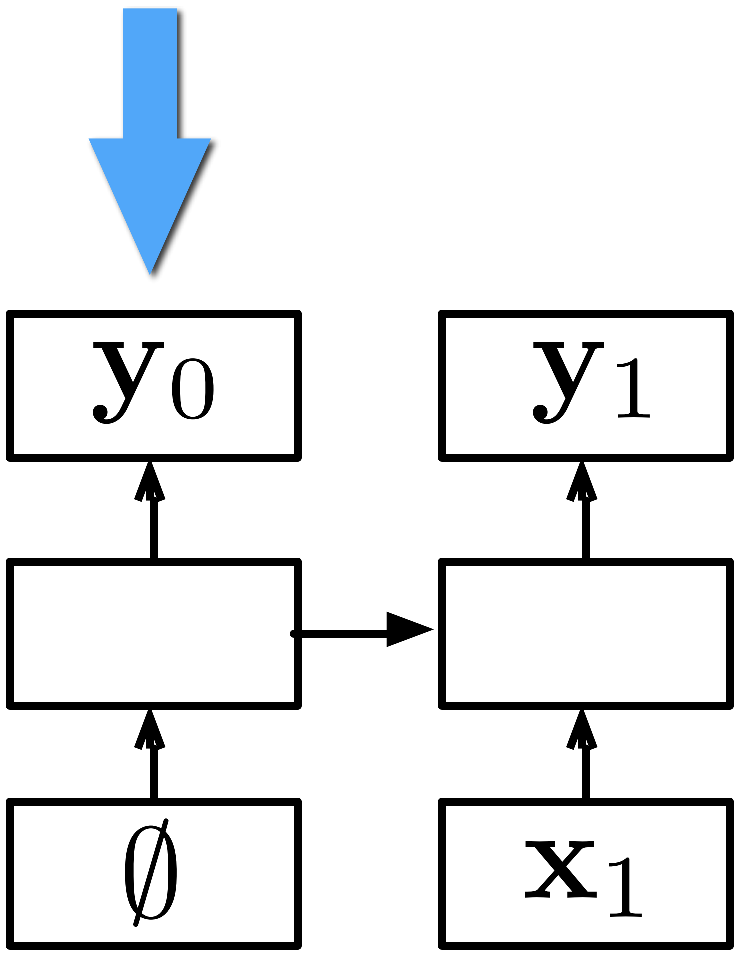


DyNet:

```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs

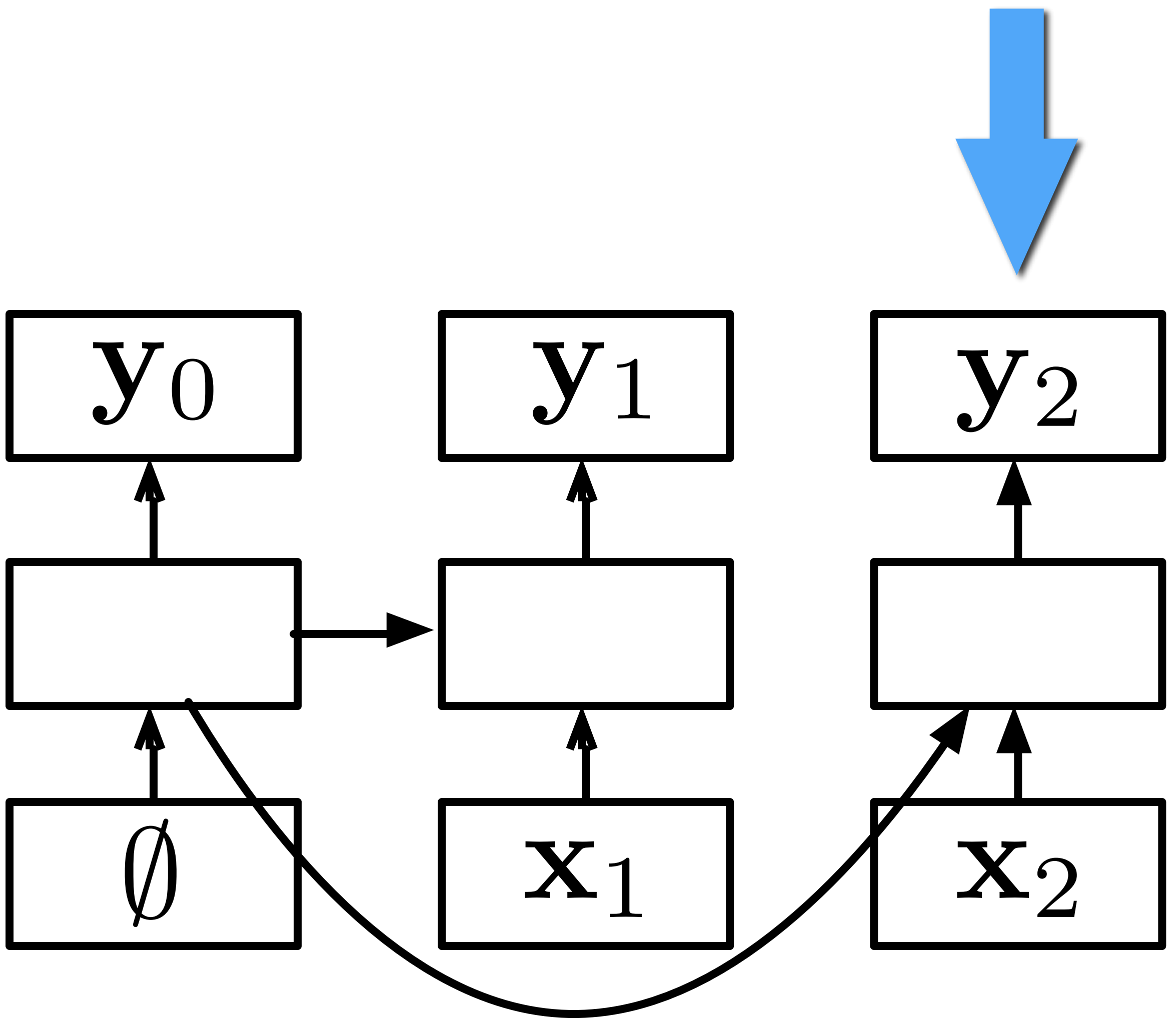


DyNet:

```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs

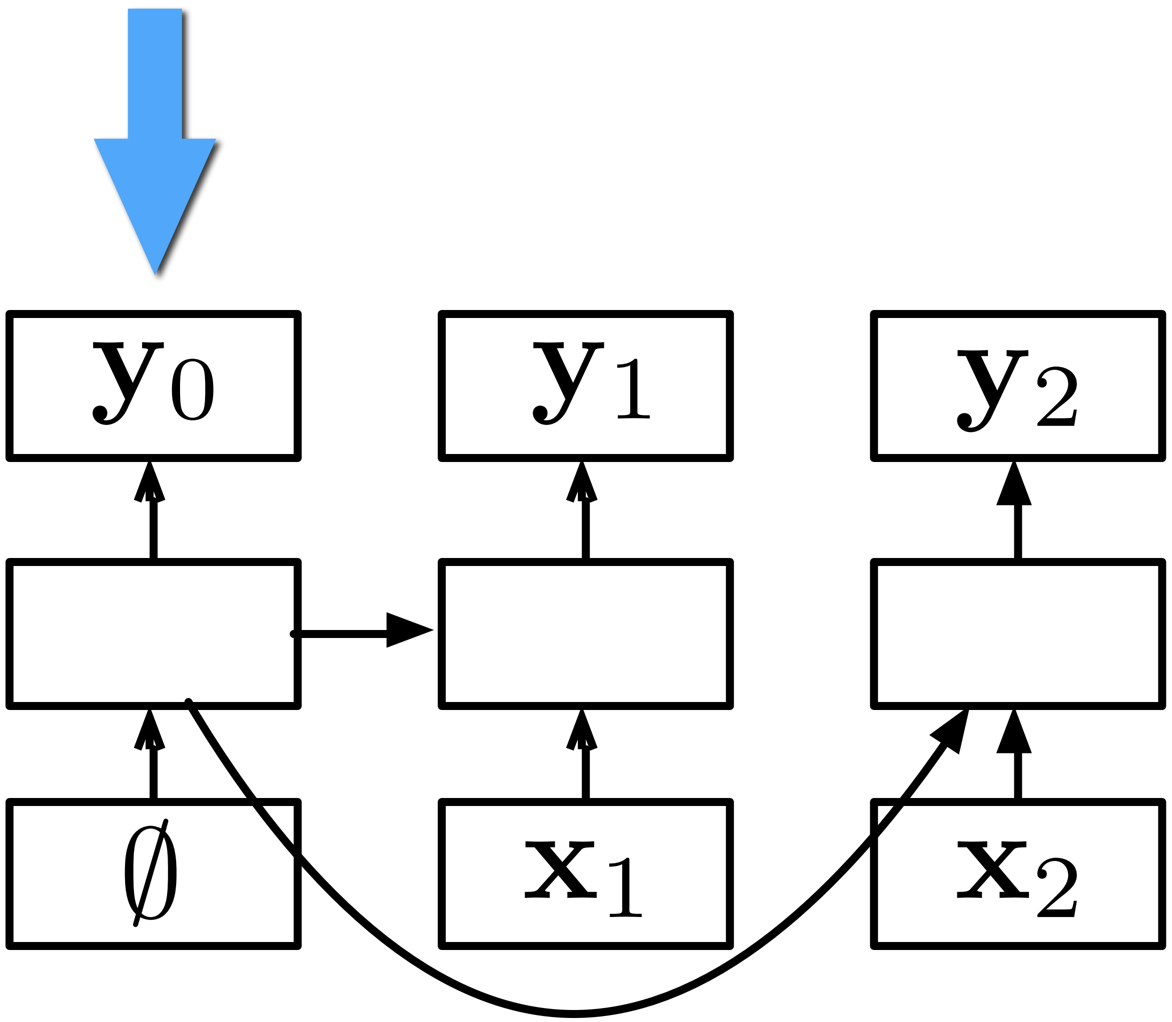


DyNet:

```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs

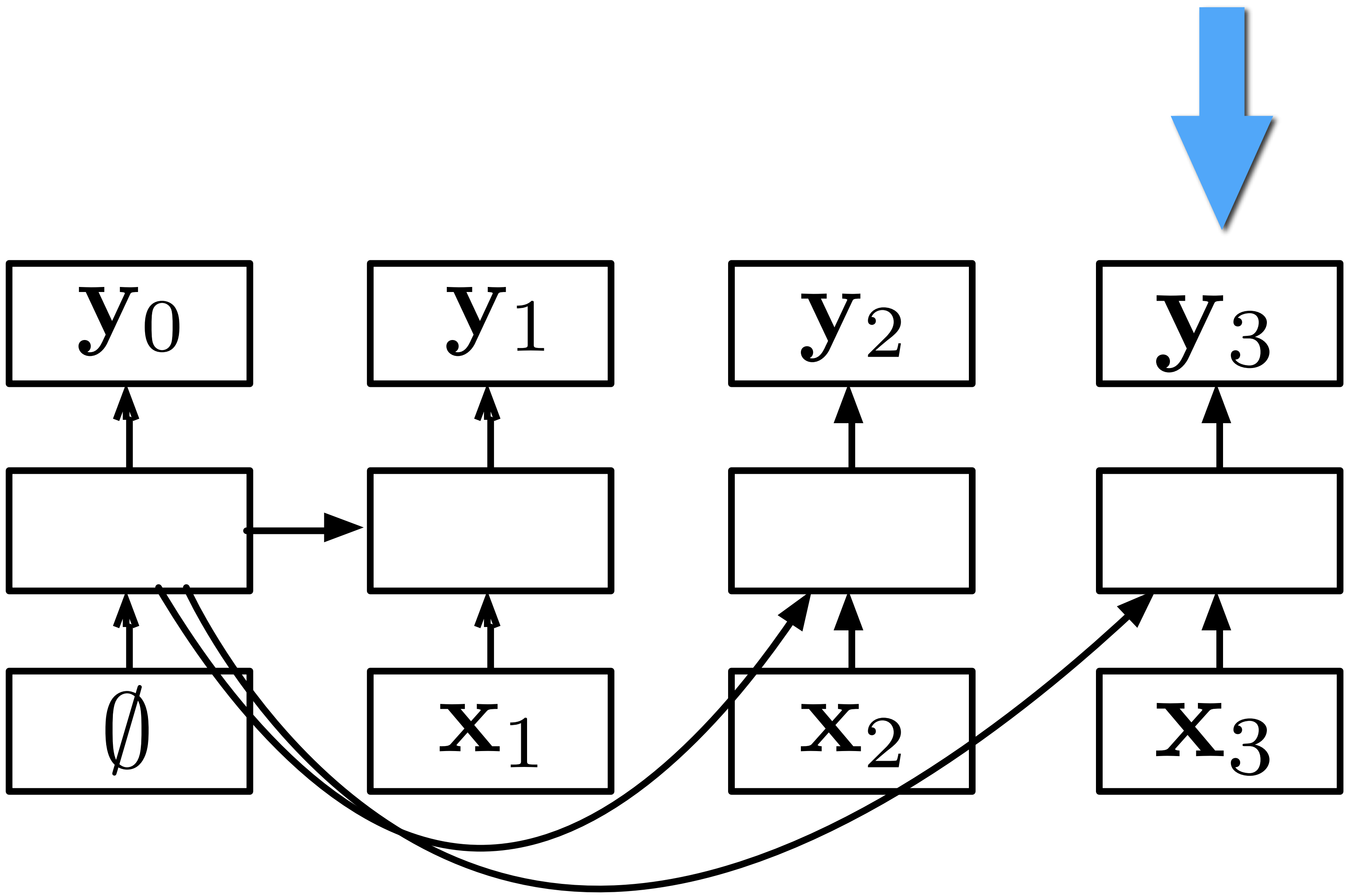


DyNet:

```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

(Slide credits: Chris Dyer)

Transition-based parsing
Stack RNNs



DyNet:

```
s=[rnn.inital_state()]
s.append[s[-1].add_input(x1)
s.pop()
s.append[s[-1].add_input(x2)
s.pop()
s.append[s[-1].add_input(x3)
```

(Slide credits: Chris Dyer)

Let's Try it Out!

`stacklstm-depparser.py`

Alternative Transition Methods

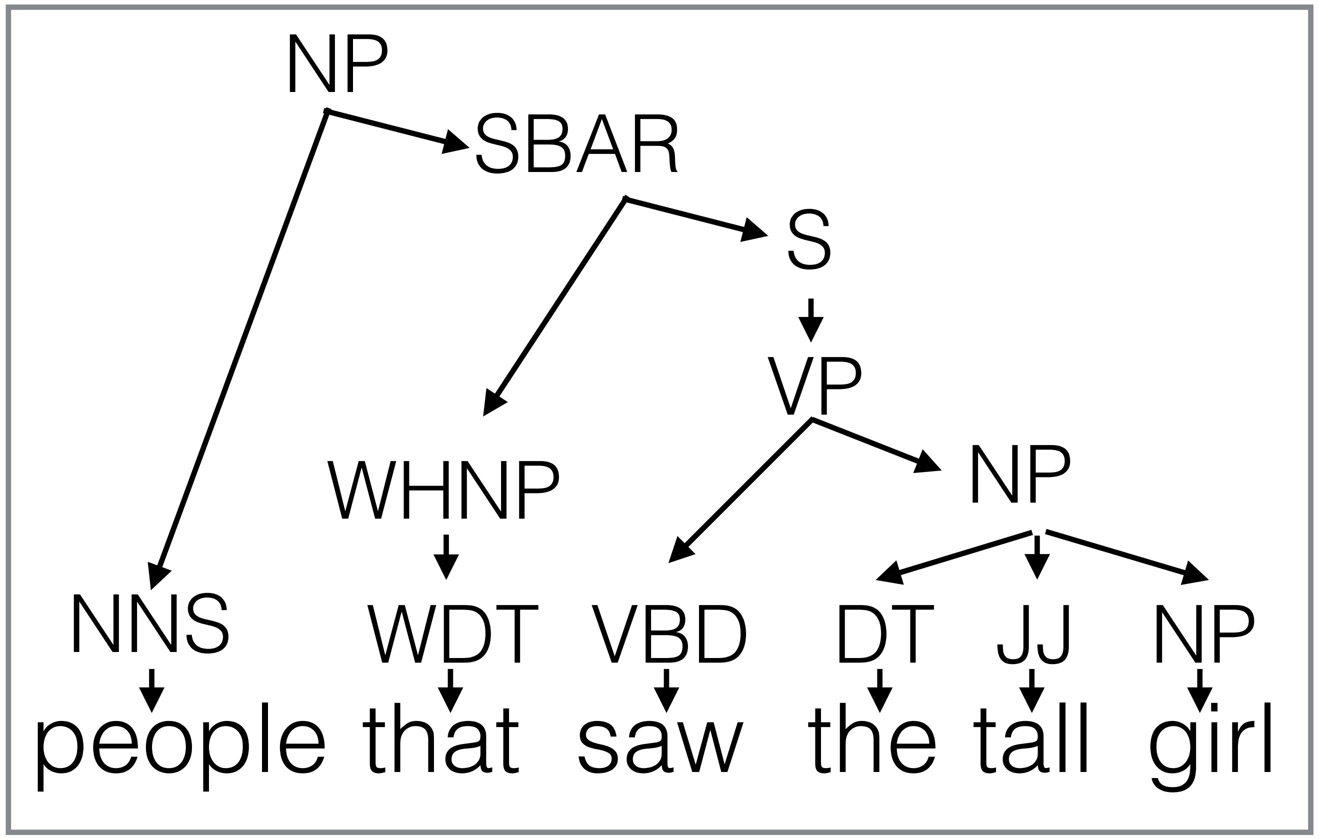
- All previous methods did left-to-right
- Also possible to do **top-down** -- pick the root first, then descend, e.g. Ma et al. (2018)
- Also can do **easy-first** -- pick the easiest link to make first, then proceed from there, e.g. Kiperwasser and Goldberg (2016)

Shift-reduce Parsing for Phrase Structure

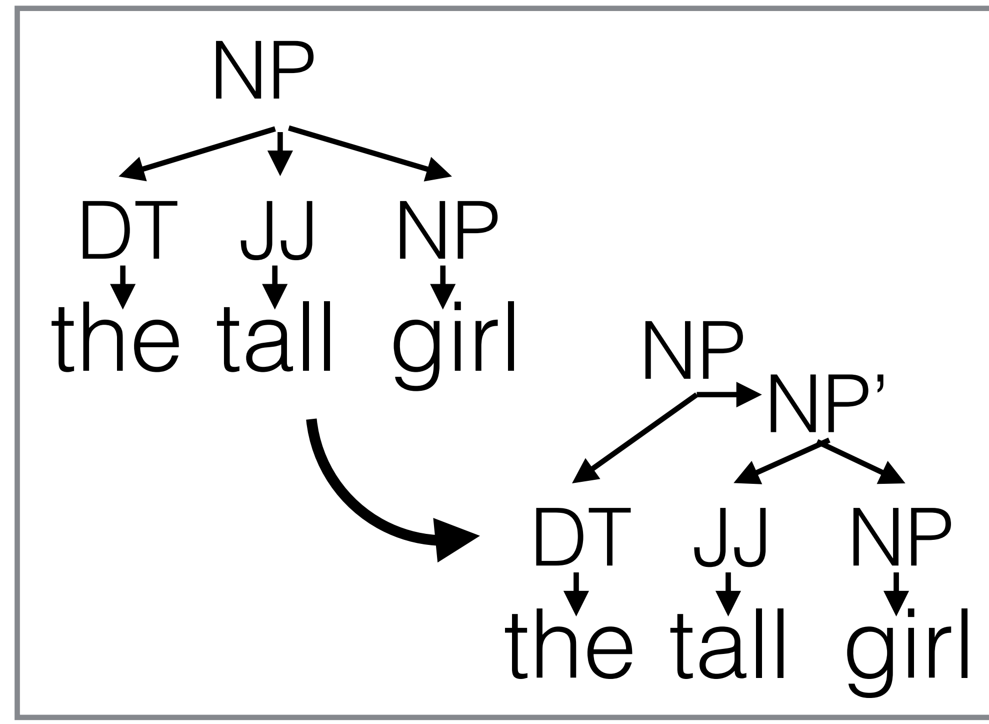
Shift-reduce Parsing for Phrase Structure

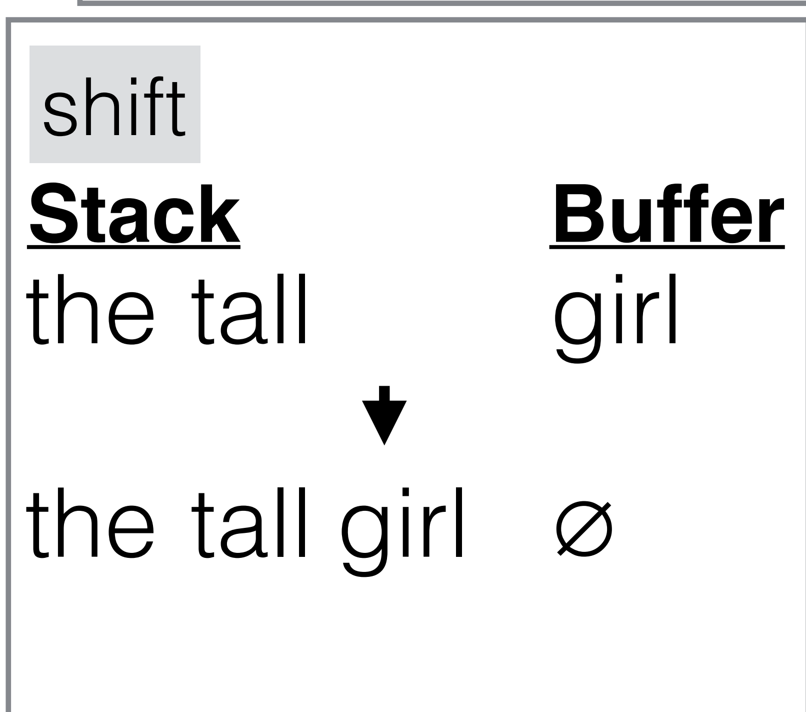
(Sagae and Lavie 2005, Watanabe 2015)

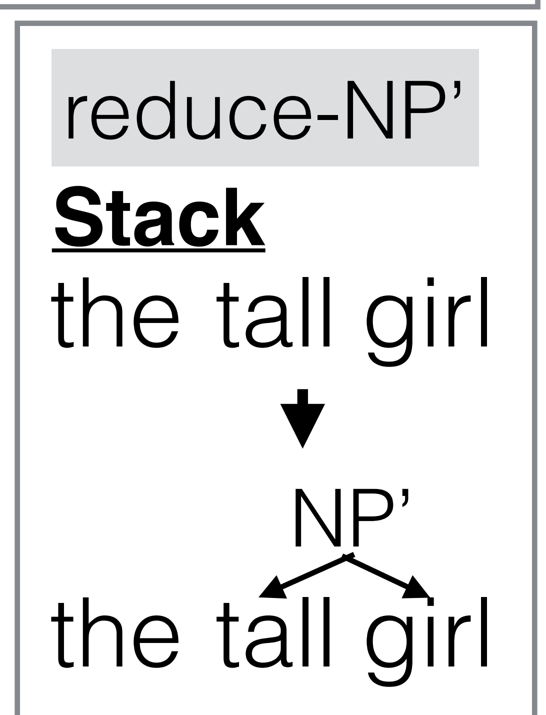
- Shift, reduce-X (binary), unary-X (unary) where X is a label

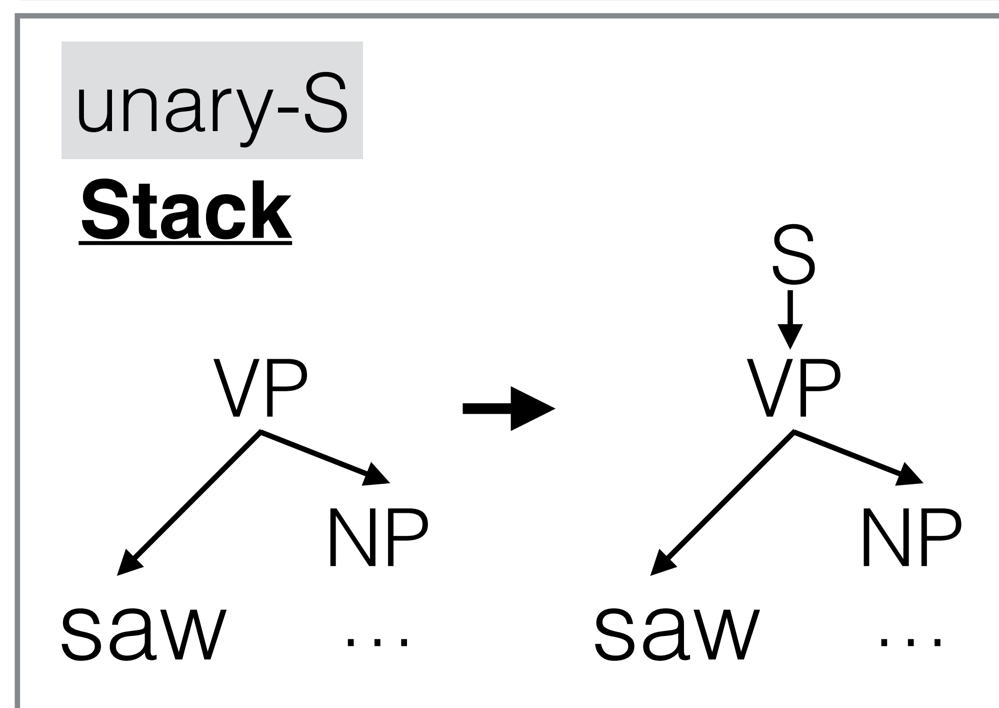


First, Binarize









Recurrent Neural Network Grammars

(Dyer et al. 2016)

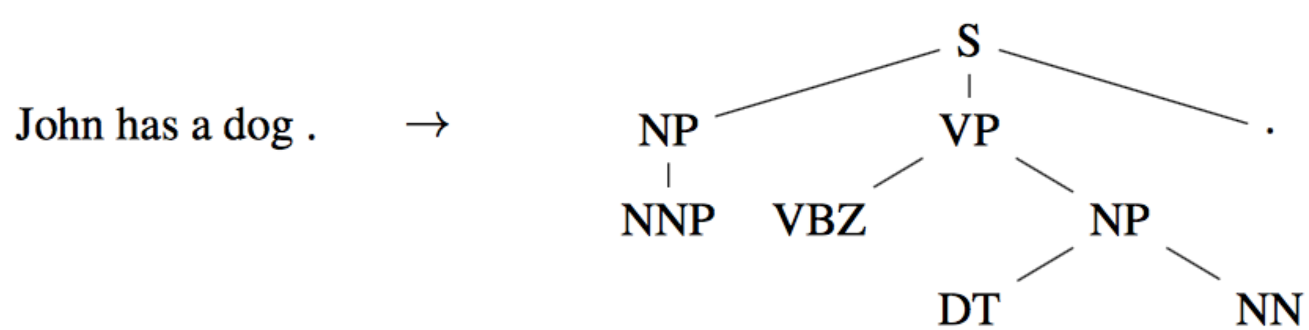
- Top-down generative models for parsing

	Stack	Terminals	Action
0			NT(S)
1	(S		NT(NP)
2	(S | (NP		GEN(*The*)
3	(S | (NP | *The*	*The*	GEN(*hungry*)
4	(S | (NP | *The* | *hungry*	*The* | *hungry*	GEN(*cat*)
5	(S | (NP | *The* | *hungry* | *cat*	*The* | *hungry* | *cat*	REDUCE
6	(S | (NP *The hungry cat*)	*The* | *hungry* | *cat*	NT(VP)
7	(S | (NP *The hungry cat*) | (VP	*The* | *hungry* | *cat*	GEN(*meows*)
8	(S | (NP *The hungry cat*) | (VP *meows*	*The* | *hungry* | *cat* | *meows*	REDUCE
9	(S | (NP *The hungry cat*) | (VP *meows*)	*The* | *hungry* | *cat* | *meows*	GEN(.)
10	(S | (NP *The hungry cat*) | (VP *meows*) | .	*The* | *hungry* | *cat* | *meows* | .	REDUCE
11	(S (NP *The hungry cat*) (VP *meows*) .)	*The* | *hungry* | *cat* | *meows* | .	

- Can serve as a language model as well
- Good parsing results
- Decoding is difficult: need to generate with discriminative model then rerank, importance sampling for LM evaluation

A Simple Approximation: Linearized Trees (Vinyals et al. 2015)

- Similar to RNNG, but generates symbols of linearized tree



John has a dog . → $(S\ (NP\ NNP\ )_{NP}\ (VP\ VBZ\ (NP\ DT\ NN\ )_{NP}\ )_{VP}\ .\ )_{S}$

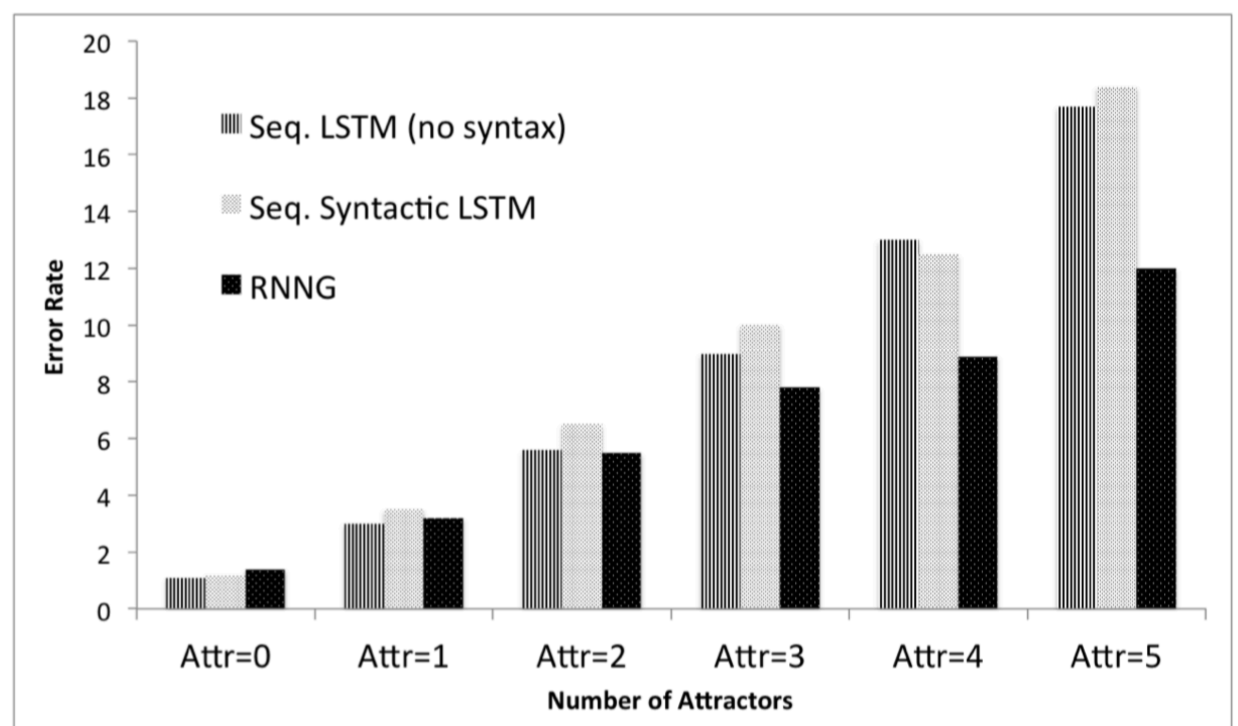
- **+** Can be done with simple sequence-to-sequence models

- **-** No explicit composition function like StackLSTM/RNNG

- **-** Not guaranteed to output well-formed trees

Why Linguistic Structure?

- Regular linear language models do quite well
- But they may not capture phenomena that inherently require structure, such as long-distance agreement
- e.g. Kuncoro et al (2018) find agreement with distractors is much better with syntactic model



Questions?