Learning Compact Lexicons for CCG Semantic Parsing

Yoav Artzi[*]
Computer Science & Engineering
University of Washington
Seattle, WA 98195
yoav@cs.washington.edu

Dipanjan Das **Slav Petrov**
Google Inc.
76 9th Avenue
New York, NY 10011
{dipanjand,slav}@google.com

Abstract

We present methods to control the lexicon size when learning a Combinatory Categorial Grammar semantic parser. Existing methods incrementally expand the lexicon by greedily adding entries, considering a single training datapoint at a time. We propose using corpus-level statistics for lexicon learning decisions. We introduce *voting* to globally consider adding entries to the lexicon, and *pruning* to remove entries no longer required to explain the training data. Our methods result in state-of-the-art performance on the task of executing sequences of natural language instructions, achieving up to 25% error reduction, with lexicons that are up to 70% smaller and are qualitatively less noisy.

1 Introduction

Combinatory Categorial Grammar (Steedman, 1996, 2000, CCG, henceforth) is a commonly used formalism for semantic parsing – the task of mapping natural language sentences to formal meaning representations (Zelle and Mooney, 1996). Recently, CCG semantic parsers have been used for numerous language understanding tasks, including querying databases (Zettlemoyer and Collins, 2005), referring to physical objects (Matuszek et al., 2012), information extraction (Krishnamurthy and Mitchell, 2012), executing instructions (Artzi and Zettlemoyer, 2013b), generating regular expressions (Kushman and Barzilay, 2013), question-answering (Cai and Yates, 2013) and textual entailment (Lewis and Steedman, 2013). In CCG, a lexicon is used to map words to formal representations of their meaning, which are then combined using bottom-up operations. In this paper we present learning techniques to explicitly control the size of the CCG lexicon, and show that this results in improved task performance and more compact models.

chair $\vdash \boldsymbol{N : \lambda x.\mathrm{chair}(x)}$
chair $\vdash \boldsymbol{N : \lambda x.\mathrm{sofa}(x)}$
chair $\vdash AP : \lambda a.\mathrm{len}(a, 3)$
chair $\vdash NP : \mathcal{A}(\lambda x.\mathrm{corner}(x))$
chair $\vdash ADJ : \lambda x.\mathrm{hall}(x)$

Figure 1: Lexical entries for the word *chair* as learned with no corpus-level statistics. Our approach is able to correctly learn only the top two bolded entries.

In most approaches for inducing CCGs for semantic parsing, lexicon learning and parameter estimation are performed jointly in an online algorithm, as introduced by Zettlemoyer and Collins (2007). To induce the lexicon, words extracted from the training data are paired with CCG categories one sample at a time (for an overview of CCG, see §2). Joint approaches have the potential advantage that only entries participating in successful parses are added to the lexicon. However, new entries are added greedily and these decisions are never revisited at later stages. In practice, this often results in a large and noisy lexicon.

Figure 1 lists a sample of CCG lexical entries learned for the word *chair* with a greedy joint algorithm (Artzi and Zettlemoyer, 2013b). In the studied navigation domain, the word *chair* is often used to refer to chairs and sofas, as captured by the first two entries. However, the system also learns several spurious meanings: the third shows an erroneous usage of *chair* as an adverbial phrase describing action length, while the fourth treats it as a noun phrase and the fifth as an adjective. In contrast, our approach is able to correctly learn only the top two lexical entries.

We present a *batch* algorithm focused on controlling the size of the lexicon when learning CCG semantic parsers (§3). Because we make updates only after processing the entire training set, we

[*] This research was carried out at Google.

Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1273–1283,
October 25-29, 2014, Doha, Qatar. ©2014 Association for Computational Linguistics

can take corpus-wide statistics into account before each lexicon update. To explicitly control the size of the lexicon, we adopt two complementary strategies: *voting* and *pruning*. First, we consider the lexical evidence each sample provides as a vote towards potential entries. We describe two voting strategies for deciding which entries to add to the model lexicon (§4). Second, even though we use voting to only conservatively add new lexicon entries, we also prune existing entries if they are no longer necessary for parsing the training data. These steps are incorporated into the learning framework, allowing us to apply stricter criteria for lexicon expansion while maintaining a single learning algorithm.

We evaluate our approach on the robot navigation semantic parsing task (Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013b). Our experimental results show that we outperform previous state of the art on executing sequences of instructions, while learning significantly more compact lexicons (§6 and Table 3).

2 Task and Inference

To present our lexicon learning techniques, we focus on the task of executing natural language navigation instructions (Chen and Mooney, 2011). This domain captures some of the fundamental difficulties in recent semantic parsing problems. In particular, it requires learning from weakly-supervised data, rather than data annotated with full logical forms, and parsing sentences in a situated environment. Additionally, successful task completion requires interpreting and executing multiple instructions in sequence, requiring accurate models to avoid cascading errors. Although this overview centers around the aforementioned task, our methods are generalizable to any semantic parsing approach that relies on CCG.

We approach the navigation task as a situated semantic parsing problem, where the meaning of instructions is represented with lambda calculus expressions, which are then deterministically executed. Both the mapping of instructions to logical forms and their execution consider the current state of the world. This problem was recently addressed by Artzi and Zettlemoyer (2013b) and our experimental setup mirrors theirs. In this section, we provide a brief background on CCG and describe the task and our inference method.

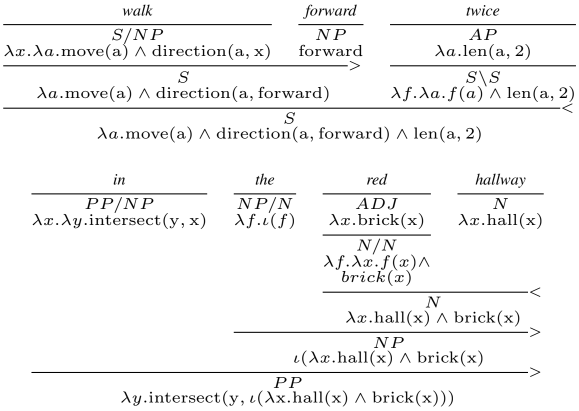

walk
forward
twice
S/NP
$\lambda x.\lambda a.\text{move(a)} \land \text{direction(a, x)}$
NP
forward
AP
$\lambda a.\text{len(a, 2)}$
>
S
$\lambda a.\text{move(a)} \land \text{direction(a, forward)}$
$S\backslash S$
$\lambda f.\lambda a.f(a) \land \text{len(a, 2)}$
<
S
$\lambda a.\text{move(a)} \land \text{direction(a, forward)} \land \text{len(a, 2)}$
in
the
red
hallway
PP/NP
$\lambda x.\lambda y.\text{intersect(y, x)}$
NP/N
$\lambda f.\iota(f)$
ADJ
$\lambda x.\text{brick(x)}$
N
$\lambda x.\text{hall(x)}$
N/N
$\lambda f.\lambda x.f(x) \land$
$brick(x)$
<
N
$\lambda x.\text{hall(x)} \land \text{brick(x)}$
>
NP
$\iota(\lambda x.\text{hall(x)} \land \text{brick(x)})$
>
PP
$\lambda y.\text{intersect(y, } \iota(\lambda \text{x.hall(x)} \land \text{brick(x)}))$

Figure 2: Two CCG parses. The top shows a complete parse with an adverbial phrase (AP), including unary type shifting and forward (>) and backward (<) application. The bottom fragment shows a prepositional phrase (PP) with an adjective (ADJ).

2.1 Combinatory Categorial Grammar

CCG is a linguistically-motivated categorial formalism for modeling a wide range of language phenomena (Steedman, 1996; Steedman, 2000). In CCG, parse tree nodes are categories, which are assigned to strings (single words or n-grams) and combined to create a complete derivation. For example, $S/NP : \lambda x.\lambda a.\text{move(a)} \land \text{direction(a, x)}$ is a CCG category describing an imperative verb phrase. The syntactic type S/NP indicates the category is expecting an argument of type NP on its right, and the returned category will have the syntax S. The directionality is indicated by the forward slash $/$, where a backward slash $\backslash$ would specify the argument is expected on the left. The logical form in the category represents its semantic meaning. For example, $\lambda x.\lambda a.\text{move(a)} \land \text{direction(a, x)}$ in the category above is a function expecting an argument, the variable x, and returning a function from events to truth-values, the semantic representation of imperatives. In this domain, the conjunction in the logical form specifies conditions on events. Specifically, the event must be a *move* event and have a specified *direction*.

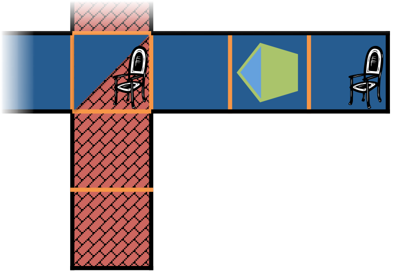
A CCG is defined by a lexicon and a set of combinators. The lexicon provides a mapping from strings to categories. Figure 2 shows two CCG parses in the navigation domain. Parse trees are read top to bottom. Parsing starts by matching categories to strings in the sentence using the lexicon. For example, the lexical entry *walk* $\vdash S/NP : \lambda x.\lambda a.\text{move(a)} \land \text{direction(a, x)}$ pairs the string *walk* with the example category above. Each intermediate parse node is constructed by applying one of a small set of binary CCG combinators or unary operators. For example, in Figure 2 the category of the span *walk forward* is combined with the category of *twice* using backward application ($<$). Parsing concludes with a logical form that captures the meaning of the complete sentence.

We adopt a factored representation for CCG lexicons (Kwiatkowski et al., 2011), where entries are dynamically generated by combining *lexemes* and *templates*. A lexeme is a pair that consists of a natural language string and a set of logical constants, while the template contains the syntactic and semantic components of a CCG category, abstracting over logical constants. For example, consider the lexical entry *walk* $\vdash S/NP : \lambda x.\lambda a.\mathrm{move(a)} \wedge \mathrm{direction(a, x)}$. Under the factored representation, this entry can be constructed by combining the lexeme $\langle$*walk*$, \{\mathrm{move}, \mathrm{direction}\}\rangle$ and the template $\lambda v_1.\lambda v_2.[S/NP : \lambda x.\lambda a.v_1(a) \wedge v_2(a, x)]$. This representation allows for better generalization over unseen lexical entries at inference time, allowing for pairings of templates and lexemes not seen during training.

2.2 Situated Log-Linear CCGs

We use a CCG to parse sentences to logical forms, which are then executed. Let $\mathcal{S}$ be a set of states, $\mathcal{X}$ be the set of all possible sentences, and $\mathcal{E}$ be the space of executions, which are $\mathcal{S} \rightarrow \mathcal{S}$ functions. For example, in the navigation task from Artzi and Zettlemoyer (2013b), $\mathcal{S}$ is a set of positions on a map, as illustrated in Figure 3. The map includes an agent that can perform four actions: LEFT, RIGHT, MOVE, and NULL. An execution e is a sequence of actions taken consecutively. Given a state $s \in \mathcal{S}$ and a sentence $x \in \mathcal{X}$, we aim to find the execution $e \in \mathcal{E}$ described in x. Let $\mathcal{Y}$ be the space of CCG parse trees and $\mathcal{Z}$ the space of all possible logical forms. Given a sentence x we generate a CCG parse $y \in \mathcal{Y}$, which includes a logical form $z \in \mathcal{Z}$. An execution e is then generated from z using a deterministic process.

Parsing with a CCG requires choosing appropriate lexical entries from an often ambiguous lexicon and the order in which operations are applied. In a situated scenario such choices must account for the current state of the world. In general, given a CCG, there are many parses for each sentence-state pair. To discriminate between competing parses, we use a situated log-linear CCG, inspired by Clark and Curran (2007).



facing the chair in the intersection move forward twice
$\lambda a.\mathrm{pre(a, front(you, \iota(\lambda x.chair(x) \wedge}$
$\mathrm{intersect(x, \iota(\lambda y.intersection(y)))))) \wedge}$
$\mathrm{move(a) \wedge len(a, 2)}$
$\langle\mathrm{FORWARD, FORWARD}\rangle$

turn left
$\lambda a.\mathrm{turn(a) \wedge direction(a, left)}$
$\langle\mathrm{LEFT}\rangle$

go to the end of the hall
$\lambda x.move(a) \wedge \mathrm{to(a, \iota(\lambda x.end(x, \iota(\lambda y.hall(y)))))}$
$\langle\mathrm{FORWARD, FORWARD}\rangle$

Figure 3: Fragment of a map and instructions for the navigation domain. The fragment includes two intersecting hallways (red and blue), two chairs and an agent facing left (green pentagon), which follows instructions such as these listed below. Each instruction is paired with a logical form representing its meaning and its execution in the map.

Let $\mathrm{GEN}(x, s; \Lambda) \subset \mathcal{Y}$ be the set of all possible CCG parses given the sentence x, the current state s and the lexicon Λ. In $\mathrm{GEN}(x, s; \Lambda)$, multiple parse trees may have the same logical form; let $\mathcal{Y}(z) \subset \mathrm{GEN}(x, s; \Lambda)$ be the subset of such parses with the logical form z at the root. Also, let $\theta \in \mathbb{R}^d$ be a d-dimensional parameter vector. We define the probability of the logical form z as:

$$p(z|x, s; \theta, \Lambda) = \sum_{y \in \mathcal{Y}(z)} p(y|x, s; \theta, \Lambda) \quad (1)$$

Above, we marginalize out the probabilities of all parse trees with the same logical form z at the root. The probability of a parse tree y is defined as:

$$p(y|x, s; \theta, \Lambda) = \frac{e^{\theta \cdot \phi(x,s,y)}}{\sum\limits_{y' \in \mathrm{GEN}(x,s;\Lambda)} e^{\theta \cdot \phi(x,s,y')}} \quad (2)$$

Where $\phi(x, s, y) \in \mathbb{R}^d$ is a feature vector. Given a logical form z, we deterministically map it to an execution $e \in \mathcal{E}$. At inference time, given a sentence x and state s, we find the best logical form z^* (and its corresponding execution) by solving:

$$z^* = \arg\max_z \; p(z|x, s; \theta, \Lambda) \quad (3)$$

The above $\arg\max$ operation sums over all trees $y \in \mathcal{Y}(z)$, as described in Equation 1. We use a CKY chart for this computation. The chart signature in each span is a CCG category. Since exact inference is prohibitively expensive, we follow previous work and perform bottom-up beam search, maintaining only the k-best categories for each span in the chart. The logical form z^* is taken from the k-best categories at the root of the chart. The partition function in Equation 2 is approximated by summing the inside scores of all categories at the root. We describe the choices of hyperparameters and details of our feature set in §5.

3 Learning

Learning a CCG semantic parser requires inducing the entries of the lexicon Λ and estimating parsing parameters θ. We describe a batch learning algorithm (Figure 4), which explicitly attempts to induce a compact lexicon, while fully explaining the training data. At training time, we assume access to a set of N examples $\mathcal{D} = \left\{d^{(i)}\right\}_1^N$, where each datapoint $d^{(i)} = \langle x^{(i)}, s^{(i)}, e^{(i)} \rangle$, consists of an instruction $x^{(i)}$, the state $s^{(i)}$ where the instruction is issued and its execution demonstration $e^{(i)}$. In particular, we know the correct execution for each state and instruction, but we do not know the correct CCG parse and logical form. We treat the choices that determine them, including selection of lexical entries and parsing operators, as latent. Since there can be many logical forms $z \in \mathcal{Z}$ that yield the same execution $e^{(i)}$, we marginalize over the logical forms (using Equation 1) when maximizing the following regularized log-likelihood:

$$\mathscr{L}(\theta, \Lambda, \mathcal{D}) = \sum_{d^{(i)} \in \mathcal{D}} \sum_{z \in \mathcal{Z}(e^{(i)})} p(z | x^{(i)}, s^{(i)}; \theta, \Lambda) - \frac{\gamma}{2} \|\theta\|_2^2 \tag{4}$$

Where $\mathcal{Z}(e^{(i)})$ is the set of logical forms that result in the execution $e^{(i)}$ and the hyperparameter γ is a regularization constant. Due to the large number of potential combinations,[1] it is impractical to consider the complete set of lexical entries, where all strings (single words and n-grams) are associated with all possible CCG categories. Therefore, similar to prior work, we gradually expand the lexicon during learning. As a result, the parameter space changes throughout training whenever the lexicon is modified. The learning problem involves jointly finding the best set of parameters and lexicon entries. In the remainder of this section, we describe how we optimize Equation 4, while explicitly controlling the lexicon size.

Algorithm 1 Batch algorithm for maximizing $\mathscr{L}(\theta, \Lambda, \mathcal{D})$. See §3.1 for details.

```
Input: Training dataset $\mathcal{D} = \left\{d^{(i)}\right\}_1^N$, number of learning
       iterations $T$, seed lexicon $\Lambda_0$, a regularization constant
       $\gamma$, and a learning rate $\mu$. VOTE is defined in §4.
Output: Lexicon $\Lambda$ and model parameters $\theta$
$\Lambda \leftarrow \Lambda_0$
for $t = 1$ to $T$ do
    » Generate lexical entries for all datapoints.
    for $i = 1$ to $N$ do
        $\lambda^{(i)} \leftarrow$ GENENTRIES($d^{(i)}, \theta, \Lambda$)
    » Add corpus-wide voted entries to model lexicon.
    $\Lambda \leftarrow \Lambda \cup$ VOTE($\Lambda, \{\lambda^{(1)}, \ldots, \lambda^{(N)}\}$)
    » Compute gradient and entries to prune.
    for $i = 1$ to $N$ do
        $\langle \lambda_-^{(i)}, \Delta^{(i)} \rangle \leftarrow$ COMPUTEUPDATE($d^{(i)}, \theta, \Lambda$)
    » Prune lexicon.
    $\Lambda \leftarrow \Lambda \setminus \bigcap_{i=1}^{N} \lambda_-^{(i)}$
    » Update model parameters.
    $\theta \leftarrow \theta + \mu \sum_{i=1}^{N} \Delta^{(i)} - \gamma\theta$
return $\Lambda$ and $\theta$
```

Algorithm 2 GENENTRIES: Algorithm to generate lexical entries from one training datapoint. See §3.2 for details.

```
Input: Single datapoint $d = \langle x, s, e \rangle$, current model parameters $\theta$ and lexicon $\Lambda$.
Output: Datapoint-specific lexicon entries $\lambda$.
» Augment lexicon with sentence-specific entries.
$\Lambda_+ \leftarrow \Lambda \cup$ GENLEX($d, \Lambda, \theta$)
» Get max-scoring parses producing correct execution.
$\boldsymbol{y}_+ \leftarrow$ GENMAX($x, s, e; \Lambda_+, \theta$)
» Extract lexicon entries from max-scoring parses.
$\lambda \leftarrow \bigcup_{y \in \boldsymbol{y}_+}$ LEX($y$)
return $\lambda$
```

Algorithm 3 COMPUTEUPDATE: Algorithm to compute the gradient and the set of lexical entries to prune for one datapoint. See §3.3 for details.

```
Input: Single datapoint $d = \langle x, s, e \rangle$, current model parameters $\theta$ and lexicon $\Lambda$.
Output: $\langle \lambda_-, \Delta \rangle$, lexical entries to prune for $d$ and gradient.
» Get max-scoring correct parses given $\Lambda$ and $\theta$.
$\boldsymbol{y}_+ \leftarrow$ GENMAX($x, s, e; \Lambda, \theta$)
» Create the set of entries to prune.
$\lambda_- \leftarrow \Lambda \setminus \bigcup_{y \in \boldsymbol{y}_+}$ LEX($y$)
» Compute gradient.
$\Delta \leftarrow \mathbb{E}(y \mid x, s, e; \theta, \Lambda) - \mathbb{E}(y \mid x, s; \theta, \Lambda)$
return $\langle \lambda_-, \Delta \rangle$
```

Figure 4: Our learning algorithm and its subroutines.

[1] For the navigation task, given the set of CCG category templates (see §2.1) and parameters used there would be between 7.5-10.2M lexical entries to consider, depending on the corpus used (§5).

3.1 Optimization Algorithm

We present a learning algorithm to optimize the data log-likelihood, where both lexicon learning and parameter updates are performed in *batch*, i.e., after observing all the training corpus. The batch formulation enables us to use information from the entire training set when updating the model lexicon. Algorithm 1 presents the outline of our optimization procedure. It takes as input a training dataset $\mathcal{D}$, number of iterations T, seed lexicon Λ_0, learning rate μ and regularization constant γ.

Learning starts with initializing the model lexicon Λ using Λ_0 (line 1). In lines 2-9, we run T iterations; in each, we make two passes over the corpus, first to generate lexical entries, and second to compute gradient updates and lexical entries to prune. To generate lexical entries (lines 3-4) we use the subroutine GENENTRIES to independently generate entries for each datapoint, as described in §3.2. Given the entries for each datapoint, we vote on which to add to the model lexicon. The subroutine VOTE (line 5) chooses a subset of the proposed entries using a particular voting strategy (see §4). Given the updated lexicon, we process the corpus a second time (lines 6-7). The subroutine COMPUTEUPDATE, as described in §3.3, computes the gradient update for each datapoint $d^{(i)}$, and also generates the set of lexical entries not included in the max-scoring parses of $d^{(i)}$, which are candidates for pruning. We prune from the model lexicon all lexical entries not used in any correct parse (line 8). During this pruning step, we ensure that no entries from Λ_0 are removed from Λ. Finally, the gradient updates are accumulated to update the model parameters (line 9).

3.2 Lexical Entries Generation

For each datapoint $d = \langle x, s, e \rangle$, the subroutine GENENTRIES, as described in Algorithm 2, generates a set of potential entries. The subroutine uses the function GENLEX, originally proposed by Zettlemoyer and Collins (2005), to generate lexical entries from sentences paired with logical forms. We use the weakly-supervised variant of Artzi and Zettlemoyer (2013b). Briefly, GENLEX uses the sentence and expected execution to generate new lexemes, which are then paired with a set of templates factored from Λ_0 to generate new lexical entries. For more details, see §8 of Artzi and Zettlemoyer (2013b).

Since GENLEX over-generates entries, we need to determine the set of entries that participate in max-scoring parses that lead to the correct execution e. We therefore create a sentence-specific lexicon Λ_+ by taking the union of the GENLEX-generated entries for the current sentence and the model lexicon (line 1). We define GENMAX$(x, s, e; \Lambda_+, \theta)$ to be the set of all max-scoring parses according to the parameters θ that are in GEN$(x, s; \Lambda_+)$ and result in the correct execution e (line 2). In line 3 we use the function LEX(y), which returns the lexical entries used in the parse y, to compute the set of all lexical entries used in these parses. This final set contains all newly generated entries for this datapoint and is returned to the optimization algorithm.

3.3 Pruning and Gradient Computation

Algorithm 3 describes the subroutine COMPUTEUPDATE that, given a datapoint d, the current model lexicon Λ and model parameters θ, returns the gradient update and the set of lexical entries to prune for d. First, similar to GENENTRIES we compute the set of correct max-scoring parses using GENMAX (line 1). This time, however, we do not use a sentence-specific lexicon, but instead use the model lexicon that has been expanded with all voted entries. As a result, the set of max-scoring parses producing the correct execution may be different compared to GENENTRIES. LEX(y) is then used to extract the lexical entries from these parses, and the set difference (λ_-) between the model lexicon and these entries is set to be pruned (line 2). Finally, the partial derivative for the datapoint is computed using the difference of two expected feature vectors, according to two distributions (line 3): (a) parses conditioned on the correct execution e, the sentence x, state s and the model, and (b) all parses not conditioned on the execution e. The derivatives are approximate due to the use of beam search, as described in §2.2.

4 Global Voting for Lexicon Learning

Our goal is to learn compact and accurate CCG lexicons. To this end, we globally reason about adding new entries to the lexicon by *voting* (VOTE, Algorithm 1, line 5), and remove entries by *pruning* the ones no longer required for explaining the training data (Algorithm 1, line 8). In voting, each datapoint can be considered as attempting to influence the learning algorithm to update the model lexicon with the entries required to parse it. In this

	Round 1	Round 2	Round 3	Round 4
$d^{(1)}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{3}$ $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$ $\frac{1}{3}$ $\langle \textit{chair}, \{\mathrm{turn}, \mathrm{direction}\}\rangle$ $\frac{1}{3}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$ $\frac{1}{2}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ 1	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ 1
$d^{(2)}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$ $\frac{1}{2}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$ $\frac{1}{2}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ 1	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ 1
$d^{(3)}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ $\frac{1}{2}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ $\frac{1}{2}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ $\frac{1}{2}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ 1
$d^{(4)}$	$\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ 1	$\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ 1	$\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ 1	$\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ 1
Votes	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $1\frac{1}{3}$ $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ $1\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$ $\frac{5}{6}$ $\langle \textit{chair}, \{\mathrm{turn}, \mathrm{direction}\}\rangle$ $\frac{1}{3}$	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $1\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ $1\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$ 1	$\langle \textit{chair}, \{\mathrm{chair}\}\rangle$ $2\frac{1}{2}$ $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ $1\frac{1}{2}$	$\boldsymbol{\langle \textit{chair}, \{\mathbf{chair}\}\rangle}$ 3 $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$ 1
Discard	$\langle \textit{chair}, \{\mathrm{turn}, \mathrm{direction}\}\rangle$	$\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$	$\langle \textit{chair}, \{\mathrm{easel}\}\rangle$	

Figure 5: Four rounds of ConsensusVote for the string *chair* for four training datapoints. For each datapoint, we specify the set of lexemes generated in the Round 1 column, and update this set after each round. At the end, the highest voted new lexeme according to the final votes is returned. In this example, MaxVote and ConsensusVote lead to different outcomes. MaxVote, based on the initial sets only, will select $\langle \textit{chair}, \{\mathrm{easel}\}\rangle$.

section we describe two alternative voting strategies. Both strategies ensure that new entries are only added when they have wide support in the training data, but count this support in different ways. For reproducibility, we also provide step-by-step pseudocode for both methods in the supplementary material.

Since we only have access to executions and treat parse trees as latent, we consider as correct all parses that produce correct executions. Frequently, however, incorrect parses spuriously lead to correct executions. Lexical entries extracted from such spurious parses generalize poorly. The goal of voting is to eliminate such entries.

Voting is formulated on the factored lexicon representation, where each lexical entry is factored into a lexeme and a template, as described in §2.1. Each lexeme is a pair containing a natural language string and a set of logical constants.[2] A lexeme is combined with a template to create a lexical entry. In our lexicon learning approach only new lexemes are generated, while the set of templates is fixed; hence, our voting strategies reason over lexemes and only create complete lexicon entries at the end. Decisions are made for each string independently of all other strings, but considering all occurrences of that string in the training data.

In lines 3-4 of Algorithm 1 GenEntries is used to propose new lexical entries for each training datapoint $d^{(i)}$. For each $d^{(i)}$ a set $\lambda^{(i)}$, that includes all lexical entries participating in parses that lead to the correct execution, is generated. In these sets, the same string can appear in multiple lexemes. To normalize its influence, each datapoint is given a vote of 1.0 for each string, which is distributed uniformly among all lexemes containing the same string.

For example, a specific $\lambda^{(i)}$ may consist of the following three lexemes: $\langle \textit{chair}, \{\mathrm{chair}\}\rangle$, $\langle \textit{chair}, \{\mathrm{hatrack}\}\rangle$, $\langle \textit{face}, \{\mathrm{post}, \mathrm{front}, \mathrm{you}\}\rangle$. In this set, the phrase *chair* has two possible meanings, which will therefore each receive a vote of 0.5, while the third lexeme will be given a vote of 1.0. Such ambiguity is common and occurs when the available supervision is insufficient to discriminate between different parses, for example, if they lead to identical executions.

Each of the two following strategies reasons over these votes to globally select the best lexemes. To avoid polluting the model lexicon, both strategies adopt a conservative approach and only select at most one lexeme for each string in each training iteration.

4.1 Strategy 1: MaxVote

The first strategy for selecting voted lexical entries is straightforward. For each string it simply aggregates all votes and selects the new lexeme with the most votes. A lexeme is considered new if it is not already in the model lexicon. If no such single lexeme exists (e.g., no new entries were used in correctly executing parses or in the case of a tie) no lexeme is selected in this iteration.

A potential limitation of MaxVote is that the votes for all rejected lexemes are lost. However, it is often reasonable to re-allocate these votes to other lexemes. For example, consider the sets of lexemes for the word *chair* in the Round 1 col-

[2] Recall, for example, that in one lexeme the string *walk* may be paired with the set of constants $\{\mathrm{move}, \mathrm{direction}\}$.

umn of Figure 5. Using MAXVOTE on these sets will select the lexeme $\langle \textit{chair}, \{\text{easel}\}\rangle$, rather than the correct lexeme $\langle \textit{chair}, \{\text{chair}\}\rangle$. This occurs when the datapoints supporting the correct lexeme distribute their votes over many spurious lexemes.

4.2 Strategy 2: CONSENSUSVOTE

Our second strategy CONSENSUSVOTE aims to capture the votes that are lost in MAXVOTE. Instead of discarding votes that do not go to the maximum scoring lexeme, voting is done in several rounds. In each round the lowest scoring lexeme is discarded and votes are re-assigned uniformly to the remaining lexemes. This procedure is continued until convergence. Finally, given the sets of lexemes in the last round, the votes are computed and the new lexeme with most votes is selected.

Figure 5 shows a complete voting process for four training datapoints. In each round, votes are aggregated over the four sets of lexemes, and the lexeme with the fewest votes is discarded. For each set of lexemes, the discarded lexeme is removed, unless it will lead to an empty set.[3] In the example, while $\langle \textit{chair}, \{\text{easel}\}\rangle$ is discarded in Round 3, it remains in the set of $d^{(4)}$. The process converges in the fourth round, when there are no more lexemes to discard. The final sets include two entries: $\langle \textit{chair}, \{\text{chair}\}\rangle$ and $\langle \textit{chair}, \{\text{easel}\}\rangle$. By avoiding wasting votes on lexemes that have no chance of being selected, the more widely supported lexeme $\langle \textit{chair}, \{\text{chair}\}\rangle$ receives the most votes, in contrast to Round 1, where $\langle \textit{chair}, \{\text{easel}\}\rangle$ was the highest voted one.

5 Experimental Setup

To isolate the effect of our lexicon learning techniques we closely follow the experimental setup of previous work (Artzi and Zettlemoyer, 2013b, §9) and use its publicly available code.[4] This includes the provided beam-search CKY parser, two-pass parsing for testing, beam search for executing sequences of instructions and the same seed lexicon, weight initialization and features. Finally, except the optimization parameters specified below, we use the same parameter settings.

Data For evaluation we use two related corpora: SAIL (Chen and Mooney, 2011) and ORACLE (Artzi and Zettlemoyer, 2013b). Due to how the original data was collected (MacMahon et al., 2006), SAIL includes many wrong executions and about 30% of all instruction sequences are infeasible (e.g., instructing the agent to walk into a wall). To better understand system performance and the effect of noise, ORACLE was created with the subset of valid instructions from SAIL paired with their gold executions. Following previous work, we use a held-out set for the ORACLE corpus and cross-validation for the SAIL corpus.

Systems We report two baselines. Our batch baseline uses the same regularized algorithm, but updates the lexicon by adding all entries without voting and skips pruning. Additionally, we added post-hoc pruning to the algorithm of Artzi and Zettlemoyer (2013b) by discarding all learned entries that are not participating in max-scoring correct parses at the end of training. For ablation, we study the influence of the two voting strategies and pruning, while keeping the same regularization setting. Finally, we compare our approach to previous published results on both corpora.

Optimization Parameters We optimized the learning parameters using cross validation on the training data to maximize recall of complete sequence execution and minimize lexicon size. We use 10 training iterations and the learning rate $\mu = 0.1$. For SAIL we set the regularization parameter $\gamma = 1.0$ and for ORACLE $\gamma = 0.5$.

Full Sequence Inference To execute sequences of instructions we use the beam search procedure of Artzi and Zettlemoyer (2013b) with an identical beam size of 10. The beam stores states, and is initialized with the starting state. Instructions are executed in order, each is attempted from all states currently in the beam, the beam is then updated and pruned to keep the 10-best states. At the end, the best scoring state in the beam is returned.

Evaluation Metrics We evaluate the end-to-end task of executing complete sequences of instructions against an oracle final state. In addition, to better understand the results, we also measure task completion for single instructions. We repeated

[3] This restriction is meant to ensure that discarding lexemes will not change the set of sentences that can be parsed. In addition, it means that the total amount of votes given to a string is invariant between rounds. Allowing for empty sets will change the sum of votes, and therefore decrease the number of datapoints contributing to the decision.

[4] Their implementation, based on the University of Washington Semantic Parsing Framework (Artzi and Zettlemoyer, 2013a), is available at `http://yoavartzi.com/navi`.

ORACLE corpus cross-validation	Single sentence			Sequence			Lexicon size
	P	R	F1	P	R	F1	
Artzi and Zettlemoyer (2013b)	84.59	82.74	83.65	68.35	58.95	63.26	5383
w/ post-hoc pruning	84.32	82.89	83.60	66.83	61.23	63.88	3104
Batch baseline	85.14	81.91	83.52	72.64	60.13	65.76	6323
w/ MaxVote	84.04	82.25	83.14	72.79	64.86	68.55	2588
w/ ConsensusVote	84.51	82.23	83.36	72.99	63.45	67.84	2446
w/ pruning	**85.58**	**83.51**	**84.53**	75.15	65.97	70.19	2791
w/ MaxVote + pruning	84.50	82.89	83.69	72.91	**66.40**	69.47	2186
w/ ConsensusVote + pruning	85.22	83.00	84.10	**75.65**	66.15	**70.55**	**2101**

Table 1: Ablation study using cross-validation on the ORACLE corpus training data. We report mean precision (P), recall (R) and harmonic mean (F1) of execution accuracy on single sentences and sequences of instructions and mean lexicon sizes. Bold numbers represent the best performing method on a given metric.

Final results		Single sentence			Sequence			Lexicon size
		P	R	F1	P	R	F1	
SAIL	Chen and Mooney (2011)		54.40			16.18		
	Chen (2012)		57.28			19.18		
	+ additional data		57.62			20.64		
	Kim and Mooney (2012)		57.22			20.17		
	Kim and Mooney (2013)		62.81			26.57		
	Artzi and Zettlemoyer (2013b)	**67.60**	**65.28**	**66.42**	38.06	31.93	34.72	10051
	Our Approach	66.67	64.36	65.49	**41.30**	**35.44**	**38.14**	**2873**
ORACLE	Artzi and Zettlemoyer (2013b)	**81.17** (0.68)	**78.63** (0.84)	**79.88** (0.76)	68.07 (2.72)	58.05 (3.12)	62.65 (2.91)	6213 (217)
	Our Approach	79.86 (0.50)	77.87 (0.41)	78.85 (0.45)	**76.05** (1.79)	**68.53** (1.76)	**72.10** (1.77)	**2365** (57)

Table 2: Our final results compared to previous work on the SAIL and ORACLE corpora. We report mean precision (P), recall (R), harmonic mean (F1) and lexicon size results and standard deviation between runs (in parenthesis) when appropriate. *Our Approach* stands for batch learning with a consensus voting and pruning. Bold numbers represent the best performing method on a given metric.

each experiment five times and report mean precision, recall,[5] harmonic mean (F1) and lexicon size. For held-out test results we also report standard deviation. For the baseline online experiments we shuffled the training data between runs.

6 Results

Table 1 shows ablation results for 5-fold cross-validation on the ORACLE training data. We evaluate against the online learning algorithm of Artzi and Zettlemoyer (2013b), an extension of it to include post-hoc pruning and a batch baseline. Our best sequence execution development result is obtained with ConsensusVote and pruning. The results provide a few insights. First, simply switching to batch learning provides mixed results: precision increases, but recall drops and the learned lexicon is larger. Second, adding pruning results in a much smaller lexicon, and, especially in batch learning, boosts performance. Adding voting further reduces the lexicon size and provides additional gains for sequence execution. Finally, while MaxVote and ConsensusVote give comparable performance on their own, ConsensusVote results in more precise and compact models when combined with pruning.

Table 2 lists our test results. We significantly outperform previous state of the art on both corpora when evaluating sequence accuracy. In both scenarios our lexicon is 60-70% smaller. In contrast to the development results, single sentence performance decreases slightly compared to Artzi and Zettlemoyer (2013b). The discrepancy between single sentence and sequence results might be due to the beam search performed when executing sequences of instructions. Models with more compact lexicons generate fewer logical forms for each sentence: we see a decrease of roughly 40% in our models compared to Artzi and Zettlemoyer (2013b). This is especially helpful during sequence execution, where we use a beam size of 10, resulting in better sequences of executions. In general, this shows the potential benefit of using more compact models in scenarios that incorporate reasoning about parsing uncertainty.

To illustrate the types of errors avoided with voting and pruning, Table 3 describes common error classes and shows example lexical entries for batch trained models with ConsensusVote and pruning and without. Quantitatively, the mean number of entries per string on development folds

[5] Recall is identical to accuracy as reported in prior work.

String	# lexical entries		Example categories
	Batch baseline	With voting and pruning	
The algorithm often treats common bigrams as multiword phrases, and later learns the more general separate entries. Without pruning the initial entries remain in the lexicon and compete with the correct ones during inference.			
octagon carpet	45	0	~~$N : \lambda x.\text{wall}(x)$~~ ~~$N : \lambda x.\text{hall}(x)$~~ ~~$N : \lambda x.\text{honeycomb}(x)$~~
carpet	51	5	$N : \lambda x.\text{hall}(x)$ ~~$N/N : \lambda f.\lambda x.x == \text{argmin}(f, \lambda y.\text{dist}(y))$~~
octagon	21	5	$N : \lambda x.\text{honeycomb}(x)$ ~~$N : \lambda x.\text{cement}(x)$~~ $ADJ : \lambda x.\text{honeycomb}(x)$
We commonly see in the lexicon a long tail of erroneous entries, which compete with correctly learned ones. With voting and pruning we are often able to avoid such noisy entries. However, some noise still exists, e.g., the entry for "intersection".			
intersection	45	7	$N : \lambda x.\text{intersection}(x)$ ~~$S\backslash N : \lambda f.\text{intersect}(\text{you}, (f))$~~ $AP : \lambda a.len(a, 1)$ ~~$N/NP : \lambda x.\lambda y.\text{intersect}(y, x)$~~
twice	46	2	$AP : \lambda a.\text{len}(a, 2)$ ~~$AP : \lambda a.\text{pass}(a, \mathcal{A}(\lambda x.\text{empty}(x)))$~~ ~~$AP : \lambda a.\text{pass}(a, \mathcal{A}(\lambda x.\text{hall}(x)))$~~
stone	31	5	$ADJ : \lambda x.\text{stone}(x)$ ~~$ADJ : \lambda x.\text{brick}(x)$~~ ~~$ADJ : \lambda x.\text{honeycomb}(x)$~~ ~~$NP/N : \lambda f.\mathcal{A}(f)$~~
Not all concepts mentioned in the corpus are relevant to the task and some of these are not semantically modeled. However, the baseline learner doesn't make this distinction and induces many erroneous entries. With voting the model better handles such cases, either by pairing such words with semantically empty entries or learning no entries for them. During inference the system can then easily skip such words.			
now	28	0	~~$AP : \lambda a.\text{len}(a, 3)$~~ ~~$AP : \lambda a.\text{direction}(a, \text{forward})$~~
only	38	0	~~$N/NP : \lambda x.\lambda y.\text{intersect}(y, x)$~~ ~~$N/NP : \lambda x.\lambda y.\text{front}(y, x)$~~
here	31	8	~~$NP : \text{you}$~~ $S/S : \lambda x.x$ ~~$S\backslash N : \lambda f.\text{intersect}(\text{you}, \mathcal{A}(f))$~~
Without pruning the learner often over-splits multiword phrases and has no way to reverse such decisions.			
coat	25	0	~~$N : \lambda x.\text{intersection}(x)$~~ ~~$ADJ : \lambda x.\text{hatrack}(x)$~~
rack	45	0	~~$N : \lambda x.\text{hatrack}(x)$~~ ~~$N : \lambda x.\text{furniture}(x)$~~
coat rack	55	5	$N : \lambda x.\text{hatrack}(x)$ ~~$N : \lambda x.\text{wall}(x)$~~ ~~$N : \lambda x.\text{furniture}(x)$~~
Voting helps to avoid learning entries for rare words when the learning signal is highly ambiguous.			
orange	20	0	~~$N : \lambda x.\text{cement}(x)$~~ ~~$N : \lambda x.\text{grass}(x)$~~
pics of towers	26	0	~~$N\lambda x.\text{intersection}(x)$~~ ~~$N : \lambda x.\text{hall}(x)$~~

Table 3: Example entries from a learned ORACLE corpus lexicon using batch learning. For each string we report the number of lexical entries without voting (CONSENSUSVOTE) and pruning and with, and provide a few examples. Struck entries were successfully avoided when using voting and pruning.

decreases from 16.77 for online training to 8.11.

Finally, the total computational cost of our approach is roughly equivalent to online approaches. In both approaches, each pass over the data makes the same number of inference calls, and in practice, Artzi and Zettlemoyer (2013b) used 6-8 iterations for online learning while we used 10. A benefit of the batch method is its insensitivity to data ordering, as expressed by the lower standard deviation between randomized runs in Table 2.[6]

7 Related Work

There has been significant work on learning for semantic parsing. The majority of approaches treat grammar induction and parameter estimation separately, e.g. Wong and Mooney (2006), Kate and Mooney (2006), Clarke et al. (2010), Goldwasser et al. (2011), Goldwasser and Roth (2011), Liang et al. (2011), Chen and Mooney (2011), and Chen (2012). In all these approaches the grammar structure is fixed prior to parameter estimation.

Zettlemoyer and Collins (2005) proposed the learning regime most related to ours. Their learner alternates between batch lexical induction and online parameter estimation. Our learning algorithm design combines aspects of previously studied approaches into a batch method, including gradient updates (Kwiatkowski et al., 2010) and using weak supervision (Artzi and Zettlemoyer, 2011). In contrast, Artzi and Zettlemoyer (2013b) use online perceptron-style updates to optimize a margin-based loss. Our work also focuses on CCG lexicon induction but differs in the use of corpus-level statistics through voting and pruning for explicitly controlling the size of the lexicon.

Our approach is also related to the grammar induction algorithm introduced by Carroll and Char-

[6] Results still vary slightly due to multi-threading.

niak (1992). Similar to our method, they process the data using two batch steps: the first proposes grammar rules, analogous to our step that generates lexical entries, and the second estimates parsing parameters. Both methods use pruning after each iteration, to remove unused entries in our approach, and low probability rules in theirs. However, while we use global voting to add entries to the lexicon, they simply introduce all the rules generated by the first step. Their approach also relies on using disjoint subsets of the data for the two steps, while we use the entire corpus.

Using voting to aggregate evidence has been studied for combining decisions from an ensemble of classifiers (Ho et al., 1994; Van Erp and Schomaker, 2000). MAXVOTE is related to approval voting (Brams and Fishburn, 1978), where voters are required to mark if they approve each candidate or not. CONSENSUSVOTE combines ideas from approval voting, Borda counting, and instant-runoff voting. Van Hasselt (2011) described all three systems and applied them to policy summation in reinforcement learning.

8 Conclusion

We considered the problem of learning for semantic parsing, and presented voting and pruning methods based on corpus-level statistics for inducing compact CCG lexicons. We incorporated these techniques into a batch modification of an existing learning approach for joint lexicon induction and parameter estimation. Our evaluation demonstrates that both voting and pruning contribute towards learning a compact lexicon and illustrates the effect of lexicon quality on task performance.

In the future, we wish to study various aspects of learning more robust lexicons. For example, in our current approach, words not appearing in the training set are treated as unknown and ignored at inference time. We would like to study the benefit of using large amounts of unlabeled text to allow the model to better hypothesize the meaning of such previously unseen words. Moreover, our model's performance is currently sensitive to the set of seed lexical templates provided. While we are able to learn the meaning of new words, the model is unable to correctly handle syntactic and semantic structures not covered by the seed templates. To alleviate this problem, we intend to further explore learning novel lexical templates.

Acknowledgements

We thank Kuzman Ganchev, Emily Pitler, Luke Zettlemoyer, Tom Kwiatkowski and Nicholas FitzGerald for their comments on earlier drafts, and the anonymous reviewers for their valuable feedback. We also wish to thank Ryan McDonald and Arturas Rozenas for their valuable input about voting procedures.

References

Yoav Artzi and Luke S. Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Yoav Artzi and Luke S. Zettlemoyer. 2013a. UW SPF: The University of Washington Semantic Parsing Framework.

Yoav Artzi and Luke S. Zettlemoyer. 2013b. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1(1):49–62.

Steven J. Brams and Peter C. Fishburn. 1978. Approval voting. *The American Political Science Review*, pages 831–847.

Qingqing Cai and Alexander Yates. 2013. Semantic parsing freebase: Towards open-domain semantic parsing. In *Proceedings of the Joint Conference on Lexical and Computational Semantics*.

Gelnn Carroll and Eugene Charniak. 1992. Two experiments on learning probabilistic dependency grammars from corpora. *Working Notes of the Workshop Statistically-Based NLP Techniques*.

David L. Chen and Raymond J. Mooney. 2011. Learning to interpret natural language navigation instructions from observations. In *Proceedings of the National Conference on Artificial Intelligence*.

David L. Chen. 2012. Fast online lexicon learning for grounded language acquisition. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Stephen Clark and James R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. 2010. Driving semantic parsing from the world's response. In *Proceedings of the Conference on Computational Natural Language Learning*.

Dan Goldwasser and Dan Roth. 2011. Learning from natural instructions. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Dan Goldwasser, Roi Reichart, James Clarke, and Dan Roth. 2011. Confidence driven unsupervised semantic parsing. In *Proceedings of the Association of Computational Linguistics*.

Tin K. Ho, Jonathan J. Hull, and Sargur N. Srihari. 1994. Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 66–75.

Rohit J. Kate and Raymond J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Joohyun Kim and Raymond J. Mooney. 2012. Unsupervised pcfg induction for grounded language learning with highly ambiguous supervision. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Joohyun Kim and Raymond J. Mooney. 2013. Adapting discriminative reranking to grounded language learning. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Jayant Krishnamurthy and Tom Mitchell. 2012. Weakly supervised training of semantic parsers. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the Human Language Technology Conference of the North American Association for Computational Linguistics*.

Tom Kwiatkowski, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Tom Kwiatkowski, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical Generalization in CCG Grammar Induction for Semantic Parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Mike Lewis and Mark Steedman. 2013. Combined distributional and logical semantics. *Transactions of the Association for Computational Linguistics*, 1(1):179–192.

Percy Liang, Michael I. Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Matt MacMahon, Brian Stankiewics, and Benjamin Kuipers. 2006. Walk the talk: Connecting language, knowledge, action in route instructions. In *Proceedings of the National Conference on Artificial Intelligence*.

Cynthia Matuszek, Nicholas FitzGerald, Luke S. Zettlemoyer, Liefeng Bo, and Dieter Fox. 2012. A joint model of language and perception for grounded attribute learning. In *Proceedings of the International Conference on Machine Learning*.

Mark Steedman. 1996. *Surface Structure and Interpretation*. The MIT Press.

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

Merijn Van Erp and Lambert Schomaker. 2000. Variants of the borda count method for combining ranked classifier hypotheses. In *In the International Workshop on Frontiers in Handwriting Recognition*.

Hado Van Hasselt. 2011. *Insights in Reinforcement Learning: formal analysis and empirical evaluation of temporal-difference learning algorithms*. Ph.D. thesis, University of Utrecht.

Yuk W. Wong and Raymond J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Association for Computational Linguistics*.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence*.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Luke S. Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*.