Good

Morning!

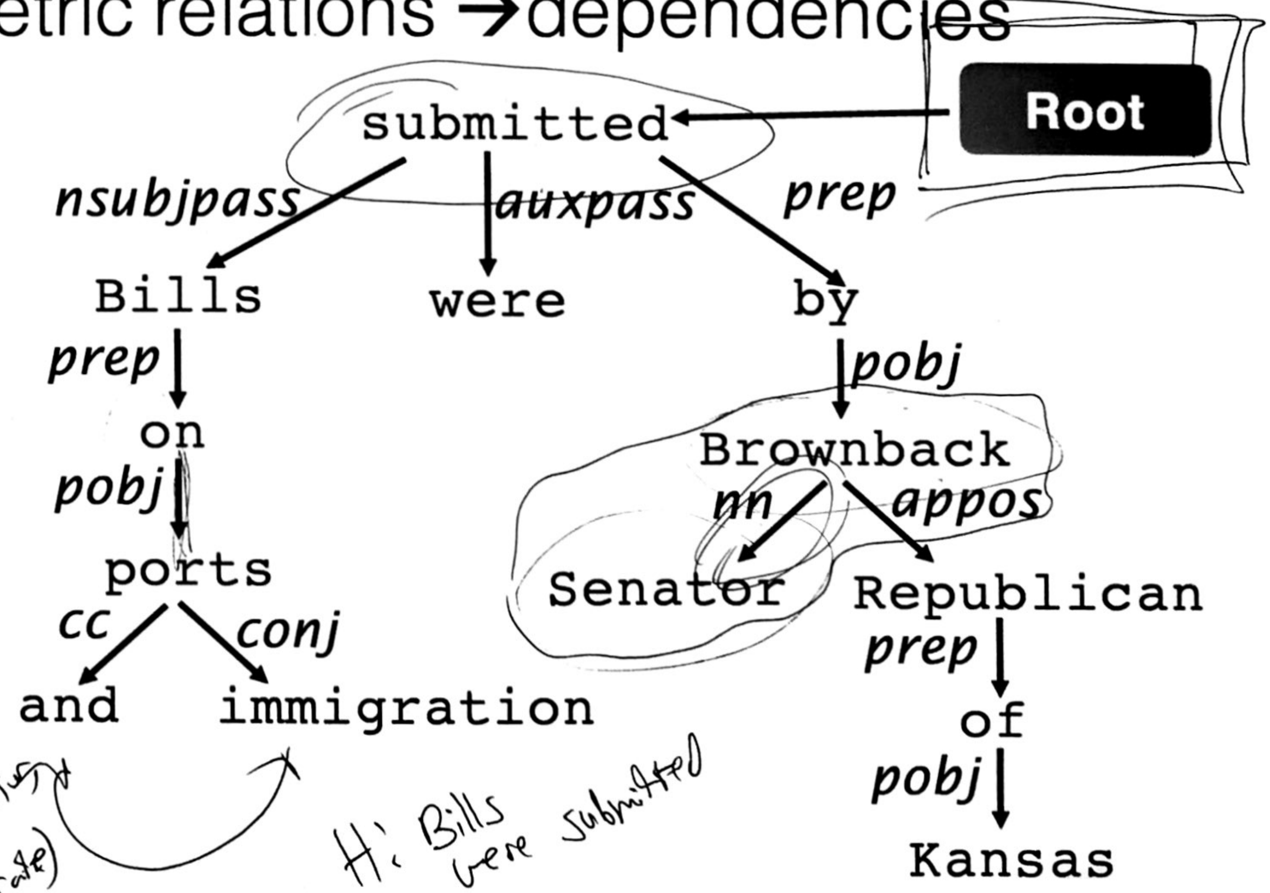
Dependency Structure

Bills on ports and immigration were submitted by Senator Brownback Rep. of Kansas

- Syntactic structure consists of:
  - Lexical items
  - Binary asymmetric relations → dependencies

Dependencies are typed with name of grammatical relation

Dependencies form a tree



Bills $\xrightarrow{\text{prep}}$ on

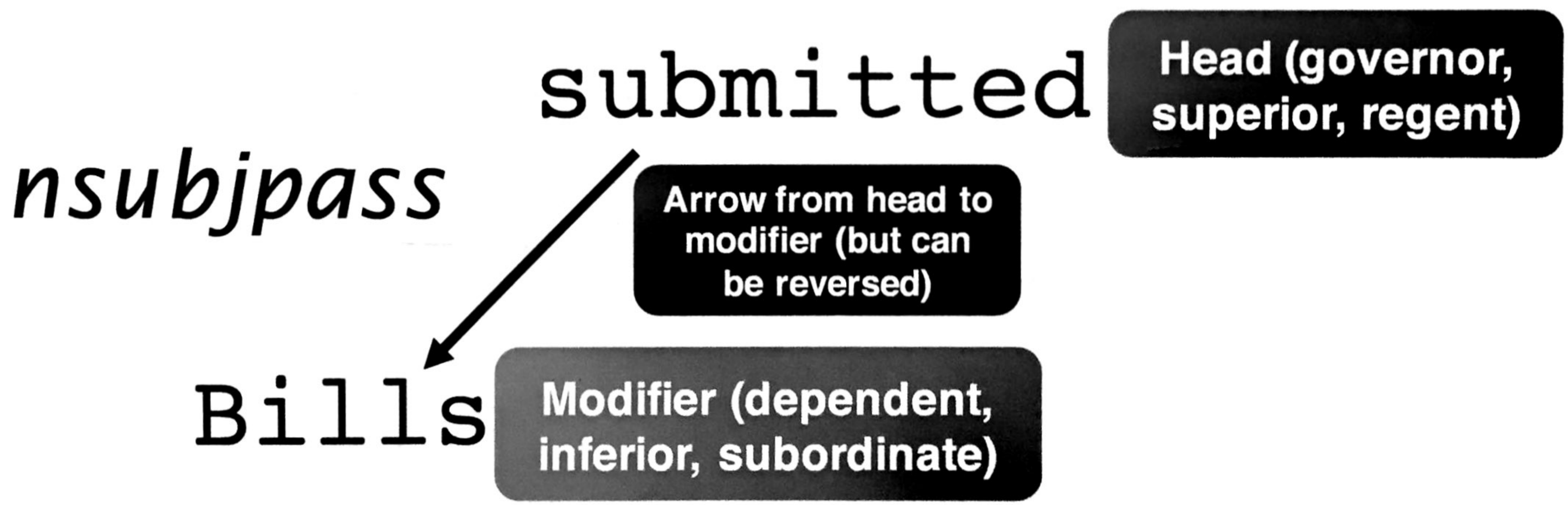
head (gov., superior, regent)

modifier (dep., inferior, subordinate)

Hi: Bills were submitted

Dependency Structure

- Syntactic structure consists of:
  - Lexical items
  - Binary asymmetric relations →dependencies



Applications:

- NER:
  edges give features
- QA:
  - relations between event and entities
  - features
  - Directly for LF
- Generation:
  - LM reranking during generation
  - Generative model
- Entailment
  - Via natural logic on trees
  - Features

- MT:
  - syntactic analysis
    → Syntax-based MT
  - Reranking ← useful
- Sentiment Analysis:
  - Entity-based sentiment
    very hard problem
- IE:
  - very important feature
  - Arch. for defining network for classification

Transforming Dependency Structures to Logical Forms for Semantic Parsing

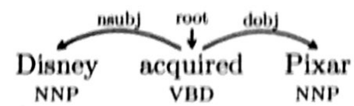
Siva Reddy[†a] **Oscar Täckström**[‡] **Michael Collins**[‡b] **Tom Kwiatkowski**[‡]
Dipanjan Das[‡] **Mark Steedman**[†] **Mirella Lapata**[†]
[†]ILCC, School of Informatics, University of Edinburgh
[‡] Google, New York
siva.reddy@ed.ac.uk
{oscart,mjcollins,tomkwiat,dipanjand}@google.com
{steedman,mlap}@inf.ed.ac.uk

Abstract

The strongly typed syntax of grammar formalisms such as CCG, TAG, LFG and HPSG offers a synchronous framework for deriving syntactic structures and semantic logical forms. In contrast—partly due to the lack of a strong type system—dependency structures are easy to annotate and have become a widely used form of syntactic analysis for many languages. However, the lack of a type system makes a formal mechanism for deriving logical forms from dependency structures challenging. We address this by introducing a robust system based on the lambda calculus for deriving neo-Davidsonian logical forms from dependency trees. These logical forms are then used for semantic parsing of natural language to Freebase. Experiments on the Free917 and WebQuestions datasets show that our representation is superior to the original dependency trees and that it outperforms a CCG-based representation on this task. Compared to prior work, we obtain the strongest result to date on Free917 and competitive results on WebQuestions.

1 Introduction

Semantic parsers map sentences onto logical forms that can be used to query databases (Zettlemoyer and Collins, 2005; Wong and Mooney, 2006), instruct robots (Chen and Mooney, 2011), extract information (Krishnamurthy and Mitchell, 2012), or describe visual scenes (Matuszek et al., 2012). Current systems accomplish this by learning task-specific grammars (Berant et al., 2013), by using strongly-typed CCG grammars (Reddy et al., 2014), or by eschewing the use of a grammar entirely (Yih et al., 2015).



(a) The dependency tree for *Disney acquired Pixar.*

(nsubj (dobj acquired Pixar) Disney)

(b) The s-expression for the dependency tree.

$$\lambda x.\ \exists yz.\ \mathrm{acquired}(x_e) \wedge \mathrm{Disney}(y_a) \wedge \mathrm{Pixar}(z_a) \wedge \mathrm{arg}_1(x_e, y_a) \wedge \mathrm{arg}_2(x_e, z_a)$$

(c) The composed lambda-calculus expression.

Figure 1: The dependency tree is binarized into its s-expression, which is then composed into the lambda expression representing the sentence logical form.

In recent years, there have been significant advances in developing fast and accurate dependency parsers for many languages (McDonald et al., 2005; Nivre et al., 2007; Martins et al., 2013, *inter alia*). Motivated by the desire to carry these advances over to semantic parsing tasks, we present a robust method for mapping dependency trees to logical forms that represent underlying predicate-argument structures.[1] We empirically validate the utility of these logical forms for question answering from databases. Since our approach uses dependency trees as input, we hypothesize that it will generalize better to domains that are well covered by dependency parsers than methods that induce semantic grammars from scratch.

The system that maps a dependency tree to its logical form (henceforth DEPLAMBDA) is illustrated in Figure 1. First, the dependency tree is binarized via an obliqueness hierarchy to give an *s-expression* that describes the application of functions to pairs

[a]Work carried out during an internship at Google.
[b]On leave from Columbia University.

[1]By "robust", we refer to the ability to gracefully handle parse errors as well as the untyped nature of dependency syntax.

Transactions of the Association for Computational Linguistics, vol. 4, pp. 127–140, 2016. Action Editor: Christopher Potts.
Submission batch: 12/2015; Published 4/2016.

Miwa and Bansal 2016

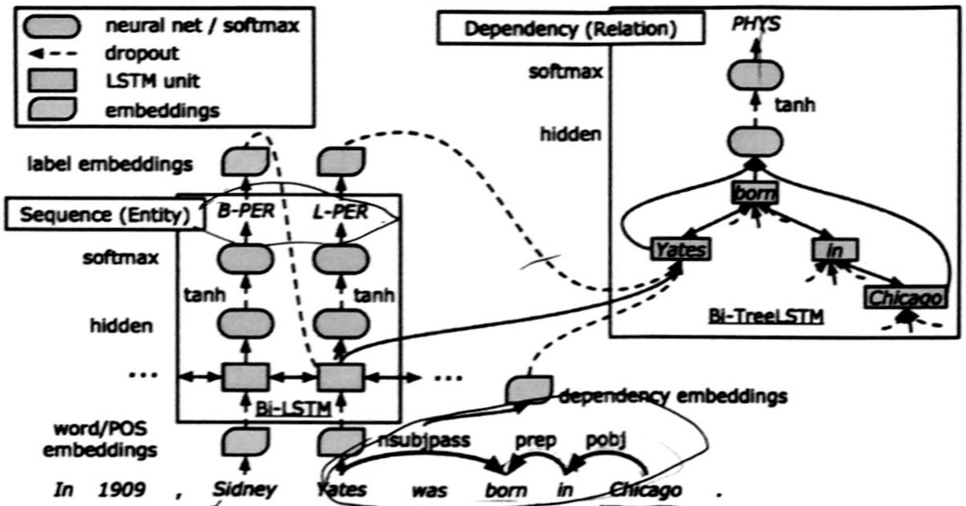


Fig. 1: Our end-to-end relation extraction model, with bidirectional sequential and bidirectional tree-structured LSTM-RNNs.

lations between entities on top of these RNNs. Fig. 1 illustrates the overview of the model. The model mainly consists of three representation layers: a word embeddings layer, a word sequence based LSTM-RNN layer, and finally a dependency subtree based LSTM-RNN layer.

3.1 Embedding Layer

The embedding layer handles word embedding representations. n_w, n_p, n_d and n_e-dimensional vectors $v^{(w)}$, $v^{(p)}$, $v^{(d)}$ and $v^{(e)}$ are embedded to words, part-of-speech (POS) tags, dependency types, and entity labels, respectively.

3.2 Sequence Layer

The sequence layer represents words in a linear sequence using the representations from the embedding layer. This layer represents sentential context information and maintains entities, as shown in bottom-left part of Fig. 1.

We employ bidirectional LSTM-RNNs (Zaremba and Sutskever, 2014) to represent the word sequence in a sentence. The LSTM unit at t-th word consists of a collection of d-dimensional vectors : an input gate i_t, a forget gate f_t, an output gate o_t, a memory cell c_t, and a hidden state h_t. The unit receives an n-dimensional input vector x_t, the previous hidden state h_{t-1}, and the memory cell c_{t-1}, and calculates the new vectors using the following equations:

$$
\begin{aligned}
i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}\right), \qquad (1)\\
f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}\right),\\
o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}\right),\\
u_t &= \tanh\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)}\right),\\
c_t &= i_t \odot u_t + f_t \odot c_{t-1},\\
h_t &= o_t \odot \tanh(c_t),
\end{aligned}
$$

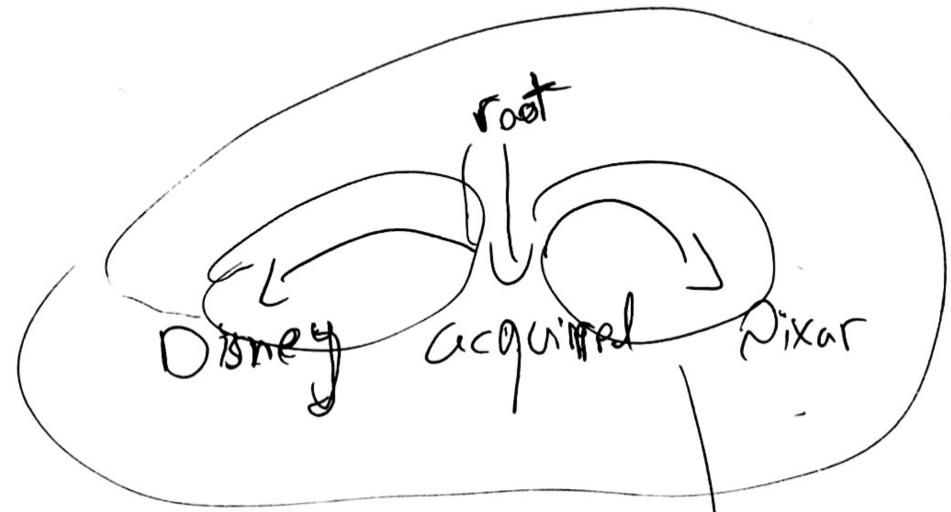
where σ denotes the logistic function, $\odot$ denotes element-wise multiplication, W and U are weight matrices, and b are bias vectors. The LSTM unit at t-th word receives the concatenation of word and POS embeddings as its input vector: $x_t = \left[v_t^{(w)}; v_t^{(p)}\right]$. We also concatenate the hidden state vectors of the two directions' LSTM units corresponding to each word (denoted as $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$) as its output vector, $s_t = \left[\overrightarrow{h_t}; \overleftarrow{h_t}\right]$, and pass it to the subsequent layers.

3.3 Entity Detection

We treat entity detection as a sequence labeling task. We assign an entity tag to each word using a commonly used encoding scheme BILOU (Begin, Inside, Last, Outside, Unit) (Ratinov and Roth, 2009), where each entity tag represents the entity type and the position of a word in the entity. For example, in Fig. 1, we assign *B-PER* and *L-PER* (which denote the beginning and last words of a person entity type, respectively) to each word in *Sidney Yates* to represent this phrase as a *PER* (person) entity type.

We realize entity detection on the top of the sequence layer. We employ a two-layered NN with an h_e-dimensional hidden layer $h^{(e)}$ and a softmax

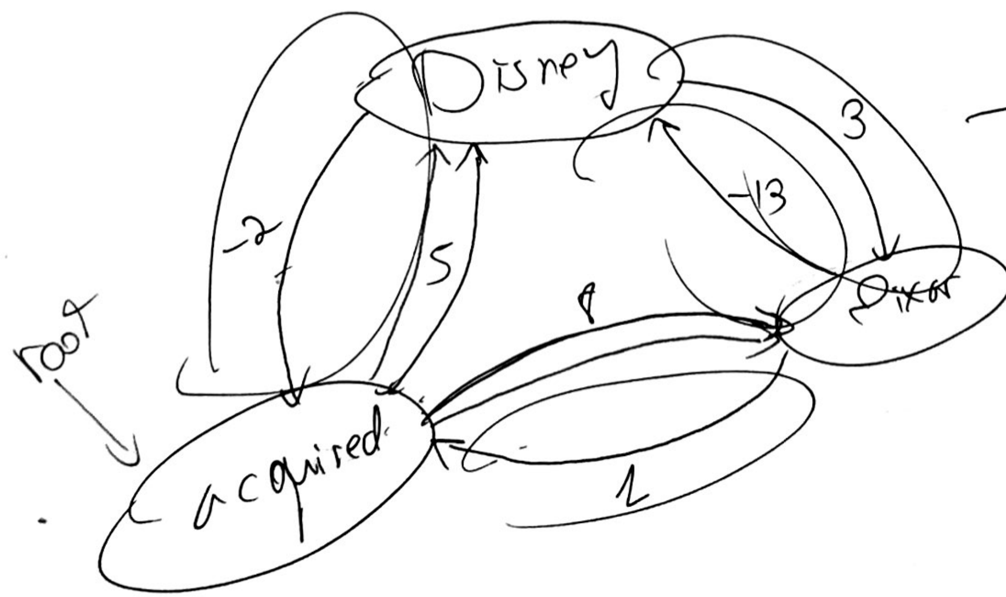
Methods:



positive

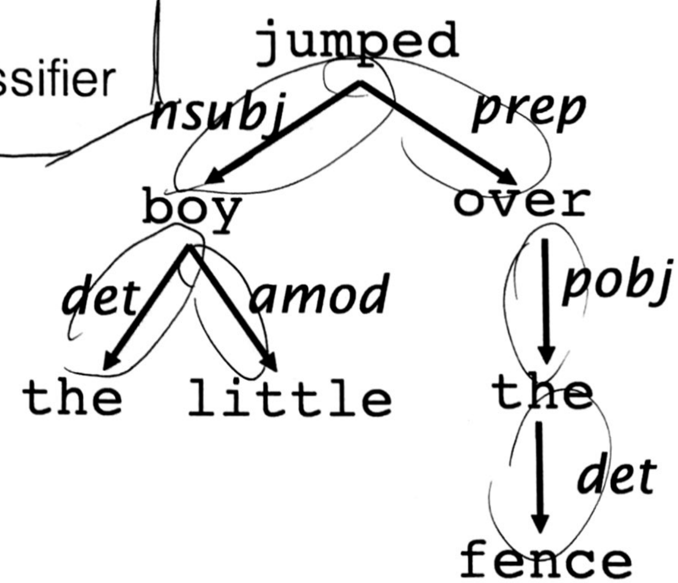
— Graph alg. → MST

— negative



Methods for Dependency Parsing

- Dynamic programming (CKY-style)
  - Similar to lexicalized PCFG: $O(n^5)$
  - Eisner (1996): $O(n^3)$
- Graph algorithms
  - McDonald et al. (2005): score edges independently using classifier and use maximum spanning tree
- Constraint satisfaction
  - Start with all edges, eliminate based on hard constraints
- "Deterministic parsing"
  - Left-to-right, each choice is done with a classifier



The house on the hill

Information:

- ~~[illegible]~~
- Class ~~of~~ of word
- surrounding
- Distance

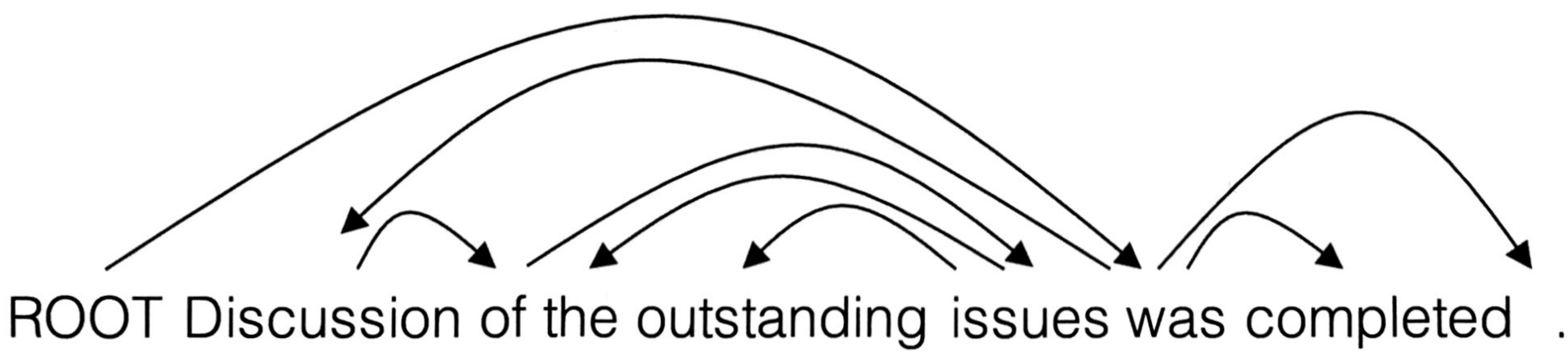
prefer shorter distances

- Material between
- ~~[illegible]~~ Connected words
- Branching/...

Making Decisions

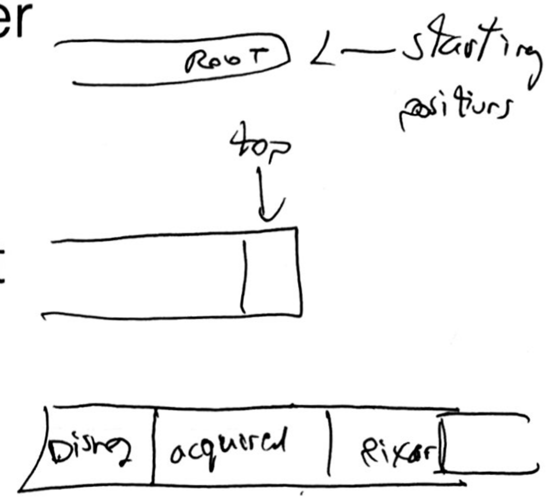
What are the sources of information for dependency parsing?

1. Bilexical affinities
   - [issues → the] is plausible
2. Dependency distance
   - mostly with nearby words
3. Intervening material
   - Dependencies rarely span intervening verbs or punctuation
4. Valency of heads
   - How many dependents on which side are usual for a head?



MaltParse (Nivre et al. 2008)

- Greedy transition-based parser
- Each decision: how to attach each word as we encounter it
  - If you are familiar: like shift-reduce parser
- Select each action with a classifier
- The parser has:
  - a stack σ, written with the top to the right
    - which starts with the ROOT symbol
  - a buffer β, written with the top to the left
    - which starts with the input sentence
  - a set of dependency arcs A
    - which starts off empty $A = \emptyset$
  - a set of actions



Arc-standard Dependency Parsing

stack buffer sentence length edges

Start: σ = [ROOT], $\beta = w_1, \ldots, w_n$, $A = \varnothing$

- Shift $\quad \sigma, w_i|\beta, A \rightarrow \boxed{\sigma|w_i}, \beta, A$
- Left-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j, w_i)\}$
- Right-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_i|\beta, A \cup \{r(w_i, w_j)\}$

head child

Finish: $\beta = \varnothing$

ROOT Joe likes Marry

root | Joe | likes | Marry | $A = \phi$

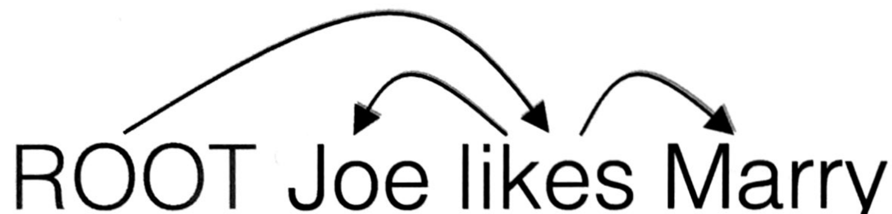
Shift	root, Joe	likes Marry	ϕ
left-arc	root	~~J~~ likes marry	$A = \{$ ~~lik~~ (likes, Joe)$\}$
Shift	root likes	Marry	A
right	root	likes	$A = A \cup \{$(likes, marry)$\}$
right		root	$A = A \cup \{$(root, likes)$\}$
shift	root		

Arc-standard Dependency Parsing

Start: σ = [ROOT], $\beta = w_1, \ldots, w_n$, A = $\varnothing$

- Shift $\sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$
- Left-Arc$_r$ $\sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j,w_i)\}$
- Right-Arc$_r$ $\sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_i|\beta, A \cup \{r(w_i,w_j)\}$

Finish: $\beta = \varnothing$



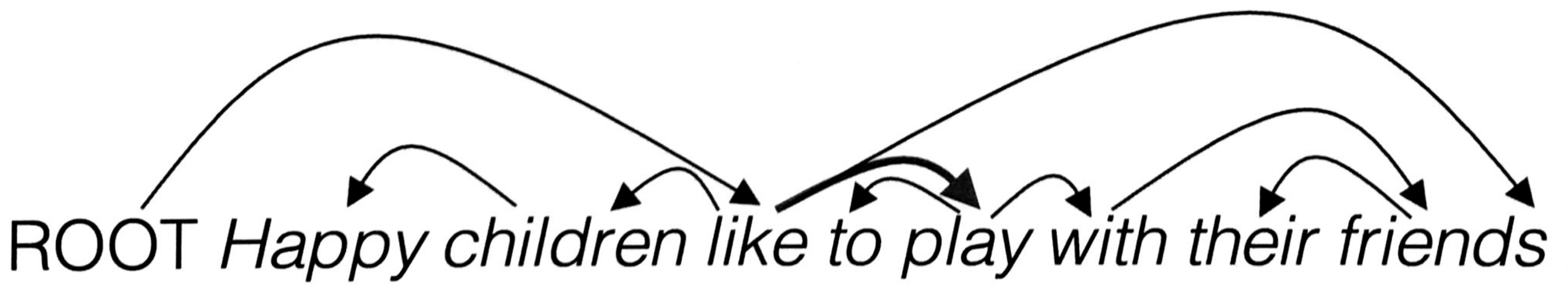
	[ROOT]	[Joe, likes, marry]	$\varnothing$
Shift	[ROOT, Joe]	[likes, marry]	$\varnothing$
Left-Arc	[ROOT]	[likes, marry]	{(likes,Joe)} = A_1
Shift	[ROOT, likes]	[marry]	A_1
Right-Arc	[ROOT]	[likes]	$A_1 \cup$ {(likes,Marry)} = A_2
Right-Arc	[]	[ROOT]	$A_2 \cup$ {(ROOT, likes)} = A_3
Shift	[ROOT]	[]	A_3

Arc-standard Dependency Parsing

Start: σ = [ROOT], $\beta = w_1, \ldots, w_n$, $A = \varnothing$

- Shift $\quad \sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$
- Left-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j,w_i)\}$
- Right-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_i|\beta, A \cup \{r(w_i,w_j)\}$

Finish: $\beta = \varnothing$



Arc-standard Dependency Parsing

Start: σ = [ROOT], $\beta = w_1, \ldots, w_n$, $A = \varnothing$

- Shift $\quad \sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$
- Left-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j,w_i)\}$
- Right-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_i|\beta, A \cup \{r(w_i,w_j)\}$

Finish: $\beta = \varnothing$

ROOT *Happy children like to play with their friends* .

have to ~~shift~~ process the rest before can attach play.

why?

Arc-eager Dependency Parsing

Start: $\sigma = [\text{ROOT}], \beta = w_1, \ldots, w_n, A = \varnothing$

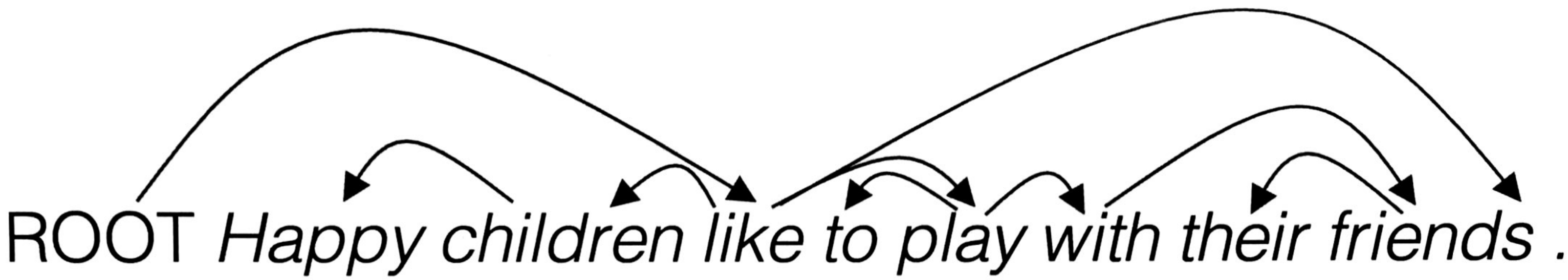
- Left-Arc$_r$ $\sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j, w_i)\}$
  - Precondition: $r'(w_k, w_i) \notin A$, $w_i \neq \text{ROOT}$
- Right-Arc$_r$ $\sigma|w_i, w_j|\beta, A \rightarrow \sigma|w_i|w_j, \beta, A \cup \{r(w_i, w_j)\}$
- Reduce $\sigma|w_i, \beta, A \rightarrow \sigma, \beta, A$
  - Precondition: $r'(w_k, w_i) \in A$
- Shift $\sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$

Finish: $\beta = \varnothing$

This is the common "arc-eager" variant: a head can immediately take a right dependent, before *its* dependents are found

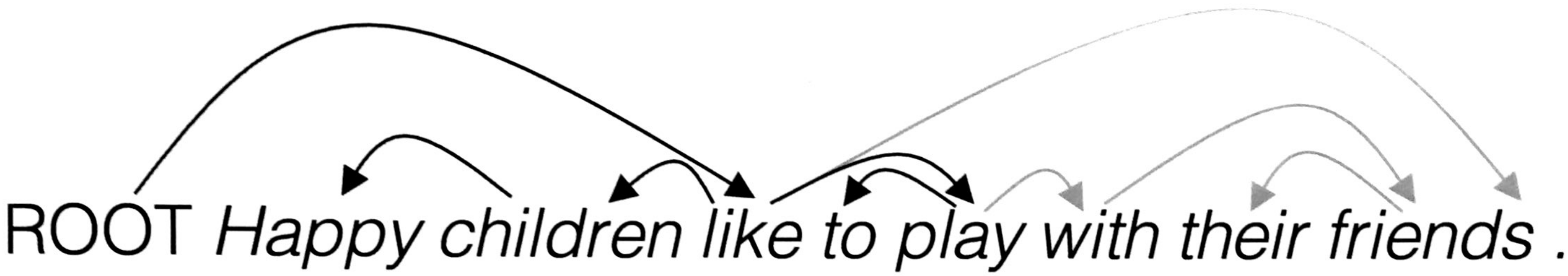
Arc-eager

1. Left-Arc$_r$ $\quad \sigma|w_i,\ w_j|\beta,\ A \Rightarrow \sigma,\ w_j|\beta,\ A \cup \{r(w_j, w_i)\}$
   Precondition: $r'(w_k, w_i) \notin A,\ w_i \neq \mathrm{ROOT}$
2. Right-Arc$_r$ $\quad \sigma|wi,\ w_j|\beta,\ A \Rightarrow \sigma|w_i|w_j,\ \beta,\ A \cup \{r(w_i, w_j)\}$
3. Reduce $\quad \sigma|w_i,\ \beta,\ A \Rightarrow \sigma,\ \beta,\ A$
   Precondition: $r'(w_k, w_i) \in A$
4. Shift $\quad \sigma,\ w_i|\beta,\ A \Rightarrow \sigma|w_i,\ \beta,\ A$



Arc-eager

1. Left-Arc$_r$ $\sigma|w_i,\ w_j|\beta,\ A \rightarrow \sigma,\ w_j|\beta,\ A \cup \{r(w_j,w_i)\}$
   Precondition: $r'(w_k,\ w_i) \notin A,\ w_i \neq \text{ROOT}$
2. Right-Arc$_r$ $\sigma|wi,\ w_j|\beta,\ A \rightarrow \sigma|w_i|w_j,\ \beta,\ A \cup \{r(w_i,w_j)\}$
3. Reduce $\sigma|w_i,\ \beta,\ A \rightarrow \sigma,\ \beta,\ A$
   Precondition: $r'(w_k,\ w_i) \in A$
4. Shift $\sigma,\ w_i|\beta,\ A \rightarrow \sigma|w_i,\ \beta,\ A$



	[ROOT]	[Happy, children, …]	∅
Shift	[ROOT, Happy]	[children, like, …]	∅
LA$_{amod}$	[ROOT]	[children, like, …]	{amod(children, happy)} = A_1
Shift	[ROOT, children]	[like, to, …]	A_1
LA$_{nsubj}$	[ROOT]	[like, to, …]	$A_1 \cup$ {nsubj(like, children)} = A_2
RA$_{root}$	[ROOT, like]	[to, play, …]	$A_2 \cup$ {root(ROOT, like) = A_3
Shift	[ROOT, like, to]	[play, with, …]	A_3
LA$_{aux}$	[ROOT, like]	[play, with, …]	$A_3 \cup$ {aux(play, to) = A_4
RA$_{xcomp}$	[ROOT, like, play]	[with their, …]	$A_4 \cup$ {xcomp(like, play) = A_5

Arg-eager

1. Left-Arc$_r$ $\sigma|w_i,\ w_j|\beta,\ A \rightarrow \sigma,\ w_j|\beta,\ A \cup \{r(w_j, w_i)\}$
   Precondition: $r'(w_k, w_i) \notin A,\ w_i \neq \text{ROOT}$
2. Right-Arc$_r$ $\sigma|wi,\ w_j|\beta,\ A \rightarrow \sigma|w_i|w_j,\ \beta,\ A \cup \{r(w_i, w_j)\}$
3. Reduce $\sigma|w_i,\ \beta,\ A \rightarrow \sigma,\ \beta,\ A$
   Precondition: $r'(w_k, w_i) \in A$
4. Shift $\sigma,\ w_i|\beta,\ A \rightarrow \sigma|w_i,\ \beta,\ A$

ROOT *Happy children like to play with their friends .*

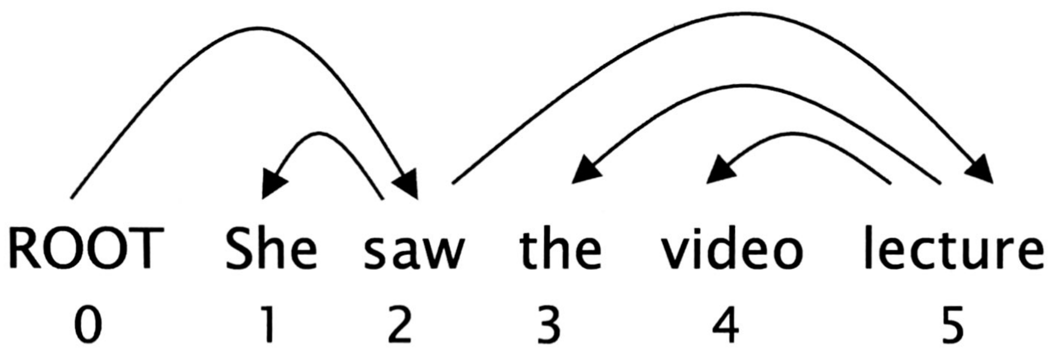
RA$_{xcomp}$	[ROOT, like, play]	[with their, …]	$A_4 \cup$ {xcomp(like, play) = A_5
RA$_{prep}$	[ROOT, like, play, with]	[their, friends, …]	$A_5 \cup$ {prep(play, with) = A_6
Shift	[ROOT, like, play, with, their]	[friends, .]	A_6
LA$_{poss}$	[ROOT, like, play, with]	[friends, .]	$A_6 \cup$ {poss(friends, their) = A_7
RA$_{pobj}$	[ROOT, like, play, with, friends]	[.]	$A_7 \cup$ {pobj(with, friends) = A_8
Reduce	[ROOT, like, play, with]	[.]	A_8
Reduce	[ROOT, like, play]	[.]	A_8
Reduce	[ROOT, like]	[.]	A_8
RA$_{punc}$	[ROOT, like, .]	[]	$A_8 \cup$ {punc(like, .) = A_9

You terminate as soon as the buffer is empty. Dependencies = A_9

MaltParser (Nivre et al. 2008)

- Selecting the next action:
  - Discriminative classifier (SVM, MaxEnt, etc.)
  - Untyped choices: 4
  - Typed choices: |R| * 2 + 2
- Features: POS tags, word in stack, word in buffer, etc.
- Greedy → no search
  - But can easily do beam search
- Close to state of the art
- Linear time parser → **very fast!**

Evaluation



$$\text{Acc} = \frac{\#\ \text{correct deps}}{\#\ \text{of deps}}$$

UAS = 4 / 5 = 80%
LAS = 2 / 5 = 40%

Gold

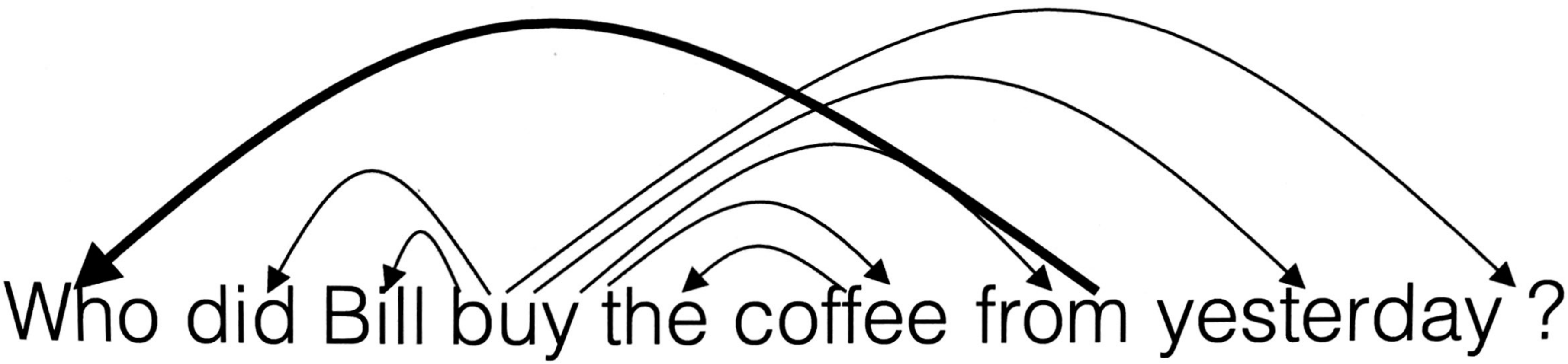
1	2	She	nsubj
2	0	saw	root
3	5	the	det
4	5	video	nn
5	2	lecture	dobj

Parsed

1	2	She	nsubj
2	0	saw	root
3	4	the	det
4	5	video	nsubj
5	2	lecture	ccomp

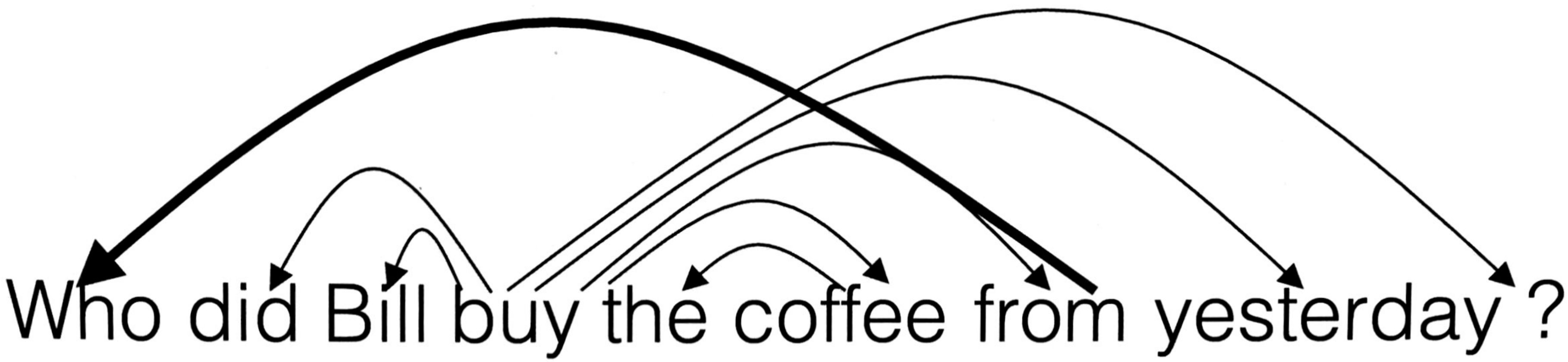
Projectivity

- Dependencies from CFG trees with head rules must be projective
  - Crossing arcs are not allowed
- But: theory allows to account for displaced constituents → non-projective structures



Projectivity

- Dependencies from CFG trees with head rules must be projective
  - Crossing arcs are not allowed
- But: theory allows to account for displaced constituents → non-projective structures



Projectivity

- Arc-eager transition system:
  - Can’t handle non-projectivity
- Possible directions:
  - Give up!
  - Post-processing
  - Add new transition types
  - Switch to a different algorithm
    - Graph-based parsers (e.g., MSTParser)