Learning Dependency-Based Compositional Semantics

Percy Liang
UC Berkeley
pliang@cs.berkeley.edu

Michael I. Jordan
UC Berkeley
jordan@cs.berkeley.edu

Dan Klein
UC Berkeley
klein@cs.berkeley.edu

Abstract

Compositional question answering begins by mapping questions to logical forms, but training a semantic parser to perform this mapping typically requires the costly annotation of the target logical forms. In this paper, we learn to map questions to answers via latent logical forms, which are induced automatically from question-answer pairs. In tackling this challenging learning problem, we introduce a new semantic representation which highlights a parallel between dependency syntax and efficient evaluation of logical forms. On two standard semantic parsing benchmarks (GEO and JOBS), our system obtains the highest published accuracies, despite requiring no annotated logical forms.

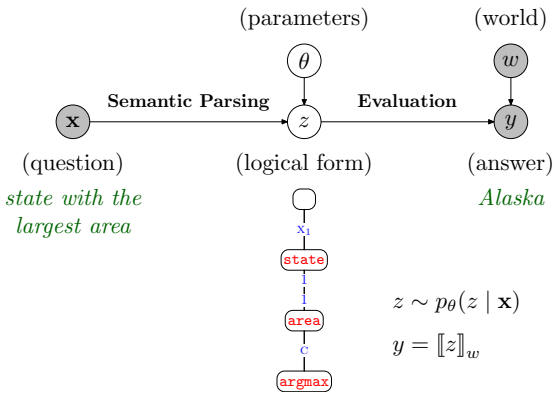


Figure 1: Our probabilistic model: a question $\mathbf{x}$ is mapped to a latent logical form z, which is then evaluated with respect to a world w (database of facts), producing an answer y. We represent logical forms z as labeled trees, induced automatically from $(\mathbf{x}, y)$ pairs.

1 Introduction

What is the total population of the ten largest capitals in the US? Answering these types of complex questions compositionally involves first mapping the questions into logical forms (semantic parsing). Supervised semantic parsers (Zelle and Mooney, 1996; Tang and Mooney, 2001; Ge and Mooney, 2005; Zettlemoyer and Collins, 2005; Kate and Mooney, 2007; Zettlemoyer and Collins, 2007; Wong and Mooney, 2007; Kwiatkowski et al., 2010) rely on manual annotation of logical forms, which is expensive. On the other hand, existing unsupervised semantic parsers (Poon and Domingos, 2009) do not handle deeper linguistic phenomena such as quantification, negation, and superlatives.

As in Clarke et al. (2010), we obviate the need for annotated logical forms by considering the end-to-end problem of mapping questions to answers. However, we still model the logical form (now as a latent variable) to capture the complexities of language. Figure 1 shows our probabilistic model: We want to induce latent logical forms z (and parameters θ) given only question-answer pairs $(\mathbf{x}, y)$, which is much cheaper to obtain than $(\mathbf{x}, z)$ pairs.

The core problem that arises in this setting is program induction: finding a logical form z (over an exponentially large space of possibilities) that produces the target answer y. Unlike standard semantic parsing, our end goal is only to generate the correct y, so we are free to choose the representation for z. Which one should we use?

The dominant paradigm in compositional semantics is Montague semantics, which constructs lambda calculus forms in a bottom-up manner. CCG is one instantiation (Steedman, 2000), which is used by many semantic parsers, e.g., Zettlemoyer and Collins (2005). However, the logical forms there can become quite complex, and in the context of program induction, this would lead to an unwieldy search space. At the same time, representations such as FunQL (Kate et al., 2005), which was used in

Clarke et al. (2010), are simpler but lack the full expressive power of lambda calculus.

The main technical contribution of this work is a new semantic representation, *dependency-based compositional semantics* (DCS), which is both simple and expressive (Section 2). The logical forms in this framework are trees, which is desirable for two reasons: (i) they parallel syntactic dependency trees, which facilitates parsing and learning; and (ii) evaluating them to obtain the answer is computationally efficient.

We trained our model using an EM-like algorithm (Section 3) on two benchmarks, GEO and JOBS (Section 4). Our system outperforms all existing systems despite using no annotated logical forms.

2 Semantic Representation

We first present a basic version (Section 2.1) of dependency-based compositional semantics (DCS), which captures the core idea of using trees to represent formal semantics. We then introduce the full version (Section 2.2), which handles linguistic phenomena such as quantification, where syntactic and semantic scope diverge.

We start with some definitions, using US geography as an example domain. Let $\mathcal{V}$ be the set of all *values*, which includes primitives (e.g., $3, \texttt{CA} \in \mathcal{V}$) as well as sets and tuples formed from other values (e.g., $3, \{3,4,7\}, (\texttt{CA}, \{5\}) \in \mathcal{V}$). Let $\mathcal{P}$ be a set of *predicates* (e.g., $\texttt{state}, \texttt{count} \in \mathcal{P}$), which are just symbols.

A *world* w is mapping from each predicate $p \in \mathcal{P}$ to a set of tuples; for example, $w(\texttt{state}) = \{(\texttt{CA}), (\texttt{OR}), \dots\}$. Conceptually, a world is a relational database where each predicate is a relation (possibly infinite). Define a special predicate $\emptyset$ with $w(\emptyset) = \mathcal{V}$. We represent functions by a set of input-output pairs, e.g., $w(\texttt{count}) = \{(S, n) : n = |S|\}$. As another example, $w(\texttt{average}) = \{(S, \bar{x}) : \bar{x} = |S_1|^{-1} \sum_{x \in S_1} S(x)\}$, where a set of pairs S is treated as a set-valued function $S(x) = \{y : (x, y) \in S\}$ with domain $S_1 = \{x : (x, y) \in S\}$.

The logical forms in DCS are called *DCS trees*, where nodes are labeled with predicates, and edges are labeled with relations. Formally:

Definition 1 (DCS trees) *Let $\mathcal{Z}$ be the set of DCS trees, where each $z \in \mathcal{Z}$ consists of (i) a predicate $z.p \in \mathcal{P}$ and (ii) a sequence of edges $z.e_1, \dots, z.e_m$, each edge e consisting of a relation $e.r \in \mathcal{R}$ (see Table 1) and a child tree $e.c \in \mathcal{Z}$.*

Relations $\mathcal{R}$

$\frac{j}{j'}$	(join)	E	(extract)
Σ	(aggregate)	Q	(quantify)
$\mathrm{X}_{\mathbf{i}}$	(execute)	C	(compare)

Table 1: Possible relations appearing on the edges of a DCS tree. Here, $j, j' \in \{1, 2, \dots\}$ and $\mathbf{i} \in \{1, 2, \dots\}^*$.

We write a DCS tree z as $\langle p; r_1 : c_1; \dots; r_m : c_m \rangle$. Figure 2(a) shows an example of a DCS tree. Although a DCS tree is a logical form, note that it looks like a syntactic dependency tree with predicates in place of words. It is this transparency between syntax and semantics provided by DCS which leads to a simple and streamlined compositional semantics suitable for program induction.

2.1 Basic Version

The basic version of DCS restricts $\mathcal{R}$ to join and aggregate relations (see Table 1). Let us start by considering a DCS tree z with only join relations. Such a z defines a constraint satisfaction problem (CSP) with nodes as variables. The CSP has two types of constraints: (i) $x \in w(p)$ for each node x labeled with predicate $p \in \mathcal{P}$; and (ii) $x_j = y_{j'}$ (the j-th component of x must equal the j'-th component of y) for each edge (x, y) labeled with $\frac{j}{j'} \in \mathcal{R}$.

A solution to the CSP is an assignment of nodes to values that satisfies all the constraints. We say a value v is *consistent* for a node x if there exists a solution that assigns v to x. The *denotation* $[\![z]\!]_w$ (z evaluated on w) is the set of consistent values of the root node (see Figure 2 for an example).

Computation We can compute the denotation $[\![z]\!]_w$ of a DCS tree z by exploiting dynamic programming on trees (Dechter, 2003). The recurrence is as follows:

$$[\![\left\langle p; \tfrac{j_1}{j'_1} : c_1; \cdots; \tfrac{j_m}{j'_m} : c_m \right\rangle]\!]_w \quad (1)$$
$$= w(p) \cap \bigcap_{i=1}^{m} \{v : v_{j_i} = t_{j'_i}, t \in [\![c_i]\!]_w\}.$$

At each node, we compute the set of tuples v consistent with the predicate at that node ($v \in w(p)$), and

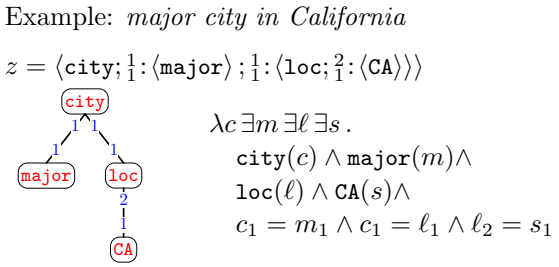


Figure 2: (a) An example of a DCS tree (written in both the mathematical and graphical notation). Each node is labeled with a predicate, and each edge is labeled with a relation. (b) A DCS tree z with only join relations encodes a constraint satisfaction problem. (c) The denotation of z is the set of consistent values for the root node.

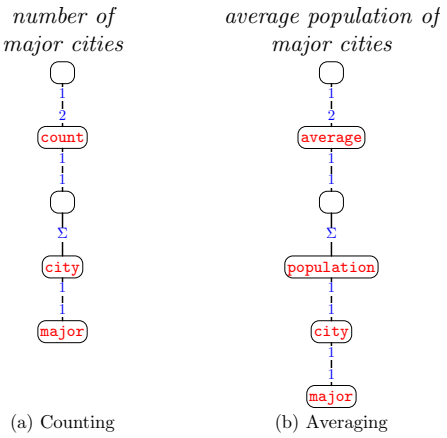


Figure 3: Examples of DCS trees that use the aggregate relation (Σ) to (a) compute the cardinality of a set and (b) take the average over a set.

for each child i, the j_i-th component of v must equal the j'_i-th component of some t in the child's denotation ($t \in [\![c_i]\!]_w$). This algorithm is linear in the number of nodes times the size of the denotations.[1]

Now the dual importance of trees in DCS is clear: We have seen that trees parallel syntactic dependency structure, which will facilitate parsing. In addition, trees enable efficient computation, thereby establishing a new connection between dependency syntax and efficient semantic evaluation.

Aggregate relation DCS trees that only use join relations can represent arbitrarily complex compositional structures, but they cannot capture higher-order phenomena in language. For example, consider the phrase *number of major cities*, and suppose that *number* corresponds to the `count` predicate. It is impossible to represent the semantics of this phrase with just a CSP, so we introduce a new *aggregate relation*, notated Σ. Consider a tree $\langle \Sigma : c \rangle$, whose root is connected to a child c via Σ. If the denotation of c is a set of values s, the parent's denotation is then a singleton set containing s. Formally:

$$[\![\langle \Sigma : c \rangle]\!]_w = \{[\![c]\!]_w\}. \quad (2)$$

Figure 3(a) shows the DCS tree for our running example. The denotation of the middle node is $\{s\}$, where s is all major cities. Having instantiated s as a value, everything above this node is an ordinary CSP: s constrains the `count` node, which in turns constrains the root node to $|s|$.

A DCS tree that contains only join and aggregate relations can be viewed as a collection of tree-structured CSPs connected via aggregate relations. The tree structure still enables us to compute denotations efficiently based on (1) and (2).

2.2 Full Version

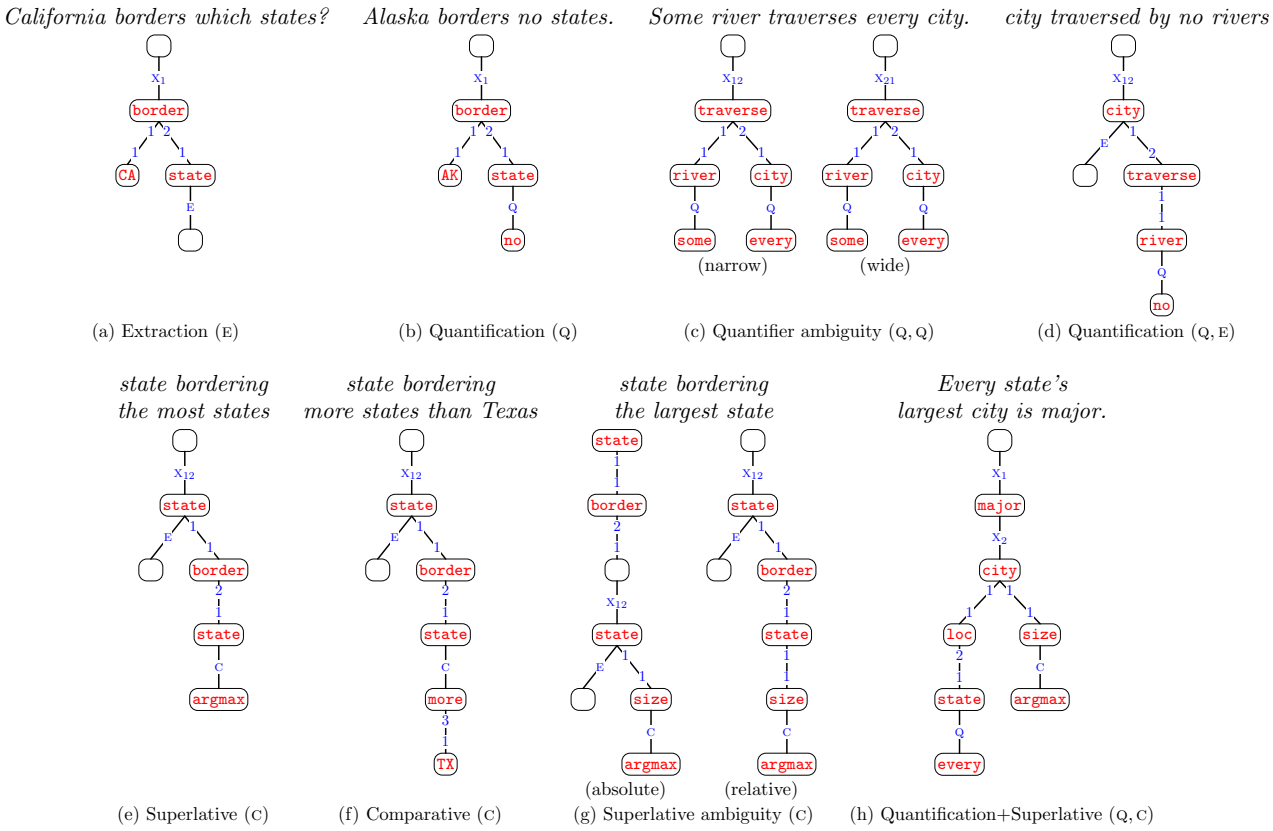
The basic version of DCS described thus far handles a core subset of language. But consider Figure 4: (a) is headed by *borders*, but *states* needs to be extracted; in (b), the quantifier *no* is syntactically dominated by the head verb *borders* but needs to take wider scope. We now present the full version of DCS which handles this type of divergence between syntactic and semantic scope.

The key idea that allows us to give semantically-scoped denotations to syntactically-scoped trees is as follows: We mark a node low in the tree with a *mark relation* (one of E, Q, or C). Then higher up in the tree, we invoke it with an *execute relation* $\text{X}_\mathbf{i}$ to create the desired semantic scope.[2]

This mark-execute construct acts non-locally, so to maintain compositionality, we must augment the

[1] Infinite denotations (such as $[\![<]\!]_w$) are represented as implicit sets on which we can perform membership queries. The intersection of two sets can be performed as long as at least one of the sets is finite.

[2] Our mark-execute construct is analogous to Montague's quantifying in, Cooper storage, and Carpenter's scoping constructor (Carpenter, 1998).



(a) Extraction (E) (b) Quantification (Q) (c) Quantifier ambiguity (Q, Q) (d) Quantification (Q, E)

(e) Superlative (C) (f) Comparative (C) (g) Superlative ambiguity (C) (h) Quantification+Superlative (Q, C)

Figure 4: Example DCS trees for utterances in which syntactic and semantic scope diverge. These trees reflect the syntactic structure, which facilitates parsing, but importantly, these trees also precisely encode the correct semantic scope. The main mechanism is using a mark relation (E, Q, or C) low in the tree paired with an execute relation ($\mathrm{X}_\mathbf{i}$) higher up at the desired semantic point.

denotation $d = [\![z]\!]_w$ to include any information about the marked nodes in z that can be accessed by an execute relation later on. In the basic version, d was simply the consistent assignments to the root. Now d contains the consistent joint assignments to the *active nodes* (which include the root and all marked nodes), as well as information stored about each marked node. Think of d as consisting of n *columns*, one for each active node according to a pre-order traversal of z. Column 1 always corresponds to the root node. Formally, a denotation is defined as follows (see Figure 5 for an example):

Definition 2 (Denotations) *Let $\mathcal{D}$ be the set of denotations, where each $d \in \mathcal{D}$ consists of*

- *a set of arrays $d.A$, where each array $\mathbf{a} = [a_1, \ldots, a_n] \in d.A$ is a sequence of n tuples ($a_i \in \mathcal{V}^*$); and*
- *a list of n* stores *$d.\boldsymbol{\alpha} = (d.\alpha_1, \ldots, d.\alpha_n)$, where each store α contains a mark relation $\alpha.r \in \{\mathrm{E}, \mathrm{Q}, \mathrm{C}, \emptyset\}$, a base denotation $\alpha.b \in \mathcal{D} \cup \{\emptyset\}$, and a child denotation $\alpha.c \in \mathcal{D} \cup \{\emptyset\}$.*

We write d as $\langle\!\langle A; (r_1, b_1, c_1); \ldots; (r_n, b_n, c_n) \rangle\!\rangle$. We use $d\{r_i = x\}$ to mean d with $d.r_i = d.\alpha_i.r = x$ (similar definitions apply for $d\{\alpha_i = x\}$, $d\{b_i = x\}$, and $d\{c_i = x\}$).

The denotation of a DCS tree can now be defined recursively:

$$[\![\langle p \rangle]\!]_w = \langle\!\langle \{[v] : v \in w(p)\}; \emptyset \rangle\!\rangle, \quad (3)$$

$$[\![\left\langle p; \mathbf{e}; \begin{smallmatrix} j \\ j' \end{smallmatrix} : c \right\rangle]\!]_w = [\![p; \mathbf{e}]\!]_w \bowtie_{j,j'} [\![c]\!]_w, \quad (4)$$

$$[\![\langle p; \mathbf{e}; \Sigma : c \rangle]\!]_w = [\![p; \mathbf{e}]\!]_w \bowtie_{*,*} \Sigma\left([\![c]\!]_w\right), \quad (5)$$

$$[\![\langle p; \mathbf{e}; \mathrm{X}_\mathbf{i} : c \rangle]\!]_w = [\![p; \mathbf{e}]\!]_w \bowtie_{*,*} \mathbf{X_i}([\![c]\!]_w), \quad (6)$$

$$[\![\langle p; \mathbf{e}; \mathrm{E} : c \rangle]\!]_w = \mathbf{M}([\![p; \mathbf{e}]\!]_w, \mathrm{E}, c), \quad (7)$$

$$[\![\langle p; \mathbf{e}; \mathrm{C} : c \rangle]\!]_w = \mathbf{M}([\![p; \mathbf{e}]\!]_w, \mathrm{C}, c), \quad (8)$$

$$[\![\langle p; \mathrm{Q} : c; \mathbf{e} \rangle]\!]_w = \mathbf{M}([\![p; \mathbf{e}]\!]_w, \mathrm{Q}, c). \quad (9)$$

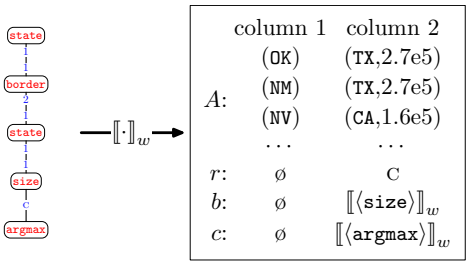


Figure 5: Example of the denotation for a DCS tree with a compare relation C. This denotation has two columns, one for each active node—the root node `state` and the marked node `size`.

The base case is defined in (3): if z is a single node with predicate p, then the denotation of z has one column with the tuples $w(p)$ and an empty store. The other six cases handle different edge relations. These definitions depend on several operations ($\bowtie_{\mathbf{j},\mathbf{j}'}, \Sigma, \mathbf{X_i}, \mathbf{M}$) which we will define shortly, but let us first get some intuition.

Let z be a DCS tree. If the last child c of z's root is a join ($\binom{j}{j'}$), aggregate (Σ), or execute ($\mathrm{X}_\mathbf{i}$) relation ((4)–(6)), then we simply recurse on z with c removed and join it with some transformation (identity, Σ, or $\mathrm{X}_\mathbf{i}$) of c's denotation. If the last (or first) child is connected via a mark relation E, C (or Q), then we strip off that child and put the appropriate information in the store by invoking $\mathbf{M}$.

We now define the operations $\bowtie_{\mathbf{j},\mathbf{j}'}, \Sigma, \mathbf{X_i}, \mathbf{M}$. Some helpful notation: For a sequence $\mathbf{v} = (v_1, \ldots, v_n)$ and indices $\mathbf{i} = (i_1, \ldots, i_k)$, let $\mathbf{v_i} = (v_{i_1}, \ldots, v_{i_k})$ be the projection of $\mathbf{v}$ onto $\mathbf{i}$; we write $\mathbf{v_{-i}}$ to mean $\mathbf{v}_{[1,\ldots,n]\setminus\mathbf{i}}$. Extending this notation to denotations, let $\langle\!\langle A; \boldsymbol{\alpha} \rangle\!\rangle[\mathbf{i}] = \langle\!\langle \{\mathbf{a_i} : \mathbf{a} \in A\}; \boldsymbol{\alpha_i} \rangle\!\rangle$. Let $d[-\emptyset] = d[-\mathbf{i}]$, where $\mathbf{i}$ are the columns with empty stores. For example, for d in Figure 5, $d[1]$ keeps column 1, $d[-\emptyset]$ keeps column 2, and $d[2,-2]$ swaps the two columns.

Join The *join* of two denotations d and d' with respect to components $\mathbf{j}$ and $\mathbf{j}'$ ($*$ means all components) is formed by concatenating all arrays $\mathbf{a}$ of d with all compatible arrays $\mathbf{a}'$ of d', where compatibility means $a_{1\mathbf{j}} = a'_{1\mathbf{j}'}$. The stores are also concatenated ($\boldsymbol{\alpha} + \boldsymbol{\alpha}'$). Non-initial columns with empty stores are projected away by applying $\cdot[1,-\emptyset]$. The full definition of join is as follows:

$$\langle\!\langle A; \boldsymbol{\alpha} \rangle\!\rangle \bowtie_{\mathbf{j},\mathbf{j}'} \langle\!\langle A'; \boldsymbol{\alpha}' \rangle\!\rangle = \langle\!\langle A''; \boldsymbol{\alpha} + \boldsymbol{\alpha}' \rangle\!\rangle[1,-\emptyset],$$
$$A'' = \{\mathbf{a} + \mathbf{a}' : \mathbf{a} \in A, \mathbf{a}' \in A', a_{1\mathbf{j}} = a'_{1\mathbf{j}'}\}. \quad (10)$$

Aggregate The aggregate operation takes a denotation and forms a set out of the tuples in the first column for each setting of the rest of the columns:

$$\Sigma\left(\langle\!\langle A; \boldsymbol{\alpha} \rangle\!\rangle\right) = \langle\!\langle A' \cup A''; \boldsymbol{\alpha} \rangle\!\rangle \quad (11)$$
$$A' = \{[S(\mathbf{a}), a_2, \ldots, a_n] : \mathbf{a} \in A\}$$
$$S(\mathbf{a}) = \{a'_1 : [a'_1, a_2, \ldots, a_n] \in A\}$$
$$A'' = \{[\emptyset, a_2, \ldots, a_n] : \neg\exists a_1, \mathbf{a} \in A,$$
$$\forall 2 \le i \le n, [a_i] \in d.b_i[1].A\}.$$

2.2.1 Mark and Execute

Now we turn to the mark ($\mathbf{M}$) and execute ($\mathbf{X_i}$) operations, which handles the divergence between syntactic and semantic scope. In some sense, this is the technical core of DCS. Marking is simple: When a node (e.g., `size` in Figure 5) is marked (e.g., with relation C), we simply put the relation r, current denotation d and child c's denotation into the store of column 1:

$$\mathbf{M}(d, r, c) = d\{r_1 = r, b_1 = d, c_1 = [\![c]\!]_w\}. \quad (12)$$

The execute operation $\mathbf{X_i}(d)$ processes columns $\mathbf{i}$ in reverse order. It suffices to define $\mathbf{X}_i(d)$ for a single column i. There are three cases:

Extraction ($d.r_i = \mathrm{E}$) In the basic version, the denotation of a tree was always the set of consistent values of the root node. Extraction allows us to return the set of consistent values of a marked non-root node. Formally, extraction simply moves the i-th column to the front: $\mathbf{X}_i(d) = d[i, -(i, \emptyset)]\{\alpha_1 = \emptyset\}$. For example, in Figure 4(a), before execution, the denotation of the DCS tree is $\langle\!\langle \{[(\texttt{CA}, \texttt{OR}), (\texttt{OR})], \ldots\}; \emptyset; (\mathrm{E}, [\![\langle \texttt{state} \rangle]\!]_w, \emptyset) \rangle\!\rangle$; after applying X_1, we have $\langle\!\langle \{[(\texttt{OR})], \ldots\}; \emptyset \rangle\!\rangle$.

Generalized Quantification ($d.r_i = \mathrm{Q}$) Generalized quantifiers are predicates on two sets, a restrictor A and a nuclear scope B. For example, $w(\texttt{no}) = \{(A, B) : A \cap B = \emptyset\}$ and $w(\texttt{most}) = \{(A, B) : |A \cap B| > \frac{1}{2}|A|\}$.

In a DCS tree, the quantifier appears as the child of a Q relation, and the restrictor is the parent (see Figure 4(b) for an example). This information is retrieved from the store when the

quantifier in column i is executed. In particular, the restrictor is $A = \Sigma(d.b_i)$ and the nuclear scope is $B = \Sigma(d[i, -(i, \emptyset)])$. We then apply $d.c_i$ to these two sets (technically, denotations) and project away the first column: $\mathbf{X}_i(d) = ((d.c_i \bowtie_{1,1} A) \bowtie_{2,1} B)[-1]$.

For the example in Figure 4(b), the denotation of the DCS tree before execution is $\langle\langle\emptyset; \emptyset; (\texttt{Q}, [\![\langle\texttt{state}\rangle]\!]_w, [\![\langle\texttt{no}\rangle]\!]_w)\rangle\rangle$. The restrictor set ($A$) is the set of all states, and the nuclear scope (B) is the empty set. Since (A, B) exists in no, the final denotation, which projects away the actual pair, is $\langle\langle\{[\,]\}\rangle\rangle$ (our representation of true).

Figure 4(c) shows an example with two interacting quantifiers. The quantifier scope ambiguity is resolved by the choice of execute relation; $\texttt{X}_{12}$ gives the narrow reading and $\texttt{X}_{21}$ gives the wide reading. Figure 4(d) shows how extraction and quantification work together.

Comparatives and Superlatives ($d.r_i = \texttt{C}$) To compare entities, we use a set S of (x, y) pairs, where x is an entity and y is a number. For superlatives, the argmax predicate denotes pairs of sets and the set's largest element(s): $w(\texttt{argmax}) = \{(S, x^*) : x^* \in \text{argmax}_{x \in S_1} \max S(x)\}$. For comparatives, $w(\texttt{more})$ contains triples (S, x, y), where x is "more than" y as measured by S; formally: $w(\texttt{more}) = \{(S, x, y) : \max S(x) > \max S(y)\}$.

In a superlative/comparative construction, the root x of the DCS tree is the entity to be compared, the child c of a C relation is the comparative or superlative, and its parent p contains the information used for comparison (see Figure 4(e) for an example). If d is the denotation of the root, its i-th column contains this information. There are two cases: (i) if the i-th column of d contains pairs (e.g., size in Figure 5), then let $d' = [\![\langle\emptyset\rangle]\!]_w \bowtie_{1,2} d[i, -i]$, which reads out the second components of these pairs; (ii) otherwise (e.g., state in Figure 4(e)), let $d' = [\![\langle\emptyset\rangle]\!]_w \bowtie_{1,2} [\![\langle\texttt{count}\rangle]\!]_w \bowtie_{1,1} \Sigma(d[i, -i])$, which counts the number of things (e.g., states) that occur with each value of the root x. Given d', we construct a denotation S by concatenating ($+_\mathbf{i}$) the second and first columns of d' ($S = \Sigma(+_{2,1}(d'\{\alpha_2 = \emptyset\}))$) and apply the superlative/comparative: $\mathbf{X}_i(d) = ([\![\langle\emptyset\rangle]\!]_w \bowtie_{1,2} (d.c_i \bowtie_{1,1} S))\{\alpha_1 = d.\alpha_1\}$.

Figure 4(f) shows that comparatives are handled using the exact same machinery as superlatives. Figure 4(g) shows that we can naturally account for superlative ambiguity based on where the scope-determining execute relation is placed.

3 Semantic Parsing

We now turn to the task of mapping natural language utterances to DCS trees. Our first question is: given an utterance $\mathbf{x}$, what trees $z \in \mathcal{Z}$ are permissible? To define the search space, we first assume a fixed set of *lexical triggers* L. Each trigger is a pair $(\mathbf{x}, p)$, where $\mathbf{x}$ is a sequence of words (usually one) and p is a predicate (e.g., $\mathbf{x}$ = *California* and p = CA). We use $L(\mathbf{x})$ to denote the set of predicates p triggered by $\mathbf{x}$ ($(\mathbf{x}, p) \in L$). Let $L(\epsilon)$ be the set of *trace predicates*, which can be introduced without an overt lexical trigger.

Given an utterance $\mathbf{x} = (x_1, \ldots, x_n)$, we define $\mathcal{Z}_L(\mathbf{x}) \subset \mathcal{Z}$, the set of permissible DCS trees for $\mathbf{x}$. The basic approach is reminiscent of projective labeled dependency parsing: For each span $i..j$, we build a set of trees $C_{i,j}$ and set $\mathcal{Z}_L(\mathbf{x}) = C_{0,n}$. Each set $C_{i,j}$ is constructed recursively by combining the trees of its subspans $C_{i,k}$ and $C_{k',j}$ for each pair of split points k, k' (words between k and k' are ignored). These combinations are then augmented via a function A and filtered via a function F, to be specified later. Formally, $C_{i,j}$ is defined recursively as follows:

$$C_{i,j} = F\Big(A\Big(L(\mathbf{x}_{i+1..j}) \cup \bigcup_{\substack{i \le k \le k' < j \\ a \in C_{i,k} \\ b \in C_{k',j}}} T_1(a, b)\Big)\Big). \tag{13}$$

In (13), $L(\mathbf{x}_{i+1..j})$ is the set of predicates triggered by the phrase under span $i..j$ (the base case), and $T_d(a, b) = \vec{T}_d(a, b) \cup \overleftarrow{T}_d(b, a)$, which returns all ways of combining trees a and b where b is a descendant of a ($\vec{T}_d$) or vice-versa ($\overleftarrow{T}_d$). The former is defined recursively as follows: $\vec{T}_0(a, b) = \emptyset$, and

$$\vec{T}_d(a, b) = \bigcup_{\substack{r \in \mathcal{R} \\ p \in L(\epsilon)}} \{\langle a; r : b\rangle\} \cup \vec{T}_{d-1}(a, \langle p; r : b\rangle).$$

The latter ($\overleftarrow{T}_k$) is defined similarly. Essentially, $\vec{T}_d(a, b)$ allows us to insert up to d trace predicates between the roots of a and b. This is useful for modeling relations in noun compounds (e.g.,

California cities), and it also allows us to underspecify L. In particular, our L will not include verbs or prepositions; rather, we rely on the predicates corresponding to those words to be triggered by traces.

The augmentation function A takes a set of trees and optionally attaches $\texttt{E}$ and $\texttt{X}_\texttt{i}$ relations to the root (e.g., $A(\langle \texttt{city} \rangle) = \{\langle \texttt{city} \rangle, \langle \texttt{city}; \texttt{E}\!:\!\emptyset \rangle\}$). The filtering function F rules out improperly-typed trees such as $\langle \texttt{city}; {}^{0}_{0}\!:\!\langle \texttt{state} \rangle \rangle$. To further reduce the search space, F imposes a few additional constraints, e.g., limiting the number of marked nodes to 2 and only allowing trace predicates between arity 1 predicates.

Model We now present our discriminative semantic parsing model, which places a log-linear distribution over $z \in \mathcal{Z}_L(\mathbf{x})$ given an utterance $\mathbf{x}$. Formally, $p_\theta(z \mid \mathbf{x}) \propto e^{\phi(\mathbf{x},z)^\top \theta}$, where θ and $\phi(\mathbf{x}, z)$ are parameter and feature vectors, respectively. As a running example, consider $\mathbf{x}$ = *city that is in California* and $z = \langle \texttt{city}; {}^{1}_{1}\!:\!\langle \texttt{loc}; {}^{2}_{1}\!:\!\langle \texttt{CA} \rangle \rangle \rangle$, where *city* triggers `city` and *California* triggers `CA`.

To define the features, we technically need to augment each tree $z \in \mathcal{Z}_L(\mathbf{x})$ with alignment information—namely, for each predicate in z, the span in $\mathbf{x}$ (if any) that triggered it. This extra information is already generated from the recursive definition in (13).

The feature vector $\phi(\mathbf{x}, z)$ is defined by sums of five simple indicator feature templates: ($\mathbf{F}_1$) a word triggers a predicate (e.g., $[\textit{city}, \texttt{city}]$); ($\mathbf{F}_2$) a word is under a relation (e.g., $[\textit{that}, {}^{1}_{1}]$); ($\mathbf{F}_3$) a word is under a trace predicate (e.g., $[\textit{in}, \texttt{loc}]$); ($\mathbf{F}_4$) two predicates are linked via a relation in the left or right direction (e.g., $[\texttt{city}, {}^{1}_{1}, \texttt{loc}, \text{RIGHT}]$); and ($\mathbf{F}_5$) a predicate has a child relation (e.g., $[\texttt{city}, {}^{1}_{1}]$).

Learning Given a training dataset $\mathcal{D}$ containing $(\mathbf{x}, y)$ pairs, we define the regularized marginal log-likelihood objective $\mathcal{O}(\theta) = \sum_{(\mathbf{x},y)\in\mathcal{D}} \log p_\theta([\![z]\!]_w = y \mid \mathbf{x}, z \in \mathcal{Z}_L(\mathbf{x})) - \lambda\|\theta\|_2^2$, which sums over all DCS trees z that evaluate to the target answer y.

Our model is arc-factored, so we can sum over all DCS trees in $\mathcal{Z}_L(\mathbf{x})$ using dynamic programming. However, in order to learn, we need to sum over $\{z \in \mathcal{Z}_L(\mathbf{x}) : [\![z]\!]_w = y\}$, and unfortunately, the additional constraint $[\![z]\!]_w = y$ does not factorize. We therefore resort to beam search. Specifically, we truncate each $C_{i,j}$ to a maximum of K candidates sorted by decreasing score based on parameters θ. Let $\tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x})$ be this approximation of $\mathcal{Z}_L(\mathbf{x})$.

Our learning algorithm alternates between (i) using the current parameters θ to generate the K-best set $\tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x})$ for each training example $\mathbf{x}$, and (ii) optimizing the parameters to put probability mass on the correct trees in these sets; sets containing no correct answers are skipped. Formally, let $\tilde{\mathcal{O}}(\theta, \theta')$ be the objective function $\mathcal{O}(\theta)$ with $\mathcal{Z}_L(\mathbf{x})$ replaced with $\tilde{\mathcal{Z}}_{L,\theta'}(\mathbf{x})$. We optimize $\tilde{\mathcal{O}}(\theta, \theta')$ by setting $\theta^{(0)} = \vec{0}$ and iteratively solving $\theta^{(t+1)} = \text{argmax}_\theta \tilde{\mathcal{O}}(\theta, \theta^{(t)})$ using L-BFGS until $t = T$. In all experiments, we set $\lambda = 0.01$, $T = 5$, and $K = 100$. After training, given a new utterance $\mathbf{x}$, our system outputs the most likely y, summing out the latent logical form z: $\text{argmax}_y\, p_{\theta^{(T)}}(y \mid \mathbf{x}, z \in \tilde{\mathcal{Z}}_{L,\theta^{(T)}})$.

4 Experiments

We tested our system on two standard datasets, GEO and JOBS. In each dataset, each sentence $\mathbf{x}$ is annotated with a Prolog logical form, which we use only to evaluate and get an answer y. This evaluation is done with respect to a world w. Recall that a world w maps each predicate $p \in \mathcal{P}$ to a set of tuples $w(p)$. There are three types of predicates in $\mathcal{P}$: *generic* (e.g., `argmax`), *data* (e.g., `city`), and *value* (e.g., `CA`). GEO has 48 non-value predicates and JOBS has 26. For GEO, w is the standard US geography database that comes with the dataset. For JOBS, if we use the standard Jobs database, close to half the y's are empty, which makes it uninteresting. We therefore generated a random Jobs database instead as follows: we created 100 job IDs. For each data predicate p (e.g., `language`), we add each possible tuple (e.g., $(\texttt{job37}, \texttt{Java})$) to $w(p)$ independently with probability 0.8.

We used the same training-test splits as Zettlemoyer and Collins (2005) (600+280 for GEO and 500+140 for JOBS). During development, we further held out a random 30% of the training sets for validation.

Our lexical triggers L include the following: (i) predicates for a small set of $\approx$ 20 function words (e.g., $(\textit{most}, \texttt{argmax})$), (ii) (x, x) for each value

System	Accuracy
Clarke et al. (2010) w/answers	73.2
Clarke et al. (2010) w/logical forms	80.4
Our system (DCS with L)	78.9
Our system (DCS with L^+)	**87.2**

Table 2: Results on GEO with 250 training and 250 test examples. Our results are averaged over 10 random 250+250 splits taken from our 600 training examples. Of the three systems that do not use logical forms, our two systems yield significant improvements. Our better system even outperforms the system that uses logical forms.

predicate x in w (e.g., $(\textit{Boston}, \texttt{Boston})$), and (iii) predicates for each POS tag in $\{\text{JJ}, \text{NN}, \text{NNS}\}$ (e.g., $(\text{JJ}, \texttt{size})$, $(\text{JJ}, \texttt{area})$, etc.).[3] Predicates corresponding to verbs and prepositions (e.g., `traverse`) are not included as overt lexical triggers, but rather in the trace predicates $L(\epsilon)$.

We also define an augmented lexicon L^+ which includes a prototype word x for each predicate appearing in (iii) above (e.g., $(\textit{large}, \texttt{size})$), which cancels the predicates triggered by x's POS tag. For GEO, there are 22 prototype words; for JOBS, there are 5. Specifying these triggers requires minimal domain-specific supervision.

Results We first compare our system with Clarke et al. (2010) (henceforth, SEMRESP), which also learns a semantic parser from question-answer pairs. Table 2 shows that our system using lexical triggers L (henceforth, DCS) outperforms SEMRESP (78.9% over 73.2%). In fact, although neither DCS nor SEMRESP uses logical forms, DCS uses even less supervision than SEMRESP. SEMRESP requires a lexicon of 1.42 words per non-value predicate, WordNet features, and syntactic parse trees; DCS requires only words for the domain-independent predicates (overall, around 0.5 words per non-value predicate), POS tags, and very simple indicator features. In fact, DCS performs comparably to even the version of SEMRESP trained using logical forms. If we add prototype triggers (use L^+), the resulting system (DCS$^+$) outperforms both versions of SEMRESP by a significant margin (87.2% over 73.2% and 80.4%).

[3]We used the Berkeley Parser (Petrov et al., 2006) to perform POS tagging. The triggers $L(x)$ for a word x thus include $L(t)$ where t is the POS tag of x.

System	GEO	JOBS
Tang and Mooney (2001)	79.4	79.8
Wong and Mooney (2007)	86.6	–
Zettlemoyer and Collins (2005)	79.3	79.3
Zettlemoyer and Collins (2007)	81.6	–
Kwiatkowski et al. (2010)	88.2	–
Kwiatkowski et al. (2010)	88.9	–
Our system (DCS with L)	88.6	91.4
Our system (DCS with L^+)	**91.1**	**95.0**

Table 3: Accuracy (recall) of systems on the two benchmarks. The systems are divided into three groups. Group 1 uses 10-fold cross-validation; groups 2 and 3 use the independent test set. Groups 1 and 2 measure accuracy of logical form; group 3 measures accuracy of the answer; but there is very small difference between the two as seen from the Kwiatkowski et al. (2010) numbers. Our best system improves substantially over past work, despite using no logical forms as training data.

Next, we compared our systems (DCS and DCS$^+$) with the state-of-the-art semantic parsers on the full dataset for both GEO and JOBS (see Table 3). All other systems require logical forms as training data, whereas ours does not. Table 3 shows that even DCS, which does not use prototypes, is comparable to the best previous system (Kwiatkowski et al., 2010), and by adding a few prototypes, DCS$^+$ offers a decisive edge (91.1% over 88.9% on GEO). Rather than using lexical triggers, several of the other systems use IBM word alignment models to produce an initial word-predicate mapping. This option is not available to us since we do not have annotated logical forms, so we must instead rely on lexical triggers to define the search space. Note that having lexical triggers is a much weaker requirement than having a CCG lexicon, and far easier to obtain than logical forms.

Intuitions How is our system learning? Initially, the weights are zero, so the beam search is essentially unguided. We find that only for a small fraction of training examples do the K-best sets contain any trees yielding the correct answer (29% for DCS on GEO). However, training on just these examples is enough to improve the parameters, and this 29% increases to 66% and then to 95% over the next few iterations. This bootstrapping behavior occurs naturally: The "easy" examples are processed first, where easy is defined by the ability of the current

model to generate the correct answer using any tree.

Our system learns lexical associations between words and predicates. For example, *area* (by virtue of being a noun) triggers many predicates: `city`, `state`, `area`, etc. Inspecting the final parameters (DCS on GEO), we find that the feature $[\textit{area}, \texttt{area}]$ has a much higher weight than $[\textit{area}, \texttt{city}]$. Trace predicates can be inserted anywhere, but the features favor some insertions depending on the words present (for example, $[\textit{in}, \texttt{loc}]$ has high weight).

The errors that the system makes stem from multiple sources, including errors in the POS tags (e.g., *states* is sometimes tagged as a verb, which triggers no predicates), confusion of Washington state with Washington D.C., learning the wrong lexical associations due to data sparsity, and having an insufficiently large K.

5 Discussion

A major focus of this work is on our semantic representation, DCS, which offers a new perspective on compositional semantics. To contrast, consider CCG (Steedman, 2000), in which semantic parsing is driven from the lexicon. The lexicon encodes information about how each word can used in context; for example, the lexical entry for *borders* is $\text{S}\backslash\text{NP}/\text{NP} : \lambda y.\lambda x.\texttt{border}(x, y)$, which means *borders* looks right for the first argument and left for the second. These rules are often too stringent, and for complex utterances, especially in free word-order languages, either disharmonic combinators are employed (Zettlemoyer and Collins, 2007) or words are given multiple lexical entries (Kwiatkowski et al., 2010).

In DCS, we start with lexical triggers, which are more basic than CCG lexical entries. A trigger for *borders* specifies only that `border` can be used, but not how. The combination rules are encoded in the features as soft preferences. This yields a more factorized and flexible representation that is easier to search through and parametrize using features. It also allows us to easily add new lexical triggers without becoming mired in the semantic formalism.

Quantifiers and superlatives significantly complicate scoping in lambda calculus, and often type raising needs to be employed. In DCS, the mark-execute construct provides a flexible framework for dealing with scope variation. Think of DCS as a higher-level programming language tailored to natural language, which results in programs (DCS trees) which are much simpler than the logically-equivalent lambda calculus formulae.

The idea of using CSPs to represent semantics is inspired by Discourse Representation Theory (DRT) (Kamp and Reyle, 1993; Kamp et al., 2005), where variables are discourse referents. The restriction to trees is similar to economical DRT (Bos, 2009).

The other major focus of this work is program induction—inferring logical forms from their denotations. There has been a fair amount of past work on this topic: Liang et al. (2010) induces combinatory logic programs in a non-linguistic setting. Eisenstein et al. (2009) induces conjunctive formulae and uses them as features in another learning problem. Piantadosi et al. (2008) induces first-order formulae using CCG in a small domain assuming observed lexical semantics. The closest work to ours is Clarke et al. (2010), which we discussed earlier.

The integration of natural language with denotations computed against a world (grounding) is becoming increasingly popular. Feedback from the world has been used to guide both syntactic parsing (Schuler, 2003) and semantic parsing (Popescu et al., 2003; Clarke et al., 2010). Past work has also focused on aligning text to a world (Liang et al., 2009), using text in reinforcement learning (Branavan et al., 2009; Branavan et al., 2010), and many others. Our work pushes the grounded language agenda towards deeper representations of language—think grounded compositional semantics.

6 Conclusion

We built a system that interprets natural language utterances much more accurately than existing systems, despite using no annotated logical forms. Our system is based on a new semantic representation, DCS, which offers a simple and expressive alternative to lambda calculus. Free from the burden of annotating logical forms, we hope to use our techniques in developing even more accurate and broader-coverage language understanding systems.

Acknowledgments We thank Luke Zettlemoyer and Tom Kwiatkowski for providing us with data and answering questions.

References

J. Bos. 2009. A controlled fragment of DRT. In *Workshop on Controlled Natural Language*, pages 1–5.

S. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay. 2009. Reinforcement learning for mapping instructions to actions. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, Singapore. Association for Computational Linguistics.

S. Branavan, L. Zettlemoyer, and R. Barzilay. 2010. Reading between the lines: Learning to map high-level instructions to commands. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics.

B. Carpenter. 1998. *Type-Logical Semantics*. MIT Press.

J. Clarke, D. Goldwasser, M. Chang, and D. Roth. 2010. Driving semantic parsing from the world's response. In *Computational Natural Language Learning (CoNLL)*.

R. Dechter. 2003. *Constraint Processing*. Morgan Kaufmann.

J. Eisenstein, J. Clarke, D. Goldwasser, and D. Roth. 2009. Reading to learn: Constructing features from semantic abstracts. In *Empirical Methods in Natural Language Processing (EMNLP)*, Singapore.

R. Ge and R. J. Mooney. 2005. A statistical semantic parser that integrates syntax and semantics. In *Computational Natural Language Learning (CoNLL)*, pages 9–16, Ann Arbor, Michigan.

H. Kamp and U. Reyle. 1993. *From Discourse to Logic: An Introduction to the Model-theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer, Dordrecht.

H. Kamp, J. v. Genabith, and U. Reyle. 2005. Discourse representation theory. In *Handbook of Philosophical Logic*.

R. J. Kate and R. J. Mooney. 2007. Learning language semantics from ambiguous supervision. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 895–900, Cambridge, MA. MIT Press.

R. J. Kate, Y. W. Wong, and R. J. Mooney. 2005. Learning to transform natural to formal languages. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1062–1068, Cambridge, MA. MIT Press.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Empirical Methods in Natural Language Processing (EMNLP)*.

P. Liang, M. I. Jordan, and D. Klein. 2009. Learning semantic correspondences with less supervision. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, Singapore. Association for Computational Linguistics.

P. Liang, M. I. Jordan, and D. Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*. Omnipress.

S. Petrov, L. Barrett, R. Thibaux, and D. Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*, pages 433–440. Association for Computational Linguistics.

S. T. Piantadosi, N. D. Goodman, B. A. Ellis, and J. B. Tenenbaum. 2008. A Bayesian model of the acquisition of compositional semantics. In *Proceedings of the Thirtieth Annual Conference of the Cognitive Science Society*.

H. Poon and P. Domingos. 2009. Unsupervised semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, Singapore.

A. Popescu, O. Etzioni, and H. Kautz. 2003. Towards a theory of natural language interfaces to databases. In *International Conference on Intelligent User Interfaces (IUI)*.

W. Schuler. 2003. Using model-theoretic semantic interpretation to guide statistical parsing and word recognition in a spoken language interface. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics.

M. Steedman. 2000. *The Syntactic Process*. MIT Press.

L. R. Tang and R. J. Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *European Conference on Machine Learning*, pages 466–477.

Y. W. Wong and R. J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Association for Computational Linguistics (ACL)*, pages 960–967, Prague, Czech Republic. Association for Computational Linguistics.

M. Zelle and R. J. Mooney. 1996. Learning to parse database queries using inductive logic proramming. In *Association for the Advancement of Artificial Intelligence (AAAI)*, Cambridge, MA. MIT Press.

L. S. Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence (UAI)*, pages 658–666.

L. S. Zettlemoyer and M. Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*, pages 678–687.