Learning Dependency-Based Compositional Semantics

Percy Liang*
University of California, Berkeley

Michael I. Jordan**
University of California, Berkeley

Dan Klein†
University of California, Berkeley

Suppose we want to build a system that answers a natural language question by representing its semantics as a logical form and computing the answer given a structured database of facts. The core part of such a system is the semantic parser that maps questions to logical forms. Semantic parsers are typically trained from examples of questions annotated with their target logical forms, but this type of annotation is expensive.

Our goal is to instead learn a semantic parser from question–answer pairs, where the logical form is modeled as a latent variable. We develop a new semantic formalism, dependency-based compositional semantics (DCS) and define a log-linear distribution over DCS logical forms. The model parameters are estimated using a simple procedure that alternates between beam search and numerical optimization. On two standard semantic parsing benchmarks, we show that our system obtains comparable accuracies to even state-of-the-art systems that do require annotated logical forms.

1. Introduction

One of the major challenges in natural language processing (NLP) is building systems that both handle complex linguistic phenomena and require minimal human effort. The difficulty of achieving both criteria is particularly evident in training semantic parsers, where annotating linguistic expressions with their associated logical forms is expensive but until recently, seemingly unavoidable. Advances in learning latent-variable models, however, have made it possible to progressively reduce the amount of supervision

* Computer Science Division, University of California, Berkeley, CA 94720, USA. E-mail: pliang@cs.stanford.edu.
** Computer Science Division and Department of Statistics, University of California, Berkeley, CA 94720, USA. E-mail: jordan@cs.berkeley.edu.
† Computer Science Division, University of California, Berkeley, CA 94720, USA. E-mail: klein@cs.berkeley.edu.

Submission received: 12 September 2011; revised submission received: 19 February 2012; accepted for publication: 18 April 2012.

doi:10.1162/COLI_a_00127

required for various semantics-related tasks (Zettlemoyer and Collins 2005; Branavan et al. 2009; Liang, Jordan, and Klein 2009; Clarke et al. 2010; Artzi and Zettlemoyer 2011; Goldwasser et al. 2011). In this article, we develop new techniques to learn accurate semantic parsers from even weaker supervision.

We demonstrate our techniques on the concrete task of building a system to answer questions given a structured database of facts; see Figure 1 for an example in the domain of U.S. geography. This problem of building natural language interfaces to databases (NLIDBs) has a long history in NLP, starting from the early days of artificial intelligence with systems such as LUNAR (Woods, Kaplan, and Webber 1972), CHAT-80 (Warren and Pereira 1982), and many others (see Androutsopoulos, Ritchie, and Thanisch [1995] for an overview). We believe NLIDBs provide an appropriate starting point for semantic parsing because they lead directly to practical systems, and they allow us to temporarily sidestep intractable philosophical questions on how to represent meaning in general. Early NLIDBs were quite successful in their respective limited domains, but because these systems were constructed from manually built rules, they became difficult to scale up, both to other domains and to more complex utterances. In response, against the backdrop of a statistical revolution in NLP during the 1990s, researchers began to build systems that could learn from examples, with the hope of overcoming the limitations of rule-based methods. One of the earliest statistical efforts was the CHILL system (Zelle and Mooney 1996), which learned a shift-reduce semantic parser. Since then, there has been a healthy line of work yielding increasingly more accurate semantic parsers by using new semantic representations and machine learning techniques (Miller et al. 1996; Zelle and Mooney 1996; Tang and Mooney 2001; Ge and Mooney 2005; Kate, Wong, and Mooney 2005; Zettlemoyer and Collins 2005; Kate and Mooney 2006; Wong and Mooney 2006; Kate and Mooney 2007; Wong and Mooney 2007; Zettlemoyer and Collins 2007; Kwiatkowski et al. 2010, 2011).

Although statistical methods provided advantages such as robustness and portability, however, their application in semantic parsing achieved only limited success. One of the main obstacles was that these methods depended crucially on having examples of utterances paired with logical forms, and this requires substantial human effort to obtain. Furthermore, the annotators must be proficient in some formal language, which drastically reduces the size of the annotator pool, dampening any hope of acquiring enough data to fulfill the vision of learning highly accurate systems.

In response to these concerns, researchers have recently begun to explore the possibility of learning a semantic parser without any annotated logical forms (Clarke et al.

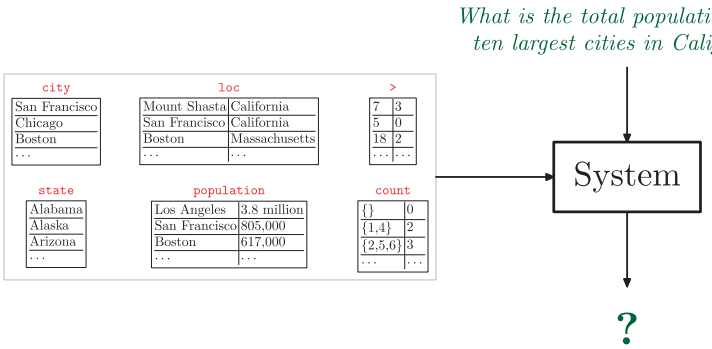


Figure 1
The concrete objective: A system that answers natural language questions given a structured database of facts. An example is shown in the domain of U.S. geography.

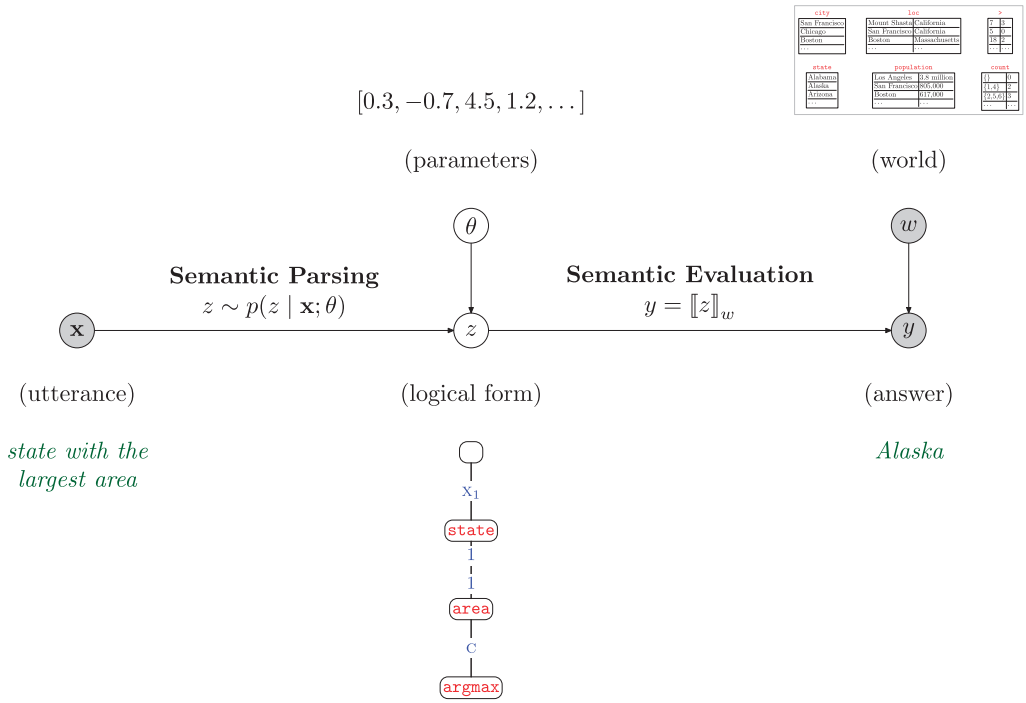


Figure 2
Our statistical methodology consists of two steps: (i) semantic parsing ($p(z \mid \mathbf{x}; \theta)$): an utterance $\mathbf{x}$ is mapped to a logical form z by drawing from a log-linear distribution parametrized by a vector θ; and (ii) evaluation ($[[z]]_w$): the logical form z is evaluated with respect to the world w (database of facts) to deterministically produce an answer y. The figure also shows an example configuration of the variables around the graphical model. Logical forms z are represented as labeled trees. During learning, we are given w and (x, y) pairs (shaded nodes) and try to infer the latent logical forms z and parameters θ.

2010; Artzi and Zettlemoyer 2011; Goldwasser et al. 2011; Liang, Jordan, and Klein 2011). It is in this vein that we develop our present work. Specifically, given a set of $(\mathbf{x}, y)$ example pairs, where $\mathbf{x}$ is an utterance (e.g., a question) and y is the corresponding answer, we wish to learn a mapping from $\mathbf{x}$ to y. What makes this mapping particularly interesting is that it passes through a latent logical form z, which is necessary to capture the semantic complexities of natural language. Also note that whereas the logical form z was the end goal in much of earlier work on semantic parsing, for us it is just an intermediate variable—a means towards an end. Figure 2 shows the graphical model which captures the learning setting we just described: The question $\mathbf{x}$, answer y, and world/database w are all observed. We want to infer the logical forms z and the parameters θ of the semantic parser, which are unknown quantities.

Although liberating ourselves from annotated logical forms reduces cost, it does increase the difficulty of the learning problem. The core challenge here is **program induction**: On each example $(\mathbf{x}, y)$, we need to efficiently search over the exponential space of possible logical forms (programs) z and find ones that produce the target answer y, a computationally daunting task. There is also a statistical challenge: How do we parametrize the mapping from utterance $\mathbf{x}$ to logical form z so that it can be learned from only the indirect signal y? To address these two challenges, we must first discuss the issue of *semantic representation*. There are two basic questions here: (i) what

should the formal language for the logical forms z be, and (ii) what are the compositional mechanisms for constructing those logical forms?

The semantic parsing literature has considered many different formal languages for representing logical forms, including SQL (Giordani and Moschitti 2009), Prolog (Zelle and Mooney 1996; Tang and Mooney 2001), a simple functional query language called FunQL (Kate, Wong, and Mooney 2005), and lambda calculus (Zettlemoyer and Collins 2005), just to name a few. The construction mechanisms are equally diverse, including synchronous grammars (Wong and Mooney 2007), hybrid trees (Lu et al. 2008), Combinatory Categorial Grammars (CCG) (Zettlemoyer and Collins 2005), and shift-reduce derivations (Zelle and Mooney 1996). It is worth pointing out that the choice of formal language and the construction mechanism are decisions which are really more orthogonal than is often assumed—the former is concerned with what the logical forms look like; the latter, with how to generate a set of possible logical forms compositionally given an utterance. (How to score these logical forms is yet another dimension.)

Existing systems are rarely based on the joint design of the formal language and the construction mechanism; one or the other is often chosen for convenience from existing implementations. For example, Prolog and SQL have often been chosen as formal languages for convenience in end applications, but they were not designed for representing the semantics of natural language, and, as a result, the construction mechanism that bridges the gap between natural language and formal language is generally complex and difficult to learn. CCG (Steedman 2000) is quite popular in computational linguistics (for example, see Bos et al. [2004] and Zettlemoyer and Collins [2005]). In CCG, logical forms are constructed compositionally using a small handful of combinators (function application, function composition, and type raising). For a wide range of canonical examples, CCG produces elegant, streamlined analyses, but its success really depends on having a good, clean lexicon. During learning, there is often a great amount of uncertainty over the lexical entries, which makes CCG more cumbersome. Furthermore, in real-world applications, we would like to handle disfluent utterances, and this further strains CCG by demanding either extra type-raising rules and disharmonic combinators (Zettlemoyer and Collins 2007) or a proliferation of redundant lexical entries for each word (Kwiatkowski et al. 2010).

To cope with the challenging demands of program induction, we break away from tradition in favor of a new formal language and construction mechanism, which we call **dependency-based compositional semantics** (DCS). The guiding principle behind DCS is to provide a simple and intuitive framework for constructing and representing logical forms. Logical forms in DCS are tree structures called **DCS trees**. The motivation is two-fold: (i) DCS trees are meant to parallel syntactic dependency trees, which facilitates parsing; and (ii) a DCS tree essentially encodes a constraint satisfaction problem, which can be solved efficiently using dynamic programming to obtain the denotation of a DCS tree. In addition, DCS provides a **mark–execute** construct, which provides a uniform way of dealing with scope variation, a major source of trouble in any semantic formalism. The construction mechanism in DCS is a generalization of labeled dependency parsing, which leads to simple and natural algorithms. To a linguist, DCS might appear unorthodox, but it is important to keep in mind that our primary goal is effective program induction, not necessarily to model new linguistic phenomena in the tradition of formal semantics.

Armed with our new semantic formalism, DCS, we then define a discriminative probabilistic model, which is depicted in Figure 2. The semantic parser is a log-linear distribution over DCS trees z given an utterance $\mathbf{x}$. Notably, z is unobserved, and we instead observe only the answer y, which is obtained by evaluating z on a world/database

w. There are an exponential number of possible trees z, and usually dynamic programming can be used to efficiently search over trees. However, in our learning setting (independent of the semantic formalism), we must enforce the global constraint that z produces y. This makes dynamic programming infeasible, so we use beam search (though dynamic programming is still used to compute the denotation of a fixed DCS tree). We estimate the model parameters with a simple procedure that alternates between beam search and optimizing a likelihood objective restricted to those beams. This yields a natural bootstrapping procedure in which learning and search are integrated.

We evaluated our DCS-based approach on two standard benchmarks, GEO, a U.S. geography domain (Zelle and Mooney 1996), and JOBS, a job queries domain (Tang and Mooney 2001). On GEO, we found that our system significantly outperforms previous work that also learns from answers instead of logical forms (Clarke et al. 2010). What is perhaps a more significant result is that our system obtains comparable accuracies to state-of-the-art systems that do rely on annotated logical forms. This demonstrates the viability of training accurate systems with much less supervision than before.

The rest of this article is organized as follows: Section 2 introduces DCS, our new semantic formalism. Section 3 presents our probabilistic model and learning algorithm. Section 4 provides an empirical evaluation of our methods. Section 5 situates this work in a broader context, and Section 6 concludes.

2. Representation

In this section, we present the main conceptual contribution of this work, dependency-based compositional semantics (DCS), using the U.S. geography domain (Zelle and Mooney 1996) as a running example. To do this, we need to define the syntax and semantics of the formal language. The syntax is defined in Section 2.2 and is quite straightforward: The logical forms in the formal language are simply trees, which we call **DCS trees**. In Section 2.3, we give a type-theoretic definition of **worlds** (also known as databases or models) with respect to which we can define the semantics of DCS trees.

The semantics, which is the heart of this article, contains two main ideas: (i) using trees to represent logical forms as constraint satisfaction problems or extensions thereof, and (ii) dealing with cases when syntactic and semantic scope diverge (e.g., for generalized quantification and superlative constructions) using a new construct which we call **mark–execute**. We start in Section 2.4 by introducing the semantics of a basic version of DCS which focuses only on (i) and then extend it to the full version (Section 2.5) to account for (ii).

Finally, having fully specified the formal language, we describe a construction mechanism for mapping a natural language utterance to a set of candidate DCS trees (Section 2.6).

2.1 Notation

Operations on tuples will play a prominent role in this article. For a sequence[1] $v = (v_1, \ldots, v_k)$, we use $|v| = k$ to denote the length of the sequence. For two sequences u and v, we use $u + v = (u_1, \ldots, u_{|u|}, v_1, \ldots, v_{|v|})$ to denote their concatenation.

1 We use the term *sequence* to refer to both tuples $(v_1, \ldots, v_k)$ and arrays $[v_1, \ldots, v_k]$. For our purposes, there is no functional difference between tuples and arrays; the distinction is convenient when we start to talk about arrays of tuples.

For a sequence of positive indices $\mathbf{i} = (i_1, \ldots, i_m)$, let $v_\mathbf{i} = (v_{i_1}, \ldots, v_{i_m})$ consist of the components of v specified by $\mathbf{i}$; we call $v_\mathbf{i}$ the projection of v onto $\mathbf{i}$. We use negative indices to exclude components: $v_{-\mathbf{i}} = (v_{(1,\ldots,|v|)\setminus\mathbf{i}})$. We can also combine sequences of indices by concatenation: $v_{\mathbf{i},\mathbf{j}} = v_\mathbf{i} + v_\mathbf{j}$. Some examples: if $v = (a, b, c, d)$, then $v_2 = b$, $v_{3,1} = (c, a)$, $v_{-3} = (a, b, d)$, $v_{3,-3} = (c, a, b, d)$.

2.2 Syntax of DCS Trees

The syntax of the DCS formal language is built from two ingredients, predicates and relations:

- Let $\mathcal{P}$ be a set of **predicates**. We assume that $\mathcal{P}$ contains a special null predicate ø, domain-independent predicates (e.g., `count`, `<`, `>`, and `=`), and domain-specific predicates (for the U.S. geography domain, `state`, `river`, `border`, etc.). Right now, think of predicates as just labels, which have yet to receive formal semantics.
- Let $\mathcal{R}$ be the set of **relations**. Note that unlike the predicates $\mathcal{P}$, which can vary across domains, the relations $\mathcal{R}$ are fixed. The full set of relations are shown in Table 1. For now, just think of relations as labels—their semantics will be defined in Section 2.4.

The logical forms in DCS are called DCS trees. A DCS tree is a directed rooted tree in which nodes are labeled with predicates and edges are labeled with relations; each node also maintains an ordering over its children. 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 $z.\mathbf{e} = (z.e_1, \ldots, z.e_m)$. Each edge e consists of a relation $e.r \in \mathcal{R}$ (see Table 1) and a child tree $e.c \in \mathcal{Z}$.

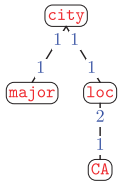
We will either draw a DCS tree graphically or write it compactly as $\langle p; r_1 : c_1; \ldots; r_m : c_m \rangle$ where p is the predicate at the root node and $c_1, \ldots, c_m$ are its m children connected via edges labeled with relations $r_1, \ldots, r_m$, respectively. Figure 3(a) shows an example of a DCS tree expressed using both graphical and compact formats.

Table 1
Possible relations that appear on edges of DCS trees. Basic DCS uses only the join and aggregate relations; the full version of DCS uses all of them.

		Relations $\mathcal{R}$
Name	Relation	Description of semantic function
join	$\overset{j}{j'}$ for $j, j' \in \{1, 2, \ldots\}$	j-th component of parent = j'-th component of child
aggregate	Σ	parent = set of feasible values of child
extract	E	mark node for extraction
quantify	Q	mark node for quantification, negation
compare	C	mark node for superlatives, comparatives
execute	$\mathrm{X}_\mathbf{i}$ for $\mathbf{i} \in \{1, 2 \ldots\}^*$	process marked nodes specified by $\mathbf{i}$

Example: *major city in California*

$z = \langle \texttt{city}; 1 : \langle \texttt{major} \rangle ; 1 : \langle \texttt{loc}; 2 : \langle \texttt{CA} \rangle \rangle \rangle$



(a) DCS tree

$$\lambda c \, \exists m \, \exists \ell \, \exists s \, . \\ \texttt{city}(c) \wedge \texttt{major}(m) \wedge \texttt{loc}(\ell) \wedge \texttt{CA}(s) \wedge \\ c_1 = m_1 \wedge c_1 = \ell_1 \wedge \ell_2 = s_1$$

(b) Lambda calculus formula

(c) Denotation: $[\![z]\!]_w = \{\texttt{SF}, \texttt{LA}, \ldots\}$

Figure 3
(a) An example of a DCS tree (written in both the mathematical and graphical notations). 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, represented here as a lambda calculus formula. For example, the root node label `city` corresponds to a unary predicate $\texttt{city}(c)$, the right child node label `loc` corresponds to a binary predicate $\texttt{loc}(\ell)$ (where ℓ is a pair), and the edge between them denotes the constraint $c_1 = \ell_1$ (where the indices correspond to the two labels on the edge). (c) The denotation of z is the set of feasible values for the root node.

A DCS tree is a logical form, but it is designed to look like a syntactic dependency tree, only with predicates in place of words. As we'll see over the course of this section, it is this transparency between syntax and semantics provided by DCS which leads to a simple and streamlined compositional semantics suitable for program induction.

2.3 Worlds

In the context of question answering, the DCS tree is a formal specification of the question. To obtain an answer, we still need to evaluate the DCS tree with respect to a database of facts (see Figure 4 for an example). We will use the term **world** to refer

w:

`city`

San Francisco
Chicago
Boston
...

`loc`

Mount Shasta	California
San Francisco	California
Boston	Massachusetts
...	...

`>`

7	3
5	0
18	2
...	...

`state`

Alabama
Alaska
Arizona
...

`population`

Los Angeles	3.8 million
San Francisco	805,000
Boston	617,000
...	...

`count`

{}	0
{1,4}	2
{2,5,6}	3
...	...

Figure 4
We use the domain of U.S. geography as a running example. The figure presents an example of a world w (database) in this domain. A world maps each predicate to a set of tuples. For example, the depicted world w maps the predicate `loc` to the set of pairs of places and their containers. Note that functions (e.g., `population`) are also represented as predicates for uniformity. Some predicates (e.g., `count`) map to an infinite number of tuples and would be represented implicitly.

to this database (it is sometimes also called a model, but we avoid this term to avoid confusion with the probabilistic model for learning that we will present in Section 3.1). Throughout this work, we assume the world is fully observed and fixed, which is a realistic assumption for building natural language interfaces to existing databases, but questionable for modeling the semantics of language in general.

2.3.1 Types and Values. To define a world, we start by constructing a set of **values** $\mathcal{V}$. The exact set of values depends on the domain (we will continue to use U.S. geography as a running example). Briefly, $\mathcal{V}$ contains numbers (e.g., $3 \in \mathcal{V}$), strings (e.g., *Washington* $\in \mathcal{V}$), tuples (e.g., (3, *Washington*) $\in \mathcal{V}$), sets (e.g., $\{3, \textit{Washington}\} \in \mathcal{V}$), and other higher-order entities.

To be more precise, we construct $\mathcal{V}$ recursively. First, define a set of primitive values $\mathcal{V}_\star$, which includes the following:

- Numeric values. Each value has the form $x{:}t \in \mathcal{V}_\star$, where $x \in \mathbb{R}$ is a real number and $t \in \{\texttt{number}, \texttt{ordinal}, \texttt{percent}, \texttt{length}, \ldots\}$ is a tag. The tag allows us to differentiate 3, 3rd, 3%, and 3 miles—this will be important in Section 2.6.3. We simply write x for the value $x{:}\texttt{number}$.
- Symbolic values. Each value has the form $x{:}t \in \mathcal{V}_\star$, where x is a string (e.g., *Washington*) and $t \in \{\texttt{string}, \texttt{city}, \texttt{state}, \texttt{river}, \ldots\}$ is a tag. Again, the tag allows us to differentiate, for example, the entities *Washington*$:\texttt{city}$ and *Washington*$:\texttt{state}$.

Now we build the full set of values $\mathcal{V}$ from the primitive values $\mathcal{V}_\star$. To define $\mathcal{V}$, we need a bit more machinery: To avoid logical paradoxes, we construct $\mathcal{V}$ in increasing order of complexity using types (see Carpenter [1998] for a similar construction). The casual reader can skip this construction without losing any intuition.

Define the set of types $\mathcal{T}$ to be the smallest set that satisfies the following properties:

1. The primitive type $\star \in \mathcal{T}$;
2. The tuple type $(t_1, \ldots, t_k) \in \mathcal{T}$ for each $k \geq 0$ and each non-tuple type $t_i \in \mathcal{T}$ for $i = 1, \ldots, k$; and
3. The set type $\{t\} \in \mathcal{T}$ for each tuple type $t \in \mathcal{T}$.

Note that $\{\star\}$, $\{\{\star\}\}$, and $((\star))$ are not valid types.

For each type $t \in \mathcal{T}$, we construct a corresponding set of values $\mathcal{V}_t$:

1. For the primitive type $t = \star$, the primitive values $\mathcal{V}_\star$ have already been specified. Note that these types are rather coarse: Primitive values with different tags are considered to have the same type $\star$.
2. For a tuple type $t = (t_1, \ldots, t_k)$, $\mathcal{V}_t$ is the cross product of the values of its component types:

$$\mathcal{V}_t = \{(v_1, \ldots, v_k) : \forall i, v_i \in \mathcal{V}_{t_i}\} \tag{1}$$

3. For a set type $t = \{t'\}$, $\mathcal{V}_t$ contains all subsets of its element type t':

$$\mathcal{V}_t = \{s : s \subset \mathcal{V}_{t'}\} \tag{2}$$

With this last condition, we ensure that all elements of a set must have the same type. Note that a set is still allowed to have values with different *tags* (e.g., $\{(\textit{Washington}:\texttt{city}), (\textit{Washington}:\texttt{state})\}$ is a valid set, which might denote the semantics of the utterance *things named Washington*). Another distinction is that types are domain-independent whereas tags tend to be more domain-specific.

Let $\mathcal{V} = \cup_{t \in \mathcal{T}} \mathcal{V}_t$ be the set of all possible values.

A world maps each predicate to its *semantics*, which is a set of tuples (see Figure 4 for an example). First, let $\mathcal{T}_{\text{TUPLE}} \subset \mathcal{T}$ be the tuple types, which are the ones of the form $(t_1, \ldots, t_k)$ for some k. Let $\mathcal{V}_{\{\text{TUPLE}\}}$ denote all the sets of tuples (with the same type):

$$\mathcal{V}_{\{\text{TUPLE}\}} \stackrel{\text{def}}{=} \bigcup_{t \in \mathcal{T}_{\text{TUPLE}}} \mathcal{V}_{\{t\}} \tag{3}$$

Now we define a world formally.

Definition 2 (World)
A world $w : \mathcal{P} \mapsto \mathcal{V}_{\{\text{TUPLE}\}} \cup \{\mathcal{V}\}$ is a function that maps each non-null predicate $p \in \mathcal{P} \backslash \{\text{ø}\}$ to a set of tuples $w(p) \in \mathcal{V}_{\{\text{TUPLE}\}}$ and maps the null predicate ø to the set of all values ($w(\text{ø}) = \mathcal{V}$).

For a set of tuples A with the same arity, let $\text{ARITY}(A) = |x|$, where $x \in A$ is arbitrary; if A is empty, then $\text{ARITY}(A)$ is undefined. Now for a predicate $p \in \mathcal{P}$ and world w, define $\text{ARITY}_w(p)$, the arity of predicate p with respect to w, as follows:

$$\text{ARITY}_w(p) = \begin{cases} 1 & \text{if } p = \text{ø} \\ \text{ARITY}(w(p)) & \text{if } p \neq \text{ø} \end{cases} \tag{4}$$

The null predicate has arity 1 by fiat; the arity of a non-null predicate p is inherited from the tuples in $w(p)$.

Remarks. In higher-order logic and lambda calculus, we construct function types and values, whereas in DCS, we construct tuple types and values. The two are equivalent in representational power, but this discrepancy does point out the fact that lambda calculus is based on function application, whereas DCS, as we will see, is based on declarative constraints. The set type $\{(\star, \star)\}$ in DCS corresponds to the function type $\star \to (\star \to \texttt{bool})$. In DCS, there is no explicit `bool` type—it is implicitly represented by using sets.

2.3.2 Examples. The world w maps each domain-specific predicate to a set of tuples (usually a finite set backed by a database). For the U.S. geography domain, w has a

predicate that maps to the set of U.S. states (`state`), another predicate that maps to the set of pairs of entities and where they are located (`loc`), and so on:

$$w(\texttt{state}) = \{(\textit{California}:\texttt{state}), (\textit{Oregon}:\texttt{state}), \dots\} \tag{5}$$

$$w(\texttt{loc}) = \{(\textit{San Francisco}:\texttt{city}, \textit{California}:\texttt{state}), \dots\} \tag{6}$$

$$\dots \tag{7}$$

To shorten notation, we use state abbreviations (e.g., $\texttt{CA} = \textit{California}:\texttt{state}$).

The world w also specifies the semantics of several domain-independent predicates (think of these as helper functions), which usually correspond to an infinite set of tuples. Functions are represented in DCS by a set of input–output pairs. For example, the semantics of the $\texttt{count}_t$ predicate (for each type $t \in \mathcal{T}$) contains pairs of sets S and their cardinalities $|S|$:

$$w(\texttt{count}_t) = \{(S, |S|) : S \in \mathcal{V}_{\{(t)\}}\} \in \mathcal{V}_{\{(\{(t)\},\star)\}} \tag{8}$$

As another example, consider the predicate $\texttt{average}_t$ (for each $t \in \mathcal{T}$), which takes a set of key–value pairs (with keys of type t) and returns the average value. For notational convenience, we treat an arbitrary set of pairs S as a set-valued function: We let $S_1 = \{x : (x, y) \in S\}$ denote the domain of the function, and abusing notation slightly, we define the function $S(x) = \{y : (x, y) \in S\}$ to be the set of values y that co-occur with the given x. The semantics of $\texttt{average}_t$ contains pairs of sets and their averages:

$$w(\texttt{average}_t) = \left\{ (S, z) : S \in \mathcal{V}_{\{(t,\star)\}}, z = |S_1|^{-1} \sum_{x \in S_1} \left[|S(x)|^{-1} \sum_{y \in S(x)} y \right] \right\} \in \mathcal{V}_{\{(\{(t,\star)\},\star)\}} \tag{9}$$

Similarly, we can define the semantics of $\texttt{argmin}_t$ and $\texttt{argmax}_t$, which each takes a set of key–value pairs and returns the keys that attain the smallest (largest) value:

$$w(\texttt{argmin}_t) = \left\{ (S, z) : S \in \mathcal{V}_{\{(t,\star)\}}, z \in \operatorname*{argmin}_{x \in S_1} \min S(x) \right\} \in \mathcal{V}_{\{(\{(t,\star)\},t)\}} \tag{10}$$

$$w(\texttt{argmax}_t) = \left\{ (S, z) : S \in \mathcal{V}_{\{(t,\star)\}}, z \in \operatorname*{argmax}_{x \in S_1} \max S(x) \right\} \in \mathcal{V}_{\{(\{(t,\star)\},t)\}} \tag{11}$$

The extra min and max is needed because $S(x)$ could contain more than one value. We also impose that $w(\texttt{argmin}_t)$ contains only (S, z) such that y is numeric for all $(x, y) \in S$; thus $\texttt{argmin}_t$ denotes a partial function (same for $\texttt{argmax}_t$).

These helper functions are monomorphic: For example, $\texttt{count}_t$ only computes cardinalities of sets of type $\{(t)\}$. In practice, we mostly operate on sets of primitives ($t = \star$). To reduce notation, we omit t to refer to this version: $\texttt{count} = \texttt{count}_\star$, $\texttt{average} = \texttt{average}_\star$, and so forth.

2.4 Semantics of DCS Trees without Mark–Execute (Basic Version)

The semantics or *denotation* of a DCS tree z with respect to a world w is denoted $[\![z]\!]_w$. First, we define the semantics of DCS trees with only join relations (Section 2.4.1). In this case, a DCS tree encodes a constraint satisfaction problem (CSP); this is important because it highlights the constraint-based nature of DCS and also naturally leads to a computationally efficient way of computing denotations (Section 2.4.2). We then allow DCS trees to have aggregate relations (Section 2.4.3). The fragment of DCS which has only join and aggregate relations is called **basic DCS**.

2.4.1 Basic DCS Trees as Constraint Satisfaction Problems. Let z be a DCS tree with only join relations on its edges. In this case, z encodes a CSP as follows: For each node x in z, the CSP has a variable with value $a(x)$; the collection of these values is referred to as an *assignment* a. The predicates and relations of z introduce constraints:

1. $a(x) \in w(p)$ for each node x labeled with predicate $p \in \mathcal{P}$; and
2. $a(x)_j = a(y)_{j'}$ for each edge (x, y) labeled with $\overset{j}{}_{j'} \in \mathcal{R}$, which says that the j-th component of $a(x)$ must equal the j'-th component of $a(y)$.

We say that an assignment a is *feasible* if it satisfies these two constraints. Next, for a node x, define $V(x) = \{a(x) : \text{assignment } a \text{ is feasible}\}$ as the set of feasible values for x—these are the ones that are consistent with at least one feasible assignment. Finally, we define the denotation of the DCS tree z with respect to the world w to be $[\![z]\!]_w = V(x_0)$, where x_0 is the root node of z.

Figure 3(a) shows an example of a DCS tree. The corresponding CSP has four variables c, m, ℓ, s.[2] In Figure 3(b), we have written the equivalent lambda calculus formula. The non-root nodes are existentially quantified, the root node c is λ-abstracted, and all constraints introduced by predicates and relations are conjoined. The λ-abstraction of c represents the fact that the denotation is the set of feasible values for c (note the equivalence between the Boolean function $\lambda c.p(c)$ and the set $\{c : p(c)\}$).

Remarks. Note that CSPs only allow existential quantification and conjunction. Why did we choose this particular logical subset as a starting point, rather than allowing universal quantification, negation, or disjunction? There seems to be something fundamental about this subset, which also appears in Discourse Representation Theory (DRT) (Kamp and Reyle 1993; Kamp, van Genabith, and Reyle 2005). Briefly, logical forms in DRT are called Discourse Representation Structures (DRSs), each of which contains (i) a set of existentially quantified discourse referents (variables), (ii) a set of conjoined discourse conditions (constraints), and (iii) nested DRSs. If we exclude nested DRSs, a DRS is exactly a CSP.[3] The default existential quantification and conjunction are quite natural for modeling cross-sentential anaphora: New variables can be added to

2 Technically, the node is c and the variable is $a(c)$, but we use c to denote the variable to simplify notation.
3 Unlike the CSPs corresponding to DCS trees, the CSPs corresponding to DRSs need not be tree-structured, though economical DRT (Bos 2009) imposes a tree-like restriction on DRSs for computational reasons.

a DRS and connected to other variables. Indeed, DRT was originally motivated by these phenomena (see Kamp and Reyle [1993] for more details).[4]

Tree-structured CSPs can capture unboundedly complex recursive structures—such as *cities in states that border states that have rivers that...*. Trees are limited, however, in that they are unable to capture long-distance dependencies such as those arising from anaphora. For example, in the phrase *a state with a river that traverses its capital*, *its* binds to *state*, but this dependence cannot be captured in a tree structure. A solution is to simply add an edge between the *its* node and the *state* node that forces the two nodes to have the same value. The result is still a well-defined CSP, though not a tree-structured one. The situation would become trickier if we were to integrate the other relations (aggregate, mark, and execute). We might be able to incorporate some ideas from Hybrid Logic Dependency Semantics (Baldridge and Kruijff 2002; White 2006), given that hybrid logic extends the tree structures of modal logic with *nominals*, thereby allowing a node to freely reference other nodes. In this article, however, we will stick to trees and leave the full exploration of non-trees for future work.

2.4.2 Computation of Join Relations. So far, we have given a declarative definition of the denotation $[\![z]\!]_w$ of a DCS tree z with only join relations. Now we will show how to compute $[\![z]\!]_w$ efficiently. Recall that the denotation is the set of feasible values for the root node. In general, finding the solution to a CSP is NP-hard, but for trees, we can exploit dynamic programming (Dechter 2003). The key is that the denotation of a tree depends on its subtrees only through their denotations:

$$[\![\left\langle p; {}^{j_1}_{j'_1}\!:c_1; \cdots; {}^{j_m}_{j'_m}\!:c_m \right\rangle]\!]_w = w(p) \cap \bigcap_{i=1}^{m} \{v : v_{j_i} = t_{j'_i}, t \in [\![c_i]\!]_w\} \tag{12}$$

On the right-hand side of Equation (12), the first term $w(p)$ is the set of values that satisfy the node constraint, and the second term consists of an intersection across all m edges of $\{v : v_{j_i} = t_{j'_i}, t \in [\![c_i]\!]_w\}$, which is the set of values v which satisfy the edge constraint with respect to some value t for the child c_i.

To further flesh out this computation, we express Equation (12) in terms of two operations: **join** and **project**. Join takes a cross product of two sets of tuples and retains the resulting tuples that match the join constraint:

$$A \bowtie_{j,j'} B = \{u + v : u \in A, v \in B, u_j = v_{j'}\} \tag{13}$$

Project takes a set of tuples and retains a fixed subset of the components:

$$A[\mathbf{i}] = \{v_{\mathbf{i}} : v \in A\} \tag{14}$$

The denotation in Equation (12) can now be expressed in terms of these join and project operations:

$$[\![\left\langle p; {}^{j_1}_{j'_1}\!:c_1; \cdots; {}^{j_m}_{j'_m}\!:c_m \right\rangle]\!]_w = ((w(p) \bowtie_{j_1,j'_1} [\![c_1]\!]_w)[\mathbf{i}] \cdots \bowtie_{j_m,j'_m} [\![c_m]\!]_w)[\mathbf{i}] \tag{15}$$

4 DRT started the dynamic semantics tradition where meanings are context-change potentials, a natural way to capture anaphora. The DCS formalism presented here does not deal with anaphora, so we give it a purely static semantics.

where $\mathbf{i} = (1, \ldots, \text{ARITY}_w(p))$. Projecting onto $\mathbf{i}$ retains only components corresponding to p.

The time complexity for computing the denotation of a DCS tree $[\![z]\!]_w$ scales linearly with the number of nodes, but there is also a dependence on the cost of performing the join and project operations. For details on how we optimize these operations and handle infinite sets of tuples (for predicates such as `count`), see Liang (2011).

The denotation of DCS trees is defined in terms of the feasible values of a CSP, and the recurrence in Equation (15) is only one way of computing this denotation. In light of the extensions to come, however, we now consider Equation (15) as the actual definition rather than just a computational mechanism. It will still be useful to refer to the CSP in order to access the intuition of using declarative constraints.

2.4.3 Aggregate Relation. Thus far, we have focused on DCS trees that only use join relations, which are insufficient for capturing higher-order phenomena in language. For example, consider the phrase *number of major cities*. Suppose that *number* corresponds to the `count` predicate, and that *major cities* maps to the DCS tree $\langle \texttt{city}; \frac{1}{1}: \langle \texttt{major} \rangle \rangle$. We cannot simply join `count` with the root of this DCS tree because `count` needs to be joined with the *set* of major cities (the denotation of $\langle \texttt{city}; \frac{1}{1}: \langle \texttt{major} \rangle \rangle$), not just a single city.

We therefore introduce the **aggregate relation** (Σ) that takes a DCS subtree and reifies its denotation so that it can be accessed by other nodes in its entirety. Consider a tree $\langle \text{ø}; \Sigma : c \rangle$, where the root is connected to a child c via Σ. The denotation of the root is simply the singleton set containing the denotation of c:

$$[\![\langle \text{ø}; \Sigma : c \rangle]\!]_w = \{([\![c]\!]_w)\} \tag{16}$$

Figure 5(a) shows the DCS tree for our running example. The denotation of the middle node is $\{(s)\}$, where s is all major cities. Everything above this node is an ordinary CSP: s constrains the `count` node, which in turns constrains the root node to $|s|$. Figure 5(b) shows another example of using the aggregate relation Σ. Here, the node right above Σ is constrained to be a set of pairs of major cities and their populations. The `average` predicate then computes the desired answer.

To represent logical disjunction in natural language, we use the aggregate relation and two predicates, `union` and `contains`, which are defined in the expected way:

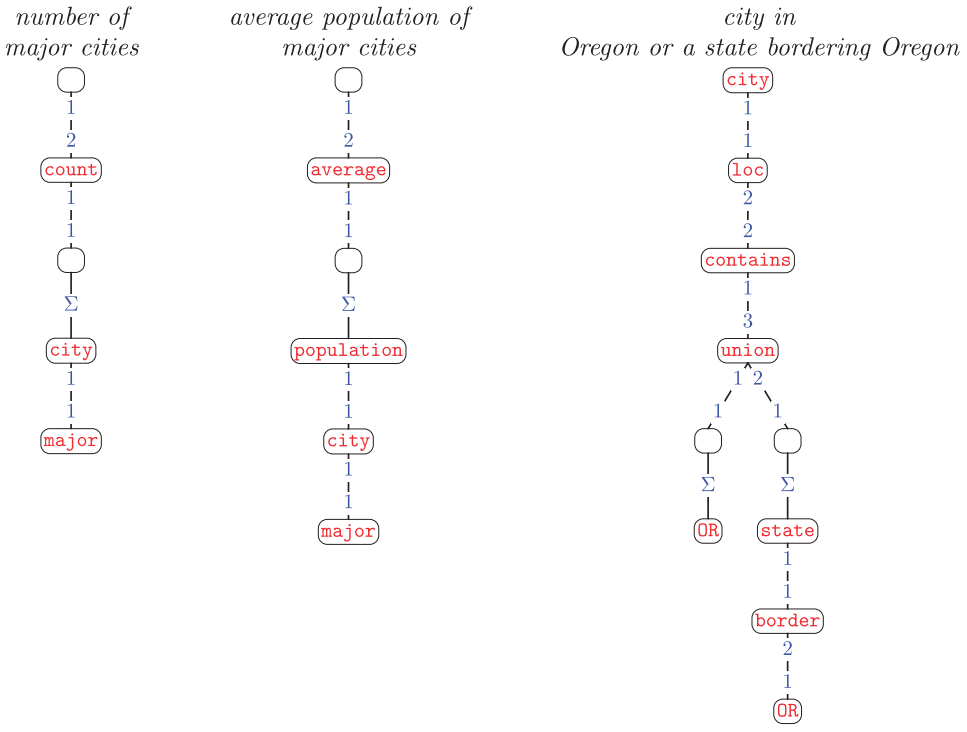
$$w(\texttt{union}) = \{(A, B, C) : C = A \cup B, A \in \mathcal{V}_{\{\star\}}, B \in \mathcal{V}_{\{\star\}}\} \tag{17}$$

$$w(\texttt{contains}) = \{(A, x) : x \in A, A \in \mathcal{V}_{\{\star\}}\} \tag{18}$$

where $A, B, C \in \mathcal{V}_{\{\star\}}$ are sets of primitive values (see Section 2.3.1). Figure 5(c) shows an example of a disjunctive construction: We use the aggregate relations to construct two sets, one containing Oregon, and the other containing states bordering Oregon. We take the union of these two sets; `contains` takes the set and reads out an element, which then constrains the `city` node.

Remarks. 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 the recurrences in Equations (15) and (16).

Recall that a DCS tree with only join relations is a DRS without nested DRSs. The aggregate relation corresponds to the abstraction operator in DRT and is one way of



(a) Count (b) Average (c) Disjunction

Figure 5
Examples of DCS trees that use the aggregate relation (Σ) to (a) compute the cardinality of a set, (b) take the average over a set, (c) represent a disjunction over two conditions. The aggregate relation sets the parent node deterministically to the denotation of the child node. Nodes with the special null predicate ø are represented as empty nodes.

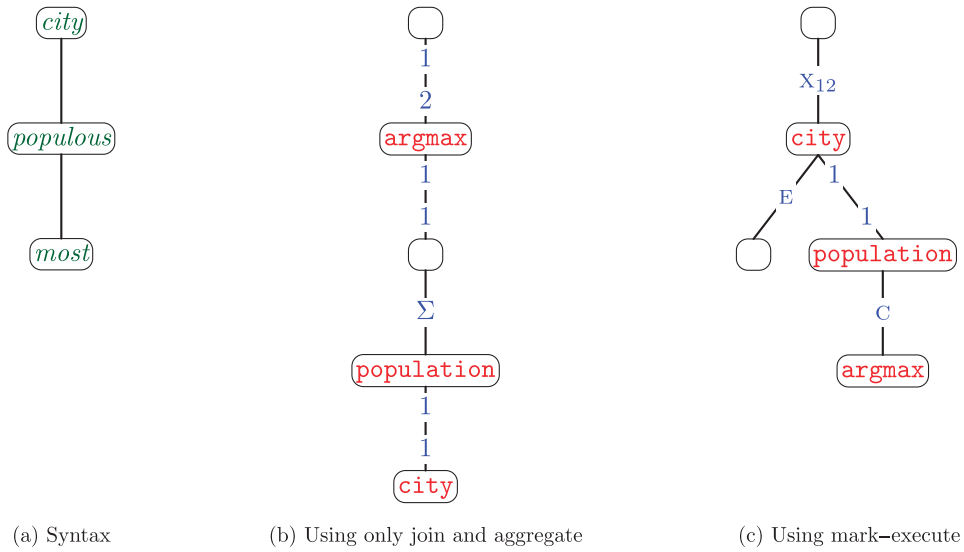
making nested DRSs. It turns out that the abstraction operator is sufficient to obtain the full representational power of DRT, and subsumes generalized quantification and disjunction constructs in DRT. By analogy, we use the aggregate relation to handle disjunction (Figure 5(c)) and generalized quantification (Section 2.5.6).

DCS restricted to join relations is less expressive than first-order logic because it does not have universal quantification, negation, and disjunction. The aggregate relation is analogous to lambda abstraction, and in basic DCS we use the aggregate relation to implement those basic constructs using higher-order predicates such as `not`, `every`, and `union`. We can also express logical statements such as generalized quantification, which go beyond first-order logic.

2.5 Semantics of DCS Trees with Mark–Execute (Full Version)

Basic DCS includes two types of relations, join and aggregate, but it is already quite expressive. In general, however, it is not enough just to be able to express the meaning of a sentence using some logical form; we must be able to derive the logical form compositionally and simply from the sentence.

Consider the superlative construction *most populous city*, which has a basic syntactic dependency structure shown in Figure 6(a). Figure 6(b) shows that we can in principle



(a) Syntax (b) Using only join and aggregate (c) Using mark–execute

Figure 6
Two semantically equivalent DCS trees are shown in (b) and (c). The DCS tree in (b), which uses the join and aggregate relations in the basic DCS, does not align well with the syntactic structure of *most populous city* (a), and thus is undesirable. The DCS tree in (c), by using the mark–execute construct, aligns much better, with `city` rightfully dominating its modifiers. The full version of DCS allows us to construct (c), which is preferable to (b).

already use a DCS tree with only join and aggregate relations to express the correct semantics of the superlative construction. Note, however, that the two structures are quite divergent—the syntactic head is *city* and the semantic head is `argmax`. This divergence runs counter to a principal desideratum of DCS, which is to create a transparent interface between coarse syntax and semantics.

In this section, we introduce mark and execute relations, which will allow us to use the DCS tree in Figure 6(c) to represent the semantics associated with Figure 6(a); these two are more similar than (a) and (b). The focus of this section is on this mark–execute construct—using mark and execute relations to give proper semantically scoped denotations to syntactically scoped tree structures.

The basic intuition of the mark–execute construct is as follows: We mark a node low in the tree with a **mark relation**; then, higher up in the tree, we invoke it with a corresponding **execute relation** (Figure 7). For our example in Figure 6(c), we mark the `population` node, which puts the child `argmax` in a temporary store; when we execute the `city` node, we fetch the superlative predicate `argmax` from the store and invoke it.

This divergence between syntactic and semantic scope arises in other linguistic contexts besides superlatives, such as quantification and negation. In each of these cases, the general template is the same: A syntactic modifier low in the tree needs to have semantic force higher in the tree. A particularly compelling case of this divergence happens with quantifier scope ambiguity (e.g., *Some river traverses every city*[5]), where the

5 The two meanings are: (i) there is a river x such that x traverses every city; and (ii) for every city x, some river traverses x.

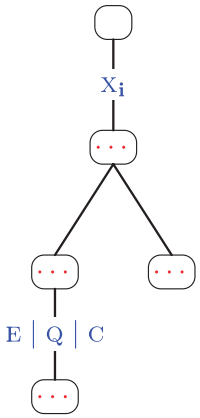


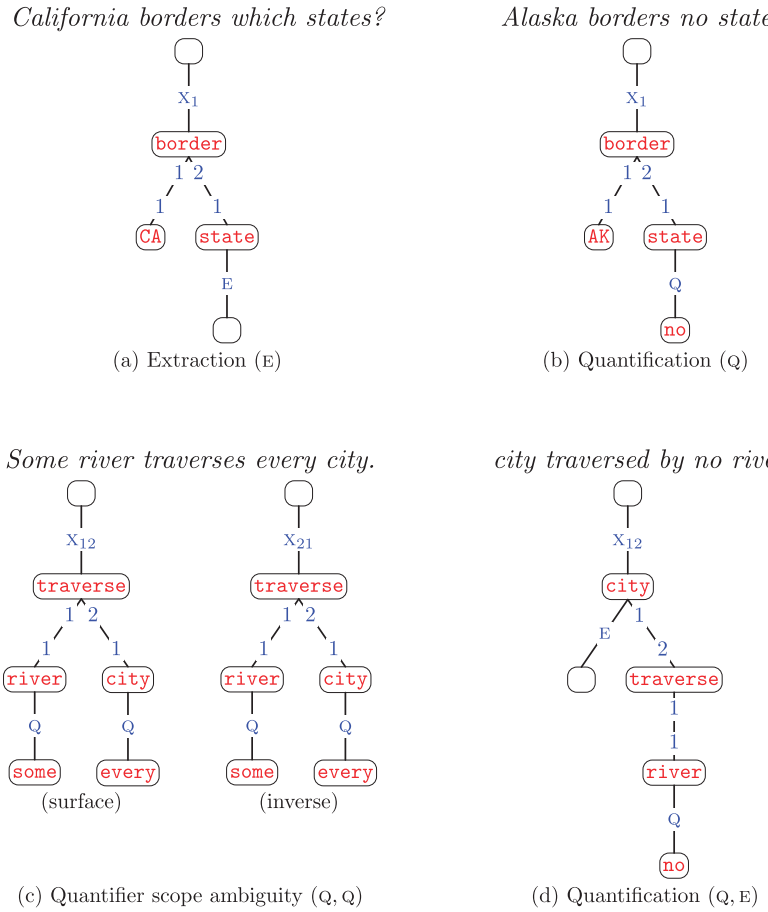
Figure 7
The template for the mark–execute construct. A mark relation (one of E, Q, C) "stores" the modifier. Then an execute relation (of the form X_i for indices **i**) higher up "recalls" the modifier and applies it at the desired semantic point.

quantifiers appear in fixed syntactic positions, but the surface and inverse scope readings correspond to different semantically scoped denotations. Analogously, a single syntactic structure involving superlatives can also yield two different semantically scoped denotations—the absolute and relative readings (e.g., *state bordering the largest state*[6]). The mark–execute construct provides a unified framework for dealing all these forms of divergence between syntactic and semantic scope. See Figures 8 and 9 for concrete examples of this construct.

2.5.1 Denotations. We now formalize the mark–execute construct. We saw that the mark–execute construct appears to act non-locally, putting things in a store and retrieving them later. This means that if we want the denotation of a DCS tree to only depend on the denotations of its subtrees, the denotations need to contain more than the set of feasible values for the root node, as was the case for basic DCS. We need to augment denotations to include information about all marked nodes, because these can be accessed by an execute relation higher up in the tree.

More specifically, let z be a DCS tree and $d = [\![z]\!]_w$ be its denotation. The denotation d consists of n **columns**. The first column always corresponds to the root node of z, and the rest of the columns correspond to non-root marked nodes in z. In the example in Figure 10, there are two columns, one for the root `state` node and the other for `size` node, which is marked by C. The columns are ordered according to a pre-order traversal of z, so column 1 always corresponds to the root node. The denotation d contains a set of arrays $d.A$, where each array represents a feasible assignment of values to the columns of d; note that we quantify over non-marked nodes, so they do not correspond to any column in the denotation. For example, in Figure 10, the first array in $d.A$ corresponds to assigning (`OK`) to the `state` node (column 1) and (`TX`, 2.7e5) to the `size` node (column 2). If there are no marked nodes, $d.A$ is basically a set of tuples, which corresponds to a denotation in basic DCS. For each marked node, the denotation d also maintains a **store**

6 The two meanings are: (i) a state that borders Alaska (which is the largest state); and (ii) a state with the highest score, where the score of a state x is the maximum size of any state that x borders (Alaska is irrelevant here because no states border it).



(a) Extraction (E) (b) Quantification (Q)

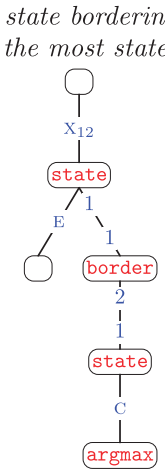
(c) Quantifier scope ambiguity (Q, Q) (d) Quantification (Q, E)

Figure 8
Examples of DCS trees that use the mark–execute construct with the E and Q mark relations. (a) The head verb *borders*, which needs to be returned, has a direct object *states* modified by *which*. (b) The quantifier *no* is syntactically dominated by *state* but needs to take wider scope. (c) Two quantifiers yield two possible readings; we build the same basic structure, marking both quantifiers; the choice of execute relation (X_{12} versus X_{21}) determines the reading. (d) We use two mark relations, Q on `river` for the negation, and E on `city` to force the quantifier to be computed for each value of `city`.

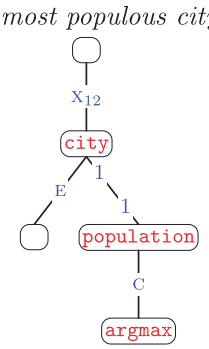
with information to be retrieved when that marked node is executed. A store σ for a marked node contains the following: (i) the mark relation $\sigma.r$ (C in the example), (ii) the base denotation $\sigma.b$, which essentially corresponds to denotation of the subtree rooted at the marked node excluding the mark relation and its subtree ($[\![\langle \texttt{size} \rangle]\!]_w$ in the example), and (iii) the denotation of the child of the mark relation ($[\![\langle \texttt{argmax} \rangle]\!]_w$ in the example). The store of any unmarked nodes is always empty ($\sigma = \emptyset$).

Definition 3 (Denotations)
Let $\mathcal{D}$ be the set of **denotations**, where each denotation $d \in \mathcal{D}$ consists of

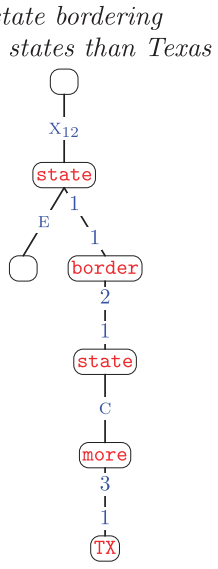
- a set of arrays $d.A$, where each array $\mathbf{a} = [a_1, \ldots, a_n] \in d.A$ is a sequence of n tuples for some $n \geq 0$; and



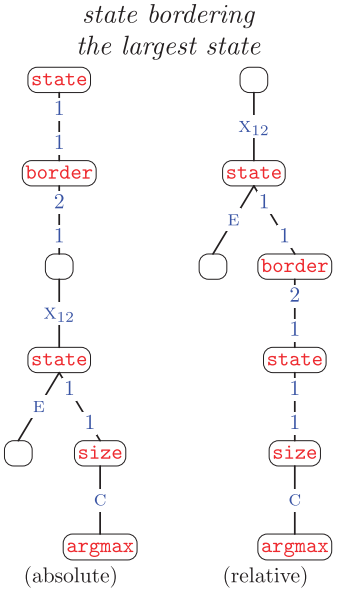
(a) Superlative (C)



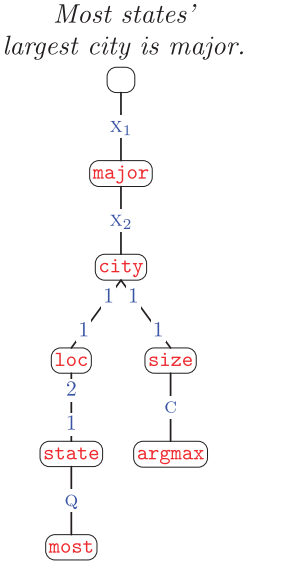
(b) Superlative (C)



(c) Comparative (C)



(d) Superlative scope ambiguity (C)



(e) Quantification+Superlative (Q, C)

Figure 9
Examples of DCS trees that use the mark–execute construct with the E and C relation. (a,b,c) Comparatives and superlatives are handled as follows: For each value of the node marked by E, we compute a number based on the node marked by C; based on this information, a subset of the values is selected as the possible values of the root node. (d) Analog of quantifier scope ambiguity for superlatives: The placement of the execute relation determines an absolute versus relative reading. (e) Interaction between a quantifier and a superlative: The lower execute relation computes the largest city for each state; the second execute relation invokes `most` and enforces that the `major` constraint holds for the majority of states.

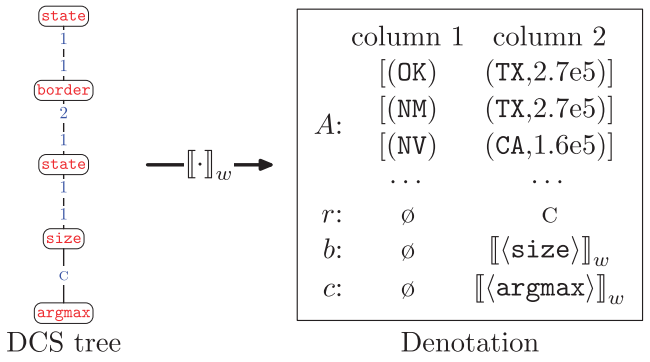


Figure 10
Example of the denotation for a DCS tree (with the compare relation C). This denotation has two columns, one for each active node—the root node `state` and the marked node `size`.

- a sequence of n *stores* $d.\sigma = (d.\sigma_1, \ldots, d.\sigma_n)$, where each store σ contains a mark relation $\sigma.r \in \{\text{E}, \text{Q}, \text{C}, \text{ø}\}$, a base denotation $\sigma.b \in \mathcal{D} \cup \{\text{ø}\}$, and a child denotation $\sigma.c \in \mathcal{D} \cup \{\text{ø}\}$.

Note that denotations are formally defined without reference to DCS trees (just as sets of tuples were in basic DCS), but it is sometimes useful to refer to the DCS tree that generates that denotation.

For notational convenience, we write d as $\langle\!\langle A; (r_1, b_1, c_1); \ldots; (r_n, b_n, c_n) \rangle\!\rangle$. Also let $d.r_i = d.\sigma_i.r$, $d.b_i = d.\sigma_i.b$, and $d.c_i = d.\sigma_i.c$. Let $d\{\sigma_i = x\}$ be the denotation which is identical to d, except with $d.\sigma_i = x$; $d\{r_i = x\}$, $d\{b_i = x\}$, and $d\{c_i = x\}$ are defined analogously. We also define a project operation for denotations: $\langle\!\langle A; \sigma \rangle\!\rangle[\mathbf{i}] \stackrel{\text{def}}{=} \langle\!\langle \{\mathbf{a_i} : \mathbf{a} \in A\}; \sigma_\mathbf{i} \rangle\!\rangle$. Extending this notation further, we use ø to denote the indices of the *non-initial columns with empty stores* ($i > 1$ such that $d.\sigma_i = \text{ø}$). We can then use $d[-\text{ø}]$ to represent projecting away the non-initial columns with empty stores. For the denotation d in Figure 10, $d[1]$ keeps column 1, $d[-\text{ø}]$ keeps both columns, and $d[2, -2]$ swaps the two columns.

In basic DCS, denotations are sets of tuples, which works quite well for representing the semantics of *wh*-questions such as *What states border Texas?* But what about polar questions such as *Does Louisiana border Texas?* The denotation should be a simple Boolean value, which basic DCS does not represent explicitly. Using our new denotations, we can represent Boolean values explicitly using zero-column structures: **true** corresponds to a singleton set containing just the empty array ($d_\text{T} = \langle\!\langle \{[\,]\} \rangle\!\rangle$) and **false** is the empty set ($d_\text{F} = \langle\!\langle \emptyset \rangle\!\rangle$).

Having described denotations as n-column structures, we now give the formal mapping from DCS trees to these structures. As in basic DCS, this mapping is defined recursively over the structure of the tree. We have a recurrence for each case (the first line is the base case, and each of the others handles a different edge relation):

$$[\![\langle p \rangle]\!]_w = \langle\!\langle \{[v] : v \in w(p)\}; \text{ø} \rangle\!\rangle \quad \text{[base case]} \tag{19}$$

$$[\![\left\langle p; \mathbf{e}; \tfrac{j}{j'} : c \right\rangle]\!]_w = [\![\langle p; \mathbf{e} \rangle]\!]_w \bowtie^{-\text{ø}}_{j,j'} [\![c]\!]_w \quad \text{[join]} \tag{20}$$

$$[\![\langle p; \mathbf{e}; \Sigma : c \rangle]\!]_w = [\![\langle p; \mathbf{e} \rangle]\!]_w \bowtie^{-\text{ø}}_{*,*} \Sigma\left([\![c]\!]_w\right) \quad \text{[aggregate]} \tag{21}$$

$$[\![\langle p; \mathbf{e}; \mathrm{X}_{\mathbf{i}} : c \rangle]\!]_w = [\![\langle p; \mathbf{e} \rangle]\!]_w \bowtie^{-\emptyset}_{*,*} \mathbf{x}_{\mathbf{i}}([\![c]\!]_w) \quad \text{[execute]} \tag{22}$$

$$[\![\langle p; \mathbf{e}; \mathrm{E} : c \rangle]\!]_w = \mathbf{M}([\![\langle p; \mathbf{e} \rangle]\!]_w, \mathrm{E}, [\![c]\!]_w) \quad \text{[extract]} \tag{23}$$

$$[\![\langle p; \mathbf{e}; \mathrm{C} : c \rangle]\!]_w = \mathbf{M}([\![\langle p; \mathbf{e} \rangle]\!]_w, \mathrm{C}, [\![c]\!]_w) \quad \text{[compare]} \tag{24}$$

$$[\![\langle p; \mathrm{Q} : c; \mathbf{e} \rangle]\!]_w = \mathbf{M}([\![\langle p; \mathbf{e} \rangle]\!]_w, \mathrm{Q}, [\![c]\!]_w) \quad \text{[quantify]} \tag{25}$$

We define the operations $\bowtie^{-\emptyset}_{\mathbf{j},\mathbf{j}'}$, Σ, $\mathbf{X_i}$, and $\mathbf{M}$ in the remainder of this section.

2.5.2 Base Case. Equation (19) defines the denotation for a DCS tree z with a single node with predicate p. The denotation of z has one column whose arrays correspond to the tuples $w(p)$; the store for that column is empty.

2.5.3 Join Relations. Equation (20) defines the recurrence for join relations. On the left-hand side, $\left\langle p; \mathbf{e}; \frac{j}{j'} : c \right\rangle$ is a DCS tree with p at the root, a sequence of edges $\mathbf{e}$ followed by a final edge with relation $\frac{j}{j'}$ connected to a child DCS tree c. On the right-hand side, we take the recursively computed denotation of $\langle p; \mathbf{e} \rangle$, the DCS tree without the final edge, and perform a **join-project-inactive** operation (notated $\bowtie^{-\emptyset}_{j,j'}$) with the denotation of the child DCS tree c.

The join-project-inactive operation joins the arrays of the two denotations (this is the core of the join operation in basic DCS—see Equation (13)), and then projects away the non-initial empty columns:[7]

$$\langle\!\langle A; \sigma \rangle\!\rangle \bowtie^{-\emptyset}_{\mathbf{j},\mathbf{j}'} \langle\!\langle A'; \sigma' \rangle\!\rangle = \langle\!\langle A''; \sigma + \sigma' \rangle\!\rangle [-\emptyset], \text{where} \tag{26}$$

$$A'' = \{\mathbf{a} + \mathbf{a}' : \mathbf{a} \in A, \mathbf{a}' \in A', a_{1\mathbf{j}} = a'_{1\mathbf{j}'}\}$$

We concatenate all arrays $\mathbf{a} \in A$ with all arrays $\mathbf{a}' \in A'$ that satisfy the join condition $a_{1\mathbf{j}} = a'_{1\mathbf{j}'}$. The sequences of stores are simply concatenated: $(\sigma + \sigma')$. Finally, any non-initial columns with empty stores are projected away by applying $\cdot[-\emptyset]$.

Note that the join works on column 1; the other columns are carried along for the ride. As another piece of convenient notation, we use $*$ to represent all components, so $\bowtie^{-\emptyset}_{*,*}$ imposes the join condition that the entire tuple has to agree ($a_1 = a'_1$).

2.5.4 Aggregate Relations. Equation (21) defines the recurrence for aggregate relations. Recall that in basic DCS, aggregate (16) simply takes the denotation (a set of tuples) and puts it into a set. Now, the denotation is not just a set, so we need to generalize this operation. Specifically, the aggregate operation applied to a denotation forms a set out of the tuples in the first column for each setting of the rest of the columns:

$$\Sigma(\langle\!\langle A; \sigma \rangle\!\rangle) = \langle\!\langle A' \cup A''; \sigma \rangle\!\rangle \tag{27}$$

$$A' = \{[S(\mathbf{a}), a_2, \ldots, a_n] : \mathbf{a} \in A\}$$

$$S(\mathbf{a}) = \{a'_1 : [a'_1, a_2, \ldots, a_n] \in A\}$$

$$A'' = \{[\emptyset, a_2, \ldots, a_n] : \forall i \in \{2, \ldots, n\}, [a_i] \in \sigma_i.b.A[1], \neg\exists a_1, \mathbf{a} \in A\}$$

7 The join and project operations are taken from relational algebra.

The aggregate operation takes the set of arrays A and produces two sets of arrays, A' and A'', which are unioned (note that the stores do not change). The set A' is the one that first comes to mind: For every setting of $a_2, \ldots, a_n$, we construct $S(\mathbf{a})$, the set of tuples a_1' in the first column which co-occur with $a_2, \ldots, a_n$ in A.

There is another case, however: what happens to settings of $a_2, \ldots, a_n$ that do not co-occur with any value of a_1' in A? Then, $S(\mathbf{a}) = \emptyset$, but note that A' by construction will not have the desired array $[\emptyset, a_2, \ldots, a_n]$. As a concrete example, suppose $A = \emptyset$ and we have one column ($n = 1$). Then $A' = \emptyset$, rather than the desired $\{[\emptyset]\}$.

Fixing this problem is slightly tricky. There are an infinite number of $a_2, \ldots, a_n$ which do not co-occur with any a_1' in A, so for which ones do we actually include $[\emptyset, a_2, \ldots, a_n]$? Certainly, the answer to this question cannot come from A, so it must come from the stores. In particular, for each column $i \in \{2, \ldots, n\}$, we have conveniently stored a base denotation $\sigma_i.b$. We consider any a_i that occurs in column 1 of the arrays of this base denotation ($[a_i] \in \sigma_i.b.A[1]$). For this $a_2, \ldots, a_n$, we include $[\emptyset, a_2, \ldots, a_n]$ in A'' as long as $a_2, \ldots, a_n$ does not co-occur with any a_1. An example is given in Figure 11.

The reason for storing base denotations is thus partially revealed: The arrays represent feasible values of a CSP and can only contain positive information. When we aggregate, we need to access possibly empty sets of feasible values—a kind of negative information, which can only be recovered from the base denotations.

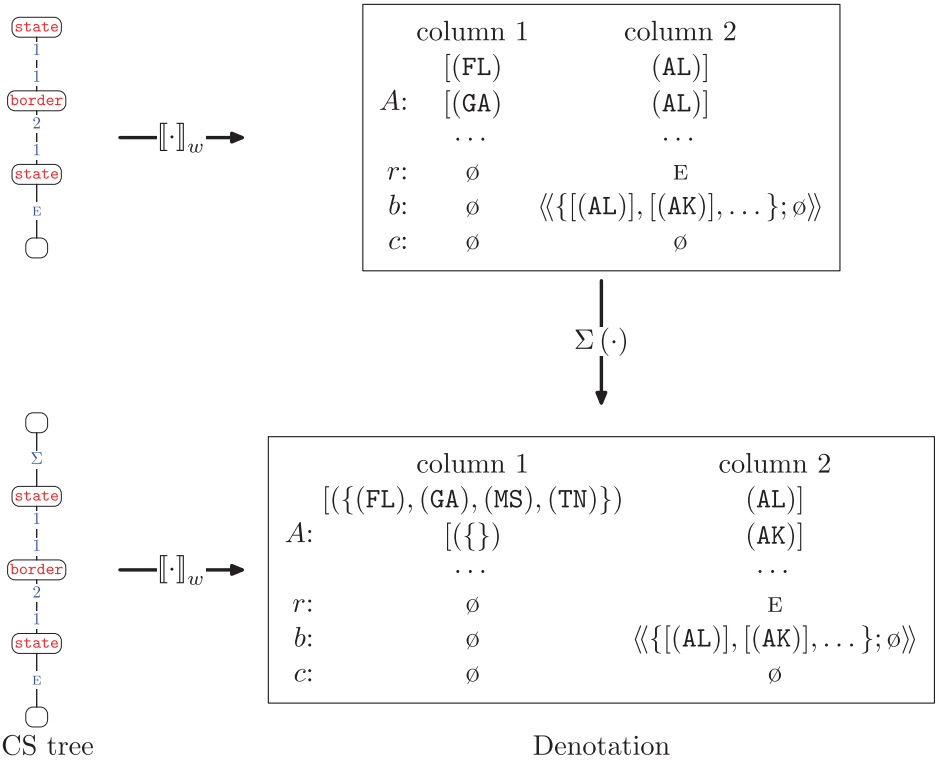


Figure 11
An example of applying the aggregate operation, which takes a denotation and aggregates the values in column 1 for every setting of the other columns. The base denotations (b) are used to put in $\{\}$ for values that do not appear in A (in this example, AK, corresponding to the fact that Alaska does not border any states).

2.5.5 Mark Relations. Equations (23), (24), and (25) each processes a different mark relation. We define a general mark operation, $\mathbf{M}(d, r, c)$ which takes a denotation d, a mark relation $r \in \{\text{E}, \text{Q}, \text{C}\}$ and a child denotation c, and sets the store of d in column 1 to be (r, d, c):

$$\mathbf{M}(d, r, c) = d\{r_1 = r, b_1 = d, c_1 = c\} \tag{28}$$

The base denotation of the first column b_1 is set to the current denotation d. This, in some sense, creates a snapshot of the current denotation. Figure 12 shows an example of the mark operation.

2.5.6 Execute Relations. Equation (22) defines the denotation of a DCS tree where the last edge of the root is an execute relation. Similar to the aggregate case (21), we recurse on the DCS tree without the last edge ($\langle p; \mathbf{e} \rangle$) and then join it to the result of applying the execute operation $\mathbf{X_i}$ to the denotation of the child ($[\![c]\!]_w$).

The execute operation $\mathbf{X_i}$ is the most intricate part of DCS and is what does the heavy lifting. The operation is parametrized by a sequence of distinct indices $\mathbf{i}$ that specifies the order in which the columns should be processed. Specifically, $\mathbf{i}$ indexes into the subsequence of columns with non-empty stores. We then process this subsequence of columns in reverse order, where processing a column means performing some operations depending on the stored relation in that column. For example, suppose that columns 2 and 3 are the only non-empty columns. Then $\mathbf{X}_{12}$ processes column 3 before column 2. On the other hand, $\mathbf{X}_{21}$ processes column 2 before column 3. We first define

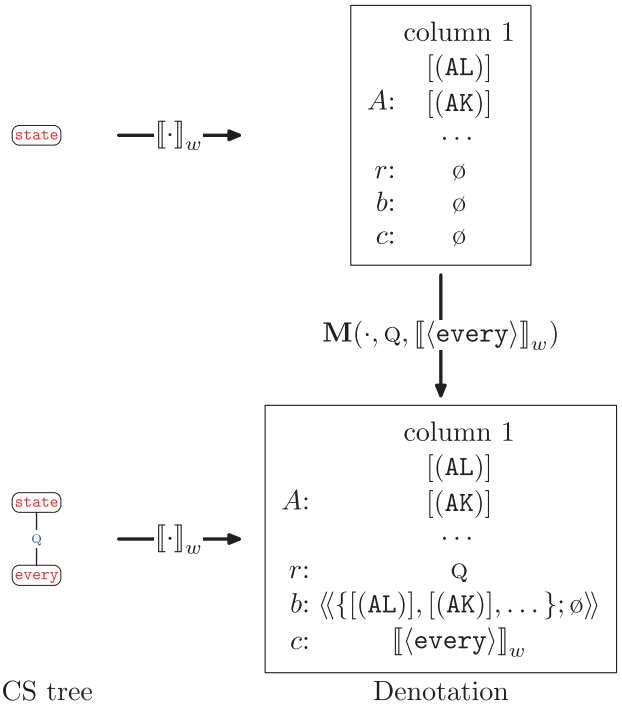


Figure 12
An example of applying the mark operation, which takes a denotation and modifies the store of the column 1. This information is used by other operations such as aggregate and execute.

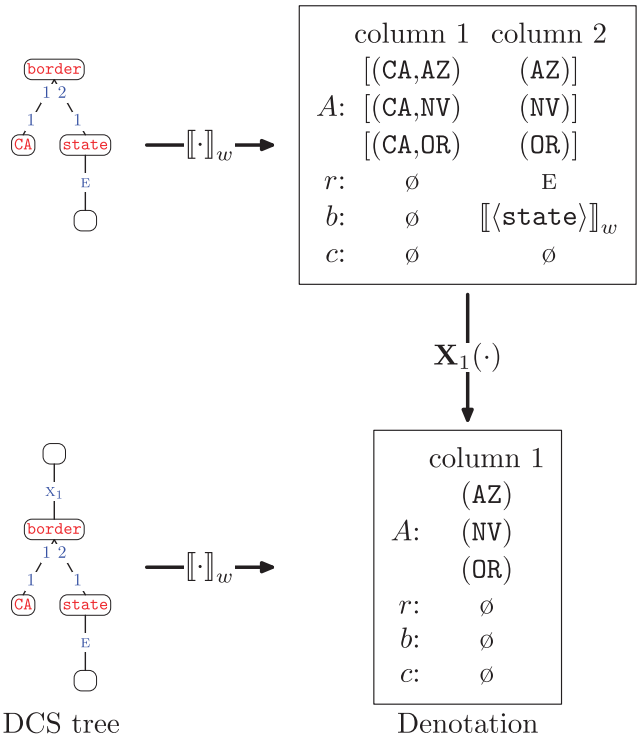


Figure 13
An example of applying the execute operation on column 1 with the extract relation E. The denotation prior to execution consists of two columns: column 1 corresponds to the `border` node; column 2 to the `state` node. The join relations and predicates `CA` and `state` constrain the arrays A in the denotation to include only the states that border California. After execution, the non-marked column 1 is projected away, leaving only the `state` column with its store emptied.

the execute operation $\mathbf{X}_i$ for a single column i. There are three distinct cases, depending on the relation stored in column i:

Extraction. For a denotation d with the extract relation E in column i, executing $\mathbf{X}_i(d)$ involves three steps: (i) moving column i to before column 1 ($\cdot[i,-i]$), (ii) projecting away non-initial empty columns ($\cdot[-\emptyset]$), and (iii) removing the store ($\cdot\{\sigma_1 = \emptyset\}$):

$$\mathbf{X}_i(d) = d[i,-i][-\emptyset]\{\sigma_1 = \emptyset\} \quad \text{if } d.r_i = \text{E} \tag{29}$$

An example is given in Figure 13. There are two main uses of extraction.

1. By default, the denotation of a DCS tree is the set of feasible values of the root node (which occupies column 1). To return the set of feasible values of another node, we mark that node with E. Upon execution, the feasible values of that node move into column 1. Extraction can be used to handle in situ questions (see Figure 8(a)).
2. Unmarked nodes (those that do not have an edge with a mark relation) are existentially quantified and have narrower scope than all marked nodes. Therefore, we can make a node x have wider scope than another node y by

marking x (with E) and executing y before x (see Figure 8(d,e) for examples). The extract relation E (in fact, any mark relation) signifies that we want to control the scope of a node, and the execute relation allows us to set that scope.

Generalized Quantification. Generalized quantifiers are predicates on two sets, a **restrictor** A and a **nuclear scope** B. For example,

$$w(\texttt{some}) = \{(A, B) : A \cap B > 0\} \tag{30}$$

$$w(\texttt{every}) = \{(A, B) : A \subset B\} \tag{31}$$

$$w(\texttt{no}) = \{(A, B) : A \cap B = \emptyset\} \tag{32}$$

$$w(\texttt{most}) = \{(A, B) : |A \cap B| > \frac{1}{2}|A|\} \tag{33}$$

We think of the quantifier as a modifier which always appears as the child of a Q relation; the restrictor is the parent. For example, in Figure 8(b), `no` corresponds to the quantifier and `state` corresponds to the restrictor. The nuclear scope should be the set of all states that Alaska borders. More generally, the nuclear scope is the set of feasible values of the restrictor node with respect to the CSP that includes all nodes between the mark and execute relations. The restrictor is also the set of feasible values of the restrictor node, but with respect to the CSP corresponding to the subtree rooted at that node.[8]

We implement generalized quantifiers as follows: Let d be a denotation and suppose we are executing column i. We first construct a denotation for the restrictor d_A and a denotation for the nuclear scope d_B. For the restrictor, we take the base denotation in column i ($d.b_i$)—remember that the base denotation represents a snapshot of the restrictor node before the nuclear scope constraints are added. For the nuclear scope, we take the complete denotation d (which includes the nuclear scope constraints) and extract column i ($d[i, -i][-\text{ø}]\{\sigma_1 = \text{ø}\}$—see (29)). We then construct d_A and d_B by applying the aggregate operation to each. Finally, we join these sets with the quantifier denotation, stored in $d.c_i$:

$$\mathbf{x}_i(d) = \left(\left(d.c_i \bowtie_{1,1}^{-\text{ø}} d_A\right) \bowtie_{2,1}^{-\text{ø}} d_B\right)[-1] \quad \text{if } d.r_i = \text{Q}, \text{where} \tag{34}$$

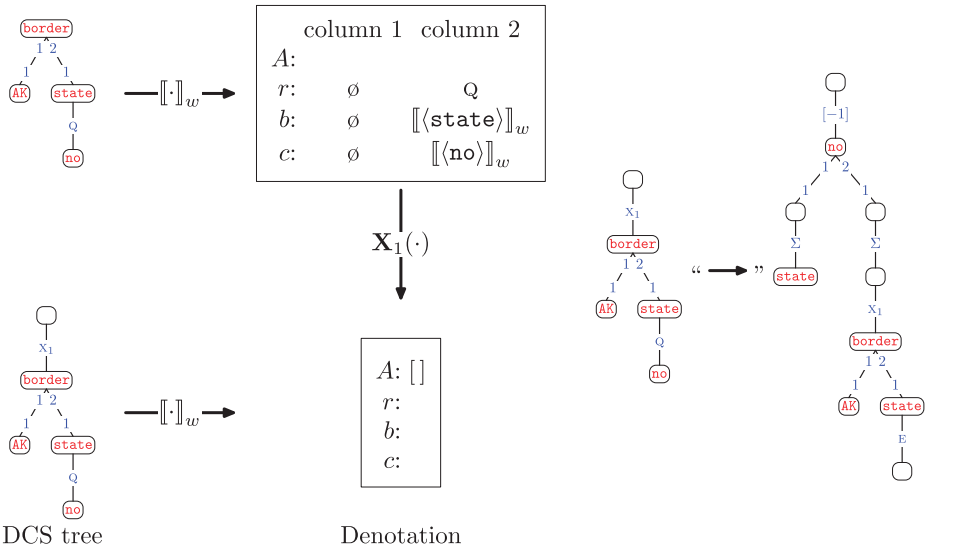
$$d_A = \Sigma\,(d.b_i) \tag{35}$$

$$d_B = \Sigma\,(d[i, -i][-\text{ø}]\{\sigma_1 = \text{ø}\}) \tag{36}$$

When there is one quantifier, think of the execute relation as performing a syntactic rewriting operation, as shown in Figure 14(b). For more complex cases, we must defer to (34).

Figure 8(c) shows an example with two interacting quantifiers. The denotation of the DCS tree before execution is the same in both readings, as shown in Figure 15. The

8 Defined this way, we can only handle conservative quantifiers, because the nuclear scope will always be a subset of the restrictor. This design decision is inspired by DRT, where it provides a way of modeling donkey anaphora. We are not treating anaphora in this work, but we can handle it by allowing pronouns in the nuclear scope to create anaphoric edges into nodes in the restrictor. These constraints naturally propagate through the nuclear scope's CSP without affecting the restrictor.



(a) Execute a quantify relation Q (b) Execute “expands the DCS tree”

Figure 14
(a) An example of applying the execute operation on column i with the quantify relation Q. Before executing, note that $A = \{\}$ (because Alaska does not border any states). The restrictor (A) is the set of all states, and the nuclear scope (B) is empty. Because the pair (A, B) does exist in w(no), the final denotation is $\langle\langle\{[\,]\}\rangle\rangle$ (which represents true). (b) Although the execute operation actually works on the denotation, think of it in terms of expanding the DCS tree. We introduce an extra projection relation $[-1]$, which projects away the first column of the child subtree's denotation.

quantifier scope ambiguity is resolved by the choice of execute relation: X_{12} gives the surface scope reading, X_{21} gives the inverse scope reading.

Figure 8(d) shows how extraction and quantification work together. First, the no quantifier is processed for each city, which is an unprocessed marked node. Here, the extract relation is a technical trick to give city wider scope.

Comparatives and Superlatives. Comparative and superlative constructions involve comparing entities, and for this we rely on a set S of entity–degree pairs (x, y), where x is an

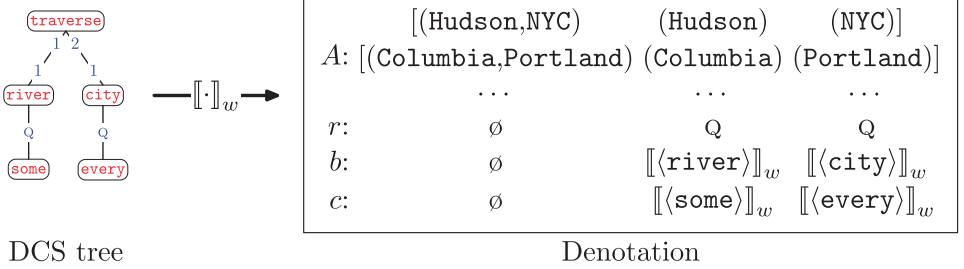


Figure 15
Denotation of Figure 8(c) before the execute relation is applied.

entity and y is a numeric degree. Recall that we can treat S as a function, which maps an entity x to the set of degrees $S(x)$ associated with x. Note that this set can contain multiple degrees. For example, in the relative reading of *state bordering the largest state*, we would have a degree for the size of each neighboring state.

Superlatives use the `argmax` and `argmin` predicates, which are defined in Section 2.3. Comparatives use the `more` and `less` predicates: $w(\texttt{more})$ contains triples (S, x, y), where x is "more than" y as measured by S; $w(\texttt{less})$ is defined analogously:

$$w(\texttt{more}) = \{(S, x, y) : \max S(x) > \max S(y)\} \tag{37}$$

$$w(\texttt{less}) = \{(S, x, y) : \min S(x) < \min S(y)\} \tag{38}$$

We use the same mark relation C for both comparative and superlative constructions. In terms of the DCS tree, there are three key parts: (i) the root x, which corresponds to the entity to be compared, (ii) the child c of a C relation, which corresponds to the comparative or superlative predicate, and (iii) c's parent p, which contains the "degree information" (which will be described later) used for comparison. We assume that the root is marked (usually with a relation E). This forces us to compute a comparison degree for each value of the root node. In terms of the denotation d corresponding to the DCS tree prior to execution, the entity to be compared occurs in column 1 of the arrays $d.A$, the degree information occurs in column i of the arrays $d.A$, and the denotation of the comparative or superlative predicate itself is the child denotation at column i ($d.c_i$).

First, we define a concatenating function $+_{\mathbf{i}}(d)$, which combines the columns $\mathbf{i}$ of d by concatenating the corresponding tuples of each array in $d.A$:

$$+_{\mathbf{i}}(\langle\!\langle A; \sigma \rangle\!\rangle) = \langle\!\langle A'; \sigma' \rangle\!\rangle, \text{where} \tag{39}$$

$$A' = \{\mathbf{a}_{(1\ldots i_1)\backslash\mathbf{i}} + [a_{i_1} + \cdots + a_{i_{|\mathbf{i}|}}] + \mathbf{a}_{(i_1\ldots n)\backslash\mathbf{i}} : \mathbf{a} \in A\}$$

$$\sigma' = \sigma_{(1\ldots i_1)\backslash\mathbf{i}} + [\sigma_{i_1}] + \sigma_{(i_1\ldots n)\backslash\mathbf{i}}$$

Note that the store of column i_1 is kept and the others are discarded. As an example:

$$+_{2,1}(\langle\!\langle \{[(1), (2), (3)], [(4), (5), (6)]\}; \sigma_1, \sigma_2, \sigma_3 \rangle\!\rangle) = \langle\!\langle \{[(2, 1), (3)], [(5, 4), (6)]\}; \sigma_2, \sigma_3 \rangle\!\rangle \tag{40}$$

We first create a denotation d' where column i, which contains the degree information, is extracted to column 1 (and thus column 2 corresponds to the entity to be compared). Next, we create a denotation d_S whose column 1 contains a set of entity-degree pairs. There are two types of degree information:

1. Suppose the degree information has arity 2 ($\text{ARITY}(d.A[i]) = 2$). This occurs, for example, in *most populous city* (see Figure 9(b)), where column i is the `population` node. In this case, we simply set the degree to the second component of `population` by projection ($[\![\langle \varnothing \rangle]\!]_w \bowtie^{-\varnothing}_{1,2} d'$). Now columns 1 and 2 contain the degrees and entities, respectively. We concatenate columns 2 and 1 ($+_{2,1}(\cdot)$) and aggregate to produce a denotation d_S which contains the set of entity–degree pairs in column 1.
2. Suppose the degree information has arity 1 ($\text{ARITY}(d.A[i]) = 1$). This occurs, for example, in *state bordering the most states* (see Figure 9(a)), where

column i is the lower marked `state` node. In this case, the degree of an entity from column 2 is the number of different values that column 1 can take. To compute this, aggregate the set of values ($\Sigma\left(d'\right)$) and apply the `count` predicate. Now with the degrees and entities in columns 1 and 2, respectively, we concatenate the columns and aggregate again to obtain d_S.

Having constructed d_S, we simply apply the comparative/superlative predicate which has been patiently waiting in $d.c_i$. Finally, the store of d's column 1 was destroyed by the concatenation operation $+_{2,1}\left(\left(\right)\cdot\right)$, so we must restore it with $\cdot\{\sigma_1 = d.\sigma_1\}$. The complete operation is as follows:

$$\mathbf{x}_i(d) = \left(\left[\!\left[\langle \text{ø}\rangle\right]\!\right]_w \bowtie_{1,2}^{-\text{ø}} \left(d.c_i \bowtie_{1,1}^{-\text{ø}} d_S\right)\right)\{\sigma_1 = d.\sigma_1\} \text{ if } d.\sigma_i = \textsc{c}, d.\sigma_1 \neq \text{ø}, \text{ where} \tag{41}$$

$$d_S = \begin{cases} \Sigma\left(+_{2,1}\left(\left[\!\left[\langle \text{ø}\rangle\right]\!\right]_w \bowtie_{1,2}^{-\text{ø}} d'\right)\right) & \text{if } \textsc{Arity}(d.A[i]) = 2 \\ \Sigma\left(+_{2,1}\left(\left[\!\left[\langle \text{ø}\rangle\right]\!\right]_w \bowtie_{1,2}^{-\text{ø}} \left(\left[\!\left[\langle \texttt{count}\rangle\right]\!\right]_w \bowtie_{1,1}^{-\text{ø}} \Sigma\left(d'\right)\right)\right)\right) & \text{if } \textsc{Arity}(d.A[i]) = 1 \end{cases} \tag{42}$$

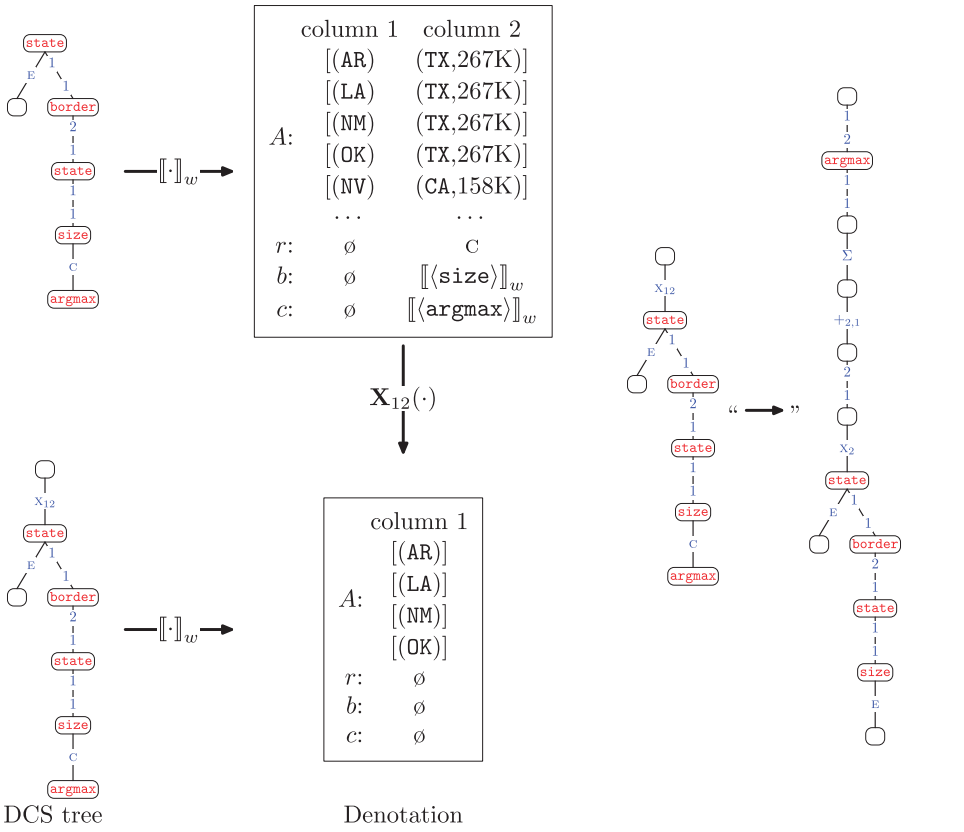
$$d' = d[i, -i][-\text{ø}]\{\sigma_1 = \text{ø}\} \tag{43}$$

An example of executing the C relation is shown in Figure 16(a). As with executing a Q relation, for simple cases we can think of executing a C relation as expanding a DCS tree, as shown in Figure 16(b).

Figure 9(a) and Figure 9(b) show examples of superlative constructions with the arity 1 and arity 2 types of degree information, respectively. Figure 9(c) shows an example of a comparative construction. Comparatives and superlatives use the same machinery, differing only in the predicate: `argmax` versus $\langle \texttt{more}; {}^{3}_{1}\texttt{:TX}\rangle$ (*more than Texas*). But both predicates have the same template behavior: Each takes a set of entity–degree pairs and returns any entity satisfying some property. For `argmax`, the property is obtaining the highest degree; for `more`, it is having a degree higher than a threshold. We can handle generalized superlatives (*the five largest* or *the fifth largest* or *the 5% largest*) as well by swapping in a different predicate; the execution mechanisms defined in Equation (41) remain the same.

We saw that the mark–execute machinery allows decisions regarding quantifier scope to be made in a clean and modular fashion. Superlatives also have scope ambiguities in the form of absolute versus relative readings. Consider the example in Figure 9(d). In the absolute reading, we first compute the superlative in a narrow scope (*the largest state* is Alaska), and then connect it with the rest of the phrase, resulting in the empty set (because no states border Alaska). In the relative reading, we consider the first *state* as the entity we want to compare, and its degree is the size of a neighboring state. In this case, the lower `state` node cannot be set to Alaska because there are no states bordering it. The result is therefore any state that borders Texas (the largest state that does have neighbors). The two DCS trees in Figure 9(d) show that we can naturally account for this form of superlative ambiguity based on where the scope-determining execute relation is placed without drastically changing the underlying tree structure.

Remarks. These scope divergence issues are not specific to DCS—every serious semantic formalism must address them. Generative grammar uses quantifier raising to move the quantifier from its original syntactic position up to the desired semantic position before semantic interpretation even occurs (Heim and Kratzer 1998). Other mechanisms such



(a) Execute a compare relation C (b) Execute "expands the DCS tree"

Figure 16
(a) Executing the compare relation C for an example superlative construction (relative reading of *state bordering the largest state* from Figure 9(d)). Before executing, column 1 contains the entity to compare, and column 2 contains the degree information, of which only the second component is relevant. After executing, the resulting denotation contains a single column with only the entities that obtain the highest degree (in this case, the states that border Texas). (b) For this example, think of the execute operation as expanding the original DCS tree, although the execute operation actually works on the denotation, not the DCS tree. The expanded DCS tree has the same denotation as the original DCS tree, and syntactically captures the essence of the execute–compare operation. Going through the relations of the expanded DCS tree from bottom to top: The X_2 relation swaps columns 1 and 2; the join relation keeps only the second component ((TX, 267*K*) becomes (267*K*)); $+_{2,1}$ concatenates columns 2 and 1 ([(267*K*), (AR)] becomes [(AR, 267*K*)]); Σ aggregates these tuples into a set; argmax operates on this set and returns the elements.

as Montague's (1973) quantifying in, Cooper storage (Cooper 1975), and Carpenter's (1998) scoping constructor handle scope divergence during semantic interpretation. Roughly speaking, these mechanisms delay application of a quantifier, "marking" its spot with a dummy pronoun (as in Montague's quantifying in) or putting it in a store (as in Cooper storage), and then "executing" the quantifier at a later point in the derivation either by performing a variable substitution or retrieving it from the store. Continuation, from programming languages, is another solution (Barker 2002; Shan 2004); this sets the semantics of a quantifier to be a function from its continuation (which captures all the semantic content of the clause minus the quantifier) to the final denotation of the clause.

Intuitively, continuations reverse the normal evaluation order, allowing a quantifier to remain in situ but still outscope the rest of the clause. In fact, the mark and execute relations of DCS are analogous to the shift and reset operators used in continuations. One of the challenges with allowing flexible scope is that free variables can yield invalid scopings, a well-known issue with Cooper storage that the continuation-based approach solves. Invalid scopings are filtered out by the construction mechanism (Section 2.6).

One difference between mark–execute in DCS and many other mechanisms is that DCS trees (which contain mark and execute relations) are the final logical forms—the handling of scope divergence occurs in the computing their denotations. The analog in the other mechanisms resides in the construction mechanism—the actually final logical form is quite simple.[9] Therefore, we have essentially pushed the inevitable complexity from the construction mechanism into the semantics of the logical form. This is a conscious design decision: We want our construction mechanism, which maps natural language to logical form, to be simple and not burdened with complex linguistic issues, for our focus is on learning this mapping. Unfortunately, the denotation of our logical forms (Section 2.5.1) do become more complex than those of lambda calculus expressions, but we believe this is a reasonable tradeoff to make for our particular application.

2.6 Construction Mechanism

We have thus far defined the syntax (Section 2.2) and semantics (Section 2.5) of DCS trees, but we have only vaguely hinted at how these DCS trees might be connected to natural language utterances by appealing to idealized examples. In this section, we formally define the construction mechanism for DCS, which takes an utterance $\mathbf{x}$ and produces a set of DCS trees $\mathcal{Z}_L(\mathbf{x})$.

Because we motivated DCS trees based on dependency syntax, it might be tempting to take a dependency parse tree of the utterance, replace the words with predicates, and attach some relations on the edges to produce a DCS tree. To a first approximation, this is what we will do, but we need to be a bit more flexible for several reasons: (i) some nodes in the DCS tree do not have predicates (e.g., children of an E relation or parent of an X_i relation); (ii) nodes have predicates that do not correspond to words (e.g., in *California cities*, there is a implicit `loc` predicate that bridges `CA` and `city`); (iii) some words might not correspond to any predicates in our world (e.g., *please*); and (iv) the DCS tree might not always be aligned with the syntactic structure depending on which syntactic formalism one ascribes to. Although syntax was the inspiration for the DCS formalism, we will not actually use it in construction.

It is also worth stressing the purpose of the construction mechanism. In linguistics, the purpose of the construction mechanism is to try to generate the exact set of valid logical forms for a sentence. We view the construction mechanism instead as simply a way of creating a set of candidate logical forms. A separate step defines a distribution over this set to favor certain logical forms over others. The construction mechanism should therefore simply overapproximate the set of logical forms. Linguistic constraints that are normally encoded in the construction mechanism (for example, in CCG, that the disharmonic pair S/NP and S\NP cannot be coordinated, or that non-indefinite quantifiers cannot extend their scope beyond clause boundaries) would be instead

9 In the continuation-based approach, this difference corresponds to the difference between assigning a denotational versus an operational semantics.

encoded as features (Section 3.1.1). Because feature weights are estimated from data, one can view our approach as automatically learning the linguistic constraints relevant to our end task.

2.6.1 Lexical Triggers. The construction mechanism assumes a fixed set of **lexical triggers** L. Each trigger is a pair $(\mathbf{s}, p)$, where $\mathbf{s}$ is a sequence of words (usually one) and p is a predicate (e.g., $\mathbf{s} = \textit{California}$ and $p = \texttt{CA}$). We use $L(\mathbf{s})$ to denote the set of predicates p triggered by $\mathbf{s}$ ($(\mathbf{s}, p) \in L$). We should think of the lexical triggers L not as pinning down the precise predicate for each word, but rather as producing an overapproximation. For example, L might contain $\{(\textit{city}, \texttt{city}), (\textit{city}, \texttt{state}), (\textit{city}, \texttt{river}), \ldots\}$, reflecting our initial ignorance prior to learning.

We also define a set of **trace predicates** $L(\epsilon)$, which can be introduced without an overt lexical element. Their name is inspired by trace/null elements in syntax, but they serve a more practical rather than a theoretical role here. As we shall see in Section 2.6.2, trace predicates provide more flexibility in the construction of logical forms, allowing us to insert a predicate based on the partial logical form constructed thus far and assess its compatibility with the words afterwards (based on features), rather than insisting on a purely lexically driven formalism. Section 4.1.3 describes the lexical triggers and trace predicates that we use in our experiments.

2.6.2 Recursive Construction of DCS Trees. Given a set of lexical triggers L, we will now describe a recursive mechanism for mapping an utterance $\mathbf{x} = (x_1, \ldots, x_n)$ to $\mathcal{Z}_L(\mathbf{x})$, a set of candidate DCS trees for $\mathbf{x}$. The basic approach is reminiscent of projective labeled dependency parsing: For each span $i..j$ of the utterance, we build a set of trees $C_{i,j}(\mathbf{x})$. The set of trees for the span $0..n$ is the final result:

$$\mathcal{Z}_L(\mathbf{x}) = C_{0,n}(\mathbf{x}) \tag{44}$$

Each set of DCS trees $C_{i,j}(\mathbf{x})$ is constructed recursively by combining the trees of its subspans $C_{i,k}(\mathbf{x})$ and $C_{k',j}(\mathbf{x})$ 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; these functions will be specified later. Formally, $C_{i,j}(\mathbf{x})$ is defined recursively as follows:

$$C_{i,j}(\mathbf{x}) = F\Big(A\Big(\{\langle p\rangle_{i..j} : p \in L(\mathbf{x}_{i+1..j})\} \cup \bigcup_{\substack{i \le k \le k' < j \\ a \in C_{i,k}(\mathbf{x}) \\ b \in C_{k',j}(\mathbf{x})}} T_1(a, b)\Big)\Big) \tag{45}$$

This recurrence has two parts:

- The base case: we take the phrase (sequence of words) over span $i..j$ and look up the set of predicates p in the set of lexical triggers. For each predicate, we construct a one-node DCS tree. We also extend the definition of DCS trees in Section 2.2 to allow each node to store the indices of the span $i..j$ that triggered the predicate at that node; this is denoted by $\langle p\rangle_{i..j}$. This span information will be useful in Section 3.1.1, where we will need to talk about how an utterance $\mathbf{x}$ is aligned with a DCS tree z.

- The recursive case: $T_1(a, b)$, which we will define shortly, that takes two DCS trees, a and b, and returns a set of new DCS trees formed by combining a and b. Figure 17 shows this recurrence graphically.

We now focus on how to combine two DCS trees. Define $T_d(a, b)$ as the set of DCS trees that result by making either a or b the root and connecting the other via a chain of relations and at most d trace predicates (d is a small integer that keeps the set of DCS trees manageable):

$$T_d(a, b) = T_d^{\searrow}(a, b) \cup T_d^{\swarrow}(b, a) \tag{46}$$

Here, $T_d^{\searrow}(a, b)$ is the set of DCS trees where a is the root; for $T_d^{\swarrow}(a, b)$, b is the root. The former is defined recursively as follows:

$$T_0^{\searrow}(a, b) = \emptyset, \tag{47}$$

$$T_d^{\searrow}(a, b) = \bigcup_{\substack{r \in \mathcal{R} \\ p \in L(\epsilon)}} \{\langle a.p; a.\mathbf{e}; r : b\rangle, \langle a.p; a.\mathbf{e}; r : \langle \Sigma : b\rangle\rangle\} \cup T_{d-1}^{\searrow}(a, \langle p; r : b\rangle)$$

First, we consider all possible relations $r \in \mathcal{R}$ and try appending an edge to a with relation r and child b ($\langle a.p; a.\mathbf{e}; r : b\rangle$); an aggregate relation Σ can be inserted in addition ($\langle a.p; a.\mathbf{e}; r : \langle \Sigma : b\rangle\rangle$). Of course, $\mathcal{R}$ contains an infinite number of join and execute relations, but only a small finite number of them make sense: We consider join relations $\overset{j}{j'}$ only for $j \in \{1, \ldots, \text{ARITY}(a.p)\}$ and $j' \in \{1, \ldots, \text{ARITY}(b.p)\}$, and execute relations $\text{X}_\mathbf{i}$ for which $\mathbf{i}$ does not contain indices larger than the number of columns of $[\![b]\!]_w$. Next, we further consider all possible trace predicates $p \in L(\epsilon)$, and recursively try to connect

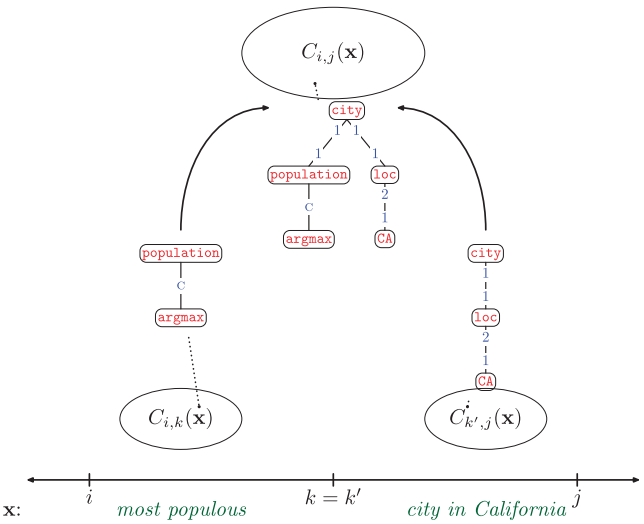


Figure 17
An example of the recursive construction of $C_{i,j}(\mathbf{x})$, a set of DCS trees for span $i..j$.

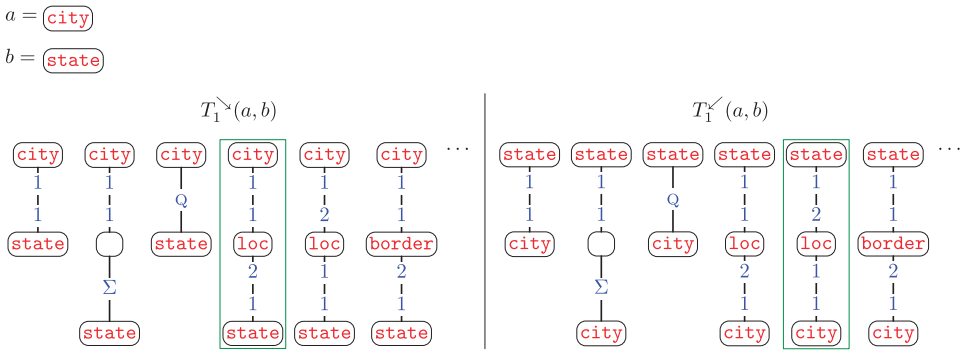


Figure 18
Given two DCS trees, a and b, $T_1^{\searrow}(a,b)$ and $T_1^{\swarrow}(a,b)$ are the two sets of DCS trees formed by combining a and b with a at the root and b at the root, respectively; one trace predicate can be inserted in between. In this example, the DCS trees which survive filtering (Section 2.6.3) are shown.

a with the intermediate $\langle p; r{:}b \rangle$, now allowing $d-1$ additional predicates. See Figure 18 for an example. In the other direction, $T_d^{\swarrow}$ is defined similarly:

$$T_0^{\swarrow}(a,b) = \emptyset \tag{48}$$
$$T_d^{\swarrow}(a,b) = \bigcup_{\substack{r \in \mathcal{R} \\ p \in L(\epsilon)}} \{\langle b.p; r{:}a; b.\mathbf{e} \rangle, \langle b.p; r{:}\langle \Sigma{:}a \rangle; b.\mathbf{e} \rangle\} \cup T_{d-1}^{\swarrow}(a, \langle p; r{:}b \rangle)$$

Inserting trace predicates allows us to build logical forms with more predicates than are explicitly triggered by the words. This ability is useful for several reasons. Sometimes, there is a predicate not overtly expressed, especially in noun compounds (e.g., *California cities*). For semantically light words such as prepositions (e.g., *for*) it is difficult to enumerate all the possible predicates that they might trigger; it is simpler computationally to try to insert trace predicates. We can even omit lexical triggers for transitive verbs such as *border* because the corresponding predicate `border` can be inserted as a trace predicate.

The function $T_1(a,b)$ connects two DCS trees via a path of relations and trace predicates. The augmentation function A adds additional relations (specifically, $\textsc{e}$ and/or $\textsc{x}_\mathbf{i}$) on a single DCS tree:

$$A(Z) = \bigcup_{\substack{z \in Z \\ \textsc{x}_\mathbf{i} \in \mathcal{R}}} \{z, \langle z; \textsc{e}{:}\langle \emptyset \rangle \rangle, \langle \textsc{x}_\mathbf{i}{:}z \rangle, \langle \textsc{x}_\mathbf{i}{:}\langle z; \textsc{e}{:}\langle \emptyset \rangle \rangle \rangle\} \tag{49}$$

2.6.3 Filtering using Abstract Interpretation. The construction procedure as described thus far is extremely permissive, generating many DCS trees which are obviously wrong—for example, $\langle \texttt{state}; \frac{1}{1}{:}\langle > ; \frac{2}{1}\langle 3 \rangle \rangle \rangle$, which tries to compare a state with the number 3. There is nothing wrong with this expression syntactically: Its denotation will simply be empty (with respect to the world). But semantically, this DCS tree is anomalous.

We cannot simply just discard DCS trees with empty denotations, because we would incorrectly rule out $\langle \texttt{state}; \frac{1}{1}{:}\langle \texttt{border}; \frac{2}{1}\langle \texttt{AK} \rangle \rangle \rangle$. The difference here is that even though the denotation is empty in this world, it is possible that it might not be empty

in a different world where history and geology took another turn, whereas it is simply impossible to compare cities and numbers.

Now let us quickly flesh out this intuition before falling into a philosophical discussion about possible worlds. Given a world w, we define an abstract world $\alpha(w)$, to be described shortly. We compute the denotation of a DCS tree z with respect to this abstract world. If at any point in the computation we create an empty denotation, we judge z to be impossible and throw it away. The filtering function F is defined as follows:[10]

$$F(Z) = \{z \in Z : \forall z' \text{ subtree of z }, [\![z']\!]_{\alpha(w)}.A \neq \emptyset\} \tag{50}$$

Now we need to define the abstract world $\alpha(w)$. The intuition is to map concrete values to abstract values: $3\texttt{:length}$ becomes $*\texttt{:length}$, $\mathit{Oregon}\texttt{:state}$ becomes $*\texttt{:state}$, and in general, primitive value $x\texttt{:}t$ becomes $*\texttt{:}t$. We perform abstraction on tuples componentwise, so that $(\mathit{Oregon}\texttt{:state}, 3\texttt{:length})$ becomes $(*\texttt{:state}, *\texttt{:length})$. Our abstraction of sets is slightly more complex: The empty set maps to the empty set, a set containing values all with the same abstract value a maps to $\{a\}$, and a set containing values with more than one abstract value maps to $\{\text{MIXED}\}$. Finally, a world maps each predicate onto a set of (concrete) tuples; the corresponding abstract world maps each predicate onto the set of abstract tuples. Formally, the abstraction function is defined as follows:

$$\alpha(x : t) = * : t \qquad \text{[primitive value]} \tag{51}$$

$$\alpha((v_1, \ldots, v_n)) = (\alpha(v_1), \ldots, \alpha(v_n)) \qquad \text{[tuple]} \tag{52}$$

$$\alpha(A) = \begin{cases} \emptyset & \text{if } A = \emptyset \\ \{\alpha(x) : x \in A\} & \text{if } |\{\alpha(x) : x \in A\}| = 1 \\ \{\text{MIXED}\} & \text{otherwise} \end{cases} \qquad \text{[set]} \tag{53}$$

$$\alpha(w) = \lambda p.\{\alpha(x) : x \in w(p)\} \qquad \text{[world]} \tag{54}$$

As an example, the abstract world might look like this:

$$\alpha(w)(\texttt{>}) = \{(*\texttt{:number}, *\texttt{:number}, *\texttt{:number}) \tag{55}$$
$$(*\texttt{:length}, *\texttt{:length}, *\texttt{:length}), \ldots\}$$

$$\alpha(w)(\texttt{state}) = \{(*\texttt{:state})\} \tag{56}$$

$$\alpha(w)(\texttt{AK}) = \{(*\texttt{:state})\} \tag{57}$$

$$\alpha(w)(\texttt{border}) = \{(*\texttt{:state}, *\texttt{:state})\} \tag{58}$$

Now returning to our motivating example at the beginning of this section, we see that the bad DCS tree has an empty abstract denotation $[\![\langle \texttt{state}; 1\texttt{:}\langle \texttt{>}; 2\texttt{:}\langle 3\rangle\rangle\rangle]\!]_{\alpha(w)} = \langle\langle \emptyset; \varnothing\rangle\rangle$. The good DCS tree has a non-empty abstract denotation: $[\![\langle \texttt{state}; 1\texttt{:}\langle \texttt{border}; 2\texttt{:}\langle \texttt{AK}\rangle\rangle\rangle]\!]_{\alpha(w)} = \langle\langle \{(*\texttt{:state})\}; \varnothing\rangle\rangle$, as desired.

10 To further reduce the search space, F imposes a few additional constraints: for example, limiting the number of columns to 2, and only allowing trace predicates between arity 1 predicates.

Remarks. Computing denotations on an abstract world is called **abstract interpretation** (Cousot and Cousot 1977) and is a very powerful framework commonly used in the programming languages community. The idea is to obtain information about a program (in our case, a DCS tree) without running it concretely, but rather just by running it abstractly. It is closely related to type systems, but the type of abstractions one uses is often much richer than standard type systems.

2.6.4 Comparison with CCG. We now compare our construction mechanism with CCG (see Figure 19 for an example). The main difference is that our lexical triggers contain less information than a lexical entry in a CCG. In CCG, the lexicon would have an entry such as

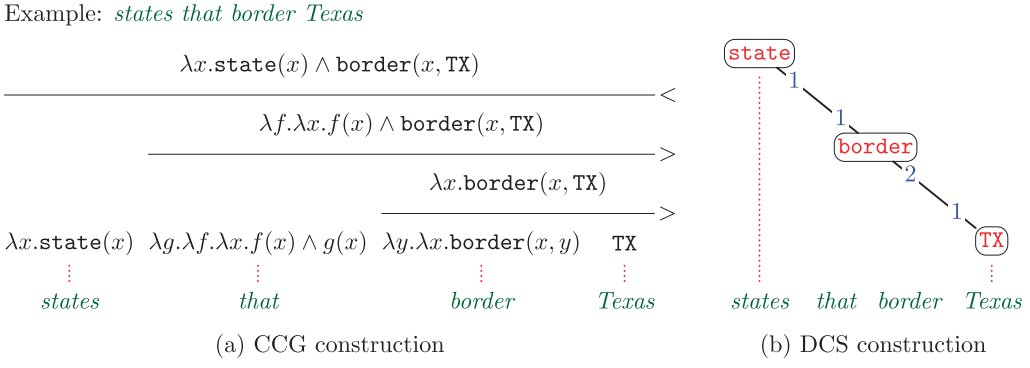
$$major \vdash \text{N/N} : \lambda f.\lambda x.\texttt{major}(x) \wedge f(x) \tag{59}$$

which gives detailed information about how this word should interact with its context. In DCS construction, however, each lexical trigger only has the minimal amount of information:

$$major \vdash \texttt{major} \tag{60}$$

A lexical trigger specifies a pre-theoretic "meaning" of a word which does not commit to any formalisms. One advantage of this minimality is that lexical triggers could be easily obtained from non-expert supervision: One would only have to associate words with database table names (predicates).

In some sense, the DCS construction mechanism pushes the complexity out of the lexicon. In linguistics, this complexity usually would end up in the grammar, which would be undesirable. We do not have to respect this tradeoff, however, because the



(a) CCG construction (b) DCS construction

Figure 19
Comparison between the construction mechanisms of CCG and DCS. There are three principal differences: First, in CCG, words are mapped onto lambda calculus expressions; in DCS, words are just mapped onto predicates. Second, in CCG, lambda calculus expressions are built by combining (e.g., via function application) two smaller expressions; in DCS, trees are combined by inserting relations (and possibly other predicates between them). Third, in CCG, all words map to logical expressions; in DCS, only a small subset of words (e.g., *state* and `Texas`) map to predicates; the rest participate in features for scoring DCS trees.

construction mechanism only produces an overapproximation, which means it is possible to have both a simple "lexicon" and a simple "grammar."

There is an important practical rationale for this design decision. During learning, we never just have one clean lexical entry per word. Rather, there are often many possible lexical entries (and to handle disfluent utterances or utterances in free word-order languages, we might actually need many of them [Kwiatkowski et al. 2010]):

$$\textit{major} \vdash \text{N} : \lambda x.\texttt{major}(x) \tag{61}$$

$$\textit{major} \vdash \text{N}/\text{N} : \lambda f.\lambda x.\texttt{major}(x) \wedge f(x) \tag{62}$$

$$\textit{major} \vdash \text{N}\backslash\text{N} : \lambda f.\lambda x.\texttt{major}(x) \wedge f(x) \tag{63}$$

$$\ldots \tag{64}$$

Now think of a DCS lexical trigger $\textit{major} \vdash \texttt{major}$ as simply a *compact representation* for a set of CCG lexical entries. Furthermore, the choice of the lexical entry is made not at the initial lexical base case, but rather during the recursive construction by inserting relations between DCS subtrees. It is exactly at this point that the choice *can* be made, because after all, the choice is one that depends on context. The general principle is to compactly represent the indeterminacy until one can resolve it. Compactly representing a set of CCG lexical entries can also be done within the CCG framework by factoring lexical entries into a **lexeme** and a **lexical template** (Kwiatkowski et al. 2011).

Type raising is a combinator in CCG that traditionally converts x to $\lambda f.f(x)$. In recent work, Zettlemoyer and Collins (2007) introduced more general type-changing combinators to allow conversion from one entity into a related entity in general (a kind of generalized metonymy). For example, in order to parse *Boston flights*, `Boston` is transformed to $\lambda x.\texttt{to}(x, \texttt{Boston})$. This type changing is analogous to inserting trace predicates in DCS, but there is an important distinction: Type changing is a unary operation and is unconstrained in that it changes logical forms into new ones without regard for how they will be used downstream. Inserting trace predicates is a binary operation that is constrained by the two predicates that it is mediating. In the example, `to` would only be inserted to combine `Boston` with `flight`. This is another instance of the general principle of delaying uncertain decisions until there is more information.

3. Learning

In Section 2, we defined DCS trees and a construction mechanism for producing a set of candidate DCS trees given an utterance. We now define a probability distribution over that set (Section 3.1) and an algorithm for estimating the parameters (Section 3.2). The number of candidate DCS trees grows exponentially, so we use beam search to control this growth. The final learning algorithm alternates between beam search and optimization of the parameters, leading to a natural bootstrapping procedure which integrates learning and search.

3.1 Semantic Parsing Model

The semantic parsing model specifies a conditional distribution over a set of candidate DCS trees $C(\mathbf{x})$ given an utterance $\mathbf{x}$. This distribution depends on a function $\phi(\mathbf{x}, z) \in \mathbb{R}^d$, which takes a $(\mathbf{x}, z)$ pair and extracts a set of local features (see Section 3.1.1

for a full specification). Associated with this feature vector is a parameter vector $\theta \in \mathbb{R}^d$. The inner product between the two vectors, $\phi(\mathbf{x}, z)^\top \theta$, yields a numerical score, which intuitively measures the compatibility of the utterance $\mathbf{x}$ with the DCS tree z. We exponentiate the score and normalize over $C(\mathbf{x})$ to obtain a proper probability distribution:

$$p(z \mid \mathbf{x}; C, \theta) = \exp\{\phi(\mathbf{x}, z)^\top \theta - \mathbf{A}(\theta; \mathbf{x}, C)\} \tag{65}$$

$$\mathbf{A}(\theta; \mathbf{x}, C) = \log \sum_{z \in C(\mathbf{x})} \exp\{\phi(\mathbf{x}, z)^\top \theta\} \tag{66}$$

where $\mathbf{A}(\theta; \mathbf{x}, C)$ is the log-partition function with respect to the candidate set function $C(\mathbf{x})$.

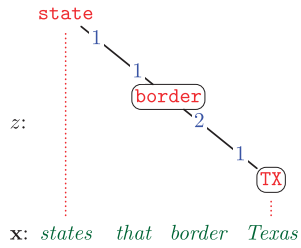
3.1.1 Features. We now define the feature vector $\phi(\mathbf{x}, z) \in \mathbb{R}^d$, the core part of the semantic parsing model. Each component $j = 1, \ldots, d$ of this vector is a feature, and $\phi(\mathbf{x}, z)_j$ is the number of times that feature occurs in $(\mathbf{x}, z)$. Rather than working with indices, we treat features as symbols (e.g., TRIGGERPRED[*states*, `state`]). Each feature captures some property about $(\mathbf{x}, z)$ that abstracts away from the details of the specific instance and allows us to generalize to new instances that share common features.

The features are organized into feature templates, where each feature template instantiates a set of features. Figure 20 shows all the feature templates for a concrete example. The feature templates are as follows:

- PREDHIT contains the single feature PREDHIT, which fires for each predicate in z.
- PRED contains features $\{\text{PRED}[\alpha(p)] : p \in \mathcal{P}\}$, each of which fires on $\alpha(p)$, the abstraction of predicate p, where

$$\alpha(p) = \begin{cases} *{:}t & \text{if } p = x{:}t \\ p & \text{otherwise} \end{cases} \tag{67}$$

  The purpose of the abstraction is to abstract away the details of concrete values such as `TX` = *Texas*:`state`.
- PREDREL contains features $\{\text{PREDREL}[\alpha(p), \mathbf{q}] : p \in \mathcal{P}, \mathbf{q} \in (\{\swarrow, \searrow\} \times \mathcal{R})^*\}$. A feature fires when a node x has predicate p and is connected via some path $\mathbf{q} = (d_1, r_1), \ldots, (d_m, r_m)$ to the lowest descendant node y with the property that each node between x and y has a null predicate. Each (d, r) on the path represents an edge labeled with relation r connecting to a left ($d = \swarrow$) or right ($d = \searrow$) child. If x has no children, then $m = 0$. The most common case is when $m = 1$, but $m = 2$ also occurs with the aggregate and execute relations (e.g., PREDREL[`count`, $\searrow\, {}^{1}_{1}\, \searrow\, \Sigma$] fires for Figure 5(a)).
- PREDRELPRED contains features $\{\text{PREDRELPRED}[\alpha(p), \mathbf{q}, \alpha(p')] : p, p' \in \mathcal{P}, \mathbf{q} \in (\{\swarrow, \searrow\} \times \mathcal{R})^*\}$, which are the same as PREDREL, except that we include both the predicate p of x and the predicate p' of the descendant node y. These features do not fire if $m = 0$.



Feature template	Feature j	Count $\phi(\mathbf{x}, z)_j$	Parameter θ_j
[Number of predicates]	PredHit	3	2.721
[Predicate]	Pred[state]	1	0.570
	Pred[border]	1	−2.596
	Pred[∗:state]	1	1.511
[Predicate + relation]	PredRel[state $\searrow^{1}_{1}$]	1	−0.262
	PredRel[border $\searrow^{2}_{1}$]	1	−2.248
	PredRel[∗:state]	1	1.059
[Predicate + relation + predicate]	PredRelPred[state $\searrow^{1}_{1}$ border]	1	2.119
	PredRelPred[border $\searrow^{2}_{1}$ ∗:state]	1	1.090
[Word + trigger predicate]	TriggerPred[*states*, state]	1	3.262
	TriggerPred[*Texas*, *Texas*:state]	1	−2.272
[Word + trace predicate]	TracePred[*that*, $\searrow$ border]	1	3.041
	TracePred[*border*, $\searrow$ border]	1	−0.253
[Word + trace relation]	TraceRel[*that*, $\searrow^{1}_{1}$]	1	0.000
	TraceRel[*border*, $\searrow^{1}_{1}$]	1	0.000
[Word + trace predicate + relation]	TracePredRel[*that*, state $\searrow^{1}_{1}$]	1	0.000
	TracePredRel[*border*, state $\searrow^{1}_{1}$]	1	0.000

Score: $\phi(\mathbf{x}, z)^\top \theta = 13.184$

Figure 20
For each utterance–DCS tree pair $(\mathbf{x}, z)$, we define a feature vector $\phi(\mathbf{x}, z)$, whose j-th component is the number of times a feature j occurs in $(\mathbf{x}, z)$. Each feature has an associated parameter θ_j, which is estimated from data in Section 3.2. The inner product of the feature vector and parameter vector yields a compatibility score.

- TriggerPred contains features $\{\text{TriggerPred}[\mathbf{s}, p] : \mathbf{s} \in W^*, p \in \mathcal{P}\}$, where $W = \{it, Texas, \ldots\}$ is the set of words. Each of these features fires when a span of the utterance with words $\mathbf{s}$ triggers the predicate p—more precisely, when a subtree $\langle p; \mathbf{e} \rangle_{i..j}$ exists with $\mathbf{s} = \mathbf{x}_{i+1..j}$. Note that these lexicalized features use the predicate p rather than the abstracted version $\alpha(p)$.
- TracePred contains features $\{\text{TracePred}[s, p, d] : s \in W^*, p \in \mathcal{P}, d \in \{\swarrow, \searrow\}\}$, each of which fires when a trace predicate p has been inserted

over a word s. The situation is the following: Suppose we have a subtree a that ends at position k (there is a predicate in a that is triggered by a phrase with right endpoint k) and another subtree b that begins at k'. Recall that in the construction mechanism (46), we can insert a trace predicate $p \in L(\epsilon)$ between the roots of a and b. Then, for every word $\mathbf{x}_j$ between the spans of the two subtrees ($j = \{k+1, \ldots, k'\}$), the feature TRACEPRED$[\mathbf{x}_j, p, d]$ fires ($d = \swarrow$ if b dominates a and $d = \searrow$ if a dominates b).

- TRACEREL contains features $\{\text{TRACEREL}[s, d, r] : s \in W^*, d \in \{\swarrow, \searrow\}, r \in \mathcal{R}\}$, each of which fires when some trace predicate with parent relation r has been inserted over a word s.
- TRACEPREDREL contains features $\{\text{TRACEPREDREL}[s, p, d, r] : s \in W^*, p \in \mathcal{P}, d \in \{\swarrow, \searrow\}, r \in \mathcal{R}\}$, each of which fires when a predicate p is connected via child relation r to some trace predicate over a word s.

These features are simple generic patterns which can be applied for modeling essentially any distribution over sequences and labeled trees—there is nothing specific to DCS at all. The first half of the feature templates (PREDHIT, PRED, PREDREL, PREDRELPRED) capture properties of the tree independent of the utterance, and are similar to those used for syntactic dependency parsing. The other feature templates (TRIGGERPRED, TRACEPRED, TRACEREL, TRACEPREDREL) connect predicates in the DCS tree with words in the utterance, similar to those in a model of machine translation.

3.2 Parameter Estimation

We have now fully specified the details of the graphical model in Figure 2: Section 3.1 described semantic parsing and Section 2 described semantic evaluation. Next, we focus on the inferential problem of estimating the parameters θ of the model from data.

3.2.1 Objective Function. We assume that our learning algorithm is given a training data set $\mathcal{D}$ containing question–answer pairs $(\mathbf{x}, y)$. Because the logical forms are unobserved, we work with $\log p(y \mid \mathbf{x}; C, \theta)$, the marginal log-likelihood of obtaining the correct answer y given an utterance $\mathbf{x}$. This marginal log-likelihood sums over all $z \in C(\mathbf{x})$ that evaluate to y:

$$\log p(y \mid \mathbf{x}; C, \theta) = \log p(z \in C^y(\mathbf{x}) \mid \mathbf{x}; C, \theta) \tag{68}$$

$$= \mathbf{A}(\theta; \mathbf{x}, C^y) - \mathbf{A}(\theta, \mathbf{x}, C), \text{ where} \tag{69}$$

$$C^y(\mathbf{x}) \stackrel{\text{def}}{=} \{z \in C(\mathbf{x}) : [\![z]\!]_w = y\} \tag{70}$$

Here, $C^y(\mathbf{x})$ is the set of DCS trees z with denotation y.

We call an example $(\mathbf{x}, y) \in \mathcal{D}$ **feasible** if the candidate set of $\mathbf{x}$ contains a DCS tree that evaluates to y ($C^y(\mathbf{x}) \neq \emptyset$). Define an objective function $\mathcal{O}(\theta, C)$ containing two terms. The first term is the sum of the marginal log-likelihood over all feasible

training examples. The second term is a quadratic penalty on the parameters θ with regularization parameter λ. Formally:

$$\mathcal{O}(\theta, C) \stackrel{\text{def}}{=} \sum_{\substack{(\mathbf{x},y)\in\mathcal{D} \\ C^y(\mathbf{x})\neq\emptyset}} \log p(y \mid \mathbf{x}; C, \theta) - \frac{\lambda}{2}\|\theta\|_2^2 \tag{71}$$

$$= \sum_{\substack{(\mathbf{x},y)\in\mathcal{D} \\ C^y(\mathbf{x})\neq\emptyset}} (\mathbf{A}(\theta; \mathbf{x}, C^y) - \mathbf{A}(\theta; \mathbf{x}, C)) - \frac{\lambda}{2}\|\theta\|_2^2$$

We would like to maximize $\mathcal{O}(\theta, C)$. The log-partition function $\mathbf{A}(\theta; \cdot, \cdot)$ is convex, but $\mathcal{O}(\theta, C)$ is the difference of two log-partition functions and hence is not concave (nor convex). Thus we resort to gradient-based optimization. A standard result is that the derivative of the log-partition function is the expected feature vector (Wainwright and Jordan 2008). Using this, we obtain the gradient of our objective function:[11]

$$\frac{\partial \mathcal{O}(\theta, C)}{\partial \theta} = \sum_{\substack{(\mathbf{x},y)\in\mathcal{D} \\ C^y(\mathbf{x})\neq\emptyset}} \left(\mathbb{E}_{p(z|\mathbf{x};C^y,\theta)}[\phi(\mathbf{x}, z)] - \mathbb{E}_{p(z|\mathbf{x};C,\theta)}[\phi(\mathbf{x}, z)]\right) - \lambda\theta \tag{72}$$

Updating the parameters in the direction of the gradient would move the parameters towards the DCS trees that yield the correct answer (C^y) and away from overall candidate DCS trees (C). We can use any standard numerical optimization algorithm that requires only black-box access to a gradient. Section 4.3.4 will discuss the empirical ramifications of the choice of optimization algorithm.

3.2.2 Algorithm. Given a candidate set function $C(\mathbf{x})$, we can optimize Equation (71) to obtain estimates of the parameters θ. Ideally, we would use $C(\mathbf{x}) = \mathcal{Z}_L(\mathbf{x})$, the candidate sets from our construction mechanism in Section 2.6, but we quickly run into the problem of computing Equation (72) efficiently. Note that $\mathcal{Z}_L(\mathbf{x})$ (defined in Equation (44)) grows exponentially with the length of $\mathbf{x}$. This by itself is not a show-stopper. Our features (Section 3.1.1) decompose along the edges of the DCS tree, so it is possible to use dynamic programming[12] to compute the second expectation $\mathbb{E}_{p(z|\mathbf{x};\mathcal{Z}_L,\theta)}[\phi(\mathbf{x}, z)]$ of Equation (72). The problem is computing the first expectation $\mathbb{E}_{p(z|\mathbf{x};\mathcal{Z}_L^y,\theta)}[\phi(\mathbf{x}, z)]$, which sums over the subset of candidate DCS trees z satisfying the constraint $[\![z]\!]_w = y$. Though this is a smaller set, there is no efficient dynamic program for this set because the constraint does not decompose along the structure of the DCS tree. Therefore, we need to approximate $\mathcal{Z}_L^y$, and, in fact, we will approximate $\mathcal{Z}_L$ as well so that the two expectations in Equation (72) are coherent.

Recall that $\mathcal{Z}_L(\mathbf{x})$ was built by recursively constructing a set of DCS trees $C_{i,j}(\mathbf{x})$ for each span $i..j$. In our approximation, we simply use beam search, which truncates each $C_{i,j}(\mathbf{x})$ to include the (at most) K DCS trees with the highest score $\phi(\mathbf{x}, z)^\top\theta$. We

11 Notation: $\mathbb{E}_{p(x)}[f(x)] = \sum_x p(x)f(x)$.

12 The state of the dynamic program would be the span $i..j$ and the head predicate over that span.

let $\tilde{C}_{i,j,\theta}(\mathbf{x})$ denote this approximation and define the set of candidate DCS trees with respect to the beam search:

$$\tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x}) = \tilde{C}_{0,n,\theta}(\mathbf{x}) \tag{73}$$

We now have a chicken-and-egg problem: If we had good parameters θ, we could generate good candidate sets $C(\mathbf{x})$ using beam search $\tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x})$. If we had good candidate sets $C(\mathbf{x})$, we could generate good parameters by optimizing our objective $\mathcal{O}(\theta, C)$ in Equation (71). This problem leads to a natural solution: simply alternate between the two steps (Figure 21). This procedure is not guaranteed to converge, due to the heuristic nature of the beam search, but we have found it to be convergent in practice.

Finally, we use the trained model with parameters θ to answer new questions $\mathbf{x}$ by choosing the most likely answer y, summing out the latent logical form z:

$$F_\theta(\mathbf{x}) \stackrel{\text{def}}{=} \operatorname*{argmax}_y p(y \mid \mathbf{x}; \theta, \tilde{\mathcal{Z}}_{L,\theta}) \tag{74}$$

$$= \operatorname*{argmax}_y \sum_{\substack{z \in \tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x}) \\ [\![z]\!]_w = y}} p(z \mid \mathbf{x}; \theta, \tilde{\mathcal{Z}}_{L,\theta}) \tag{75}$$

4. Experiments

We have now completed the conceptual part of this article—using DCS trees to represent logical forms (Section 2), and learning a probabilistic model over these trees (Section 3). In this section, we evaluate and study our approach empirically. Our main result is that our system can obtain comparable accuracies to state-of-the-art systems that require annotated logical forms. All the code and data are available at `cs.stanford.edu/~pliang/software/`.

4.1 Experimental Set-up

We first describe the data sets (Section 4.1.1) that we use to train and evaluate our system. We then mention various choices in the model and learning algorithm (Section 4.1.2). One of these choices is the lexical triggers, which are further discussed in Section 4.1.3.

Learning Algorithm

Initialize: $\theta^{(0)} \leftarrow (0, \ldots, 0)$
For each iteration $t = 1, \ldots, T$:
  Update candidate sets: $C^{(t)}(\mathbf{x}) \leftarrow \tilde{\mathcal{Z}}_{L,\theta^{(t-1)}}(\mathbf{x})$
  Update parameters: $\theta^{(t)} \leftarrow \operatorname{argmax}_\theta \mathcal{O}(\theta, C^{(t)})$
Return $\theta^{(T)}$

Figure 21
The learning algorithm alternates between updating the candidate sets based on beam search and updating the parameters using standard numerical optimization.

4.1.1 Data sets. We tested our methods on two standard data sets, referred to in this article as GEO and JOBS. These data sets were created by Ray Mooney's group during the 1990s and have been used to evaluate semantic parsers for over a decade.

U.S. Geography. The GEO data set, originally created by Zelle and Mooney (1996), contains 880 questions about U.S. geography and a database of facts encoded in Prolog. The questions in GEO ask about general properties (e.g., area, elevation, and population) of geographical entities (e.g., cities, states, rivers, and mountains). Across all the questions, there are 280 word types, and the length of an utterance ranges from 4 to 19 words, with an average of 8.5 words. The questions involve conjunctions, superlatives, and negation, but no generalized quantification. Each question is annotated with a logical form in Prolog, for example:

Utterance: *What is the highest point in Florida?*
Logical form: `answer(A,highest(A,(place(A),loc(A,B),const(B,stateid(florida)))))`

Because our approach learns from answers, not logical forms, we evaluated the annotated logical forms on the provided database to obtain the correct answers.

Recall that a world/database w maps each predicate $p \in \mathcal{P}$ to a set of tuples $w(p)$. Some predicates contain the set of tuples explicitly (e.g., `mountain`); others can be derived (e.g., `higher` takes two entities x and y and returns true if $\texttt{elevation}(x) > \texttt{elevation}(y)$). Other predicates are higher-order (e.g., `sum`, `highest`) in that they take other predicates as arguments. We do not use the provided domain-specific higher-order predicates (e.g., `highest`), but rather provide domain-independent higher-order predicates (e.g., `argmax`) and the ordinary domain-specific predicates (e.g., `elevation`). This provides more compositionality and therefore better generalization. Similarly, we use `more` and `elevation` instead of `higher`. Altogether, $\mathcal{P}$ contains 43 predicates plus one predicate for each value (e.g., `CA`).

Job Queries. The JOBS data set (Tang and Mooney 2001) contains 640 natural language queries about job postings. Most of the questions ask for jobs matching various criteria: job title, company, recruiter, location, salary, languages and platforms used, areas of expertise, required/desired degrees, and required/desired years of experience. Across all utterances, there are 388 word types, and the length of an utterance ranges from 2 to 23 words, with an average of 9.8 words.

The utterances are mostly based on conjunctions of criteria, with a sprinkling of negation and disjunction. Here is an example:

Utterance: *Are there any jobs using Java that are not with IBM?*
Logical form: `answer(A,(job(A),language(A,'java'),¬company(A,'IBM')))`

The JOBS data set comes with a database, which we can use as the world w. When the logical forms are evaluated on this database, however, close to half of the answers are empty (no jobs match the requested criteria). Therefore, there is a large discrepancy between obtaining the correct logical form (which has been the focus of most work on semantic parsing) and obtaining the correct answer (our focus).

To bring these two into better alignment, we generated a random database as follows: We created $m = 100$ jobs. For each job j, we go through each predicate p (e.g., `company`) that takes two arguments, a job, and a target value. For each of the possible target values v, we add (j, v) to $w(p)$ independently with probability $\alpha = 0.8$. For example, for $p = \texttt{company}$, $j = \texttt{job37}$, we might add $(\texttt{job37}, \texttt{IBM})$ to $w(\texttt{company})$. The result is

a database with a total of 23 predicates (which includes the domain-independent ones) in addition to the value predicates (e.g., IBM).

The goal of using randomness is to ensure that two different logical forms will most likely yield different answers. For example, consider two logical forms:

$$z_1 = \lambda j.\texttt{job}(j) \wedge \texttt{company}(j, \texttt{IBM}), \tag{76}$$

$$z_2 = \lambda j.\texttt{job}(j) \wedge \texttt{language}(j, \texttt{Java}). \tag{77}$$

Under the random construction, the denotation of z_1 is S_1, a random subset of the jobs, where each job is included in S_1 independently with probability α, and the denotation of z_2 is S_2, which has the same distribution as S_1 but importantly is independent of S_1. Therefore, the probability that $S_1 = S_2$ is $[\alpha^2 + (1-\alpha)^2]^m$, which is exponentially small in m. This construction yields a world that is not entirely "realistic" (a job might have multiple employers), but it ensures that if we get the correct answer, we probably also obtain the correct logical form.

4.1.2 Settings. There are a number of settings that control the tradeoffs between computation, expressiveness, and generalization power of our model, shown here. For now, we will use generic settings chosen rather crudely; Section 4.3.4 will explore the effect of changing these settings.

Lexical Triggers The lexical triggers L (Section 2.6.1) define the set of candidate DCS trees for each utterance. There is a tradeoff between expressiveness and computational complexity: The more triggers we have, the more DCS trees we can consider for a given utterance, but then either the candidate sets become too large or beam search starts dropping the good DCS trees. Choosing lexical triggers is important and requires additional supervision (Section 4.1.3).

Features Our probabilistic semantic parsing model is defined in terms of feature templates (Section 3.1.1). Richer features increase expressiveness but also might lead to overfitting. By default, we include all the feature templates.

Number of training examples (n) An important property of any learning algorithm is its sample complexity—how many training examples are required to obtain a certain level of accuracy? By default, all training examples are used.

Number of training iterations (T) Our learning algorithm (Figure 21) alternates between updating candidate sets and updating parameters for T iterations. We use $T = 5$ as the default value.

Beam size (K) The computation of the candidate sets in Figure 21 is based on beam search where each intermediate state keeps at most K DCS trees. The default value is $K = 100$.

Optimization algorithm To optimize the objective function $\mathcal{O}(\theta, C)$ our default is to use the standard L-BFGS algorithm (Nocedal 1980) with a backtracking line search for choosing the step size.

Regularization (λ) The regularization parameter $\lambda > 0$ in the objective function $\mathcal{O}(\theta, C)$ is another knob for controlling the tradeoff between fitting and overfitting. The default is $\lambda = 0.01$.

4.1.3 Lexical Triggers. The lexical trigger set L (Section 2.6.1) is a set of entries $(\mathbf{s}, p)$, where $\mathbf{s}$ is a sequence of words and p is a predicate. We run experiments on two sets of lexical triggers: **base triggers** L_{B} and **augmented triggers** $L_{\text{B+P}}$.

Base Triggers. The base trigger set L_{B} includes three types of entries:

- Domain-independent triggers: For each domain-independent predicate (e.g., `argmax`), we manually specify a few words associated with that predicate (e.g., *most*). The full list is shown at the top of Figure 22.
- Values: For each value x that appears in the world (specifically, $x \in v_j \in w(p)$ for some tuple v, index j, and predicate p), L_{B} contains an entry (x, x) (e.g., (*Boston*, *Boston*:`city`)). Note that this rule implicitly specifies an infinite number of triggers.

  Regarding predicate names, we do *not* add entries such as (*city*, `city`), because we want our system to be language-independent. In Turkish, for instance, we would not have the luxury of lexicographical cues that associate `city` with *şehir*. So we should think of the predicates as just symbols `predicate1`, `predicate2`, and so on. On the other hand, values in the database are generally proper nouns (e.g., city names) for which there are generally strong cross-linguistic lexicographic similarities.
- Part-of-speech (POS) triggers:[13] For each domain-specific predicate p, we specify a set of POS tags T. Implicitly, L_{B} contains all pairs (x, p) where the word x has a POS tag $t \in T$. For example, for `city`, we would specify NN and NNS, which means that any word which is a singular or plural common noun triggers the predicate `city`. Note that *city* triggers `city` as desired, but *state* also triggers `city`.

  The POS triggers for GEO and JOBS domains are shown in the left side of Figure 22. Note that some predicates such as `traverse` and `loc` are not associated with any POS tags. Predicates corresponding to verbs and prepositions are not included as overt lexical triggers, but rather included as trace predicates $L(\epsilon)$. In constructing the logical forms, nouns and adjectives serve as anchor points. Trace predicates can be inserted between these anchors. This strategy is more flexible than requiring each predicate to spring from some word.

Augmented Triggers. We now define the augmented trigger set $L_{\text{B+P}}$, which contains more domain-specific information than L_{B}. Specifically, for each domain-specific predicate (e.g., `city`), we manually specify a single prototype word (e.g., *city*) associated with that predicate. Under $L_{\text{B+P}}$, *city* would trigger only `city` because *city* is a prototype word, but *town* would trigger all the NN predicates (`city`, `state`, `country`, etc.) because it is not a prototype word.

Prototype triggers require only a modest amount of domain-specific supervision (see the right side of Figure 22 for the entire list for GEO and JOBS). In fact, as we'll see in Section 4.2, prototype triggers are not absolutely required to obtain good accuracies, but they give an extra boost and also improve computational efficiency by reducing the set of candidate DCS trees.

13 To perform POS tagging, we used the Berkeley Parser (Petrov et al. 2006), trained on the WSJ Treebank (Marcus, Marcinkiewicz, and Santorini 1993) and the Question Treebank (Judge, Cahill, and v. Genabith 2006)—thanks to Slav Petrov for providing the trained parser.

Domain-independent triggers

no | *not* | *dont* | *doesnt* | *outside* | *exclude*	⊢	`not`
each | *every*	⊢	`every`
most	⊢	`argmax`
least | *fewest*	⊢	`argmin`
count | *number* | *many*	⊢	`count`
large | *high* | *great*	⊢	`affirm`
small | *low*	⊢	`negate`
sum | *combined* | *total*	⊢	`sum`
less | *at most*	⊢	`less`
more | *at least*	⊢	`more`
called | *named*	⊢	`nameObj`

GEO **POS triggers**

NN | NNS	⊢	`city` | `state` | `country` | `lake` | `mountain` | `river` | `place` | `person` | `capital` | `population`
NN | NNS | JJ	⊢	`len` | `negLen` | `size` | `negSize` | `elevation` | `negElevation` | `density` | `negDensity` | `area` | `negArea`
NN | NNS	⊢	`usa:country`
JJ	⊢	`major`
WRB	⊢	`loc`
ϵ	⊢	`loc` | `next_to` | `traverse` | `hasInhabitant`

GEO **prototypes**

city	⊢ `city`	*person*	⊢ `person`
state	⊢ `state`	*population*	⊢ `population`
country	⊢ `country`	*long*	⊢ `len`
lake	⊢ `lake`	*short*	⊢ `negLen`
mountain	⊢ `mountain`	*large*	⊢ `size`
river	⊢ `river`	*small*	⊢ `negSize`
point	⊢ `place`	*high*	⊢ `elevation`
where	⊢ `loc`	*low*	⊢ `negElevation`
major	⊢ `major`	*dense*	⊢ `density`
capital	⊢ `capital`	*sparse*	⊢ `negDensity`
high point	⊢ `high_point`	*area*	⊢ `area`

JOBS **POS triggers**

NN | NNS	⊢	`job` | `deg` | `exp` | `language` | `loc`
ϵ	⊢	`salary_greater_than` | `require` | `desire` | `title` | `company` | `recruiter` | `area` | `platform` | `application` | `language` | `loc`

JOBS **prototypes**

(beginning of utterance)	⊢	`job`
degree	⊢	`deg`
experience	⊢	`exp`
language	⊢	`language`
location	⊢	`loc`

Figure 22
Lexical triggers used in our experiments.

Finally, to determine triggering, we stem all words using the Porter stemmer (Porter 1980), so that *mountains* triggers the same predicates as *mountain*. We also decompose superlatives into two words (e.g., *largest* is mapped to *most large*), allowing us to construct the logical form more compositionally.

4.2 Comparison with Other Systems

We now compare our approach with existing methods. We used the same training-test splits as Zettlemoyer and Collins (2005) (600 training and 280 test examples for GEO, 500 training and 140 test examples for JOBS). For development, we created five random splits of the training data. For each split, we put 70% of the examples into a *development training set* and the remaining 30% into a *development test set*. The actual test set was only used for obtaining final numbers.

4.2.1 Systems that Learn from Question–Answer Pairs. We first compare our system (henceforth, LJK11) with Clarke et al. (2010) (henceforth, CGCR10), which is most similar to our work in that it also learns from question–answer pairs without using annotated logical forms. CGCR10 works with the FunQL language and casts semantic parsing as integer linear programming (ILP). In each iteration, the learning algorithm solves the

Table 2
Results on GEO with 250 training and 250 test examples. Our system (LJK11 with base triggers and no logical forms) obtains higher test accuracy than CGCR10, even when CGCR10 is trained using logical forms.

System		Accuracy (%)
CGCR10 w/answers	(Clarke et al. 2010)	73.2
CGCR10 w/logical forms	(Clarke et al. 2010)	80.4
LJK11 w/base triggers	(Liang, Jordan, and Klein 2011)	84.0
LJK11 w/augmented triggers	(Liang, Jordan, and Klein 2011)	**87.6**

ILP to predict the logical form for each training example. The examples with correct predictions are fed to a structural support vector machine (SVM) and the model parameters are updated.

Though similar in spirit, there are some important differences between CGCR10 and our approach. They use ILP instead of beam search and structural SVM instead of log-linear models, but the main difference is which examples are used for learning. Our approach learns on any feasible example (Section 3.2.1), one where the candidate set contains a logical form that evaluates to the correct answer. CGCR10 uses a much more stringent criterion: The highest scoring logical form must evaluate to the correct answer. Therefore, for their algorithm to progress, the model already must be non-trivially good before learning even starts. This is reflected in the amount of prior knowledge and initialization that CGCR10 uses before learning starts: WordNet features, syntactic parse trees, and a set of lexical triggers with 1.42 words per non-value predicate. Our system with base triggers requires only simple indicator features, POS tags, and 0.5 words per non-value predicate.

CGCR10 created a version of GEO which contains 250 training and 250 test examples. Table 2 compares the empirical results of this split. We see that our system (LJK11) with base triggers significantly outperforms CGCR10 (84% vs. 73.2%), and it even outperforms the version of CGCR10 that is trained using logical forms (84.0% vs. 80.4%). If we use augmented triggers, we widen the gap by another 3.6 percentage points.[14]

4.2.2 State-of-the-Art Systems. We now compare our system (LJK11) with state-of-the-art systems, which all require annotated logical forms (except PRECISE). Here is a brief overview of the systems:

- COCKTAIL (Tang and Mooney 2001) uses inductive logic programming to learn rules for driving the decisions of a shift-reduce semantic parser. It assumes that a lexicon (mapping from words to predicates) is provided.
- PRECISE (Popescu, Etzioni, and Kautz 2003) does not use learning, but instead relies on matching words to strings in the database using various heuristics based on WordNet and the Charniak parser. Like our work, it also uses database type constraints to rule out spurious logical forms. One of the unique features of PRECISE is that it has 100% precision—it refuses to parse an utterance which it deems *semantically intractable*.

14 Note that the numbers for LJK11 differ from those presented in Liang, Jordan, and Klein (2011), which reports results based on 10 different splits rather than the set-up used by CGCR10.

- SCISSOR (Ge and Mooney 2005) learns a generative probabilistic model that extends the Collins (1999) models with semantic labels, so that syntactic and semantic parsing can be done jointly.
- SILT (Kate, Wong, and Mooney 2005) learns a set of transformation rules for mapping utterances to logical forms.
- KRISP (Kate and Mooney 2006) uses SVMs with string kernels to drive the local decisions of a chart-based semantic parser.
- WASP (Wong and Mooney 2006) uses log-linear synchronous grammars to transform utterances into logical forms, starting with word alignments obtained from the IBM models.
- λ-WASP (Wong and Mooney 2007) extends WASP to work with logical forms that contain bound variables (lambda abstraction).
- LNLZ08 (Lu et al. 2008) learns a generative model over **hybrid trees**, which are logical forms augmented with natural language words. IBM model 1 is used to initialize the parameters, and a discriminative reranking step works on top of the generative model.
- ZC05 (Zettlemoyer and Collins 2005) learns a discriminative log-linear model over CCG derivations. Starting with a manually constructed domain-independent lexicon, the training procedure grows the lexicon by adding lexical entries derived from associating parts of an utterance with parts of the annotated logical form.
- ZC07 (Zettlemoyer and Collins 2007) extends ZC05 with extra (disharmonic) combinators to increase the expressive power of the model.
- KZGS10 (Kwiatkowski et al. 2010) uses a restricted higher-order unification procedure, which iteratively breaks up a logical form into smaller pieces. This approach gradually adds lexical entries of increasing generality, thus obviating the need for the manually specified templates used by ZC05 and ZC07 for growing the lexicon. IBM model 1 is used to initialize the parameters.
- KZGS11 (Kwiatkowski et al. 2011) extends KZGS10 by factoring lexical entries into a template plus a sequence of predicates that fill the slots of the template. This factorization improves generalization.

With the exception of PRECISE, all other systems require annotated logical forms, whereas our system learns only from annotated answers. On the other hand, our system does rely on a few manually specified lexical triggers, whereas many of the later systems essentially require no manually crafted lexica. For us, the lexical triggers play a crucial role in the initial stages of learning because they constrain the set of candidate DCS trees; otherwise we would face a hopelessly intractable search problem. The other systems induce lexica using unsupervised word alignment (Wong and Mooney 2006, 2007; Kwiatkowski et al. 2010, 2011) and/or on-line lexicon learning (Zettlemoyer and Collins 2005, 2007; Kwiatkowski et al. 2010, 2011). Unfortunately, we cannot use these automatic techniques because they rely on having annotated logical forms.

Table 3 shows the results for GEO. Semantic parsers are typically evaluated on the accuracy of the logical forms: precision (the accuracy on utterances which are

Table 3
Results on GEO: Logical form accuracy (LF) and answer accuracy (Answer) of the various systems. The first group of systems are evaluated using 10-fold cross-validation on all 880 examples; the second are evaluated on the 680 + 200 split of Zettlemoyer and Collins (2005). Our system (LJK11) with base triggers obtains comparable accuracy to past work, whereas with augmented triggers, our system obtains the highest overall accuracy.

System		LF (%)	Answer (%)
COCKTAIL	(Tang and Mooney 2001)	79.4	–
PRECISE	(Popescu, Etzioni, and Kautz 2003)	77.5	77.5
SCISSOR	(Ge and Mooney 2005)	72.3	–
SILT	(Kate, Wong, and Mooney 2005)	54.1	–
KRISP	(Kate and Mooney 2006)	71.7	–
WASP	(Wong and Mooney 2006)	74.8	–
λ-WASP	(Wong and Mooney 2007)	86.6	–
LNLZ08	(Lu et al. 2008)	81.8	–
ZC05	(Zettlemoyer and Collins 2005)	79.3	–
ZC07	(Zettlemoyer and Collins 2007)	86.1	–
KZGS10	(Kwiatkowski et al. 2010)	88.2	88.9
KZGS11	(Kwiatkowski et al. 2010)	**88.6**	–
LJK11 w/base triggers	(Liang, Jordan, and Klein 2011)	–	87.9
LJK11 w/augmented triggers	(Liang, Jordan, and Klein 2011)	–	**91.4**

successfully parsed) and recall (the accuracy on all utterances). We only focus on recall (a lower bound on precision) and simply use the word *accuracy* to refer to recall.[15] Our system is evaluated only on answer accuracy because our model marginalizes out the latent logical form. All other systems are evaluated on the accuracy of logical forms. To calibrate, we also evaluated KZGS10 on answer accuracy and found that it was quite similar to its logical form accuracy (88.9% vs. 88.2%).[16] This does not imply that our system would necessarily have a high logical form accuracy because multiple logical forms can produce the same answer, and our system does not receive a training signal to tease them apart. Even with only base triggers, our system (LJK11) outperforms all but two of the systems, falling short of KZGS10 by only one percentage point (87.9% vs. 88.9%).[17] With augmented triggers, our system takes the lead (91.4% vs. 88.9%).

Table 4 shows the results for JOBS. The two learning-based systems (COCKTAIL and ZC05) are actually outperformed by PRECISE, which is able to use strong database type constraints. By exploiting this information and doing learning, we obtain the best results.

4.3 Empirical Properties

In this section, we try to gain intuition into properties of our approach. All experiments in this section were performed on random development splits. Throughout this section, "accuracy" means development test accuracy.

15 Our system produces a logical form for every utterance, and thus our precision is the same as our recall.
16 The 88.2% corresponds to 87.9% in Kwiatkowski et al. (2010). The difference is due to using a slightly newer version of the code.
17 The 87.9% and 91.4% correspond to 88.6% and 91.1% in Liang, Jordan, and Klein (2011). These differences are due to minor differences in the code.

Table 4
Results on JOBS: Both PRECISE and our system use database type constraints, which results in a decisive advantage over the other systems. In addition, LJK11 incorporates learning and therefore obtains the highest accuracies.

System		LF (%)	Answer (%)
COCKTAIL	(Tang and Mooney 2001)	79.4	–
PRECISE	(Popescu, Etzioni, and Kautz 2003)	**88.0**	88.0
ZC05	(Zettlemoyer and Collins 2005)	79.3	–
LJK11 w/base triggers	(Liang, Jordan, and Klein 2011)	–	90.7
LJK11 w/augmented triggers	(Liang, Jordan, and Klein 2011)	–	**95.0**

4.3.1 Error Analysis. To understand the type of errors our system makes, we examined one of the development runs, which had 34 errors on the test set. We classified these errors into the following categories (the number of errors in each category is shown in parentheses):

- Incorrect POS tags (8): GEO is out-of-domain for our POS tagger, so the tagger makes some basic errors that adversely affect the predicates that can be lexically triggered. For example, the question *What states border states ...* is tagged as WP VBZ NN NNS ..., which means that the first *states* cannot trigger `state`. In another example, *major river* is tagged as NNP NNP, so these cannot trigger the appropriate predicates either, and thus the desired DCS tree cannot even be constructed.
- Non-projectivity (3): The candidate DCS trees are defined by a projective construction mechanism (Section 2.6) that prohibits edges in the DCS tree from crossing. This means we cannot handle utterances such as *largest city by area*, because the desired DCS tree would have `city` dominating `area` dominating `argmax`. To construct this DCS tree, we could allow local reordering of the words.
- Unseen words (2): We never saw *at least* or *sea level* at training time. The former has the correct lexical trigger, but not a sufficiently large feature weight (0) to encourage its use. For the latter, the problem is more structural: We have no lexical triggers for 0:`length`, and only adding more lexical triggers can solve this problem.
- Wrong lexical triggers (7): Sometimes the error is localized to a single lexical trigger. For example, the model incorrectly thinks *Mississippi* is the state rather than the river, and that *Rochester* is the city in New York rather than the name, even though there are contextual cues to disambiguate in these cases.
- Extra words (5): Sometimes, words trigger predicates that should be ignored. For example, for *population density*, the first word triggers `population`, which is used rather than `density`.
- Over-smoothing of DCS tree (9): The first half of our features (Figure 20) are defined on the DCS tree alone; these produce a form of smoothing that encourages DCS trees to look alike regardless of the words. We found

> several instances where this essential tool for generalization went too far. For example, in *state of Nevada*, the trace predicate `border` is inserted between the two nouns, because it creates a structure more similar to that of the common question *what states border Nevada?*

4.3.2 Visualization of Features. Having analyzed the behavior of our system for individual utterances, let us move from the token level to the type level and analyze the learned parameters of our model. We do not look at raw feature weights, because there are complex interactions between them not represented by examining individual weights. Instead, we look at expected feature counts, which we think are more interpretable.

Consider a group of "competing" features J, for example $J = \{\text{TRIGGERPRED}[\textit{city}, p] : p \in \mathcal{P}\}$. We define a distribution $q(\cdot)$ over J as follows:

$$q(j) = \frac{N_j}{\sum_{j' \in J} N_{j'}}, \text{ where} \tag{78}$$

$$N_j = \sum_{(\mathbf{x},y)\in\mathcal{D}} \mathbb{E}_{p(z|\mathbf{x},\tilde{\mathcal{Z}}_{L,\theta},\theta)}[\phi(\mathbf{x},z)]$$

Think of $q(j)$ as a marginal distribution (because all our features are positive) that represents the relative frequencies with which the features $j \in J$ fire with respect to our training data set $\mathcal{D}$ and trained model $p(z \mid \mathbf{x}, \tilde{\mathcal{Z}}_{L,\theta}, \theta)$. To appreciate the difference between what this distribution and raw feature weights capture, suppose we had two features, j_1 and j_2, which are identical ($\phi(\mathbf{x},z)_{j_1} \equiv \phi(\mathbf{x},z)_{j_2}$). The weights would be split across the two features, but the features would have the same marginal distribution ($q(j_1) = q(j_2)$). Figure 23 shows some of the feature distributions learned.

4.3.3 Learning, Search, Bootstrapping. Recall from Section 3.2.1 that a training example is feasible (with respect to our beam search) if the resulting candidate set contains a DCS tree with the correct answer. Infeasible examples are skipped, but an example may become feasible in a later iteration. A natural question is how many training examples are feasible in each iteration. Figure 24 shows the answer: Initially, only around 30% of the training examples are feasible; this is not surprising given that all the parameters are zero, so our beam search is essentially unguided. Training on just these examples improves the parameters, however, and over the next few iterations, the number of feasible examples steadily increases to around 97%.

In our algorithm, learning and search are deeply intertwined. Search is of course needed to learn, but learning also improves search. The general approach is similar in spirit to Searn (Daume, Langford, and Marcu 2009), although we do not have any formal guarantees at this point.

Our algorithm also has a bootstrapping flavor. The "easy" examples are processed first, where easy is defined by the ability of beam search to generate the correct answer. This bootstrapping occurs quite naturally: Unlike most bootstrapping algorithms, we do not have to set a confidence threshold for accepting new training examples, something that can be quite tricky to do. Instead, our threshold falls out of the discrete nature of the beam search.

4.3.4 Effect of Various Settings. So far, we have used our approach with default settings (Section 4.1.2). How sensitive is the approach to these choices? Table 5 shows the impact of the feature templates. Figure 25 shows the effect of the number of training examples,

TriggerPred[*city*, ·]	
city	1.00
river	0.00
capital	0.00
...	...

TriggerPred[*peak*, ·]	
mountain	0.92
place	0.08
city	0.00
...	...

TriggerPred[*sparse*, ·]	
elevation	1.00
density	0.00
size	0.00
...	...

TracePred[*in*, ·, ·]	
loc ↘	0.99
traverse ↘	0.01
border ↘	0.00
...	...

TracePred[*have*, ·, ·]	
loc ↘	0.68
border ↘	0.20
traverse ↘	0.12
...	...

TracePred[*flow*, ·, ·]	
traverse ↘	0.71
border ↘	0.18
loc ↘	0.11
...	...

PredRelPred[·, ·, city]	
ø $x_{1,2}$	0.38
ø Σ	0.19
count ↘ $\frac{1}{1}$ ø Σ	0.19
...	...

PredRelPred[·, ·, loc]	
city ↘ $\frac{1}{1}$	0.25
state ↘ $\frac{1}{2}$	0.25
place ↘ $\frac{1}{1}$	0.17
...	...

PredRelPred[·, ·, elevation]	
place ↙ $\frac{1}{1}$	0.65
mountain ↙ $\frac{1}{1}$	0.27
ø $\frac{1}{2}$	0.08
...	...

Figure 23
Learned feature distributions. In a feature group (e.g., TriggerPred[*city*, ·]), each feature is associated with the marginal probability that the feature fires according to Equation (78). Note that we have successfully learned that *city* means `city`, but incorrectly learned that *sparse* means `elevation` (due to the confounding fact that Alaska is the most sparse state and has the highest elevation).

number of training iterations, beam size, and regularization parameter. The overall conclusion is that there are no big surprises: Our default settings could be improved on slightly, but these differences are often smaller than the variation across different development splits.

We now consider the choice of optimization algorithm to update the parameters given candidate sets (see Figure 21). Thus far, we have been using L-BFGS (Nocedal 1980), which is a batch algorithm. Each iteration, we construct the candidate

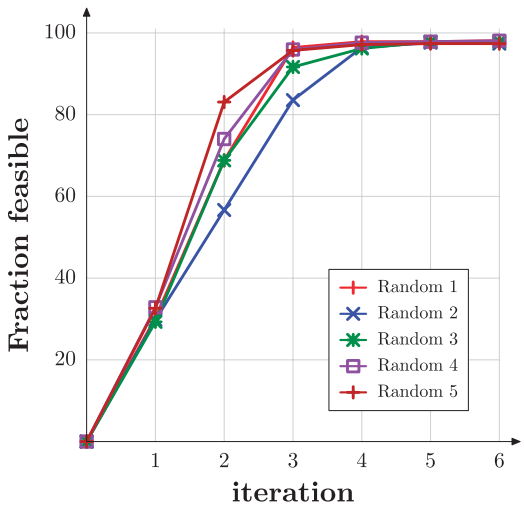


Figure 24
The fraction of feasible training examples increases steadily as the parameters, and thus the beam search improves. Each curve corresponds to a run on a different development split.

Table 5
There are two classes of feature templates: lexical features (TRIGGERPRED,TRACE*) and non-lexical features (PREDREL,PREDRELPRED). The lexical features are relatively much more important for obtaining good accuracy (76.4% vs. 23.1%), but adding the non-lexical features makes a significant contribution as well (84.7% vs. 76.4%).

Features	Accuracy (%)
PRED	13.4 ± 1.6
PRED + PREDREL	18.4 ± 3.5
PRED + PREDREL + PREDRELPRED	23.1 ± 5.0
PRED + TRIGGERPRED	61.3 ± 1.1
PRED + TRIGGERPRED + TRACE*	76.4 ± 2.3
PRED + PREDREL + PREDRELPRED + TRIGGERPRED + TRACE*	84.7 ± 3.5

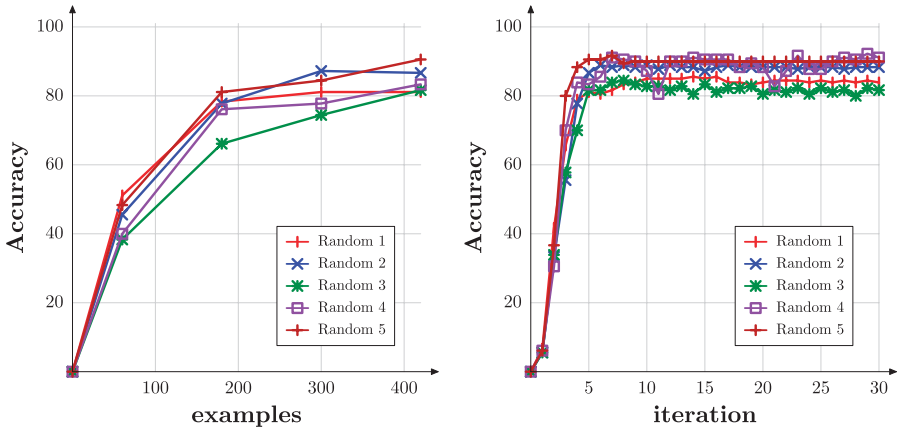
sets $C^{(t)}(\mathbf{x})$ for all the training examples before solving the optimization problem $\text{argmax}_\theta \mathcal{O}(\theta, C^{(t)})$. We now consider an on-line algorithm, stochastic gradient descent (SGD) (Robbins and Monro 1951), which updates the parameters after computing the candidate set for each example. In particular, we iteratively scan through the training examples in a random order. For each example $(\mathbf{x}, y)$, we compute the candidate set using beam search. We then update the parameters in the direction of the gradient of the marginal log-likelihood for that example (see Equation (72)) with step size $t^{-\alpha}$:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + t^{-\alpha} \left(\frac{\partial \log p(y \mid \mathbf{x}; \tilde{\mathcal{Z}}_{L,\theta^{(t)}}, \theta)}{\partial \theta} \Big|_{\theta=\theta^{(t)}} \right) \tag{79}$$

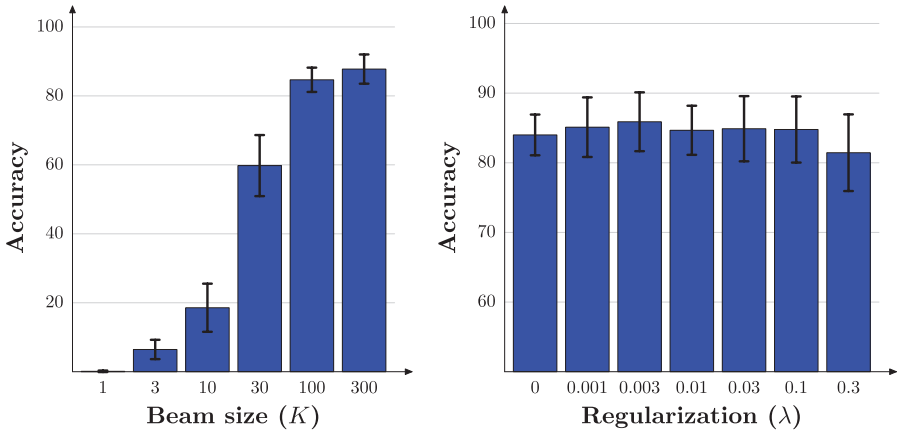
The trickiest aspect of using SGD is selecting the correct step size: A small α leads to quick progress but also instability; a large α leads to the opposite. We let L-BFGS and SGD both take the same number of iterations (passes over the training set). Figure 26 shows that a very small value of α (less than 0.2) is best for our task, even though only values between 0.5 and 1 guarantee convergence. Our setting is slightly different because we are interleaving the SGD updates with beam search, which might also lead to unpredictable consequences. Furthermore, the non-convexity of the objective function exacerbates the unpredictability (Liang and Klein 2009). Nonetheless, with a proper α, SGD converges much faster than L-BFGS and even to a slightly better solution.

5. Discussion

The work we have presented in this article addresses three important themes. The first theme is *semantic representation* (Section 5.1): How do we parametrize the mapping from utterances to their meanings? The second theme is *program induction* (Section 5.2): How do we efficiently search through the space of logical structures given a weak feedback signal? Finally, the last theme is *grounded language* (Section 5.3): How do we use constraints from the world to guide learning of language and conversely use language to interact with the world?



(a) Effect of number of training examples (b) Effect of number of training iterations

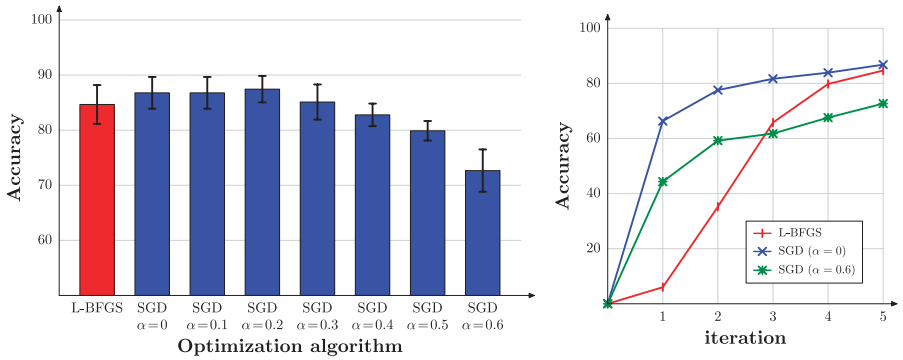


(c) Effect of beam size (d) Effect of regularization

Figure 25
(a) The learning curve shows test accuracy as the number of training examples increases; about 300 examples suffices to get around 80% accuracy. (b) Although our algorithm is not guaranteed to converge, the test accuracy is fairly stable (with one exception) with more training iterations—hardly any overfitting occurs. (c) As the beam size increases, the accuracy increases monotonically, although the computational burden also increases. There is a small gain from our default setting of $K = 100$ to the more expensive $K = 300$. (d) The accuracy is relatively insensitive to the choice of the regularization parameter for a wide range of values. In fact, no regularization is also acceptable. This is probably because the features are simple, and the lexical triggers and beam search already provide some helpful biases.

5.1 Semantic Representation

Since the late nineteenth century, philosophers and linguists have worked on elucidating the relationship between an utterance and its meaning. One of the pillars of formal semantics is Frege's principle of compositionality, that the meaning of an utterance is built by composing the meaning of its parts. What these parts are and how they are composed is the main question. The dominant paradigm, which stems from the seminal work of Richard Montague (1973) in the early 1970s, states that parts are lambda calculus expressions that correspond to syntactic constituents, and composition is function application.



(a) Effect of optimization algorithm (b) Batch versus online

Figure 26
(a) Given the same number of iterations, compared to default batch algorithm (L-BFGS), the on-line algorithm (stochastic gradient descent) is slightly better for aggressive step sizes (small α) and worse for conservative step sizes (large α). (b) The on-line algorithm (with an appropriate choice of α) obtains a reasonable accuracy much faster than L-BFGS.

Consider the compositionality principle from a statistical point of view, where we construe compositionality as factorization. Factorization, the way a statistical model breaks into features, is necessary for generalization: It enables us to learn from previously seen examples and interpret new utterances. Projecting back to Frege's original principle, the parts are the features (Section 3.1.1), and composition is the DCS construction mechanism (Section 2.6) driven by parameters learned from training examples.

Taking the statistical view of compositionality, finding a good semantic representation becomes designing a good statistical model. But statistical modeling must also deal with the additional issue of language acquisition or learning, which presents complications: In absorbing training examples, our learning algorithm must inevitably traverse through intermediate models that are wrong or incomplete. The algorithms must therefore tolerate this degradation, and do so in a computationally efficient way. For example, in the line of work on learning probabilistic CCGs (Zettlemoyer and Collins 2005, 2007; Kwiatkowski et al. 2010), many candidate lexical entries must be entertained for each word even when polysemy does not actually exist (Section 2.6.4).

To improve generalization, the lexicon can be further factorized (Kwiatkowski et al. 2011), but this is all done within the constraints of CCG. DCS represents a departure from this tradition, which replaces a heavily lexicalized constituency-based formalism with a lightly-lexicalized dependency-based formalism. We can think of DCS as a shift in linguistic coordinate systems, which makes certain factorizations or features more accessible. For example, we can define features on paths between predicates in a DCS tree which capture certain lexical patterns much more easily than in a lambda calculus expression or a CCG derivation.

DCS has a family resemblance to a semantic representation called *natural logic form* (Alshawi, Chang, and Ringgaard 2011), which is also motivated by the benefits of working with dependency-based logical forms. The goals and the detailed structure of the two semantic formalisms are different, however. Alshawi, Chang, and Ringgaard (2011) focus on parsing complex sentences in an open domain where a structured database or world does not exist. Whereas they do equip their logical forms with a full model-theoretic semantics, the logical forms are actually closer to dependency trees: Quantifier scope is left unspecified, and the predicates are simply the words.

Perhaps not immediately apparent is the fact that DCS draws an important idea from Discourse Representation Theory (DRT) (Kamp and Reyle 1993)—not from the treatment of anaphora and presupposition which it is known for, but something closer to its core. This is the idea of having a logical form where all variables are existentially quantified and constraints are combined via conjunction—a Discourse Representation Structure (DRS) in DRT, or a basic DCS tree with only join relations. Computationally, these logical structures conveniently encode CSPs. Linguistically, it appears that existential quantifiers play an important role and should be treated specially (Kamp and Reyle 1993). DCS takes this core and focuses on semantic compositionality and computation, whereas DRT focuses more on discourse and pragmatics.

In addition to the statistical view of DCS as a semantic representation, it is useful to think about DCS from the perspective of programming language design. Two programming languages can be equally expressive, but what matters is how simple it is to express a desired type of computation in a given language. In some sense, we designed the DCS formal language to make it easy to represent computations expressed by natural language. An important part of DCS is the mark–execute construct, a uniform framework for dealing with the divergence between syntactic and semantic scope. This construct allows us to build simple DCS tree structures and still handle the complexities of phenomena such as quantifier scope variation. Compared to lambda calculus, think of DCS as a higher-level programming language tailored to natural language, which results in simpler programs (DCS trees). Simpler programs are easier for us to work with and easier for an algorithm to learn.

5.2 Program Induction

Searching over the space of programs is challenging. This is the central computational challenge of program induction, that of inferring programs (logical forms) from their behavior (denotations). This problem has been tackled by different communities in various forms: program induction in AI, programming by demonstration in Human–Computer Interaction, and program synthesis in programming languages. The core computational difficulty is that the supervision signal—the behavior—is a complex function of the program that cannot be easily inverted. What program generated the output *Arizona*, *Nevada*, and *Oregon*?

Perhaps somewhat counterintuitively, program induction is easier if we infer programs for not a single task but for multiple tasks. The intuition is that when the tasks are related, the solution to one task can help another task, both computationally in navigating the program space and statistically in choosing the appropriate program if there are multiple feasible possibilities (Liang, Jordan, and Klein 2010). In our semantic parsing work, we want to infer a logical form for each utterance (task). Clearly the tasks are related because they use the same vocabulary to talk about the same domain.

Natural language also makes program induction easier by providing side information (words) which can be used to guide the search. There have been several papers that induce programs in this setting: Eisenstein et al. (2009) induce conjunctive formulae from natural language instructions, Piantadosi et al. (2008) induce first-order logic formulae using CCG in a small domain assuming observed lexical semantics, and Clarke et al. (2010) induce logical forms in semantic parsing. In the ideal case, the words would determine the program predicates, and the utterance would determine the entire program compositionally. But of course, this mapping is not given and must be learned.

5.3 Grounded Language

In recent years, there has been an increased interest in connecting language with the world.[18] One of the primary issues in grounded language is alignment—figuring out what fragments of utterances refer to what aspects of the world. In fact, semantic parsers trained on examples of utterances and annotated logical form (those discussed in Section 4.2.2) need to solve the task of aligning words to predicates. Some can learn from utterances paired with a set of logical forms, one of which is correct (Kate and Mooney 2007; Chen and Mooney 2008). Liang, Jordan, and Klein (2009) tackle the even more difficult alignment problem of segmenting and aligning a discourse to a database of facts, where many parts on either side are irrelevant.

If we know how the world relates to language, we can leverage structure in the world to guide the learning and interpretation of language. We saw that type constraints from the database/world reduce the set of candidate logical forms and lead to more accurate systems (Popescu, Etzioni, and Kautz 2003; Liang, Jordan, and Klein 2011). Even for syntactic parsing, information from the denotation of an utterance can be helpful (Schuler 2003).

One of the exciting aspects about using the world for learning language is that it opens the door to many new types of supervision. We can obtain answers given a world, which are cheaper to obtain than logical forms (Clarke et al. 2010; Liang, Jordan, and Klein 2011). Other researchers have also pushed in this direction in various ways: learning a semantic parser based on bootstrapping and estimating the confidence of its own predictions (Goldwasser et al. 2011), learning a semantic parser from user interactions with a dialog system (Artzi and Zettlemoyer 2011), and learning to execute natural language instructions from just a reward signal using reinforcement learning (Branavan et al. 2009; Branavan, Zettlemoyer, and Barzilay 2010; Branavan, Silver, and Barzilay 2011). In general, supervision from the world is indirectly related to the learning task, but it is often much more plentiful and natural to obtain.

The benefits can also flow from language to the world. For example, previous work learned to interpret language to troubleshoot a Windows machine (Branavan et al. 2009; Branavan, Zettlemoyer, and Barzilay 2010), win a game of Civilization (Branavan, Silver, and Barzilay 2011), play a legal game of solitaire (Eisenstein et al. 2009; Goldwasser and Roth 2011), and navigate a map by following directions (Vogel and Jurafsky 2010; Chen and Mooney 2011). Even when the objective in the world is defined independently of language (e.g., in Civilization), language can provide a useful bias towards the non-linguistic end goal.

6. Conclusions

The main conceptual contribution of this article is a new semantic formalism, dependency-based compositional semantics (DCS), and techniques to learn a semantic parser from question–answer pairs where the intermediate logical form (a DCS tree) is induced in an unsupervised manner. Our final question–answering system was able to match the accuracies of state-of-the-art systems that learn from annotated logical forms.

There is currently a significant conceptual gap between our question–answering system (which can be construed as a natural language interface to a database) and

18 Here, *world* need not refer to the physical world, but could be any virtual world. The point is that the world has non-trivial structure and exists extra-linguistically.

open-domain question–answering systems. The former focuses on understanding a question compositionally and computing the answer compositionally, whereas the latter focuses on retrieving and ranking answers from a large unstructured textual corpus. The former has depth; the latter has breadth. Developing methods that can both model the semantic richness of language and scale up to an open-domain setting remains an open challenge.

We believe that it is possible to push our approach in the open-domain direction. Neither DCS nor the learning algorithm is tied to having a clean rigid database, which could instead be a database generated from a noisy information extraction process. The key is to drive the learning with the desired behavior, the question–answer pairs. The latent variable is the logical form or program, which just tries to compute the desired answer by piecing together whatever information is available. Of course, there are many open challenges ahead, but with the proper combination of linguistic, statistical, and computational insight, we hope to eventually build systems with both breadth and depth.

Acknowledgments

We thank Luke Zettlemoyer and Tom Kwiatkowski for providing us with data and answering questions, as well as the anonymous reviewers for their detailed feedback. P. L. was supported by an NSF Graduate Research Fellowship.

References

Alshawi, H., P. Chang, and M. Ringgaard. 2011. Deterministic statistical mapping of sentences to underspecified semantics. In *International Conference on Compositional Semantics (IWCS)*, pages 15–24, Oxford.

Androutsopoulos, I., G. D. Ritchie, and P. Thanisch. 1995. Natural language interfaces to databases—an introduction. *Journal of Natural Language Engineering*, 1:29–81.

Artzi, Y. and L. Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 421–432, Edinburgh.

Baldridge, J. and G. M. Kruijff. 2002. Coupling CCG with hybrid logic dependency semantics. In *Association for Computational Linguistics (ACL)*, pages 319–326, Philadelphia, PA.

Barker, C. 2002. Continuations and the nature of quantification. *Natural Language Semantics*, 10:211–242.

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

Bos, J., S. Clark, M. Steedman, J. R. Curran, and J. Hockenmaier. 2004. Wide-coverage semantic representations from a CCG parser. In *International Conference on Computational Linguistics (COLING)*, pages 1240–1246, Geneva.

Branavan, S., 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)*, pages 82–90, Singapore.

Branavan, S., D. Silver, and R. Barzilay. 2011. Learning to win by reading manuals in a Monte-Carlo framework. In *Association for Computational Linguistics (ACL)*, pages 268–277.

Branavan, S., L. Zettlemoyer, and R. Barzilay. 2010. Reading between the lines: Learning to map high-level instructions to commands. In *Association for Computational Linguistics (ACL)*, pages 1268–1277, Portland, OR.

Carpenter, B. 1998. *Type-Logical Semantics*. MIT Press, Cambridge, MA.

Chen, D. L. and R. J. Mooney. 2008. Learning to sportscast: A test of grounded language acquisition. In *International Conference on Machine Learning (ICML)*, pages 128–135, Helsinki.

Chen, D. L. and R. J. Mooney. 2011. Learning to interpret natural language navigation instructions from observations. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 128–135, Cambridge, MA.

Clarke, J., D. Goldwasser, M. Chang, and D. Roth. 2010. Driving semantic parsing from the world's response. In *Computational Natural Language Learning (CoNLL)*, pages 18–27, Uppsala.

Collins, M. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

Cooper, R. 1975. *Montague's semantic theory and transformational syntax*. Ph.D. thesis, University of Massachusetts at Amherst.

Cousot, P. and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA.

Daume, H., J. Langford, and D. Marcu. 2009. Search-based structured prediction. *Machine Learning Journal (MLJ)*, 75:297–325.

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

Eisenstein, J., J. Clarke, D. Goldwasser, and D. Roth. 2009. Reading to learn: Constructing features from semantic abstracts. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 958–967, Singapore.

Ge, R. 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, MI.

Giordani, A. and A. Moschitti. 2009. Semantic mapping between natural language questions and SQL queries via syntactic pairing. In *International Conference on Applications of Natural Language to Information Systems*, pages 207–221, Saarbrücken.

Goldwasser, D., R. Reichart, J. Clarke, and D. Roth. 2011. Confidence driven unsupervised semantic parsing. In *Association for Computational Linguistics (ACL)*, pages 1486–1495, Barcelona.

Goldwasser, D. and D. Roth. 2011. Learning from natural instructions. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1794–1800, Portland, OR.

Heim, I. and A. Kratzer. 1998. *Semantics in Generative Grammar*. Wiley-Blackwell, Oxford.

Judge, J., A. Cahill, and J. v. Genabith. 2006. Question-bank: Creating a corpus of parse-annotated questions. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*, pages 497–504, Sydney.

Kamp, H. 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.

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

Kate, R. J. and R. J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*, pages 913–920, Sydney.

Kate, R. J. 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.

Kate, R. J., 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.

Kwiatkowski, T., 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)*, pages1223–1233, Cambridge, MA.

Kwiatkowski, T., L. Zettlemoyer, S. Goldwater, and M. Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1512–1523, Cambridge, MA.

Liang, P. 2011. *Learning Dependency-Based Compositional Semantics*. Ph.D. thesis, University of California at Berkeley.

Liang, P., 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)*, pages 91–99, Singapore.

Liang, P., M. I. Jordan, and D. Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*, pages 639–646, Haifa.

Liang, P., M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599, Portland, OR.

Liang, P. and D. Klein. 2009. Online EM for unsupervised models. In *North American Association for Computational Linguistics (NAACL)*, pages 611–619, Boulder, CO.

Lu, W., H. T. Ng, W. S. Lee, and L. S. Zettlemoyer. 2008. A generative model for parsing natural language to meaning representations. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 783–792, Honolulu, HI.

Marcus, M. P., M. A. Marcinkiewicz, and B. Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330.

Miller, S., D. Stallard, R. Bobrow, and R. Schwartz. 1996. A fully statistical approach to natural language interfaces. In *Association for Computational Linguistics (ACL)*, pages 55–61, Santa Cruz, CA.

Montague, R. 1973. The proper treatment of quantification in ordinary English. In J. Hiutikka, J. Moravcsik, and P. Suppes, editors, *Approaches to Natural Language*, pages 221–242, Dordrecht, The Netherlands.

Nocedal, J. 1980. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35:773–782.

Petrov, S., 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, Sydney.

Piantadosi, S. T., 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*, pages 1620–1625, Washington, DC.

Popescu, A., O. Etzioni, and H. Kautz. 2003. Towards a theory of natural language interfaces to databases. In *International Conference on Intelligent User Interfaces (IUI)*, pages 149–157, Miami, FL.

Porter, M. F. 1980. An algorithm for suffix stripping. *Program*, 14:130–137.

Robbins, H. and S. Monro. 1951. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407.

Schuler, W. 2003. Using model-theoretic semantic interpretation to guide statistical parsing and word recognition in a spoken language interface. In *Association for Computational Linguistics (ACL)*, pages 529–536, Sapporo.

Shan, C. 2004. Delimited continuations in natural language. Technical report, ArXiv. Available at `http://arvix.org/abs/cs.CL/0404006`.

Steedman, M. 2000. *The Syntactic Process*. MIT Press, Cambridge, MA.

Tang, L. R. 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, Freiburg.

Vogel, A. and D. Jurafsky. 2010. Learning to follow navigational directions. In *Association for Computational Linguistics (ACL)*, pages 806–814, Uppsala.

Wainwright, M. and M. I. Jordan. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1:1–307.

Warren, D. and F. Pereira. 1982. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8:110–122.

White, M. 2006. Efficient realization of coordinate structures in combinatory categorial grammar. *Research on Language and Computation*, 4:39–75.

Wong, Y. W. and R. J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *North American Association for Computational Linguistics (NAACL)*, pages 439–446, New York, NY.

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

Woods, W. A., R. M. Kaplan, and B. N. Webber. 1972. The lunar sciences natural language information system: Final report. Technical Report 2378, Bolt Beranek and Newman Inc., Cambridge, MA.

Zelle, M. and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1050–1055, Cambridge, MA.

Zettlemoyer, L. S. 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.

Zettlemoyer, L. S. 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, Prague.