A Probabilistic Generative Grammar for Semantic Parsing

Abulhair Saparov
Carnegie Mellon University

Domain-general semantic parsing is a long-standing goal in natural language processing, where the semantic parser is capable of robustly parsing sentences from domains outside of which it was trained. Current approaches largely rely on additional supervision from new domains in order to generalize to those domains. We present a generative model of natural language utterances and logical forms and demonstrate its application to semantic parsing. Our approach relies on domain-independent supervision to generalize to new domains. We derive and implement efficient algorithms for training, parsing, and sentence generation. The work relies on a novel application of hierarchical Dirichlet processes (HDPs) for structured prediction, which we also present in this manuscript.

This manuscript is an excerpt of chapter 4 from the Ph.D. thesis of Saparov (2022), where the model plays a central role in a larger natural language understanding system.

This manuscript provides a new simplified and more complete presentation of the work first introduced in Saparov, Saraswat, and Mitchell (2017). The description and proofs of correctness of the training algorithm, parsing algorithm, and sentence generation algorithm are much simplified in this new presentation. We also describe the novel application of hierarchical Dirichlet processes for structured prediction. In addition, we extend the earlier work with a new model of word morphology, which utilizes the comprehensive morphological data from Wiktionary.

1. Introduction

Accurate and efficient semantic parsing is a long-standing goal in natural language processing. Existing approaches are quite successful in particular domains (Zettlemoyer and Collins 2005, 2007; Wong and Mooney 2007; Liang, Jordan, and Klein 2013; Kwiatkowski et al. 2010, 2011, 2013; Li, Liu, and Sun 2013; Wang, Kwiatkowski, and Zettlemoyer 2014; Zhao and Huang 2015; Dong and Lapata 2016; Rabinovich, Stern, and Klein 2017). However, they are largely domain-specific, relying on additional supervision such as a lexicon that provides the semantics or the type of each token in a set (Zettlemoyer and Collins 2005, 2007; Kwiatkowski et al. 2010, 2011; Liang, Jordan, and Klein 2013; Wang, Kwiatkowski, and Zettlemoyer 2014; Zhao and Huang 2015; Dong and Lapata 2016; Rabinovich, Stern, and Klein 2017), or a set of initial synchronous context-free grammar rules (Wong and Mooney 2007; Li, Liu, and Sun 2013). To apply the above systems to a new domain, additional supervision is necessary. When beginning to read text from a new domain, humans do not need to re-learn basic English grammar. Rather, they may encounter novel terminology. With this in mind, our approach is akin to that of (Kwiatkowski et al. 2013) where we provide domain-independent supervision to help train a semantic parser. More specifically, our semantic parsing model restricts the rules that may be learned during training to a set that characterizes the general syntax

of English. While we do not explicitly present and evaluate an open-domain semantic parser, we hope our work provides *a step* in that direction.

Knowledge plays a critical role in natural language understanding. Even seemingly trivial sentences may have a large number of ambiguous interpretations. Consider the sentence *"Ada started the machine with the GPU,"* for example. Without additional knowledge, such as the fact that "machine" can refer to computing devices that contain GPUs, or that computers generally contain devices such as GPUs, the reader cannot determine whether the GPU is part of the machine or if the GPU is a device that is used to start machines. Context is highly instrumental to quickly and unambiguously understand sentences.

In contrast to most semantic parsers, which are built on discriminative models, our model is fully generative: To generate a sentence, the logical form is first drawn from a prior. A grammar then recursively constructs a derivation tree top-down, probabilistically selecting production rules from distributions that depend on the logical form. The semantic prior distribution provides a straightforward way to incorporate background knowledge, such as information about the types of entities and predicates, or the context of the utterance. Additionally, our generative model presents a promising direction to *jointly* learn to understand and generate natural language. Further, our parser can return *partial* parses of sentences, which is useful for sentences that contain a small number of unseen words, such as definitions of new tokens. This can be exploited to learn new tokens and concepts outside of training.

In section 2, we present a novel application of hierarchical Dirichlet processes (HDPs) to structured prediction. We use this HDP model within our semantic parsing model, where HDPs are used to model dependence on logical forms. A mathematical description of the semantic parsing model is given in section 3. In section 4.1, we describe the algorithms for training, parsing, and generation, including details on their implementation. In section 5, we apply this parsing approach to the GEOQUERY and JOBS datasets (Zelle and Mooney 1996; Tang and Mooney 2000), using the Datalog representation of the provided logical form labels, and demonstrate that the accuracy of the parsed logical forms is comparable to that of the state-of-the-art on these datasets.

2. Hierarchical Dirichlet processes for structured prediction

In order to describe our novel application of HDPs for structured prediction, which play a central role in our semantic parsing model, we must first define some notation as well as useful properties of Dirichlet processes.

The *Dirichlet process* (DP) (Ferguson 1973) is a distribution over probability distributions (i.e. samples from a DP are themselves distributions). If a distribution G is drawn from a DP, we can write

$$G \sim \mathrm{DP}(\alpha, H), \tag{1}$$

where the DP is characterized by two parameters: a concentration parameter $\alpha > 0$ and a base distribution H. The DP has the useful property that $\mathbb{E}[G] = H$, and the concentration parameter α describes the "closeness" of G to the base distribution H. If α is small, G is more different from the base distribution H. If α is large, G is more similar to H.

DPs are often used in statistical machine learning models where observations $y_1, y_2, \ldots$ are distributed according to G, such as in:

$$G \sim \text{DP}(\alpha, H), \tag{2}$$

$$y_1, y_2, \ldots \sim G. \tag{3}$$

The joint probability of $y_1, \ldots, y_n$ is given by:

$$p(y_1, \ldots, y_n) = \frac{\alpha^n \Gamma(\alpha)}{\Gamma(\alpha + n)} \prod_{k=1}^{m} H(y_k^*)\Gamma(n_k), \tag{4}$$

where y_k^* are the unique values of $y_1, \ldots, y_n$, m is the number of such values, $n_k \triangleq \#\{i : y_i = y_k^*\}$ is the number of times y_k^* appears in $y_1, \ldots, y_n$, and $\alpha^n\Gamma(\alpha)/\Gamma(\alpha + n)$ is the normalization term.

In these models, the *Chinese restaurant process* (CRP) (Aldous 1985) provides a convenient equivalent description:

$$\phi_1, \phi_2, \ldots \sim H, \tag{5}$$

$$z_1 = 1, \tag{6}$$

$$z_{i+1} = \begin{cases} k & \text{with probability } \frac{n_k}{\alpha+i}, \\ k^{\text{new}} & \text{with probability } \frac{\alpha}{\alpha+i}, \end{cases} \tag{7}$$

$$y_i = \phi_{z_i}, \tag{8}$$

where $n_k \triangleq \#\{j \leq i : z_j = k\}$ is the number of times k appears in $\{z_1, \ldots, z_i\}$, $k^{\text{new}} \triangleq \max\{z_1, \ldots, z_i\} + 1$ is the next integer that doesn't appear in $\{z_1, \ldots, z_i\}$. The analogy to a restaurant is to imagine a restaurant with a countably infinite sequence of tables, labeled $1, 2, 3, \ldots$ The first person comes into the restaurant and sits at table 1. For each subsequent person that enters the restaurant, they choose to sit at a table with probability proportional to the number of people already sitting at that table. Otherwise, they choose to sit at an empty table with probability proportional to α. z_i indicates which table the i^{th} customer chose to sit, n_k is the number of people sitting at table k, and k^{new} is the index of the next unoccupied table. Each table is assigned a sample from H, independently and identically distributed (i.i.d.), where ϕ_i is the sample assigned to table i. Each observation y_i is the sample from H that is assigned to the table that the i^{th} customer chose to sit (i.e. table z_i). The CRP provides a simple algorithm to generate samples from a DP model. Notice that if α is very large, every customer is likely to choose to sit at a new table, and so each y_i is likely to be drawn i.i.d. from H (and therefore, G would be very similar to H). The opposite is true in the case where α is small, where G would be heavily concentrated on a small handful of observations, as each customer is more likely to sit at a table with existing customers. The CRP is *exchangeable* which is useful property where the joint distribution of table assignments $\boldsymbol{z}$ is independent of their order. That is, for any permutation of the integers σ:

$$p(z_1, z_2, \ldots) = p(z_{\sigma(1)}, z_{\sigma(2)}, \ldots). \tag{9}$$

Note that this presentation of the DP differs from the classical presentation, where the DP is part of a mixture model, as in:

$$G \sim \text{DP}(\alpha, H), \tag{10}$$

$$\theta_1, \theta_2, \ldots \sim G, \tag{11}$$

$$y_i \sim F(\theta_i), \tag{12}$$

where $F(\theta_i)$ is a distribution with parameter θ_i. If H is a conjugate prior of F, then an efficient Gibbs sampling algorithm is available, for example if H is a Dirichlet distribution and F is a multinomial, or if both H and F are normal distributions. In this manuscript, F is assumed to be the delta function (the distribution whose samples are identical to the input parameter), and no assumptions are made on H other than there exists an efficient way to compute the prior probability $p(\phi_i)$.

2.1 Hierarchical Dirichlet processes

The DP can be used as a component in larger models. The *hierarchical Dirichlet process* (HDP) (Teh et al. 2006) is a hierarchy of random variables, where each random variable is a distributed according to a Dirichlet process whose base distribution is given by the parent node in the hierarchy. Suppose each observation y_i is coupled with a parameter x_i that indicates the source node from which to sample the observation. Let the label of the root node in the hierarchy be $\mathbf{0}$, and the model can be written:

$$G^{\mathbf{n}} \sim \begin{cases} \text{DP}(\alpha^{\mathbf{0}}, H) & \text{if } \mathbf{n} = \mathbf{0}, \\ \text{DP}(\alpha^{\mathbf{n}}, G^{\text{parent}(\mathbf{n})}) & \text{otherwise}, \end{cases} \tag{13}$$

$$y_i \sim G^{x_i}, \tag{14}$$

for all nodes in the hierarchy $\mathbf{n}$. An equivalent "Chinese restaurant" representation may be written, which is coined a *Chinese restaurant franchise* (CRF), where each node $\mathbf{n}$ has a restaurant. For simplicity, assume that all x_i are leaf nodes, then the CRF is written:

$$\phi_1, \phi_2, \ldots \sim H, \tag{15}$$

$$z_1^{\mathbf{n}} = 1, \tag{16}$$

$$z_{i+1}^{\mathbf{n}} = \begin{cases} k & \text{with probability } \frac{n_k^{\mathbf{n}}}{\alpha^{\mathbf{n}}+i}, \\ k^{\text{new}} & \text{with probability } \frac{\alpha^{\mathbf{n}}}{\alpha^{\mathbf{n}}+i}, \end{cases} \tag{17}$$

$$\psi_i^{\mathbf{n}} = \begin{cases} \phi_{z_i^{\mathbf{0}}} & \text{if } \mathbf{n} = \mathbf{0}, \\ \psi_{z_i^{\mathbf{n}}}^{\text{parent}(\mathbf{n})} & \text{otherwise}, \end{cases} \tag{18}$$

$$y_i = \psi_{u_i+1}^{x_i}, \tag{19}$$

for all nodes in the hierarchy $\mathbf{n}$, where $n_k^{\mathbf{n}} \triangleq \#\{j \leq i : z_j^{\mathbf{n}} = k\}$ is the number of customers at node $\mathbf{n}$ sitting at table k, $k^{\text{new}} \triangleq \max\{z_1^{\mathbf{n}}, \ldots, z_i^{\mathbf{n}}\} + 1$ is the next available table at node $\mathbf{n}$, and $u_i \triangleq \#\{j < i : x_j = x_i\}$ is the number of previous observations drawn from node $\mathbf{n}$. In this extended metaphor, whenever a customer sits at a new table in the restaurant at node $\mathbf{n} \neq \mathbf{0}$, a "new customer" appears in the parent node parent($\mathbf{n}$) which

corresponds to this table. The $\psi_i^{\mathbf{n}}$ are the samples from $G^{\mathbf{n}}$. Note that the above model is valid only when x_i is a leaf node. If x_i were a parent node, then the output samples $\psi_j^{x_i}$ are used by both the child nodes of x_i as well as the observations y_i. In the restaurant metaphor, the customers at node x_i not only come from its child nodes but also from the observations. In this case, the $\psi_j^{x_i}$ that are assigned to the observations come after those assigned to child nodes (the order does not actually matter thanks to exchangeability, so long as the samples/customers are partitioned between the two). More precisely, y_i would be equal to $\psi_{c^{\mathbf{n}}+u_i+1}^{x_i}$ where $c^{\mathbf{n}} = \max\{z_i^{\mathbf{c}} : \mathbf{c} \in \text{children}(\mathbf{n})\}$ is the number of $\psi_j^{x_i}$ used by the child nodes of $\mathbf{n}$ (i.e. the number of customers that come from the child nodes of $\mathbf{n}$).

2.2 Inferring the source node x

Sections A and B describes how to efficiently obtain posterior samples of $\boldsymbol{z}$ (and therefore, $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$) using *Markov chain Monte Carlo* (MCMC), given a set of observations y_i and the corresponding nodes x_i from which they were sampled. But now consider the case where the $\boldsymbol{x}$ are random variables, and we encounter a new observation y^*, but the source node x^* (from which y^* was sampled) is unknown, and we would like to infer it. That is, we would like to compute:

$$\arg\max_{x^*} p(x^*|y^*, \boldsymbol{x}, \boldsymbol{y}) = \arg\max_{x^*} p(x^*) \int p(y^*|x^*, \boldsymbol{z}) p(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{z}, \tag{20}$$

$$\approx \arg\max_{x^*} \frac{p(x^*)}{N_{\text{samples}}} \sum_{\boldsymbol{z}^{(t)} \sim \boldsymbol{z}|\boldsymbol{x},\boldsymbol{y}} p(y^* \mid x^*, \boldsymbol{z}^{(t)}, \boldsymbol{\psi}^{(t)}, \boldsymbol{\phi}^{(t)}), \tag{21}$$

$$\text{where } p(y^* \mid x^*, \boldsymbol{z}, \boldsymbol{\psi}, \boldsymbol{\phi}) = p(\psi_{\text{new}}^{x^*} = y^* \mid \boldsymbol{z}, \boldsymbol{\psi}, \boldsymbol{\phi}).$$

This quantity is computed as in equations 59 and 60. The $\arg\max$ over this objective is a discrete optimization problem, which, if solved naïvely, would require computing the objective function for every node $\mathbf{n}$ in the tree. This is intractable if the tree is very large. Therefore, we present a branch-and-bound algorithm to perform this optimization efficiently.

Branch-and-bound (Land and Doig 1960) is a method for solving discrete optimization problems. Pseudocode is shown in algorithm 1. Given an objective function f, heuristic h, and search space X, the algorithm returns the k-best elements of X that maximize the objective f. The algorithm requires that the heuristic h be an *upper bound* for f. That is, for any set S,

$$h(S) \geq \max_{x \in S} f(x). \tag{22}$$

The algorithm begins by considering the full search space X. A procedure called `branch` then partitions X into n disjoint subsets X_i (this procedure is specific to the optimization problem). Each subset is pushed onto the priority queue, with its key given by the heuristic $h(X_i)$. Then, for each iteration of the main loop, pop a set S from the priority queue, and repeat the process: using `branch`, partition S into $(S_1, \ldots, S_n)$, and then push each subset into the priority with key $h(S_i)$. If $S = \{x\}$ is a singleton set only containing the element x, then add it to a list of potential solutions. The algorithm terminates when there are k potential solutions whose objective function values are at least the priority of S, or when the priority queue becomes empty. Once the algorithm terminates, the

Algorithm 1: Pseudocode for a generic brand-and-bound algorithm for k-best discrete optimization.

```
function branch_and_bound(objective function f, heuristic h, domain X)
    C is an empty list
    Q is an empty priority queue
    Q.push(X, ∞)
    while Q not empty do
        (S, v) = Q.pop()
        if S = {x} is a singleton
            C.add(x, f(x))
        else
            (S_1, ..., S_n) = branch(S)
            for i = 1, ..., n do
                Q.push(S_i, h(S_i))
        /* check termination condition */
        if there are k elements in C with priority at least v
            break
    return C /* the k elements of X that maximize f */
```

objective function values of the returned solutions are at least as large as the heuristic of the remainder of the search space. And since h is an upper bound for f, the returned solutions are guaranteed to be optimal.

We develop a branch-and-bound algorithm to perform the optimization in equation 21. The HDP hierarchy provides a convenient search tree structure for the optimization. Let $D(\mathbf{n})$ be the set of descendent nodes of $\mathbf{n}$, including $\mathbf{n}$ itself. The function $\texttt{branch}(D(\mathbf{n}))$ is defined to partition $D(\mathbf{n})$ into $(\{\mathbf{n}\}, D(\mathbf{c}_1), \ldots, D(\mathbf{c}_n))$ where $\mathbf{c}_i$ are the child nodes of $\mathbf{n}$. We define a heuristic for $D(\mathbf{n})$:

$$h(D(\mathbf{n})) = \frac{h_x(D(\mathbf{n}))}{N_{\text{samples}}} \sum_{t=1}^{N_{\text{samples}}} \max_{\{k:n_k^{\mathbf{n}}>0\}} \{\mathbb{1}\{\psi_k^{\mathbf{n}} = y^*\}, p(\psi_{\text{new}}^{\mathbf{n}} = y^*)\} \tag{23}$$

where $h_x(S)$ is an upper bound on the prior $h_x(S) \geq \max_{x \in S} p(x)$, the max is taken over all occupied tables in the restaurant at node $\mathbf{n}$, and the references to ψ within the sum are for the t^{th} sample, $\psi^{(t)}$. $D(\mathbf{n})$ can be sparsely represented in the implementation as a simple pointer to $\mathbf{n}$. The heuristic is convenient since it can be computed only using the information available at node $\mathbf{n}$, and so its running time is not a function of the size of the HDP hierarchy, as long as the heuristic on the prior $h_x(\cdot)$ is easy to compute. Furthermore, our algorithm avoids the recursion in the computation of $p(\psi_{\text{new}}^{\mathbf{n}})$, since the term $p(\psi_{\text{new}}^{\text{parent}(\mathbf{n})})$ was already computed in the computation of the heuristic for the *parent node*, and our algorithm re-uses it in future heuristic evaluations.

Thm 1
The heuristic $h(D(\mathbf{n}))$ is an upper bound on $\max_{x \in D(\mathbf{n})} f(x)$ where f is the objective function given by equation 21.

Proof. Consider any node $\mathbf{m} \in D(\mathbf{n})$ a descendant of $\mathbf{n}$, and any MCMC sample t. We first aim to show that the quantity within the sum is an upper bound:

$$\max_{\{k:n_k^{\mathbf{n}}>0\}} \{\mathbb{1}\{\psi_k^{\mathbf{n}} = y^*\}, p(\psi_{\text{new}}^{\mathbf{n}} = y^*)\} \geq p(y^*|x = \mathbf{m}, \boldsymbol{z}^{(t)}, \boldsymbol{\psi}^{(t)}, \boldsymbol{\phi}^{(t)}). \tag{24}$$

Since the right-hand side is equal to $p(\psi^{\mathbf{m}}_{\text{new}} = y^*)$, the bound is trivially true in the case where $\mathbf{m} = \mathbf{n}$. So we can assume without loss of generality that $\mathbf{m} \neq \mathbf{n}$, and the right-hand side can be written:

$$p(y^* \mid x = \mathbf{m}, \boldsymbol{z}^{(t)}, \boldsymbol{\psi}^{(t)}, \boldsymbol{\phi}^{(t)}) = p(\psi^{\mathbf{m}}_{\text{new}} = y^*), \tag{25}$$

$$= \frac{\alpha^{\mathbf{m}} p(\psi^{\text{parent}(\mathbf{m})}_{\text{new}} = y^*)}{\alpha^{\mathbf{m}} + n^{\mathbf{m}}} + \sum_{\{k' : n^{\mathbf{m}}_{k'} > 0\}} \frac{n^{\mathbf{m}}_{k'} \mathbb{1}\{\psi^{\text{parent}(\mathbf{m})}_{k'} = y^*\}}{\alpha^{\mathbf{m}} + n^{\mathbf{m}}}, \tag{26}$$

according to equation 59. Since this expression is a convex combination of $\mathbb{1}\{\psi^{\text{parent}(\mathbf{m})}_{k'} = y^*\}$ and $p(\psi^{\text{parent}(\mathbf{m})}_{\text{new}} = y^*)$, it is bounded above by:

$$\leq \max_{\{k' : n^{\mathbf{m}}_{k'} > 0\}} \left\{ \mathbb{1}\{\psi^{\text{parent}(\mathbf{m})}_{k'} = y^*\}, p(\psi^{\text{parent}(\mathbf{m})}_{\text{new}} = y^*) \right\}. \tag{27}$$

Due to equation 59, observe that $p(\psi^{\mathbf{a}}_{\text{new}} = y^*) \leq p(\psi^{\text{parent}(\mathbf{a})}_{\text{new}} = y^*)$ for any node $\mathbf{a}$. In addition, by construction of the HDP, the $\psi^{\mathbf{a}}_k$ at any node $\mathbf{a}$ are a subset of the $\psi^{\text{parent}(\mathbf{a})}_k$. That is, for all k, there is a k' such that $\psi^{\mathbf{a}}_k = \psi^{\text{parent}(\mathbf{a})}_{k'}$. These observations extend to all ancestors of $\mathbf{a}$. Applying these two observations to the node $\mathbf{m}$, we can conclude that the above expression is further bounded above by:

$$\leq \max_{\{k' : n^{\mathbf{n}}_{k'} > 0\}} \{ \mathbb{1}\{\psi^{\mathbf{n}}_{k'} = y^*\}, p(\psi^{\mathbf{n}}_{\text{new}} = y^*) \}. \tag{28}$$

We have shown that the quantity within the sum of the heuristic is an upper bound. Since by definition, $h_x(D(\mathbf{n})) \geq \max_{x \in D(\mathbf{n})} p(x) \geq p(x^* = \mathbf{m})$, the full heuristic $h(D(\mathbf{n}))$ is an upper bound on $f(\mathbf{m})$, the objective function evaluated at $\mathbf{m}$, for all $\mathbf{m} \in D(\mathbf{n})$. Therefore, $h(D(\mathbf{n})) \geq \max_{\mathbf{m} \in D(\mathbf{n})} f(\mathbf{m})$. ■

The branch-and-bound algorithm starts with the input set $D(\mathbf{0})$, which is the set of all nodes in the tree, and will efficiently compute the k most probable values of the source node x^*, from which the observation y^* was sampled. Note that the above algorithm is easily extended to the case where the HDP is part of a mixture model (i.e. F is not a delta function). To do so, replace each instance of $\mathbb{1}\{\psi^{\mathbf{n}}_k = y^*\}$ with $p(y^* \mid y^* \sim F(\theta^{\mathbf{n}}_k), \boldsymbol{z}, \boldsymbol{\phi})$, for all $\mathbf{n}$ and k.

The above algorithm can be generalized to the case where x^* is restricted to a subset of the nodes X in the hierarchy: $\arg\max_{x^*} p(x^* \mid x^* \in X, y^*, \boldsymbol{x}, \boldsymbol{y})$. In this case, the algorithm is started with the input set $D(\mathbf{0}) \cap X$. The `branch` function is modified: `branch`$(D(\mathbf{n}) \cap X)$ partitions the set $D(\mathbf{n}) \cap X$ into $(\{\mathbf{n}\} \cap X, D(\mathbf{c}_1) \cap X, \ldots, D(\mathbf{c}_n) \cap X)$ where $\mathbf{c}_i$ are the child nodes of $\mathbf{n}$.

2.3 Infinite hierarchies

To apply the HDP in our semantic parsing model, we need to be able to handle the case where the HDP hierarchy is infinite (but with finite height). That is, every non-leaf node in the hierarchy may have an infinite number of children. But this makes no difference in the MCMC algorithm to infer $\boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{\psi}$, since the number of given observations $(\boldsymbol{x}, \boldsymbol{y})$ is finite. We only need to compute and keep track of the variables that are associated with an observation (either at the current node or a descendant). Thus, the only nodes

of the tree that we need to explicitly keep in memory are those of $\boldsymbol{x}$ and their ancestors, as the restaurants at all other nodes are empty. The explicitly-stored tree size is bounded by the product of the number of distinct x_i and the height of the tree.

However, the branch-and-bound algorithm to find the most probable source node x^* needs to be adapted, since the `branch` function would otherwise return an infinite number of subsets. Consider any node $\boldsymbol{n}$ that has no observations (i.e. has an empty restaurant). Then by equation 59, $p(\psi^{\mathbf{n}}_{\text{new}}) = p(\psi^{\text{parent}(\mathbf{n})}_{\text{new}}) = \ldots = p(\psi^{\mathbf{a}}_{\text{new}})$ where $\mathbf{a}$ is the most recent non-empty ancestor of $\mathbf{n}$. For such nodes, the objective function in equation 21 can be simplified

$$\frac{p(\mathbf{n})}{N_{\text{samples}}} \sum_{\boldsymbol{z}^{(t)} \sim \boldsymbol{z}|\boldsymbol{x},\boldsymbol{y}} p(\psi^{\mathbf{a}}_{\text{new}} = y^*). \tag{29}$$

Aside from the prior term $p(\mathbf{n})$, all empty descendant nodes of $\mathbf{a}$ have the same objective function value, which is independent of $\mathbf{n}$. So to adapt the algorithm to the infinite hierarchy case, the `branch` function is modified:

$$\texttt{branch}(D(\mathbf{a})) \textit{ returns } \left(\{\mathbf{a}\}, D(\mathbf{c}_1), \ldots, D(\mathbf{c}_n), \bigcup_{i=n+1}^{\infty} D(\mathbf{c}_i)\right), \tag{30}$$

where $(\mathbf{c}_1, \ldots, \mathbf{c}_n)$ are the non-empty child nodes of $\mathbf{a}$, and $(\mathbf{c}_{n+1}, \mathbf{c}_{n+2}, \ldots)$ are the empty child nodes of $\mathbf{a}$. Next, in algorithm 1, following line 8, we add a new else-if statement to check for the case that S is a set of empty nodes. If so, S is added to C, and we don't continue the search in the empty descendant nodes. The resulting adapted branch-and-bound algorithm correctly and efficiently solves the optimization problem for infinite hierarchies.

2.4 Modeling dependence on discrete structures

HDPs can be used to learn distributions that depend on sequences of non-negative integers. Consider the data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ where each $x_i \in \mathbb{Z}^h_+$ is a sequence of h non-negative integers. The distribution of y_i is dependent on the value of x_i. We can use the HDP to learn the relationship of this dependence: construct a hierarchy of height h, where each non-leaf node has a countably infinite number of children, every child node corresponding to a non-negative integer. Here, each x_i uniquely identifies a leaf node in the hierarchy by characterizing a path from the root $\mathbf{0}$ to a leaf: the first integer in the sequence identifies the child of the root node, the second integer identifies the grandchild, and so on. The y_i are then sampled from the corresponding leaf node. We can apply MCMC to learn the distributions of the y_i, and how those distributions relate to the integer sequences x_i.

Given a new observation y^*, the branch-and-bound algorithm can be used to find the most probable corresponding integer sequence x^*, but we need to be able to convert the output of the branch-and-bound into the corresponding integer sequence. The algorithm will output a list of the k most probable source nodes from which y^* is sampled, *or* sets of empty source nodes (since the HDP hierarchy is infinite). More precisely, let $(o_1, \ldots, o_k)$ be the output of the branch-and-bound algorithm. For each o_j, there are two cases:

1. o_j is a single leaf node, in which case it is straightforward to convert the node into its corresponding integer sequence.
2. o_j is the set of empty descendants of a node **a**. In this latter case, it can be converted into an "incomplete" sequence of integers, where the first $L(\mathbf{a})$ numbers of the sequence correspond to the node **a**, where $L(\mathbf{a})$ is the level of **a**. This incomplete sequence represents the set of all integer sequences that begin with the same $L(\mathbf{a})$ integers, that do not already explicitly exist in the tree.

For example, let $\mathbf{n}_a$ be the a^{th} child of the root node $\mathbf{0}$ in the HDP hierarchy. Let $\mathbf{n}_{a,b}$ be the b^{th} child of $\mathbf{n}_a$, and so on. Suppose the training set contains only the sequences $(4, 3, 1)$, $(4, 7, 4)$, and $(4, 8, 2)$. Therefore, the nodes in the HDP with non-empty restaurants are: $\mathbf{n}_{4,3,1}$, $\mathbf{n}_{4,7,4}$, $\mathbf{n}_{4,8,2}$, $\mathbf{n}_{4,3}$, $\mathbf{n}_{4,7}$, $\mathbf{n}_{4,8}$, $\mathbf{n}_4$, and $\mathbf{0}$. If the branch-and-bound algorithm returns $\mathbf{n}_{4,7,4}$, the corresponding output integer sequence is $(4, 7, 4)$. If instead, branch-and-bound returns the set of the empty descendant nodes of $\mathbf{n}_4$, the corresponding output integer sequence is $(4, * \setminus \{3, 7, 8\}, *)$. The '$*$' is a "wildcard" symbol that represents the set of all non-negative integers. Thus, $(4, * \setminus \{3, 7, 8\}, *)$ represents the set of all integer sequences that start with $(4, \ldots)$ but do not start with $(4, 3, \ldots)$, $(4, 7, \ldots)$, or $(4, 8, \ldots)$.

This model can be extended to the case where the $x_i \in \mathcal{X}$ have richer structure (e.g. $\mathcal{X}$ is the set of labeled trees, graphs, logical forms, etc), i.e. *structured prediction*. To do so, define d functions $f_k : \mathcal{X} \to \mathbb{Z}_+$ that characterize an aspect of the input structures x_i. We call these functions f_k *feature functions*. For example, if x is a labeled binary tree, $f_1(x)$ returns the label of the root node, and $f_2(x)$ returns the label of the left child, etc. The functions serve to map the structures x_i into sequences of non-negative integers: $(f_1(x_i), \ldots, f_d(x_i))$. Then the above HDP model can be directly used to learn the relationship between these integer sequences and the distribution of the observations y_i. For a new observation y^*, the branch-and-bound algorithm will return the k most likely integer sequences (possibly with wildcard symbols) that represent the unknown structure x^*. To convert the integer sequence $(w_1, \ldots, w_d)$ into the corresponding structure in $\mathcal{X}$, we can compute:

$$f_1^{-1}(w_1) \bigcap \ldots \bigcap f_d^{-1}(w_d) \text{ where } f_k^{-1}(w_k) \triangleq \{x : f_k(x) \in w_k\}. \tag{31}$$

Our code implements three functions to perform the above mapping between integer sequences and more structured representations in $\mathcal{X}$:

1. `get_feature(`f`,` X`)`: Given a feature function f and a set $X \subseteq \mathcal{X}$, return $\{f(x) : x \in X\}$.
2. `set_feature(`f`,` X^{old}`,` w`)`: Given a feature function f, a set $X^{\text{old}} \subseteq \mathcal{X}$, and a non-negative integer $w \in \mathbb{Z}_+$, return $X^{\text{old}} \cap f^{-1}(w)$. This function is used in the case that w_k is an integer (not a wildcard).
3. `exclude_features(`f`,` X^{old}`,` W`)`: Given a feature function f, a set $X^{\text{old}} \subseteq \mathcal{X}$, and a finite set of non-negative integers $W \in \mathbb{Z}_+^*$, return $X^{\text{old}} \setminus f^{-1}(W)$. This is used in the case that w_k is a wildcard $* \setminus W$.

2.5 Related work

The HDP hierarchy in our proposed model in section 2.4 resembles a decision tree (Russell and Norvig 2010). The input features determine the path within the tree, and the output is sampled from a leaf node. Teh (2006) constructs a language model using a

hierarchical Pitman-Yor process (HPY), which is a generalization of the HDP that exhibits power-law behavior. In their model, the HPY describes the distribution of the next character in a sequence of characters, conditioned on the previous d characters. The sequence of preceding d characters corresponds to the path in the hierarchy of depth d. Our approach is a novel application of HDPs for structured prediction, where the path in the hierarchy is a random variable which corresponds to the structure we aim to predict. Since the HDP hierarchies are infinite, the model does not a priori impose a limit on the number of possible structures or logical forms. An idea for future work is to replace the HDP in our model with the HPY to better capture power-law behavior which is prevalent in natural language.

$$\begin{aligned}
\mathsf{S} &\rightarrow \mathsf{N}\text{:}\texttt{select_arg1}\ \ \mathsf{VP}\text{:}\texttt{delete_arg1} \\
\mathsf{VP} &\rightarrow \mathsf{V}\text{:}\texttt{identity}\ \ \mathsf{N}\text{:}\texttt{select_arg2} \\
\mathsf{VP} &\rightarrow \mathsf{V}\text{:}\texttt{identity}
\end{aligned}$$

N → “New Jersey”	V → “borders”
N → “NJ”	V → “bordered”
N → “Pennsylvania”	V → “has”
N → “Michael Phelps”	V → “swims”
N → “tennis”	V → “plays”

Figure 1
Example of a grammar in our framework. This example grammar operates on logical forms of the form *predicate(first argument, second argument)*. The semantic function `select_arg1` returns the first argument of the logical form. Likewise, the function `select_arg2` returns the second argument. The function `delete_arg1` removes the first argument, and `identity` returns the logical form with no change. In our work, the interior production rules (the first three listed above) are examples of rules that we specify, whereas the terminal rules and the posterior probabilities of *all* rules are learned via grammar induction. A simplified semantic representation is shown here for the sake of illustration. Our semantic parser uses a richer semantic representation. Section 3.2 provides more detail.

3. Model: semantic grammar

A grammar in our formalism operates over a set of nonterminals $\mathcal{N}$ and a set of terminal symbols $\mathcal{W}$. It can be understood as an extension of a context-free grammar (CFG) (Chomsky 1956) where the generative process for the syntax is dependent on a logical form, thereby coupling syntax with semantics. In the top-down generative process of a derivation tree, a logical form guides the selection of production rules. Production rules in our grammar have the form $A \rightarrow B_1{:}f_1 \ldots B_k{:}f_k$ where $A \in \mathcal{N}$ is a nonterminal, $B_i \in \mathcal{N} \cup \mathcal{W}$ are right-hand side symbols, and f_i are *semantic transformation functions*. These functions describe how to “decompose” this logical form when recursively generating the subtrees rooted at each B_i. Thus, they enable semantic compositionality. An example of a grammar in this framework is shown in figure 1, and a derivation tree is shown in figure 2. Let $\boldsymbol{\mathcal{R}}$ be the set of production rules in the grammar and $\boldsymbol{\mathcal{R}}_A$ be the set of production rules with left-hand nonterminal symbol A.

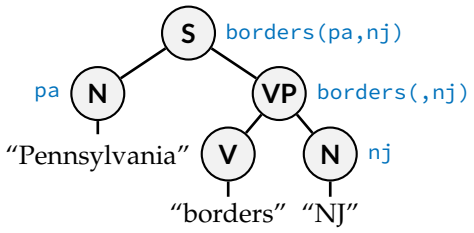

Figure 2
Example of a derivation tree under the grammar given in figure 1. The logical form corresponding to every node is shown in blue beside the respective node. The logical form for V is `borders(,nj)` and is omitted to reduce clutter.

3.1 Generative process

A *derivation tree* in this formalism is a tree where every interior node is labeled with a nonterminal symbol in $\mathcal{N}$, every leaf is labeled with a terminal in $\mathcal{W}$, and the root node is labeled with the root nonterminal S. Moreover, every node in the tree is associated with a logical form: let $x^{\mathbf{n}}$ be the logical form assigned to the tree node $\mathbf{n}$, and $x^{\mathbf{0}} = x$ for the root node $\mathbf{0}$.

The generative process to build a derivation tree begins with the root nonterminal S and a logical form x. The logical form x is drawn from a prior distribution on logical forms $p(x)$. The generative process *expands* S by randomly drawing a production rule from $\boldsymbol{\mathcal{R}}_S$, *conditioned* on the logical form x. This provides the first level of child nodes in the derivation tree. For example, if the rule $\mathsf{S} \to B_1{:}f_1 \dots B_k{:}f_k$ were drawn, the root node would have k child nodes, $\mathbf{n}_1, \dots, \mathbf{n}_k$, respectively labeled with the symbols $B_1, \dots, B_k$. The logical form associated with each node is determined by the semantic transformation function: $x^{\mathbf{n}_i} = f_i(x^{\mathbf{0}})$. These functions describe the relationship between the logical form at a child node and that of its parent node. This process repeats recursively with every right-hand side nonterminal symbol, until there are no unexpanded nonterminal nodes. The sentence is obtained by taking the *yield* (i.e. the concatenation) of the terminals in the tree.

The semantic transformation functions are specific to the semantic formalism and may be defined as appropriate to the application. In our semantic parsing experiments in section 5, we define a domain-independent set of transformation functions specific to the Datalog representation of GEOQUERY and JOBS (e.g., one function selects the left n conjuncts in a conjunction, another selects the n^{th} argument of a predicate instance, etc). Some examples of these transformation functions are:

- The function `select_left` returns the left conjunct of a conjunction. For example, given the Datalog expression `(river(A),loc(A,B),const(B,stateid(colorado)))`, this function returns `river(A)`.
- The function `delete_left` returns a conjunction where the first conjunct is removed. For example, given `(river(A),loc(A,B),const(B,stateid(colorado)))`, this function returns `(loc(A,B),const(B,stateid(colorado)))`.
- The function `select_arg2` returns the second argument in an atomic formula. For example, given `const(A,stateid(maine))`, this function returns `stateid(maine)`.

Semantic transformation functions are allowed to fail, which is useful in defining richer transformation functions and providing more flexibility when designing the production rules of the grammar. If in the generative process, a transformation function returns failure, the generative process is restarted from the root (all progress up to the failure is discarded). As an example, failure enables the definition of transformation functions that check whether the input logical form satisfies a specific property: `require_binary_conjunction` returns the input logical form, unchanged, if it is a conjunction of length 2; otherwise, it returns failure. Since failure can cause the generative process to repeatedly restart, the process of sampling using the generative process can be expensive. However, our approach does not generate sentences using this algorithm, and as we show in section 4, the performance of the parser is not adversely affected.

3.2 Selecting production rules

The above description does not specify the conditional distribution from which rules are selected from $\boldsymbol{\mathcal{R}}_A$ given the logical form. There are many modeling options available in choosing this distribution, but we need a distribution that captures complex dependencies between the logical form and selected production rule. For example, consider the grammar in figure 1 and the logical form `plays_sport( michael_phelps,tennis)`. When generating a sentence for this logical form, at the root nonterminal S, there is only one production rule available, $\mathsf{S} \rightarrow \mathsf{N}$:`select_arg1` VP:`delete_arg1`, so this rule is selected. Now consider the child node corresponding to the nonterminal VP. Its logical form is `plays_sport(,tennis)`, which is the output of the function `delete_arg1` when applied to the logical form at the root node. At this point, there are two choices of production rules with VP on the left-hand side: $\mathsf{VP} \rightarrow \mathsf{V}\ \ \mathsf{N}$ and $\mathsf{VP} \rightarrow \mathsf{V}$. In this case, we want the conditional distribution to select $\mathsf{VP} \rightarrow \mathsf{V}\ \ \mathsf{N}$, since the most likely sentence that conveys the semantics in the logical form is "plays tennis." In fact, $\mathsf{VP} \rightarrow \mathsf{V}\ \ \mathsf{N}$ should be selected even in when the logical form is `plays_sport(,baseball)` or `plays_sport(,soccer)` or almost any other sport. However, if the logical form were `plays_sport(,swimming)`, we want the conditional distribution to give higher probability to $\mathsf{VP} \rightarrow \mathsf{V}$, since the verb phrase "swims" is much more likely. Therefore, a desirable property of the conditional distribution for VP production rules is that the distribution depends primarily on the predicate symbol (e.g. `plays_sport`) but also depends secondarily on the object argument (e.g. `swimming` or `tennis`).

Our model uses the HDP to capture this dependence, as presented in section 2.4. Every nonterminal in our grammar $A \in \mathcal{N}$ will be associated with an HDP hierarchy. For each nonterminal, we specify a sequence of *semantic feature functions*, $\{g_1, \ldots, g_m\}$, each of which, when given input logical form x, returns a non-negative integer. The HDP hierarchy is a complete infinite tree of height m, where every parent node has an infinite number of child nodes, one for each non-negative integer. The base distribution H at the root of the HDP is over $\boldsymbol{\mathcal{R}}_A$.

Take, for example, the derivation in figure 2. In the generative process where the node VP is expanded, the production rule is drawn from the HDP associated with the nonterminal VP. Suppose the HDP was constructed using a sequence of two semantic features: (`predicate`, `arg2`). In the example, the feature functions are evaluated with the logical form `borders(,nj)` and they return a sequence of two integers, the first is the identifier for the symbol `borders` and the second is the identifier for the symbol `nj`. This sequence uniquely identifies a path in the HDP hierarchy from the root node $\mathbf{0}$ to a leaf node $\mathbf{n}$. The production rule $\mathsf{VP} \rightarrow \mathsf{V}\ \ \mathsf{N}$ is drawn from this leaf node $G^{\mathbf{n}}$, and the generative process continues recursively. As desired, the distribution of the selected

production rule $G^{\mathbf{n}}$ depends on the HDP source node $\mathbf{n}$, which itself depends primarily on the first feature and secondarily on the second feature and so on (in this example, the `predicate` and `arg2` of the logical form are the first and second features, respectively).

In our implementation, the set of nonterminals $\mathcal{N}$ is divided into two disjoint groups: (1) the set of "interior" nonterminals, and (2) preterminals. The production rules of preterminals are restricted such that the right-hand side contains only terminal symbols. The rules of interior nonterminals are restricted such that only nonterminal symbols appear on the right side.

1. For **preterminals**, H is a distribution over sequences of terminal symbols. The sequence of terminal symbols is distributed as follows: first sample the length of the terminal from a geometric distribution (i.e. the number of words) and then generate each word in the sequence i.i.d. from a uniform distribution over a finite set of (initially unknown) terminals. Note that we do not specify a set of domain-specific terminal symbols in defining this distribution.
2. For **interior nonterminals**, H is a discrete distribution over a domain-independent set of production rules, which we specify. Since the production rules contain transformation functions, they are specific to the semantic formalism. However, prior knowledge of the English language can be encoded in these specified production rules, which dramatically improves the statistical efficiency of our model and obviates the need for massive training sets to learn English syntax. It is nonetheless tedious to design these rules while maintaining domain-generality. Once specified, however, these rules in principle can be re-used in new tasks and domains without further changes.

We emphasize that only the prior is specified here, and our algorithm uses grammar induction to infer the posterior. In principle, a more relaxed choice of H may enable grammar induction without pre-specified production rules, and therefore without dependence on a particular semantic formalism or natural language, if an efficient inference algorithm can be developed in such cases.

3.3 Modeling morphology

The grammar model is easily modified to incorporate word morphology. To do so, we add an additional step to the generative process after generating the terminal symbols. Instead of the terminal symbols constituting the tokens of the sentence directly, the terminal symbols are instead word roots coupled with morphological flags that indicate their inflection. For example, in the grammar in figure 1, rather than having multiple rules for the various inflections of the verb "to border", such as $\mathsf{V} \to$ "borders", $\mathsf{V} \to$ "bordered", $\mathsf{V} \to$ "bordering", there would only be a single production rule for the root: $\mathsf{V} \to$ "border". In order to produce the various inflections, the logical form is augmented to carry morphology information. Semantic transformation functions may add or modify morphological flags. For example, suppose we have the rule $\mathsf{VP} \to \mathsf{V}$:`add_third_person,add_present_tense` where `add_third_person` is a function that adds the 3RD flag (indicating third person) to the logical form, and `add_present_tense` is a function that adds the PRES flag (indicating present tense) to the logical form. These morphological flags are copied into the terminal symbols, and as a final step, the roots are inflected according to the morphological flags (e.g. "border[3RD,PRES]" is inflected to "borders"). See figure 3 for an example of a derivation tree for a grammar that models morphology.

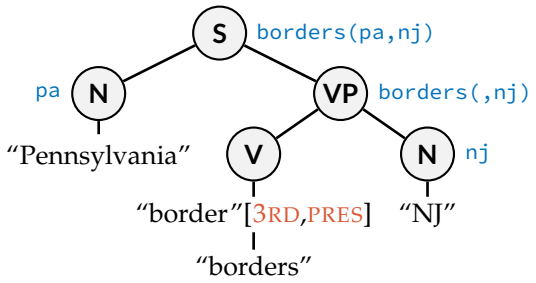

Figure 3
Example of a derivation tree under a grammar with a model of morphology. The logical form corresponding to every node is shown in blue beside the respective node. The logical form for V is `borders(,nj)`[3RD,PRES] and is omitted to reduce clutter. Morphology is not modeled for proper nouns such as "Pennsylvania" and "NJ."

If a root with morphological flags has multiple inflections, such as "octopus"[PL] (PL indicates plural), the generative process selects one uniformly at random. During inference (i.e. parsing), this morphological model has the effect of performing morphological and syntactic-semantic parsing jointly, as we will show in the next section. Wiktionary (Wikimedia Foundation 2020) provides comprehensive high-quality morphology information for English verbs, common nouns, adjectives, and adverbs. Our implementation uses Wiktionary to construct a mapping between uninflected roots and inflected words, which is used in both directions: (1) given root and morphological flags, find the corresponding set of inflections, or (2) given an inflected word, find the corresponding set of roots and morphological flags. Note that only (2) is necessary for parsing and training, whereas (1) is necessary for generation.

Although this morphology model is implemented in our code, we do not use it in our experiments on GEOQUERY and JOBS. The morphology model is used in the experiments described later in the thesis of Saparov (2022).

4. Inference and implementation

4.1 Training

In this section, we describe how to infer the latent derivation trees $\boldsymbol{t} \triangleq \{t_1, \ldots, t_n\}$, given a collection of sentences $\boldsymbol{y} \triangleq \{y_1, \ldots, y_n\}$ and logical form labels $\boldsymbol{x} \triangleq \{x_1, \ldots, x_n\}$, where each derivation tree t_i is distributed according to the conditional distribution described by the generative process in section 3.1 above.

We describe grammar induction independently of the choice of rule distribution. We wish to compute the posterior $p(\boldsymbol{t} \mid \boldsymbol{x}, \boldsymbol{y})$ of the latent derivation trees. This is intractable to compute exactly, and so we resort to MCMC. To perform blocked Gibbs sampling, we pick initial values for $\boldsymbol{t}$ and repeat the following: For $i = 1, \ldots, n$, sample $t_i \mid \boldsymbol{t}_{-i}, x_i, y_i$ where $\boldsymbol{t}_{-i} = \boldsymbol{t} \setminus \{t_i\}$.

$$p(t_i \mid \boldsymbol{t}_{-i}, x_i, y_i) \propto \mathbb{1}\{\text{yield}(t_i) = y_i\} \prod_{A \in \mathcal{N}} p\left(\bigcap_{\substack{\{\mathbf{n} \in t_i \,:\, \mathbf{n} \\ \text{has label } A\}}} r^{\mathbf{n}} \;\middle|\; \boldsymbol{t}_{-i}, x_i \right), \tag{32}$$

where $\mathcal{N}$ is the set of nonterminals, and the intersection is taken over the nodes $\mathbf{n} \in t_i$ labeled with the nonterminal A in the i^{th} derivation tree, and $r^{\mathbf{n}}$ is the production rule at node $\mathbf{n}$. Note that the probability does not necessarily factorize over rules, as is the case when using the HDP. So in order to sample t_i, we use Metropolis-Hastings (MH), where the proposal distribution is given by the fully factorized form:

$$p(t_i^* \mid \boldsymbol{t}_{-i}, x_i, y_i) \propto \mathbb{1}\{\text{yield}(t_i^*) = y_i\} \prod_{\mathbf{n} \in t_i^*} p\left(r^{\mathbf{n}} \mid \boldsymbol{t}_{-i}, x_i^{\mathbf{n}}\right). \tag{33}$$

The algorithm for sampling t_i^* is detailed in section 4.1.1. After sampling t_i^*, we choose to accept the new sample with probability

$$\min\left\{1, \frac{\prod_{\mathbf{n} \in t_i} p(r^{\mathbf{n}} \mid x^{\mathbf{n}}, \boldsymbol{t}_{-i})}{p\left(\bigcap_{\mathbf{n} \in t_i} r^{\mathbf{n}} \mid x, \boldsymbol{t}_{-i}\right)} \frac{p\left(\bigcap_{\mathbf{n} \in t_i^*} r^{\mathbf{n}} \mid x, \boldsymbol{t}_{-i}\right)}{\prod_{\mathbf{n} \in t_i^*} p(r^{\mathbf{n}} \mid x^{\mathbf{n}}, \boldsymbol{t}_{-i})}\right\}, \tag{34}$$

where t_i, here, is the old sample, and t_i^* is the newly proposed sample. In practice, this acceptance probability is very high. This approach is very similar in structure to that in Johnson, Griffiths, and Goldwater (2007); Blunsom and Cohn (2010); Cohn, Blunsom, and Goldwater (2010).

Computing the conditional probabilities of the production rules $p(r^{\mathbf{n}} \mid x^{\mathbf{n}}, \boldsymbol{t}_{-i})$ and $p(\bigcap_{\mathbf{n} \in t_i} r^{\mathbf{n}} \mid x, \boldsymbol{t}_{-i})$ (as well as the quantities required in sampling t_i^*) depends on the model for selecting production rules. In our semantic parsing model, which uses an HDP model, these quantities can be computed using equations 61 and 64. Our parsing method only keeps the last MCMC sample ($N_{\text{samples}} = 1$), so for each node in every derivation tree $\mathbf{m} \in t_j$, the production rule at that node $r^{\mathbf{m}}$ corresponds to a customer in the Chinese restaurant representation of the HDP associated with the nonterminal at node $\mathbf{m}$. When resampling the derivation tree t_i, our method removes all customers that correspond to a production rule in t_i. Then it is straightforward to compute conditional probabilities according to equations 61 and 64 with the remaining customers. Once a new t_i is sampled, the customers corresponding to production rules in t_i are added to their respective restaurants.

There may be additional random variables in the grammar apart from the derivation trees, such as $\boldsymbol{\alpha}$ in the HDPs. We perform Gibbs sampling steps for these variables after each loop of resampling the trees $t_i, i = 1, \ldots, n$. The grammar induction algorithm is summarized: Pick initial values for $\boldsymbol{t}$ and $\boldsymbol{\alpha}$ and repeat the following,

1. For $i = 1, \ldots, n$, sample $t_i^* \mid \boldsymbol{\alpha}, \boldsymbol{t}_{-i}, x_i, y_i$ from the distribution given by equation 33. Then accept this sample as the new value for t_i with probability given by equation 34.
2. Perform the Gibbs sampling step for $\boldsymbol{\alpha} \mid \boldsymbol{t}$.

In all our experiments, we run the above loop for 10 iterations. Note that this algorithm requires no further supervision beyond the utterances $\boldsymbol{y}$ and logical forms $\boldsymbol{x}$. However, it is able to exploit additional information such as supervised derivation trees: if $\bar{\boldsymbol{t}} \subseteq \boldsymbol{t}$ is a subset of derivation trees that are supervised, the Gibbs sampling algorithm simply avoids resampling the trees in $\bar{\boldsymbol{t}}$. These supervised derivation trees do not necessarily need to be rooted in the nonterminal S. For example, a lexicon can be provided where each entry is a terminal symbol y_i with a corresponding logical form label x_i. In our

experiments on GEOQUERY and JOBS, we evaluate our method with and without such a lexicon.

4.1.1 Sampling t_i^*. To sample from equation 33, we use *inside-outside sampling* (Finkel, Manning, and Ng 2006; Johnson, Griffiths, and Goldwater 2007), a dynamic programming approach. For every nonterminal $A \in \mathcal{N}$, sentence start position i, end position j, and logical form x, let $I_{(A,i,j,x)}$ be the probability that t_i^* has a node $\mathbf{n}$ with the label A and logical form x and spans the sentence from i to j. This is known as the *inside probability*. Similarly, for all production rules in the grammar $A \to B_1{:}f_1 \dots B_K{:}f_K$, sentence boundary positions between the right-hand side nonterminals $l_1 < \dots < l_{K+1}$, and logical forms x, let $I_{(\mathsf{S} \to B_1:f_1 \dots B_K:f_K, \boldsymbol{l}, x)}$ be the probability that t_i^* has a node $\mathbf{n}$ with the label A and logical form x and has child nodes B_u, each with logical forms $f_u(x)$, and each spanning the sentence from l_u to l_{u+1}. This is known as the *inside rule probability*. Note that we don't need to compute all possible inside probabilities for all logical forms (in many applications, the set of logical forms is infinite). Therefore, we compute these inside probabilities top-down, beginning at the root nonterminal $I_{(\mathsf{S},0,|y_i|,x_i)}$ with the known logical form x_i where $|y_i|$ is the length of sentence y_i. The following formula can be used to compute this quantity recursively:

$$I_{(A,i,j,x)} = \sum_{A \to B_1:f_1 \dots B_K:f_K} \sum_{i=l_1<\dots<l_{K+1}=j} I_{(A \to B_1:f_1 \dots B_K:f_K, \boldsymbol{l}, x)}. \tag{35}$$

$$I_{(A \to B_1:f_1 \dots B_K:f_K, \boldsymbol{l}, x)} = p(A \to B_1{:}f_1 \dots B_K{:}f_K \mid x, \boldsymbol{t}_{-i}) \prod_{u=1}^{K} I_{(B_u, l_u, l_{u+1}, f_u(x))}. \tag{36}$$

If $f_u(x)$ returns failure, then $I_{(B_u,l_u,l_{u+1},f_u(x))} = 0$. Note that in the case that A is a preterminal,

$$I_{(A \to w, l_1, l_2, x)} = \mathbb{1}\{w \text{ matches } y_i \text{ at } (l_1, l_2)\}\, p(A \to w \mid \boldsymbol{t}_{-i}). \tag{37}$$

where w is a terminal. Aside from the inside probabilities that were required to compute the root inside probability $I_{(\mathsf{S},0,|y_i|,x_i)}$, all other inside probabilities are 0. In our code, this recursion is implemented iteratively, in order to avoid any issues with limited stack size and to share code with the parsing and generation algorithms. We also take care not to recompute previously computed inside probabilities.

All that remains is the outside step: sample the derivation tree using the computed inside probabilities. To do so, start with the root nonterminal S at positions $i = 0$ to $j = |y_i|$ and logical form x_i, and consider all production rules with S on the left-hand side $\mathsf{S} \to B_1{:}f_1 \dots B_K{:}f_K$ and all sentence boundaries between the right-hand side nonterminals $l_1 < \dots < l_K < l_{K+1}$ where $l_1 = i$ and $l_{K+1} = j$. Sample a production rule and sentence boundaries with probability proportional to the inside rule probability $I_{(\mathsf{S} \to B_1:f_1 \dots B_K:f_K, \boldsymbol{l}, x_i)}$. Next, consider each right-hand side nonterminal of the selected rule B_u, start position l_u, end position l_{u+1}, and logical form $f_u(x_i)$, and recursively repeat the sampling procedure. The end result is a tree sampled from equation 33.

4.2 Parsing

For a new sentence y_*, we aim to find the logical form x_* and derivation t_* that maximizes

$$p(x_*, t_* \mid y_*, \boldsymbol{x}, \boldsymbol{y}) = \int p(x_*, t_* \mid y_*, \boldsymbol{t}) p(\boldsymbol{t} \mid \boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{t}, \tag{38}$$

$$\approx \frac{1}{N_{\text{samples}}} \sum_{\boldsymbol{y} \sim \boldsymbol{t} \mid \boldsymbol{x}, \boldsymbol{y}} p(x_*, t_* \mid y_*, \boldsymbol{t}). \tag{39}$$

These samples of $\boldsymbol{t}$ are obtained from the above training procedure. For the parsing approach presented in this section, it is assumed that $N_{\text{samples}} = 1$, and so $p(x_*, t_* \mid y_*, \boldsymbol{x}, \boldsymbol{y}) \approx p(x_*, t_* \mid y_*, \boldsymbol{t})$, where $\boldsymbol{t}$ is the last MH sample from the training procedure. Thus, we can write the objective function for parsing:

$$\begin{aligned} p(x_*, t_* \mid y_*, \boldsymbol{t}) &\propto p(x_*) p(y_* \mid t_*) p(t_* \mid x_*, \boldsymbol{t}), \\ &= \mathbb{1}\{\text{yield}(t_*) = y_*\} p(x_*) p\left(\bigcap_{\mathbf{n} \in t_*} r^{\mathbf{n}} \;\middle|\; x_*^{\mathbf{n}}, \boldsymbol{t} \right), \qquad (40) \\ &\approx \mathbb{1}\{\text{yield}(t_*) = y_*\} p(x_*) \prod_{\mathbf{n} \in t_*} p(r^{\mathbf{n}} \mid x_*^{\mathbf{n}}, \boldsymbol{t}). [1] \qquad (41) \end{aligned}$$

This is a discrete optimization problem, which we solve using *branch-and-bound* (see algorithm 1). The algorithm starts by considering the set of all derivation trees of y_* and partitions it into a number of subsets (the "branch" step). For each subset S, we compute an upper bound on the log probability of any derivation in S (the "bound" step). This bound is given by equations 43, 44, and 45. Having the computed the bound for each subset, we push them onto a priority queue, prioritized by the bound. We then pop the subset with the highest bound and repeat this process, further subdividing this set into subsets, computing the bound for each subset, and pushing them onto the queue. Eventually, we will pop a subset containing a single derivation which is provably optimal, if its objective function value according to the above equation is at least the priority of the next item in the queue. We can continue the algorithm to obtain the top-k derivations/logical forms. Since this algorithm operates over *sets* of logical forms (where each set is possibly infinite), we must implement a data structure to sparsely represent such sets of formulas, as well as algorithms to perform set operations, such as intersection and subtraction.

Each set of derivations is sparsely represented in our implementation as a single incomplete derivation tree (i.e. the leaf nodes may be either terminals or nonterminals) and a logical form set. The logical form set represents the logical form of the root node of every derivation tree in the set. The logical forms at the other nodes can be computed by using the semantic transformation functions. In addition, every nonterminal node with non-zero children has two integer indices that indicate its start and end positions in the sentence y_*. This data structure represents the set of all derivation trees whose

1 Equations 40 and 41 are not equal since the conditional distribution of production rules is not necessarily i.i.d. Our semantic parsing model uses an HDP for this conditional distribution, which has the nice property that as the size of the training set increases, this approximation becomes more exact.

nodes match the nodes in the incomplete derivation tree at the given sentence positions. In addition, each derivation tree set has an integer counter to indicate to the `branch` function how to subdivide the set. Each such set of derivation trees is also called a *search state*. As an example, consider the input sentence "Trenton is the capital of New Jersey." Now consider a search state where the incomplete derivation tree contains only a single node labeled NP with start position 3 and end position 7 (corresponding to "capital of New Jersey") and the logical form set is the set of all logical forms. This search state represents the set of all derivation trees that have a node with label NP and is the common ancestor of the terminals in "capital of New Jersey."

Given a set of derivation trees, the `branch` function is defined in algorithm 2. The branch-and-bound algorithm is started with a derivation tree set whose incomplete derivation tree has a single root node with nonterminal S, the set of all logical forms, start position 0, and end position $|y_*|$.

Algorithm 2: Pseudocode for `branch` in the branch-and-bound algorithm for the parser, which aims to maximize equation 41.

function `branch`(*derivation tree set* S)
  L is an empty list
  n is the root of the incomplete derivation tree of S
  X is the logical form set at **n**
  i is the start sentence position of **n**
  j is the end sentence position of **n**
  if ***n*** *has no child nodes*
    A is the nonterminal symbol of **n**
    return `expand(`A, i, j, X`) /* see algorithm 3 */`
  else if ***n*** *has a nonterminal child node with no children*
    $A \to B_1{:}f_1 \ldots B_K{:}f_K$ is the production rule at **n**
    $\mathbf{c}_k$ is the first nonterminal child node of **n** with no children
    B_k is the nonterminal symbol of $\mathbf{c}_k$
    i_k is the start sentence position of $\mathbf{c}_k$
    j_k is the end sentence position of $\mathbf{c}_k$
    m is the counter of S
    S_m is the m^{th} most probable set of derivation trees with root nonterminal B_k, start position i_k, end position j_k, whose logical forms are a subset of $f_k(X) \triangleq \{f_k(x) \neq \textit{fail} : x \in X\}$, according to equation 41
    if S_m *exists*
      for *sentence positions* j_{k+1} *such that* $j_k < j_{k+1} < j$ **do**
        `/* the operation` $X \cap f_k^{-1}(X_m)$ `can return a union of sets */`
        let $X_{k,1} \cup \ldots \cup X_{k,r}$ be the output of $X \cap f_k^{-1}(X_m)$ where X_m is the logical form set of S_m, and $f_k^{-1}(X_m) \triangleq \{x : f_k(x) \in X_m\}$
        for $X_{k,l} \in \{X_{k,1}, \ldots, X_{k,r}\}$ **do**
          S^* is a new derivation tree set with counter 1, the incomplete derivation tree is identical to that of S except $\mathbf{c}_k$ is substituted with the incomplete derivation tree of S_m, the logical form set at the root is $X_{k,l}$, and the end position of $\mathbf{c}_{k+1}$ is j_{k+1}
          `L.add(`S^*`)`
        `L.add(` a new derivation tree set identical to S except its counter is $m+1$`)`
  else
    $A \to B_1{:}f_1 \ldots B_K{:}f_K$ is the production rule at **n**
    m is the counter of S
    X_m is the m^{th} most likely set of logical forms according to $p(A \to B_1{:}f_1 \ldots B_K{:}f_K \mid x \in X, \boldsymbol{t})$
    if X_m *exists*
      `L.add(` a new derivation tree set identical to S except its logical form set is X_m, and is marked as `COMPLETE )`
      `L.add(` a new derivation tree set identical to S except its counter is $m+1$`)`
  return L

Algorithm 3: Pseudocode for the `expand` helper function, which algorithm 2 invokes.

```
1  function expand(nonterminal A, start position i, end position j, logical form set X)
2      L is an empty list
3      if A is a preterminal
4          for rules A → w where the tokens in the sentence y* matches the terminal w at positions i to j do
               /* if using the morphology model, we instead require that w is a valid
                  morphological parse of the tokens in the sentence y* at positions i to j */
5              S* is a new derivation tree set where the incomplete derivation tree consists of a root node n with
                nonterminal A, start position i, end position j, logical form set X, and child node w
6              L.add(S*)
7      else
8          for rules A → B1:f1 ... BK:fK and sentence positions k such that i < k < j do
9              S* is a new derivation tree set with counter 1, the incomplete derivation tree consists of a root
                node n with nonterminal A, start position i, end position j, logical form set X, and for each child
                node c_i, the nonterminal is B_i, and logical form set is f_i(X) ≜ {f_i(x) ≠ fail : x ∈ X}; the start
                position of c_1 is i, the end position of c_1 is k, and the end position of c_K is j
10             L.add(S*)
11     return L
```

Algorithm 4: A modified branch-and-bound algorithm to return the k^{th} best element(s) that maximize(s) the function f. Before the first call to this function, C is initialized as an empty list, and Q is initialized with a single element: Q.`push`$(X, h(X))$ where X is the domain on which to maximize f. The changes to C and Q persist across subsequent calls to `get_kth_best`.

```
1  function get_kth_best(objective function f,
                         heuristic h,
                         priority queue Q,
                         list of completed elements C,
                         integer k)
2      if there are at least k elements in C with priority at least the highest priority in Q
3          return k-best element in C
4      while Q not empty do
5          (S, v) = Q.pop()
6          if S is marked as COMPLETE
7              C.add(x, f(x))
8          else
9              (S_1, ..., S_n) = branch(S)
10             for i = 1, ..., n do
11                 Q.push(S_i, h(S_i))
           /* check termination condition */
12         if there are at least k elements in C with priority at least v
13             return k-best element in C
14     return ∅
```

There are two missing pieces in algorithm 2: the first is on line 17. To compute this, we can augment the branch-and-bound algorithm to return the m^{th} best element(s) that maximize(s) an objective function over a set. This augmented function is shown in algorithm 4. Whenever line 17 is first executed for a given nonterminal B_k, start position i_k, end position j_k and logical form set $f_k(X)$, initialize the priority queue Q in algorithm 4 with: Q.`push`$(S^*, h(S^*))$ where S^* is the search state with an incomplete derivation tree consisting of a single node at the root with nonterminal B_k, start position i_k, end position j_k, and logical form set $f_k(X)$.

The other missing piece in algorithm 2 is line 28, which depends on the model for selecting production rules. Our semantic parsing model uses an HDP model, and is able to directly use the algorithm described in section 2.2 to compute X^*. Algorithm 4 may also be used here to return the m^{th} most likely logical form(s).

The above branch-and-bound algorithm requires a heuristic function that, for an input search state (set of derivation trees), returns an upper bound on the objective function in equation 41 over all derivation trees in the set. This heuristic function determines the order of the search states to visit. The product in the objective $\prod_{\mathbf{n} \in t_*} p(r^{\mathbf{n}} \mid x_*^{\mathbf{n}}, \boldsymbol{t})$ can be decomposed accordingly into a product of two components: (1) the *inner probability* at a node $\mathbf{n} \in t^*$ is the product of the terms that correspond to the subtree rooted at $\mathbf{n}$, and (2) the *outer probability* is the product of the remaining terms, which correspond to the parts of t_* outside of the subtree rooted at $\mathbf{n}$.

To help define this heuristic, we define an upper bound on the log inner probability $I_{(A,i,j)}$ for any derivation tree rooted at nonterminal A at start position i and end position j in the sentence.

$$I_{(A,i,j)} \triangleq \max_{A \to B_1 \ldots B_K} \Bigg(\max_{x'} \log p(A \to B_1, \ldots, B_K \mid x', \boldsymbol{t}) + \max_{l_2 < \ldots < l_K} \sum_{k=1}^{K} I_{(B_k, l_k, l_{k+1})} \Bigg), \tag{42}$$

where $l_1 = i$, $l_{K+1} = j$. Note that the left term is a maximum over all logical forms x', and so this upper bound only considers syntactic information. Computing the left term depends on the model for selecting production rules. Since our semantic parsing model uses an HDP, it uses the branch-and-bound approach in section 2.2 to compute this term. The right term can be maximized using dynamic programming with running time $\mathcal{O}(K^2)$. As such, classical syntactic parsing algorithms can be applied to compute I for every chart cell in $\mathcal{O}(n^3)$. For any terminal symbol $w \in \mathcal{W}$, we define $I_{(w,i,j)} = 0$.

We now define the upper bound heuristic on any search state S with an incomplete derivation tree that has root node $\mathbf{n}$, start position i, end position j, and logical form set X. If $\mathbf{n}$ has no child nodes:

$$\log h(S) \triangleq h_x^{\mathbf{n}}(X) + I_{(A,i,j)}. \tag{43}$$

Else, if $\mathbf{n}$ has a nonterminal child node without children, the production rule at $\mathbf{n}$ is $A \to B_1{:}f_1 \ldots B_K{:}f_K$, k is the smallest index of a nonterminal child node, and m is the value of the counter:

$$\log h(S) \triangleq \min\{h_x^{\mathbf{n}}(X) + I_{(B_k,l_k,l_{k+1})}, \log \eta_{m-1}\} + \rho + \max_{l_{k+2} < \ldots < l_{K+1}} \sum_{u=k+1}^{K} I_{(B_u,l_u,l_{u+1})}. \tag{44}$$

Else, if all the nonterminal child nodes of $\mathbf{n}$ has children, and m is the value of the counter:

$$\log h(S) \triangleq h_x^{\mathbf{n}}(X) + \rho + \log \mu_{m-1}. \tag{45}$$

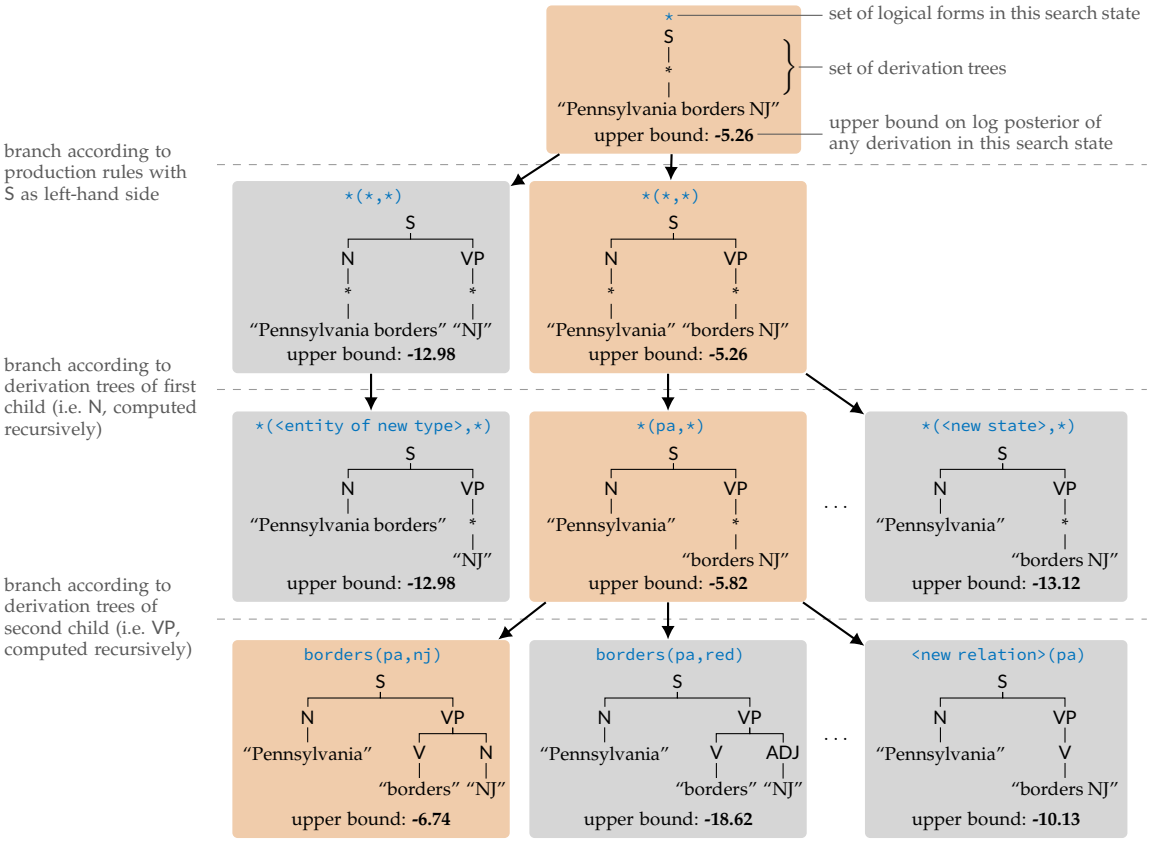

Figure 4
The search tree of the branch-and-bound algorithm during parsing. In this diagram, each block is a search state, which represents a set of derivation trees. The blue asterisk $\star$ denotes the set of all possible logical forms, whereas the black asterisk * denotes the set of all possible derivation (sub)trees. Note only the logical form at the root node is shown. The gray-colored search states are unvisited by the parser, since their upper bounds on the log posterior are smaller than that of the completed parse at the bottom of the diagram (-6.74), thus allowing the parser to ignore a very large number of improbable logical forms and derivations. In this example, we use the grammar from figure 1. The branching steps here are simplified for the sake of illustration. The recursive optimization of the derivation subtrees for N and VP are not shown, which have their own respective search trees.

where

$$\rho \triangleq \sum_{\mathbf{m} \in S \setminus \mathbf{n}} \log p(r^{\mathbf{m}} \mid x \in X, \mathbf{t}),$$

$h_x^{\mathbf{n}}(X) \geq \max_{x \in X} \log p(x^{\mathbf{n}})$ is an upper bound on the semantic prior,

$\eta_m \triangleq$ the objective function value of the m^{th} most probable derivation trees obtained on line 17,

$\mu_m \triangleq m^{th}$ highest value of $p(A \to B_1{:}f_1 \dots B_K{:}f_K \mid x \in X, \boldsymbol{t})$ obtained on line 28.

The max in the third term of equation 44 can be computed via dynamic programming with running time $\mathcal{O}(K^2)$. In the equation for ρ, the sum over $\mathbf{m} \in S \setminus \mathbf{n}$ is over all nodes in the incomplete derivation tree of S, excluding $\mathbf{n}$. To avoid recomputing ρ every time h is invoked, our implementation stores it in every search state. Its initial value is 0. In algorithm 2, on line 22, the log probability of the new search state $S*$ is equal to the sum of the log probability of the old state S and the log probability of S_m. In line 30, the log probability of the new search state is equal to the sum of the log probability of the old search state S and $\log p(A \to B_1{:}f_1 \ldots B_K{:}f_K \mid x \in X, \boldsymbol{t})$. Our implementation then uses this quantity directly as ρ in the above heuristic. The heuristic also has the nice property that when a search state is marked `COMPLETE`, its heuristic value is equal to the logarithm of the objective, aside from the prior term. Thus, when computing the objective function, such as checking the termination condition in the branch-and-bound, we only need to compute the prior term.

With a sufficiently tight upper bound on the objective, this algorithm ignores a very large number of subproblems whose upper bound is too high. Figure 4 shows the search tree for the branch-and-bound algorithm. By ignoring sets of derivation trees with an upper bound smaller than that of the highest-scoring element in the search queue, the parser can ignore a large number of improbable logical forms and derivations. Thus, with a good upper bound, the parser can run in sublinear time with respect to the size of the theory. The parser resembles a generalized version of the Earley parsing algorithm (Earley 1970).

4.3 Generating sentences

In contrast with parsing, given a new logical form x_*, natural language generation is the task of finding the unknown sentence y_* and derivation tree t_*. A straightforward way to do this in our model is to sample $t_* \mid \boldsymbol{t}, x_*$, and simply compute $y_* = \text{yield}(t_*)$. The sampling follows the generative process directly.

However, in many situations, it is desirable to find the sentence y_* and derivation t_* that maximize:

$$p(y_*, t_* \mid x_*, \boldsymbol{x}, \boldsymbol{y}) = \int p(y_*, t_* \mid x_*, \boldsymbol{t}) p(\boldsymbol{t} \mid \boldsymbol{x}, \boldsymbol{y}), \tag{46}$$

$$\approx \frac{1}{N_{\text{samples}}} \sum_{\boldsymbol{t} \sim \boldsymbol{t} \mid \boldsymbol{x}, \boldsymbol{y}} p(y_*, t_* \mid x_*, \boldsymbol{t}), \tag{47}$$

$$p(y_*, t_* \mid x_*, \boldsymbol{t}) \approx \mathbb{1}\{\text{yield}(t_*) = y_*\} \prod_{\mathbf{n} \in t_*} p(r^{\mathbf{n}} \mid x_*^{\mathbf{n}}, \boldsymbol{t}). \tag{48}$$

As with parsing, we assume that $N_{\text{samples}} = 1$. This is also a discrete optimization problem, albeit simpler than parsing, and we again apply branch-and-bound. Similar to the case in parsing, each search state represents a set of derivation trees, represented by an incomplete derivation tree, except that the nodes do not have any sentence positions, since y_* is not known, and there is only a single logical form rather than a set of logical forms, since x_* is known. The `branch` function for generation is shown in algorithm 5. The algorithm is started with a derivation tree set whose incomplete derivation tree consists of a single node labeled S with logical form x_*.

Algorithm 5: Pseudocode for `branch` and `expand` in the branch-and-bound algorithm for generating the most likely sentence(s), given a logical form, which aims to maximize equation 48.

```
function branch(derivation tree set S)
    L is an empty list
    n is the root of the incomplete derivation tree of S
    x is the logical form at n
    if n has no child nodes
        A is the nonterminal symbol of n
        return expand(A, x)
    else if n has a nonterminal child node with no children
        A → B_1:f_1 ... B_K:f_K is the production rule at n
        c_k is the first nonterminal child node of n with no children
        B_k is the nonterminal symbol of c_k
        m is the counter of S
        ρ is the log probability of S
        if f_k(x) fails return ∅
        S_m is the m^th most probable derivation tree with root nonterminal B_k and logical form f_k(x),
          according to equation 48
        if S_m exists
            S* = S ∩ S_m is a new derivation tree set with counter 1, the incomplete derivation tree is identical
              to that of S except c_k is substituted with the incomplete derivation tree of S_m, and the log
              probability is the sum of ρ and the log probability of S_m
            L.add(S*)
            L.add( a new derivation tree set identical to S except its counter is m + 1)
    else
        A → B_1:f_1 ... B_K:f_K is the production rule at n
        ρ is the log probability of S
        L.add( a new derivation tree set identical to S except its log probability is the sum of ρ and
          p(A → B_1:f_1 ... B_K:f_K | x, t), and is marked COMPLETE )
    return L
function expand(nonterminal A, logical form x)
    L is an empty list
    if A is a preterminal
        for rules A → w do
            S* is a new derivation tree set where the incomplete derivation tree consists of a root node n with
              nonterminal A, logical form x, and child node w
            L.add(S*)
    else
        for rules A → B_1:f_1 ... B_K:f_K do
            S* is a new derivation tree set with counter 1, the incomplete derivation tree consists of a root
              node n with nonterminal A, logical form x, and for each child node c_i, the nonterminal is B_i, and
              logical form is f_i(x)
            if f_i(x) did not fail for all i = 1, ..., K
                L.add(S*)
    return L
```

The heuristic upper bound for a search state S is simply:

$$\log h(S) \triangleq \sum_{\mathbf{m} \in S \setminus \mathbf{n}} \log p(r^{\mathbf{m}} \mid x_*^{\mathbf{m}}, \boldsymbol{t}) = \rho, \tag{49}$$

where the sum is over all nodes in the incomplete derivation tree of S, excluding the root node $\mathbf{n}$. Note that, just as in parsing, the algorithm keeps track of this quantity in each search state as the log probability ρ, and so $\log h(S) = \rho$.

Sentence	"How large is Alaska?"
Logical form	`answer(A,(size(B,A),const(B,stateid(alaska))))`
Sentence	"How many people lived in Austin?"
Logical form	`answer(A,(population(B,A),const(B,cityid(austin,_))))`
Sentence	"What is the biggest city in Nebraska?"
Logical form	`answer(A,largest(A,(city(A),` `loc(A,B),const(B,stateid(nebraska)))))`
Sentence	"Give me the cities in USA?"
Logical form	`answer(A,(city(A),loc(A,B),const(B,countryid(usa))))`

Figure 5
Examples of sentences and logical form labels from GEOQUERY.

Sentence	"Show me programmer jobs in Tulsa?"
Logical form	`answer(A,(job(A),title(A,T),` `const(T,'Programmer'),loc(A,C),const(C,'tulsa')))`
Sentence	"What jobs are there with a salary of more than 50000 dollars per year?"
Logical form	`answer(A,(job(A),salary_greater_than(A,50000,year)))`
Sentence	"What jobs in Austin require more than 10 years of experience?"
Logical form	`answer(A,(job(A),loc(A,P),` `const(P,'austin'),req_exp(A,E),const(E,10)))`
Sentence	"Can I find a job making more than 40000 a year without a degree?"
Logical form	`answer(A,(job(A),` `salary_greater_than(A,40000,year),\+ req_deg(A)))`

Figure 6
Examples of sentences and logical form labels from JOBS.

Just as in parsing, in order to execute line 15, we can use the augmented branch-and-bound in algorithm 4 to return the m^{th} best derivation tree that maximizes the objective over the set of derivation trees rooted at B_k with logical form $f_k(x)$. Whenever line 17 is first executed for a given nonterminal B_k and logical form $f_k(x)$, initialize the priority queue Q in algorithm 4 with: Q`.push(`$S^*, h(S^*)$`)` where S^* is the search state with an incomplete derivation tree consisting of a single node at the root with nonterminal B_k and logical form $f_k(x)$. The implementation for our inside-outside sampler, branch-and-bound parser and generator is available at `github.com/asaparov/grammar`.

5. Semantic parsing experiments on GEOQUERY and JOBS

To evaluate our parser, we use the GEOQUERY and JOBS datasets (Zelle and Mooney 1996; Tang and Mooney 2000). GEOQUERY contains 880 questions about U.S. geography. Each question is labeled with a logical form in Datalog. The dataset includes a database called GEOBASE, which, when each logical form is executed, returns the answer to the corresponding question. The JOBS dataset contains 640 questions about computer-related job postings (from the USENET group `austin.jobs`). Each question is also labeled with a Datalog logical form, similar to the semantic formalism of GEOQUERY. Most question in the two datasets are interrogative sentences, but there are some imperative sentences. Figures 5 and 6 showcases some examples from each dataset, respectively. The task is semantic parsing: given each sentence, predict the logical form that represents its meaning.

We created a semantic grammar for the Datalog representation of GEOQUERY and JOBS, specifying the "interior" production rules and implementing the semantic transformation functions and their inverses.[2] We experiment with a simple prior for the logical forms: Let x be a Datalog logical form, and $x^{a,i}$ is the i^{th} predicate or "function" node in x in prefix order whose smallest variable is a ("smallest" in the sense that A is smaller than B is smaller than C etc). For example, `size(A,B)` is a predicate node whose smallest variable is `A`, and `most(B,C,...)` is a "function" node whose smallest variable is `B`. The prior probability of x is given by $p(x) \propto \prod_{a,i} p(x^{a,i} \mid x^{a,i-1})$ where the conditional $p(x^{a,i} \mid x^{a,i-1})$ is modeled with an HDP as in section 2.4. This HDP has height 2: the first feature function is the predicate or "function" symbol of the input node (e.g. `size` or `most`), and the second feature function is the arity and "order" of the arguments (e.g. `size(A)` vs `size(A,B)` vs `size(B,A)`).

We also follow Wong and Mooney (2007); Li, Liu, and Sun (2013); Zhao and Huang (2015) and experiment with type-checking, where every entity is assigned a type from a type hierarchy (e.g. `alaska` has type `state`, `state` has supertype `polity`, etc), and every predicate is assigned a functional type (e.g. `population` has type `polity` $\rightarrow$ `int` $\rightarrow$ `bool`, etc). We incorporate type-checking into the semantic prior by assigning zero probability to type-incorrect logical forms. More precisely, logical forms are distributed according to the original prior, conditioned on the fact that the logical form is type-correct. Type-checking requires the specification of a type hierarchy. Our hierarchy contains 11 types for GEOQUERY and 12 for JOBS. We run experiments with and without type-checking for comparison.

Following Zettlemoyer and Collins (2007), we use the same 600 GEOQUERY sentences for training and an independent test set of 280 sentences. On JOBS, we use the same 500 sentences for training and 140 for testing. We run our parser with two setups: (1) with no domain-specific supervision, and (2) using a small domain-specific lexicon and a set of beliefs (such as the fact that Portland is a city). For each setup, we run the experiments with and without type-checking, for a total of 4 experimental setups. A given output logical form is considered correct if it is semantically equivalent to the true logical form.[3] In these experiments, we did not use a model of morphology in the grammar. We measure the precision and recall of our method, where precision is the number of correct parses divided by the number of sentences for which our parser provided output, and recall is the number of correct parses divided by the total number of sentences in each dataset. Our results are shown compared against many other semantic parsers in table 1. Our method is labeled PWL-LM. The numbers for the baselines were copied from their respective papers, and so their specified lexicons/type hierarchies may differ slightly. All code for these experiments is available at `github.com/asaparov/parser`.

Many sentences in the test set contain tokens previously unseen in the training set. In such cases, the maximum possible recall is 88.2 and 82.3 on GEOQUERY and JOBS, respectively. Therefore, we also measure the effect of adding a domain-specific lexicon, which maps semantic constants like `maine` to the noun "Maine" for example. This lexicon is analogous to the string-matching and argument identification steps in some other semantic parsers. We constructed the lexicon manually, with an entry for

2 This grammar is available at `github.com/asaparov/parser/blob/master/english.gram`

3 The result of execution of the output logical form is identical to that of the true logical form, for any grounding knowledge base/possible world.

Method	Additional Supervision	GeoQuery P	GeoQuery R	GeoQuery F1	Jobs P	Jobs R	Jobs F1
WASP (Wong and Mooney 2006)	A,B	87.2	74.8	80.5			
λ-WASP (Wong and Mooney 2007)	A,B,F	92.0	86.6	89.2			
Extended GHKM (Li, Liu, and Sun 2013)	A,B,F	93.0	87.6	90.2			
Zettlemoyer and Collins (2005)	C,E,F	**96.3**	79.3	87.0	97.3	79.3	87.4
Zettlemoyer and Collins (2007)	C,E,F	91.6	86.1	88.8			
UBL (Kwiatkowski et al. 2010)	E	94.1	85.0	89.3			
FUBL (Kwiatkowski et al. 2011)	E	88.6	88.6	88.6			
Wang, Kwiatkowski, and Zettlemoyer (2014)	C,E		91.1				
TISP (Zhao and Huang 2015)	E,F	92.9	88.9	90.9	85.0	85.0	85.0
Rabinovich, Stern, and Klein (2017)	E,F		87.1			**92.9**	
Coarse2Fine (Dong and Lapata 2018)	E,F		88.2				
Platanios et al. (2021)	E,F,G		91.4			91.4	
NQG-T5-3B (Shaw et al. 2021)	E,F,G		**93.7**				
PWL-LM − lexicon − type-checking	D	86.9	75.7	80.9	89.5	67.1	76.7
PWL-LM + lexicon − type-checking	D,E	88.4	81.8	85.0	91.4	75.7	82.8
PWL-LM − lexicon + type-checking	D,F	89.3	77.9	83.2	93.2	69.3	79.5
PWL-LM + lexicon + type-checking	D,E,F	90.7	83.9	87.2	**97.4**	81.4	88.7

Legend for sources of additional supervision:
A. Training set containing 792 examples,
B. Domain-specific set of initial synchronous CFG rules,
C. Domain-independent set of lexical templates,
D. Domain-independent set of interior production rules,
E. Domain-specific initial lexicon,
F. Type-checking and type specification for entities,
G. Pre-trained on large web corpus.

Table 1
Results of semantic parsing experiments on the GeoQuery and Jobs datasets (Saparov, Saraswat, and Mitchell 2017). Precision, recall, and F1 scores are shown. The methods in the top portion of the table were evaluated using 10-fold cross validation, whereas those in the bottom portion were evaluated with an independent test set. As a consequence, the methods evaluated using 10-fold cross validation were trained on 792 GeoQuery examples and tested on 88 examples for each fold (hence the additional supervision label "A" in the above table). In contrast, the methods evaluated using an independent test set were trained on 600 GeoQuery examples and tested on 280 examples. The domain-independent set of interior production rules (labeled "D" in the above table) is described in section 3.2. Some of the above methods use the preprocessed version of data from Dong and Lapata (2016), where entity names and numbers in the training and test sets are replaced with typed placeholders. This provides the same additional information as a typed domain-specific lexicon.

Logical form:	`answer(A,smallest(A,state(A)))`	`answer(A,largest(B,(state(A),population(A,B))))`
Test sentence:	"Which state is the smallest?"	"Which state has the most population?"
Generated:	"What state is the smallest?"	"What is the state with the largest population?"

Figure 7
Examples of sentences generated from our trained grammar on logical forms in the GeoQuery test set (Saparov, Saraswat, and Mitchell 2017). Generation is performed by computing $\arg\max_{y_*, t_*} p(y_*, t_* \mid x_*, \boldsymbol{t})$ as described in section 4.3.

every city, state, river, and mountain in GeoQuery (141 entries), and an entry for every city, company, position, and platform in Jobs (180 entries).

Aside from the lexicon and type hierarchy, the only training information is given by the set of sentences $\boldsymbol{y}$, corresponding logical forms $\boldsymbol{x}$, and the domain-independent set of interior production rules, as described in section 3.2. In our experiments, we found that the sampler converges rapidly, with only 10 passes over the data. This is largely due to our restriction of the interior production rules to a domain-independent set, which provides significant information about English syntax.

We emphasize that the addition of type-checking and a lexicon are mainly to enable a fair comparison with past approaches. As expected, their addition greatly improves parsing performance. At the time of the publication of our method (Saparov, Saraswat, and Mitchell 2017), we achieved state-of-the-art F1 on the JOBS dataset. However, even without such domain-specific supervision, the parser performs reasonably well. This is a promising indication that this parser will work effectively in the broader NLU system (described in Saparov (2022)), and is able to correctly parse sentences with complex and nested semantics. However, we notice a common error is the incorrect determination of scope of functions like `highest`, `shortest`, etc. This is likely due to the fact that the semantic prior does not explicitly model the scope of these functions (it assumes a uniform probability on all possible scopes). Thus, a more explicit model of scope might further improve parsing performance. We found that the semantic parsing problem is easier if the logical forms of each sentence are more similar to the syntactic structure of that sentence. In the extreme case, the logical forms would be identical to the sentences themselves, in which case parsing would be trivial. Thus, there is an inevitable balancing act in designing a semantic grammar and logical formalism for natural language, where on one hand we want the parsing problem to be as simple as possible, but on the other hand, we want the logical forms to be useful for downstream tasks, such as question-answering and reasoning, and ideally the logical forms of two distinct sentences that have the same meaning should be equivalent. These are important lessons to keep in mind when designing a domain-general semantic grammar and logical formalism.

6. Related work

Our grammar formalism can be related to synchronous CFGs (SCFGs) (Aho and Ullman 1972), where the semantics and syntax are generated simultaneously. However, instead of modeling the joint probability of the logical form and natural language utterance $p(x, y)$, we model the factorized probability $p(x)p(y \mid x)$, where the logical form x may have its own complex prior distribution $p(x)$. Modeling each component in isolation provides a cleaner division between syntax and semantics, and one half of the model can be modified without affecting the other, and this is instrumental in larger NLU model (described in Saparov (2022)) since in that model, the logical form is derived from a larger theory containing background knowledge. We used a CFG in the syntactic portion of our model. Note that due to the coupling with semantics, our formalism is more powerful than purely syntactic CFGs: The sets of strings generated by grammars in our formalism is strictly larger than those generated by plain CFGs. In fact, any indexed grammar can be converted into a grammar in our formalism, where the stack of indices can be interpreted as the logical form (Aho 1968). Linear indexed grammars (LIGs) are strictly less powerful than indexed grammars, and are weakly equivalent to combinatory categorial grammars (CCGs), head grammars, and tree-adjoining grammars (Vijay-Shanker and Weir 1994), which in turn are strictly more powerful than CFGs. Richer syntactic formalisms such as CCGs (Steedman 1997) or head-driven phrase structure grammars (HPSGs) (Proudian and Pollard 1985) could replace the syntactic component in our framework and may provide a more uniform analysis across languages. Our model is similar to lexical functional grammar (LFG) (Kaplan and Bresnan 1995), where f-structures are replaced with logical forms. Nothing in our model precludes incorporating syntactic information like f-structures into the logical form, and as such, LFG is realized in our framework. Including a model of morphology in our grammar furthers the comparison to LFG. Our approach can be

used to define new generative models of these grammatical formalisms. We implemented our method with a particular semantic formalism, but the grammatical model is agnostic to the choice of semantic formalism or the language. As in some previous parsers, our parsing problem can be related to the problem of finding shortest paths in hypergraphs using A* search (Klein and Manning 2001, 2003; Pauls and Klein 2009; Pauls, Klein, and Quirk 2010; Gallo, Longo, and Pallottino 1993).

7. Future work

There is significant room for future work and exploration in the subject presented in this manuscript. In this section, we discuss shortcomings of various aspects of our approach, and give suggestions for how to overcome them.

The performance of our parser and generator depend heavily on the production rules of the grammar. Although the preterminal production rules are induced during training, we had to specify the other production rules by hand. While this does give us a great deal of control over the grammar, and enables us to incorporate prior knowledge about the English language into the grammar, it is very time-consuming. It would be valuable to look into ways in which these production rules can be induced from data. Recall that every production rule in our grammar is annotated with semantic transformation functions. These functions are intimately tied with the semantic formalism and effectively implement a theory of formal semantics. It would also be valuable to explore whether these transformation functions can be learned as well. One promising direction would be to decompose the semantic transformation functions into a sequence of elementary "instructions." Each semantic transformation function could then be equivalently written as short programs in a simple programming language. We could then induce the semantic transformation functions by searching over the space of these short programs, perhaps by attempting to add or remove instructions, etc. However, it is not clear how much grammar induction would improve our current grammar for English. But such an approach would certainly help to learn grammar for other languages, about which we have much less knowledge. The statistical efficiency of our approach could greatly aid in natural language processing for low-resource languages, for which training data is very scarce.

During parsing, our method uses an upper bound on the objective function (as defined in equations 43, 44, and 45) that takes into account syntactic information. While this works well enough for our purposes, it may be possible to further improve the performance of the parser by defining tighter upper bound, possibly by taking into account semantic information.

Our semantic parsing model assumes that the sentences are noise-less: there are no spelling or grammatical errors in the utterances. This assumption helps to simplify the problem and to focus the scope of the thesis more onto language understanding and reasoning. But real-world language is noisy, and thus further work to extend the semantic parsing model to noisy settings is warranted. To properly handle grammatical errors, additional "incorrect" production rules must be added to the grammar, such as a rule where the grammatical number of the subject noun and the verb do not agree, or a rule where the subject is dropped entirely (and left to be inferred from context). Grammar induction could be used to learn these "incorrect" production rules. A possible way to handle spelling errors is to add another step to the generative process (as described in section 3.1). This extra step would take the correctly-spelled sentence as its input and create errors, such as insertions, deletions, or substitutions of characters. During inference, this process is inverted: Given the noisy sentence as input, the parser

first needs to infer the correctly-spelled sentence (which is now latent), and then proceed with the parsing algorithm as described earlier in this chapter.

The syntactic component of our grammatical formalism is a CFG, which is a *projective* model of grammar. That is, in any derivation tree of a sentence, the leaves of any subtree form a contiguous substring of the sentence. For example, in the sentence "John saw a dog which was a Yorkshire Terrier yesterday," the object noun phrase is "dog which was a Yorkshire Terrier," which appears contiguously in the sentence. However, natural languages exhibit non-projectivity, such as in the example "John saw a dog yesterday which was a Yorkshire Terrier," where the object noun phrase is now split by the adverb "yesterday" (McDonald et al. 2005). However, techniques such as feature passing can be used to model non-projective phenomena such as syntactic movement in non-transformational models of grammar (Gazdar 1981). In principle it is also possible to replace the CFG with a non-projective grammar formalism such as a mildly non-projective dependency grammar (Kuhlmann 2013; Bodirsky, Kuhlmann, and Möhl 2005).

7.1 Modeling context

The logical forms in our model are assumed to be context-independent. Conditioned on the theory, they are independently and identically distributed. This assumption greatly simplifies the natural language that we need to be able to parse. While it helps to focus the scope of the thesis, it is not representative of real-world language. In real language, the distribution of a sentence is highly dependent on the sentences that precede it, even when conditioned on the theory, which contains all of the background knowledge. For example, this assumption disallows inter-sentential coreference (e.g. pronouns that can refer to objects mentioned in other sentences). Our model also assumes that the universe of discourse does not vary, and so the sentence "All of the children are asleep" would mean that, literally, every child in the universe is sleeping. The more likely meaning of the sentence is that all of the children within the local area, such as the home or town, are sleeping. The definite article "the" often indicates the uniqueness of an object: "the tallest mountain" indicates that there is exactly one tallest mountain. However, this is not the case in the example: "A cat walked into the room. The cat purred." Here, "the cat" does not imply that there is exactly one cat in the universe. Rather, it means that the cat is unique in the context. The universe of discourse can change across sentences (and sometimes even within sentences). Relaxing the assumption that logical forms are context-independent would enable our parser to correctly understand these example sentences. To relax this assumption, our model must be augmented with a model of context. See Saparov (2022) for a more concrete proposal for a model of context.

A. Gibbs sampling for the Dirichlet process

The *Chinese restaurant process* (CRP) representation of the Dirichlet process enables efficient inference using *Markov chain Monte Carlo* (MCMC) methods. Suppose that we are given $\boldsymbol{y} \triangleq \{y_1, \ldots, y_n\}$ observations and we wish to infer the values of the latent variables: ϕ_i and z_i. A Gibbs sampling algorithm can be derived, where initial values for ϕ_i and z_i are selected, $\phi_i^{(0)}$ and $z_i^{(0)}$, and for each iteration t, we sample new values of $\phi_i^{(t)}$ and $z_i^{(t)}$. One straightforward initialization for $\phi_i^{(0)}$ and $z_i^{(0)}$ is to assign each observation to its own table: $\phi_i^{(0)} = y_i$ and $z_i^{(0)} = i$ for $i = 1, \ldots, n$. Note that the value of $\phi_i^{(t)}$ is deterministic and equal to $y_{z_j^{(t)}}$ for all $z_j^{(t)} = i$. Thus, only $z_i^{(t)}$ needs to be sampled at each iteration (for all $i = 1, \ldots, n$). In Gibbs sampling, each random variable is sampled

from its conditional distribution given all other variables: $z_i^{(t+1)} \sim z_i \mid \boldsymbol{\phi}^{(t)}, \boldsymbol{z}_{-i}^{(t)}, \boldsymbol{y}$.

$$p(z_i \mid \boldsymbol{\phi}, \boldsymbol{z}_{-i}, \boldsymbol{y}) \propto p(\boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{y}), \tag{50}$$

$$= p(z_1, \ldots, z_n) \prod_{j=1} p(\phi_j) \prod_{j=1}^{n} p(y_j \mid \boldsymbol{\phi}, z_j), \tag{51}$$

$$\propto p(z_{\pi(1)}, \ldots, z_{\pi(n)}) \mathbb{1}\{y_i = \phi_{z_i}\}, \tag{52}$$

$$p(z_i = k \mid \boldsymbol{\phi}, \boldsymbol{z}_{-i}, \boldsymbol{y}) \propto \begin{cases} \mathbb{1}\{y_i = \phi_k\} \frac{n_k}{\alpha + n - 1} & \text{if } n_k > 0, \\ \mathbb{1}\{y_i = \phi_k\} \frac{p(\phi_k = y_i)\alpha}{\alpha + n - 1} & \text{if } n_k = 0, \end{cases} \tag{53}$$

where n_k is the number of customers sitting at table k not including the i^{th} customer, $\boldsymbol{\phi} \triangleq \{\phi_1, \phi_2, \ldots\}$ and $\boldsymbol{z}_{-i} = \boldsymbol{z} \setminus \{z_i\}$ is the set of all z_j *except* z_i, and $\mathbb{1}\{\cdot\}$ is 1 if the condition is true and zero otherwise. In this derivation, we used exchangeability to change the order of the table assignments $\boldsymbol{z}$ so that z_i is the last assignment. After sufficiently many iterations, the distribution of the samples $\phi_i^{(t)}$ and $z_i^{(t)}$ will approach the true posterior $p(\boldsymbol{\phi}, \boldsymbol{z} \mid \boldsymbol{y})$.

B. Gibbs sampling for the hierarchical Dirichlet process

The Gibbs sampling update can be derived similarly to the DP case: Given $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ can be computed deterministically. Thus we only need to sample each $z_i^{\mathbf{n}}$:

$$p(z_i^{\mathbf{n}} \mid \boldsymbol{\phi}, \boldsymbol{\psi}, \boldsymbol{z}^{-\mathbf{n}}, \boldsymbol{z}_{-i}^{\mathbf{n}}, \boldsymbol{x}, \boldsymbol{y}) \propto p(\boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{\psi}, \boldsymbol{x}, \boldsymbol{y}), \tag{54}$$

$$= \prod_{j=1} p(\phi_j) \prod_{\mathbf{n}} p(z_1^{\mathbf{n}}, z_2^{\mathbf{n}}, \ldots) \prod_{j=1} p(\psi_j^{\mathbf{n}} | \boldsymbol{\phi}, \boldsymbol{\psi}^{-\mathbf{n}}, z_j^{\mathbf{n}}) \prod_{j=1} p(y_j | \boldsymbol{\psi}, x_j), \tag{55}$$

$$\propto \begin{cases} p(z_{\pi(1)}^{\mathbf{0}}, z_{\pi(2)}^{\mathbf{0}}, \ldots) \mathbb{1}\{\psi_i^{\mathbf{0}} = \phi_{z_i^{\mathbf{0}}}\} & \text{if } \mathbf{n} = \mathbf{0}, \\ p(z_{\pi(1)}^{\mathbf{n}}, z_{\pi(2)}^{\mathbf{n}}, \ldots) \mathbb{1}\{\psi_i^{\mathbf{n}} = \psi_{z_i^{\mathbf{n}}}^{\text{parent}(\mathbf{n})}\} & \text{otherwise}, \end{cases} \tag{56}$$

$$p(z_i^{\mathbf{n}} = k \mid \boldsymbol{\phi}, \boldsymbol{\psi}, \boldsymbol{z}^{-\mathbf{n}}, \boldsymbol{z}_{-i}^{\mathbf{n}}, \boldsymbol{x}, \boldsymbol{y}) \propto \tag{57}$$

$$\begin{cases} \mathbb{1}\{\psi_i^{\mathbf{0}} = \phi_k\} \frac{n_k^{\mathbf{0}}}{\alpha^{\mathbf{0}} + n^{\mathbf{0}}} & \text{if } \mathbf{n} = \mathbf{0}, n_k^{\mathbf{n}} > 0, \\ \mathbb{1}\{\psi_i^{\mathbf{0}} = \phi_{\text{new}}\} \frac{p(\phi_{\text{new}} = \psi_i^{\mathbf{0}})\alpha^{\mathbf{0}}}{\alpha^{\mathbf{0}} + n^{\mathbf{0}}} & \text{if } \mathbf{n} = \mathbf{0}, n_k^{\mathbf{n}} = 0, \\ \mathbb{1}\{\psi_i^{\mathbf{n}} = \psi_k^{\text{parent}(\mathbf{n})}\} \frac{n_k^{\mathbf{n}}}{\alpha^{\mathbf{n}} + n^{\mathbf{n}}} & \text{if } \mathbf{n} \neq \mathbf{0}, n_k^{\mathbf{n}} > 0, \\ \mathbb{1}\{\psi_i^{\mathbf{n}} = \psi_{\text{new}}^{\text{parent}(\mathbf{n})}\} \frac{f^{\text{parent}(\mathbf{n})}(\psi_i^{\mathbf{n}})\alpha^{\mathbf{n}}}{\alpha^{\mathbf{n}} + n^{\mathbf{n}}} & \text{if } \mathbf{n} \neq \mathbf{0}, n_k^{\mathbf{n}} = 0, \end{cases} \tag{58}$$

where $n_k^{\mathbf{n}}$ is the number of customers at node $\mathbf{n}$ sitting at table k *not including* the customer currently being resampled, $n^{\mathbf{n}}$ is the total number of customers at node $\mathbf{n}$ (also not including the current customer), and $f^{\mathbf{m}}(v)$ is shorthand for $p(\psi_{\text{new}}^{\mathbf{m}} = v \mid \boldsymbol{\phi}, \boldsymbol{\psi}, \boldsymbol{z}^{-\mathbf{n}}, \boldsymbol{z}_{-i}^{\mathbf{n}}, \boldsymbol{x}, \boldsymbol{y})$ for any node $\mathbf{m}$. Note that in the case where $\mathbf{n} \neq \mathbf{0}$, sampling $z_i^{\mathbf{n}}$ requires computing $f^{\text{parent}(\mathbf{n})}(\psi_i^{\mathbf{n}})$, i.e. the probability of a customer at node $\mathbf{n}$ choosing to sit a "new" table $p(\psi_{\text{new}}^{\text{parent}(\mathbf{n})})$. This value can be computed recursively, so if the node

$\mathbf{m} \neq \mathbf{0}$:

$$f^{\mathbf{m}}(v) = \frac{\alpha^{\mathbf{m}} f^{\text{parent}(\mathbf{m})}(v)}{\alpha^{\mathbf{m}} + n^{\mathbf{m}}} + \sum_{\{k' : n^{\mathbf{m}}_{k'} > 0\}} \frac{n^{\mathbf{m}}_{k'} \mathbb{1}\{\psi^{\text{parent}(\mathbf{m})}_{k'} = v\}}{\alpha^{\mathbf{m}} + n^{\mathbf{m}}}. \tag{59}$$

In the case where $\mathbf{m} = \mathbf{0}$:

$$f^{\mathbf{0}}(v) = \frac{\alpha^{\mathbf{0}} p(\phi_{\text{new}} = v)}{\alpha^{\mathbf{0}} + n^{\mathbf{0}}} + \sum_{\{k' : n^{\mathbf{0}}_{k'} > 0\}} \frac{n^{\mathbf{0}}_{k'} \mathbb{1}\{\phi_{k'} = v\}}{\alpha^{\mathbf{0}} + n^{\mathbf{0}}}. \tag{60}$$

If $z^{\mathbf{n}}_i$ is sampled to be a "new" table, a new customer will appear in the parent node of $\mathbf{n}$, and its table assignment must be sampled next. This new customer may itself be assigned to a new table, and so this process continues recursively until a customer sits at a non-empty table, or a customer sits at an empty table at the root node $\mathbf{0}$. The computation required in this recursive sampling procedure overlaps heavily with that in computing the probabilities in equations 59 and 60, so they should be done simultaneously to avoid wasted computation.

In our code, for each iteration of Gibbs sampling, we traverse the tree nodes $\mathbf{n}$ in prefix order, and resample $z^{\mathbf{n}}_i$ in random order.

In many applications, including in our semantic parsing approach, we need to compute the probability of a new observation y_{n+1}, given its source node x_{n+1} and previous observations $(\boldsymbol{x}, \boldsymbol{y})$:

$$p(y_{n+1} \mid x_{n+1}, \boldsymbol{x}, \boldsymbol{y}) = \int p(y_{n+1} \mid x_{n+1}, \boldsymbol{z}) p(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{z}, \tag{61}$$

$$\approx \frac{1}{N_{\text{samples}}} \sum_{\boldsymbol{z}^{(t)} \sim \boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{y}} p(y_{n+1} \mid x_{n+1}, \boldsymbol{z}^{(t)}, \boldsymbol{\phi}^{(t)}, \boldsymbol{\psi}^{(t)}). \tag{62}$$

The integral is approximated as a sum over posterior samples of $\boldsymbol{z}$, which can be obtained using the MCMC algorithm described above. However, we find in our experiments that the posterior is concentrated at a single point, and it suffices to keep only the final sample (i.e. $N_{\text{samples}} = 1$) as a point estimate of $\boldsymbol{z}, \boldsymbol{\psi}, \boldsymbol{\phi}$. In either case, we can compute the quantity within the sum:

$$p(y_{n+1} \mid x_{n+1}, \boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{\psi}) = p(\psi^{x_{n+1}}_{\text{new}} = y_{n+1} \mid \boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{\psi}). \tag{63}$$

This quantity can be computed as in equations 59 and 60 (but since we are not resampling $z^{\mathbf{n}}_i$, we don't exclude any customers in the $n^{\mathbf{m}}_k$ terms).

The above can be extended to the case where rather than 1 new observation, there are k new observations, and we want to compute their joint probability:

$$p\left(\bigcap_{i=1}^{k} y_{n+i} \,\middle|\, \boldsymbol{x}, \boldsymbol{y}, \bigcap_{i=1}^{k} x_{n+i} \right) = \prod_{i=1}^{k} p\left(y_{n+i} \,\middle|\, \boldsymbol{x}, \boldsymbol{y}, \bigcap_{j=1}^{i} x_{n+j}, \bigcap_{j=1}^{i-1} x_{n+j} \right). \tag{64}$$

So to compute this, first compute the probability of the first observation y_{n+1} alone. Next, add (x_{n+1}, y_{n+1}) to the HDP (treat them as part of $\boldsymbol{x}$ and $\boldsymbol{y}$) and compute the probability of y_{n+2} alone. Repeat until all k probabilities are computed and then return the

product. We observe that the joint probability does not factorize over each observation. This is due to the "rich get richer" effect observed in the Chinese restaurant process: If one observation is sampled, the same observation is more likely to be sampled in the future, since future customers are more likely to sit at tables with existing customers. And so the distribution is not i.i.d.

But as the number of customers n becomes very large, the effect of α and any single customer on the distribution of the next observation becomes negligible, and so the distribution becomes more i.i.d.:

$$\lim_{n\to\infty} p\left(\bigcap_{i=1}^{k} y_{n+i} \middle| \boldsymbol{x}, \boldsymbol{y}, \bigcap_{i=1}^{k} x_{n+i}\right) = \lim_{n\to\infty} \prod_{i=1}^{k} p\left(y_{n+i} \middle| \boldsymbol{x}, \boldsymbol{y}, \bigcap_{i=1}^{k} x_{n+i}\right). \tag{65}$$

This fact can be useful when approximating

$$p\left(\bigcap_{i=1}^{k} y_{n+i} \,\middle|\, \boldsymbol{x}, \boldsymbol{y}, \bigcap_{i=1}^{k} x_{n+i}\right) \approx \prod_{i=1}^{k} p\left(y_{n+i} \,\middle|\, \boldsymbol{x}, \boldsymbol{y}, \bigcap_{i=1}^{k} x_{n+i}\right), \tag{66}$$

when n is large.

B.1 Learning the concentration parameter α

We learn the concentration parameter from the data by placing a Gamma prior on α:

$$\alpha^{\mathbf{n}} \sim \text{Gamma}(a^{\mathbf{n}}, b^{\mathbf{n}}). \tag{67}$$

An auxiliary variable sampling method can be used to infer α, which is described in appendix A of Teh et al. (2006) and section 6 of Escobar and West (1995). The Gibbs sampling step for $\alpha^{\mathbf{n}}$ is:

$$s^{\mathbf{n}} \sim \text{Bernoulli}\left(\frac{n^{\mathbf{n}}}{\alpha^{\mathbf{n}} + n^{\mathbf{n}}}\right), \tag{68}$$

$$w^{\mathbf{n}} \sim \text{Beta}(\alpha^{\mathbf{n}} + 1, n^{\mathbf{n}}), \tag{69}$$

$$\alpha^{\mathbf{n}} \sim \text{Gamma}(a^{\mathbf{n}} + \max\{z_i^{\mathbf{n}}\} - s^{\mathbf{n}}, b^{\mathbf{n}} - \log w^{\mathbf{n}}). \tag{70}$$

Here, $\max\{z_i^{\mathbf{n}}\}$ is the number of occupied tables in restaurant $\mathbf{n}$. The above updates assume that each node $\mathbf{n}$ has an independent $\alpha^{\mathbf{n}}$. However, in many scenarios, we wish to tie the concentration parameters together to improve statistical efficiency. Suppose we constrain all the concentration parameters at each level in the hierarchy to be equal. Let $L(\mathbf{n})$ be defined as the level of the node $\mathbf{n}$ (i.e. $L(\mathbf{0}) = 0$ and $L(\mathbf{n}) = L(\text{parent}(\mathbf{n})) + 1$). Let α_i be the concentration parameter at level i, and so $\alpha^{\mathbf{n}} = \alpha_{L(\mathbf{n})}$. Let its prior be $\alpha_i \sim \text{Gamma}(a_i, b_i)$. Then, the Gibbs sampling step for α_i is:

$$\alpha_i \sim \text{Gamma}\left(a_i + \sum_{\{\mathbf{n}: L(\mathbf{n})=i\}} (\max\{z_i^{\mathbf{n}}\} - s^{\mathbf{n}}), b_i - \sum_{\{\mathbf{n}: L(\mathbf{n})=i\}} \log w^{\mathbf{n}}\right). \tag{71}$$

This is the approach we implement in our semantic parsing model during training. Our semantic parsing model has several HDP hierarchies, and each has its own set of hyperparameters a and b. As an example, for one such hierarchy in our semantic parsing model (corresponding to the nonterminal VP_R), the hyperparameters are $a_1 = 100, a_2 = 10, b_1 = 0.1, b_2 = 1$, but the other hierarchies have similar values for their hyperparameters.

References

Aho, Alfred V. 1968. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671.

Aho, Alfred V. and Jeffrey D. Ullman. 1972. *The theory of parsing, translation, and compiling. 1: Parsing*. Prentice-Hall.

Aldous, David J. 1985. Exchangeability and related topics. In *Lecture Notes in Mathematics*. Springer Berlin Heidelberg, pages 1–198.

Blunsom, Phil and Trevor Cohn. 2010. Inducing synchronous grammars with slice sampling. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA*, pages 238–241, The Association for Computational Linguistics.

Bodirsky, Manuel, Marco Kuhlmann, and Mathias Möhl. 2005. Well-nested drawings as models of syntactic structure. In *In 10th Conference on Formal Grammar and 9th Meeting on Mathematics of Language (FGMOL'05)*, page 195–203.

Chomsky, Noam. 1956. Three models for the description of language. *IRE Trans. Inf. Theory*, 2(3):113–124.

Cohn, Trevor, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *J. Mach. Learn. Res.*, 11:3053–3096.

Dong, Li and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, The Association for Computer Linguistics.

Dong, Li and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 731–742, Association for Computational Linguistics.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Escobar, Michael D. and Mike West. 1995. Bayesian density estimation and inference using mixtures. *Journal of the American Statistical Association*, 90(430):577–588.

Ferguson, Thomas S. 1973. A Bayesian Analysis of Some Nonparametric Problems. *The Annals of Statistics*, 1(2):209 – 230.

Finkel, Jenny Rose, Christopher D. Manning, and Andrew Y. Ng. 2006. Solving the problem of cascading errors: Approximate bayesian inference for linguistic annotation pipelines. In *EMNLP 2006, Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing, 22-23 July 2006, Sydney, Australia*, pages 618–626, ACL.

Gallo, Giorgio, Giustino Longo, and Stefano Pallottino. 1993. Directed hypergraphs and applications. *Discret. Appl. Math.*, 42(2):177–201.

Gazdar, Gerald. 1981. Unbounded dependencies and coordinate structure. *Linguistic Inquiry*, 12:155–184.

Johnson, Mark, Thomas L. Griffiths, and Sharon Goldwater. 2007. Bayesian inference for pcfgs via markov chain monte carlo. In *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, April 22-27, 2007, Rochester, New York, USA*, pages 139–146, The Association for Computational Linguistics.

Kaplan, Ronald M. and Joan Bresnan. 1995. Lexical-functional grammar: A formal system for grammatical representation.

Klein, Dan and Christopher D. Manning. 2001. Parsing and hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT-2001), 17-19 October 2001, Beijing, China*, Tsinghua University Press.

Klein, Dan and Christopher D. Manning. 2003. A* parsing: Fast exact viterbi parse selection. In *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL 2003, Edmonton, Canada, May 27 - June 1, 2003*, The Association for

Computational Linguistics.
Kuhlmann, Marco. 2013. Mildly non-projective dependency grammar. *Comput. Linguistics*, 39(2):355–387.
Kwiatkowski, Tom, Eunsol Choi, Yoav Artzi, and Luke S. Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1545–1556, ACL.
Kwiatkowski, Tom, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, EMNLP 2010, 9-11 October 2010, MIT Stata Center, Massachusetts, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1223–1233, ACL.
Kwiatkowski, Tom, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP 2011, 27-31 July 2011, John McIntyre Conference Centre, Edinburgh, UK, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1512–1523, ACL.
Land, A. H. and A. G. Doig. 1960. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497.
Li, Peng, Yang Liu, and Maosong Sun. 2013. An extended GHKM algorithm for inducing lambda-scfg. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, AAAI Press.
Liang, Percy, Michael I. Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Comput. Linguistics*, 39(2):389–446.
McDonald, Ryan T., Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT/EMNLP 2005, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 6-8 October 2005, Vancouver, British Columbia, Canada*, pages 523–530, The Association for Computational Linguistics.
Pauls, Adam and Dan Klein. 2009. K-best a* parsing. In *ACL 2009, Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2-7 August 2009, Singapore*, pages 958–966, The Association for Computer Linguistics.
Pauls, Adam, Dan Klein, and Chris Quirk. 2010. Top-down k-best a* parsing. In *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden, Short Papers*, pages 200–204, The Association for Computer Linguistics.
Platanios, Emmanouil Antonios, Adam Pauls, Subhro Roy, Yuchen Zhang, Alexander Kyte, Alan Guo, Sam Thomson, Jayant Krishnamurthy, Jason Andrew Wolfe, Jacob Andreas, and Dan Klein. 2021. Value-agnostic conversational semantic parsing. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 3666–3681, Association for Computational Linguistics.
Proudian, Derek and Carl Pollard. 1985. Parsing head-driven phrase structure grammar. In *23rd Annual Meeting of the Association for Computational Linguistics, 8-12 July 1985, University of Chicago, Chicago, Illinois, USA, Proceedings*, pages 167–171, ACL.
Rabinovich, Maxim, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1139–1149, Association for Computational Linguistics.
Russell, Stuart J. and Peter Norvig. 2010. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education.
Saparov, Abulhair. 2022. *Towards General Natural Language Understanding with Probabilistic Worldbuilding*. Ph.D. thesis, Carnegie Mellon University.
Saparov, Abulhair, Vijay A. Saraswat, and Tom M. Mitchell. 2017. A probabilistic generative grammar for semantic parsing. In *Proceedings of the 21st Conference on*

Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, August 3-4, 2017, pages 248–259, Association for Computational Linguistics.

Shaw, Peter, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 922–938, Association for Computational Linguistics.

Steedman, Mark. 1997. *Surface structure and interpretation*, volume 30 of *Linguistic inquiry*. MIT Press.

Tang, Lappoon R. and Raymond J. Mooney. 2000. Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing. In *Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, EMNLP 2000, Hong Kong, October 7-8, 2000*, pages 133–141, Association for Computational Linguistics.

Teh, Yee Whye. 2006. A hierarchical bayesian language model based on pitman-yor processes. In *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Sydney, Australia, 17-21 July 2006*, The Association for Computer Linguistics.

Teh, Yee Whye, Michael I. Jordan, Matthew J. Beal, and David M. Blei. 2006. Hierarchical dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581.

Vijay-Shanker, K. and David J. Weir. 1994. The equivalence of four extensions of context-free grammars. *Math. Syst. Theory*, 27(6):511–546.

Wang, Adrienne, Tom Kwiatkowski, and Luke S. Zettlemoyer. 2014. Morpho-syntactic lexical generalization for CCG semantic parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1284–1295, ACL.

Wikimedia Foundation. 2020. Wiktionary data dumps.

Wong, Yuk Wah and Raymond J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 4-9, 2006, New York, New York, USA*, The Association for Computational Linguistics.

Wong, Yuk Wah and Raymond J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, June 23-30, 2007, Prague, Czech Republic*, The Association for Computational Linguistics.

Zelle, John M. and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pages 1050–1055, AAAI Press / The MIT Press.

Zettlemoyer, Luke S. and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*, pages 658–666, AUAI Press.

Zettlemoyer, Luke S. and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 678–687, ACL.

Zhao, Kai and Liang Huang. 2015. Type-driven incremental semantic parsing with polymorphism. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 1416–1421, The Association for Computational Linguistics.