Good Morning!
Dependency Structure

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

- Dependencies are typed with name of grammatical relation
- Dependencies form a tree

Example:

Bills were submitted by Brownback on ports and immigration of Kansas.
Dependency Structure

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

- **submitted**
  - Head (governor, superior, regent)
  - Arrow from head to modifier (but can be reversed)
  - Modifier (dependent, inferior, subordinate)

- **Bills**
  - nsubjpass
- **MT**:
  - Syntax analysis
  - Syntax-based MT
  - Re-ranking → helpful

- **Sentiment Analysis**:
  - Entity-based sentiment
    - Very hard problem

- **IE**:
  - Very important feature
  - Arch. for designing network for classification

---

**Application**:

- **NER**:
  - Edges give features
  - QA:
    - Relations between event
    - Directly for LP
  - Generation:
    - LM re-ranking during generation
    - Generative model
  - Entailment
    - No natural logic on trees
    - Features
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).

†Work carried out during an internship at Google.
‡On leave from Columbia University.

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

1By “robust”, we refer to the ability to gracefully handle parse errors as well as the untyped nature of dependency syntax.
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:

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

where $\sigma$ 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: $x_t = \left[ v^{(w)}_t; v^{(p)}_t \right]$. We also concatenate the hidden state vectors of the two directions' LSTM units corresponding to each word (denoted as $h^L_t$ and $h^R_t$) as its output vector, $s_t = \left[ h^L_t; h^R_t \right]$, 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:

- Graph alg. → MST

- Root

- Disney acquired Pixar

- Positive

- 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
Making Decisions

What are the sources of information for dependency parsing?

1. Bilexical affinities
   - [issues \(\rightarrow\) 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?

Discussion of the outstanding issues was completed.
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 $\sigma$, written with the top to the right
    - which starts with the ROOT symbol
  - a buffer $\beta$, written with the top to the left
    - which starts with the input sentence
  - a set of dependency arcs $A$
    - which starts off empty
  - a set of actions $A = \emptyset$
Arc-standard Dependency Parsing

Start: \( \sigma = [\text{ROOT}], \beta = w_1, \ldots, w_n, A = \emptyset \)

- **Shift**
  \( \sigma, w_i | \beta, A \rightarrow [\sigma | w_i, \beta, A] \)

- **Left-Arc**
  \( \sigma | w_i, w_j | \beta, A \rightarrow \sigma, w_j | \beta, A \cup \{r(w_i, w_j)\} \)

- **Right-Arc**
  \( \sigma | w_i, w_j | \beta, A \rightarrow \sigma, w_i | \beta, A \cup \{r(w_i, w_j)\} \)

Finish: \( \beta = \emptyset \).

ROOT Joe likes Marry

```
Shift
root
root, Joe

left-arc
root, Joe
root

right
root
root likes

\[ A = \emptyset \]

likes Marry

\[ A = A \cup \{(\text{likes}, \text{Marry})\} \]

Marry likes

\[ A = A \cup \{(\text{Marry}, \text{likes})\} \]
```
Arc-standard Dependency Parsing

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

- **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: $\beta = \emptyset$

**Example Parsing**

ROOT Joe likes Marry

<table>
<thead>
<tr>
<th>Shift</th>
<th>[ROOT]</th>
<th>[Joe, likes, marry]</th>
<th>$\emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-Arc</td>
<td>[ROOT]</td>
<td>[likes, marry]</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Shift</td>
<td>[ROOT, likes]</td>
<td>[marry]</td>
<td>${(\text{likes}, \text{Joe})} = A_1$</td>
</tr>
<tr>
<td>Right-Arc</td>
<td>[ROOT]</td>
<td>[likes]</td>
<td>$A_1$</td>
</tr>
<tr>
<td>Right-Arc</td>
<td>[]</td>
<td>[ROOT]</td>
<td>$A_1 \cup {(\text{likes}, \text{Marry})} = A_2$</td>
</tr>
<tr>
<td>Shift</td>
<td>[ROOT]</td>
<td>[]</td>
<td>$A_2 \cup {(\text{ROOT, likes})} = A_3$</td>
</tr>
</tbody>
</table>
Arc-standard Dependency Parsing

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

- 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: $\beta = \emptyset$

ROOT Happy children like to play with their friends.
Arc-standard Dependency Parsing

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

- **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: $\beta = \emptyset$

ROOT Happy children like to play with their friends.
Arc-eager Dependency Parsing

Start: \( \sigma = [\text{ROOT}], \beta = w_1, \ldots, w_n, A = \emptyset \)

- **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: \( \beta = \emptyset \)

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

ROOT Happy children like to play with their friends.
1. Left-Arc, $\sigma|w_i, w_j|, A \xrightarrow{\sigma, w_j|, A} \{r(w_i, w_j)\}$
   Precondition: $r(w_i, w_j) \in A, w_i \neq \text{ROOT}$

2. Right-Arc, $\sigma|w_i, w_j|, A \xrightarrow{\sigma|w_i|w_j, \beta, A} \{r(w_i, w_j)\}$

3. Reduce, $\sigma|w_i, \beta, A \xrightarrow{\sigma, \beta, A}$
   Precondition: $r(w_k, w_i) \in A$

4. Shift, $\sigma, w_j|, A \xrightarrow{\sigma|w_i, \beta, A}$

ROOT Happy children like to play with their friends.

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>String</th>
<th>Updated Stack</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>[ROOT]</td>
<td>[Happy, children, ...]</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>LA$_{amod}$</td>
<td>[ROOT]</td>
<td>[children, like, ...]</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>[ROOT, children]</td>
<td>[like, to, ...]</td>
<td>${\text{amod(children, happy)}} = A_1$</td>
<td></td>
</tr>
<tr>
<td>LA$_{nsubj}$</td>
<td>[ROOT]</td>
<td>[like, to, ...]</td>
<td>$A_1$</td>
<td></td>
</tr>
<tr>
<td>RA$_{root}$</td>
<td>[ROOT, like]</td>
<td>[to, play, ...]</td>
<td>$A_1 \cup {\text{nsubj(like, children)}} = A_2$</td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>[ROOT, like, to]</td>
<td>[play, with, ...]</td>
<td>$A_2 \cup {\text{root(ROOT, like)}} = A_3$</td>
<td></td>
</tr>
<tr>
<td>LA$_{aux}$</td>
<td>[ROOT, like]</td>
<td>[play, with, ...]</td>
<td>$A_3$</td>
<td></td>
</tr>
<tr>
<td>RA$_{xcomp}$</td>
<td>[ROOT, like, play]</td>
<td>[with their, ...]</td>
<td>$A_4 \cup {\text{xcomp(like, play)}} = A_5$</td>
<td></td>
</tr>
</tbody>
</table>
Arg-eager

ROOT Happy children like to play with their friends.

RA\textsubscript{xcomp} [ROOT, like, play] [with their, ...] A\textsubscript{4} \cup \{xcomp(like, play) = A\textsubscript{5}\}

RA\textsubscript{prep} [ROOT, like, play, with] [their, friends, ...] A\textsubscript{5} \cup \{prep(play, with) = A\textsubscript{6}\}

Shift [ROOT, like, play, with, their] [friends, .] A\textsubscript{6}

LA\textsubscript{poss} [ROOT, like, play, with] [friends, .] A\textsubscript{6} \cup \{poss(friends, their) = A\textsubscript{7}\}

RA\textsubscript{pobj} [ROOT, like, play, with, friends] [.] A\textsubscript{7} \cup \{pobj(with, friends) = A\textsubscript{8}\}

Reduce [ROOT, like, play, with] [.] A\textsubscript{8}

Reduce [ROOT, like, play] [.] A\textsubscript{8}

Reduce [ROOT, like] [.] A\textsubscript{8}

RA\textsubscript{punc} [ROOT, like, .] [.] A\textsubscript{8} \cup \{punc(like, .) = A\textsubscript{9}\}

You terminate as soon as the buffer is empty. Dependencies = A\textsubscript{9}
MaltParser (Nivre et al. 2008)

• Selecting the next action:
  – Discriminative classifier (SVM, MaxEnt, etc.)
  – Untyped choices: 4
  – Typed choices: $|R| \times 2 + 2$

• Features: POS tags, word in stack, word in buffer, etc.

• Greedy $\rightarrow$ no search
  – But can easily do beam search

• Close to state of the art

• Linear time parser $\rightarrow$ very fast!
Evaluation

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

UAS = \frac{4}{5} = 80\%

LAS = \frac{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 $\rightarrow$ non-projective structures

Who did Bill buy the coffee from yesterday?
Projectivity

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

Who did Bill buy the coffee from yesterday?
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)