# Neural Semantic Parsing with Type Constraints for Semi-Structured Tables

**Jayant Krishnamurthy,[1] Pradeep Dasigi,[2] and Matt Gardner[1]**
[1]Allen Institute for Artificial Intelligence
[2]Carnegie Mellon University
{jayantk, mattg}@allenai.org, pdasigi@cs.cmu.edu

## Abstract

We present a new semantic parsing model for answering compositional questions on semi-structured Wikipedia tables. Our parser is an encoder-decoder neural network with two key technical innovations: (1) a grammar for the decoder that only generates well-typed logical forms; and (2) an entity embedding and linking module that identifies entity mentions while generalizing across tables. We also introduce a novel method for training our neural model with question-answer supervision. On the WIKITABLEQUESTIONS data set, our parser achieves a state-of-the-art accuracy of 43.3% for a single model and 45.9% for a 5-model ensemble, improving on the best prior score of 38.7% set by a 15-model ensemble. These results suggest that type constraints and entity linking are valuable components to incorporate in neural semantic parsers.

## 1 Introduction

Semantic parsing is the problem of translating human language into computer language, and therefore is at the heart of natural language understanding. A typical semantic parsing task is question answering against a database, which is accomplished by translating questions into executable logical forms (i.e., programs) that output their answers. Recent work has shown that recurrent neural networks can be used for semantic parsing by encoding the question then predicting each token of the logical form in sequence (Jia and Liang, 2016; Dong and Lapata, 2016). These approaches, while effective, have two major limitations. First, they treat the logical form as an unstructured sequence, thereby ignoring type constraints on well-formed programs. Second, they do not address entity linking, which is a critical subproblem of semantic parsing (Yih et al., 2015).

This paper introduces a novel neural semantic parsing model that addresses these limitations of prior work. Our parser uses an encoder-decoder architecture with two key innovations. First, the decoder generates from a grammar that guarantees that generated logical forms are well-typed. This grammar is automatically induced from typed logical forms, and does not require any manual engineering to produce. Second, the encoder incorporates an entity linking and embedding module that enables it to learn to identify which question spans should be linked to entities. Finally, we also introduce a new approach for training neural semantic parsers from question-answer supervision.

We evaluate our parser on WIKITABLEQUESTIONS, a challenging data set for question answering against semi-structured Wikipedia tables (Pasupat and Liang, 2015). This data set has a broad variety of entities and relations across different tables, along with complex questions that necessitate long logical forms. On this data set, our parser achieves a question answering accuracy of 43.3% and an ensemble of 5 parsers achieves 45.9%, both of which outperform the previous state-of-the-art of 38.7% set by an ensemble of 15 models (Haug et al., 2017). We further perform several ablation studies that demonstrate the importance of both type constraints and entity linking to achieving high accuracy on this task.

## 2 Related Work

Semantic parsers vary along a few important dimensions:

**Formalism** Early work on semantic parsing used lexicalized grammar formalisms such as Combinatory Categorial Grammar (Zettlemoyer

and Collins, 2005, 2007; Kwiatkowski et al., 2011, 2013; Krishnamurthy and Mitchell, 2012; Artzi and Zettlemoyer, 2013) and others (Liang et al., 2011; Berant et al., 2013; Zhao and Huang, 2015; Wong and Mooney, 2006, 2007). These formalisms have the advantage of only generating well-typed logical forms, but the disadvantage of introducing latent syntactic variables that make learning difficult. Another approach is to treat semantic parsing as a machine translation problem, where the logical form is linearized then predicted as an unstructured sequence of tokens (Andreas et al., 2013). This approach is taken by recent neural semantic parsers (Jia and Liang, 2016; Dong and Lapata, 2016; Locascio et al., 2016; Ling et al., 2016). This approach has the advantage of predicting the logical form directly from the question without latent variables, which simplifies learning, but the disadvantage of ignoring type constraints on logical forms. Our type-constrained neural semantic parser inherits the advantages of both approaches: it only generates well-typed logical forms and has no syntactic latent variables as every logical form has a unique derivation. Recent work has explored similar ideas to ours in the context of Python code generation (Yin and Neubig, 2017; Rabinovich et al., 2017).

**Entity Linking** Identifying the entities mentioned in a question is a critical subproblem of semantic parsing in broad domains and proper entity linking can lead to large accuracy improvements (Yih et al., 2015). However, semantic parsers have typically ignored this problem by assuming that entity linking is done beforehand (as the neural parsers above do) or using a simple parameterization for the entity linking portion (as the lexicalized parsers do). Our parser explicitly includes an entity linking module that enables it to model the highly ambiguous and implicit entity mentions in WIKITABLEQUESTIONS.

**Supervision** Semantic parsers can be trained from labeled logical forms (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005) or question-answer pairs (Liang et al., 2011; Berant et al., 2013). Question-answer pairs were considered easier to obtain than labeled logical forms, though recent work has demonstrated that logical forms can be collected efficiently and are more effective (Yih et al., 2016). However, a key advantage of question-answer pairs is that they are agnostic to the domain representation and logical form language (e.g., lambda calculus or $\lambda$-DCS). This property is important for problems such as semi-structured tables where the proper domain representation is unclear.

**Data Sets** We use WIKITABLEQUESTIONS to evaluate our parser as this data set exhibits both a broad domain and complex questions. Early data sets, such as GEOQUERY (Zelle and Mooney, 1996) and ATIS (Dahl et al., 1994), have small domains with only a handful of different predicates. More recent data sets for question answering against Freebase have a much broader domain, but simple questions (Berant et al., 2013; Cai and Yates, 2013).

## 3 Model

This section describes our semantic parsing model and training procedure. For clarity, we describe the model on WIKITABLEQUESTIONS, though it is also applicable to other problems. The input to our model is a natural language question and a context in which it is to be answered, which in our task is a table. The model predicts the answer to the question by semantically parsing it to a logical form then executing it against the table.

Our model follows an encoder-decoder architecture using recurrent neural networks with Long Short Term Memory (LSTM) cells (Hochreiter and Schmidhuber, 1997). The input question and table entities are first encoded as vectors that are then decoded into a logical form (Figure 1). We make two significant additions to the encoder-decoder architecture. First, the encoder includes a special entity embedding and linking module that produces a *link embedding* for each question token that represents the table entities it links to (Section 3.2). Second, the action space of the decoder is defined by a *type-constrained grammar* which guarantees that generated logical forms satisfy type constraints (Section 3.3).

We train the parser using question-answer pairs as supervision using an approximation of marginal loglikelihood based on enumerating logical forms via dynamic programming on denotations (Pasupat and Liang, 2016) (Section 3.4). This approximation makes it possible to train neural models with question-answer supervision, which is otherwise difficult for efficiency and gradient variance reasons.

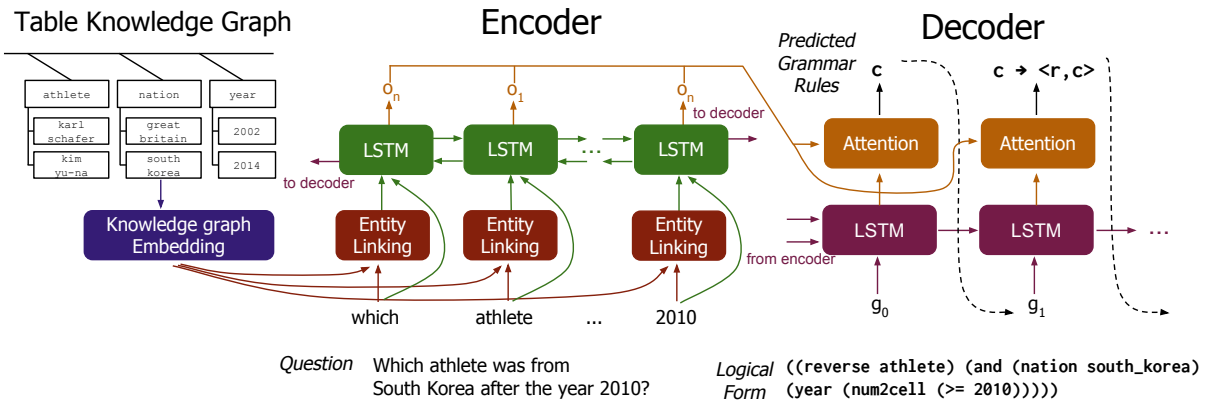


Figure 1: Overview of our semantic parsing model. The encoder performs entity embedding and linking before encoding the question with a bidirectional LSTM. The decoder predicts a sequence of grammar rules that generate a well-typed logical form.

### 3.1 Preliminaries

We follow (Pasupat and Liang, 2015) in using the same table structure representation and $\lambda$-DCS language for expressing logical forms. In this representation, tables are expressed as knowledge graphs over 6 types of entities: cells, cell parts, rows, columns, numbers and dates. Each entity also has a name, which is typically a string value in the table. Our parser uses both the entity names and the knowledge graph structure to construct embeddings for each entity.

The logical form language consists of a collection of named sets and entities, along with operators on them. The named sets are used to select table cells, e.g., `united_states` is the set of cells that contain the text "united states". The operators include functions from sets to sets, e.g., the `next` operator maps a row to the next row. Columns are treated as functions from cells to their rows, e.g., `(country united_states)` generates the rows whose `country` column contains "united states". Other operators include reversing relations (e.g., in order to map rows to cells in a certain column), relations that interpret cells as numbers and dates, and set and arithmetic operations. The language also includes aggregation and quantification operations such as `count` and `argmax`, along with $\lambda$ abstractions that can be used to join binary relations.

Our parser also assigns a type to every $\lambda$-DCS expression, which is used to enforce type constraints on generated logical forms. The base types are cells `c`, parts `p`, rows `r`, numbers `i`, and dates `d`. Columns such as `country` have the functional type $\langle \mathtt{c}, \mathtt{r} \rangle$, representing functions from cells `c` to rows `r`. Other operations have more complex functional types, e.g., `reverse` has type $\langle\langle \mathtt{c}, \mathtt{r} \rangle, \langle \mathtt{r}, \mathtt{c} \rangle\rangle$, which enables us to write `(reverse country)`.[1] The parser assigns every $\lambda$-DCS constant a type, then applies standard programming language type inference algorithms (Pierce, 2002) to automatically assign types to larger expressions.

### 3.2 Encoder

The encoder is a bidirectional LSTM augmented with an entity embedding and linking module.

**Notation.** Throughout this section, we denote entities as $e$, and their corresponding types as $\tau(e)$. The set of all entities is denoted as $E$, and the entities with type $\tau$ as $E_\tau$. $E$ includes the cells, cell parts, and columns from the table in addition to numeric entities detected in the question by NER. The question is denoted as a sequence of tokens $[q_1, ..., q_n]$. We use $v_w$ to denote a learned vector representation (embedding) of word $w$, e.g., $v_{q_i}$ denotes the vector representation of the $i$th question token.

**Entity Embedding.** The encoder first constructs an embedding for each entity in the knowledge graph given its type and position in the graph. Let $W(e)$ denote the set of words in the name of en-

[1] Technically, `reverse` has the parametric polymorphic type $\langle\langle \alpha, \beta \rangle, \langle \beta, \alpha \rangle\rangle$, where $\alpha$ and $\beta$ are *type variables* that can be any type. This type allows `reverse` to reverse any function. However, this is a detail that can largely be ignored. We only use parametric polymorphism when typing logical forms to generate the type-constrained grammar; the grammar itself does not have type variables, but rather a fixed number of concrete instances – such as $\langle\langle \mathtt{c}, \mathtt{r} \rangle, \langle \mathtt{r}, \mathtt{c} \rangle\rangle$ – of the above polymorphic type.

tity $e$ and $N(e)$ the neighbors of entity $e$ in the knowledge graph. Specifically, the neighbors of a column are the cells it contains, and the neighbors of a cell are the columns it belongs to. Each entity's embedding $r_e$ is a nonlinear projection of a type vector $v_{\tau(e)}$ and a neighbor vector $v_{N(e)}$:

$$v_{N(e)} = \frac{1}{|N(e)|} \sum_{e' \in N(e)} \frac{1}{|W(e')|} \sum_{w \in W(e')} v_w$$

$$r_e = \tanh\left(P_\tau v_{\tau(e)} + P_N v_{N(e)}\right)$$

The type vector $v_{\tau(e)}$ is a one-hot vector for $\tau(e)$, with dimension equal to the number of entity types in the grammar. The neighbor vector $v_{N(e)}$ is simply an average of the word vectors in the names of $e$'s neighbors. $P_\tau$ and $P_N$ are learned parameter matrices for combining these two vectors.

**Entity Linking.** This module generates a *link embedding* $l_i$ for each question token representing the entities it links to. The first part of this module generates an entity linking score $s(e, i)$ for each entity $e$ and token index $i$:

$$s(e, i) = \max_{w \in W(e)} v_w^\intercal v_{q_i} + \psi^\intercal \phi(e, i)$$

This score has two terms. The first represents similarity in word embedding space between the token and entity name, computed as function of the embeddings of words in $e$'s name, $W(e)$, and the word embedding of the $i$th token, $v_{q_i}$. The max-pooling architecture allows each question token to pick its best match in the entity's name. The second term represents a linear classifier with parameters $\psi$ on features $\phi(e, i)$. The feature function $\phi$ includes only a few features: indicators for exact token and lemma match, edit distance, an NER tag indicator, and a bias feature. It also includes "related column" versions of the token and lemma features that are active when $e$ is a column and the original feature matches a cell entity in column $e$. We found that features were an effective way to address sparsity in the entity name tokens, many of which appear too infrequently to learn embeddings for. We produce an independent score for each entity and token index even though we expect entities to link to multi-token spans in order to avoid the quadratic computational complexity of scoring each span.

Finally, the entity embeddings and linking scores are combined to produce a link embedding for each token. The scores $s(e, i)$ are then fed into a softmax layer over all entities $e$ of the same type, and the link embedding $l_i$ is an average of entity vectors $r_e$ weighted by the resulting distribution, then summed over all types. We include a null entity, $\varnothing$, in each softmax layer to permit the model to identify tokens that do not refer to an entity. The null entity's embedding is the all-zero vector and its score $s(\varnothing, \cdot) = 0$. Note that the null entity may still be assigned high probability as the other entity scores of may be negative. The link embedding $\ell_i$ for the $i$th question token is computed as:

$$p(e|i, \tau) = \frac{\exp s(e, i)}{\sum_{e' \in E_\tau \cup \{\varnothing\}} \exp s(e', i)}$$

$$l_i = \sum_\tau \sum_{e \in E_\tau} r_e p(e|i, \tau)$$

For WIKITABLEQUESTIONS, we ran the entity embedding and linking module over every entity. However, this approach may be prohibitively expensive in applications with a very large number of entities. In these cases, our method can be applied by adding a preliminary filtering step to identify a subset of entities that may be mentioned in the question. This filter need not have high precision, and therefore could rely on simple text overlap or similarity heuristics. This reduced set of entities can then be fed into the entity embedding and linking module, which will learn to further prune this set of candidates.

**Bidirectional LSTM.** We concatenate the link embedding $l_i$ and the word embedding $v_{q_i}$ of each token in the question, and feed them into a bidirectional LSTM:

$$x_i = \begin{bmatrix} l_i \\ v_{q_i} \end{bmatrix}$$

$$(o_i^f, f_i) = \text{LSTM}(f_{i-1}, x_i)$$

$$(o_i^b, b_i) = \text{LSTM}(b_{i+1}, x_i)$$

$$o_i = \begin{bmatrix} o_i^f \\ o_i^b \end{bmatrix}$$

This process produces an encoded vector representation of each token $o_i$. The final LSTM hidden states $f_{n+1}$ and $b_0$ are concatenated and used to initialize the decoder.

### 3.3 Decoder

The decoder is an LSTM with attention that selects parsing actions from a grammar over well-typed logical forms.

**Type-Constrained Grammar.** The parser maintains a state at each step of decoding that consists of a logical form with nonterminals standing for portions that are yet to be generated. Each nonterminal is a tuple $[\tau, \Gamma]$ of a type $\tau$ and a *scope* $\Gamma$ that contains typed variable bindings, $(x : \alpha) \in \Gamma$, where $x$ is a variable name and $\alpha$ is a type. The scope is used to store and generate the arguments of lambda expressions. The grammar consists of a collection of four kinds of production rules on nonterminals:

1. **Application** $[\tau, \Gamma] \rightarrow (\,[\langle \beta, \tau \rangle, \Gamma] \quad [\beta, \Gamma]\,)$ rewrites a nonterminal of type $\tau$ by applying a function from $\beta$ to $\tau$ to an argument of type $\beta$. We also permit applications with more than one argument.

2. **Constant** $[\tau, \Gamma] \rightarrow$ `const` where constant `const` has type $\tau$. This rule generates both table-independent operations such as `argmax` and table-specific entities such as `united_states`.

3. **Lambda** $[\langle \alpha, \tau \rangle, \Gamma] \rightarrow \lambda x.\, [\tau, \Gamma \cup \{(x : \alpha)\}]$ generates a lambda expression where the argument has type $\alpha$. $x$ represents a fresh variable name. The right hand side of this rule extends the scope $\Gamma$ with a binding for $x$ then generates an expression of type $\tau$.

4. **Variable** $[\tau, \Gamma] \rightarrow\ x$ where $(x : \tau) \in \Gamma$. This rule generates a variable bound in a previously-generated lambda expression that is currently in scope.

We instantiate each of the four rules above by replacing the type variables $\tau, \alpha, \beta$ with concrete types, producing, e.g., $[\mathtt{c}, \Gamma] \rightarrow ([\langle \mathtt{r}, \mathtt{c} \rangle, \Gamma]\ [\mathtt{r}, \Gamma])$ from the application rule. The set of instantiated rules is automatically derived from a corpus of logical forms, which we in turn produce by running dynamic programming on denotations (see Section 3.4). Every logical form can be derived in exactly one way using the four kinds of rules above; this derivation is combined with the (automatically-assigned) type of each of the logical form's subexpressions to instantiate the type variables in each rule. Finally, as a postprocessing step, we filter out table-dependent rules, such as those that generate table cells, to produce a table-independent grammar. The table-dependent rules are handled specially by the decoder in order to guarantee that they are only generated when answering questions against the appropriate table. The table-independent grammar generates well-typed expressions that include functions such as `next` and quantifiers such as `argmax`; however, it cannot generate cells, columns, or other table entities.

The first action of the parser is to predict a root type for the logical form, and then decoding proceeds according to the production rules above. Each time step of decoding fills the leftmost nonterminal in the logical form, and decoding terminates when no nonterminals remain. Figure 2 shows the sequence of decoder actions used to generate an example logical form.

**Network Architecture.** The decoder is an LSTM that outputs a distribution over grammar actions using an attention mechanism over the encoded question tokens. The decoder also uses a copy-like mechanism on the entity linking scores to generate entities. Say that, during the $j$th time step, the current nonterminal has type $\tau$. The decoder generates a score for each grammar action whose left-hand side is $\tau$ using the following equations:

$$(y_j, h_j) = \text{LSTM}(h_{j-1}, \begin{bmatrix} g_{j-1} \\ o_{j-1} \end{bmatrix}) \tag{1}$$

$$a_j = \text{softmax}(OW^a y_j) \tag{2}$$

$$o_j = (a_j)^T O \tag{3}$$

$$s_j = W^2_\tau \,\text{relu}(W^1 \begin{bmatrix} y_j \\ o_j \end{bmatrix} + b^1) + b^2_\tau \tag{4}$$

$$s_j(e_k) = \sum_i s(e_k, i) a_{ji} \tag{5}$$

$$p_j = \text{softmax}(\begin{bmatrix} s_j \\ s_j(e_1) \\ s_j(e_2) \\ \cdots \end{bmatrix}) \tag{6}$$

The input to the LSTM $g_{j-1}$ is a grammar action embedding for the action chosen in previous time step. $g_0$ is a learned parameter vector, and $h_0$ is the concatenated final hidden states of the encoder LSTMs. The matrix $O$ contains the encoded token vectors $o_1, ..., , o_n$ from the encoder. The first three lines above perform a softmax attention over $O$ using a learned parameter matrix $W^a$. The fourth line generates scores $s_j$ for the table-independent grammar rules applicable to type $\tau$ using a multilayer perceptron with

*Question*: name the largest lake
*Logical Form*:
`((reverse name) (argmax allrows (reverse (λ (x)`
`((reverse num2cell) ((reverse area_in_km) x))))))`
*Derivation of Logical Form*:

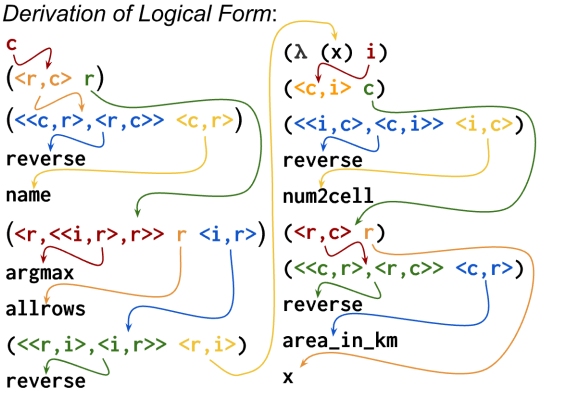


Figure 2: The derivation of a logical form using the type-constrained grammar. The nonterminals in the left column have empty scope, and those in the right column have scope $\Gamma = \{(x : \mathtt{r})\}$

weights $W^1$,$b^1$,$W^2_\tau$,$b^2_\tau$. The fifth line generates a score for each entity $e$ with type $\tau$ by averaging the entity linking scores with the current attention $a_j$. Finally, the table-independent and -dependent scores are concatenated and softmaxed to produce a probability distribution $p_j$ over grammar actions. If a table-independent action is chosen, $g_j$ is a learned parameter vector for that action. Otherwise $g_j = g^\tau$, which is a learned parameter representing the selection of an entity with type $\tau$.

### 3.4 DPD Training

Our parser is trained from question-answer pairs, treating logical forms as a latent variable. We use an approximate marginal loglikelihood objective function that first automatically enumerates a set of correct logical forms for each example, then trains on these logical forms. This objective simplifies the search problem during training and is well-suited to training our neural model.

The training data consists of a collection of $n$ question-answer-table triples, $\{(q^i, a^i, T^i)\}_{i=1}^n$. We first run dynamic programming on denotations (Pasupat and Liang, 2016) on each table $T^i$ and answer $a^i$ to generate a set of logical forms $\ell \in \mathcal{L}^i$ that execute to the correct answer. Dynamic programming on denotations (DPD) is an automatic procedure for enumerating logical forms that execute to produce a particular value; it leverages the observation that there are fewer denotations than logical forms to enumerate this set relatively efficiently. However, many of these logical forms are *spurious*, in the sense that they do not represent the question's meaning. Therefore, the objective must marginalize over the many logical forms generated in this fashion:

$$\mathcal{O}(\theta) = \sum_{i=1}^{n} \log \sum_{\ell \in \mathcal{L}^i} P(\ell | q^i, T^i; \theta)$$

We optimize this objective function using stochastic gradient descent. If $|\mathcal{L}^i|$ is small, e.g., 5-10, the gradient of the $i$th example can be computed exactly by simply replicating the parser's network architecture $|\mathcal{L}^i|$ times, once per logical form. However, $|\mathcal{L}^i|$ often contains many thousands of logical forms, which makes the above computation infeasible. We address this problem by truncating $\mathcal{L}^i$ to the $m = 100$ shortest logical forms, then using a beam search with a beam of $k = 5$ to approximate the sum. Section 4.5 considers the effect of varying the number of logical forms $m$ in this objective function.

We briefly contrast this approach with two other commonly-used approaches. The first is a similar marginal loglikelihood objective commonly used in prior semantic parsing work with loglinear models (Liang et al., 2011; Pasupat and Liang, 2015). However, this approach does not precompute correct logical forms. Therefore, computing its gradient requires running a wide beam search, generating, e.g., 300 logical forms, executing each one to identify which are correct, then backpropagating through a term for each. The wide beam is required to find correct logical forms; however, such a wide beam is prohibitively expensive with a neural model due to the cost of each backpropagation pass. Another approach is to train the network with REINFORCE (Williams, 1992), which essentially samples a logical form instead of using beam search. This approach is known to be difficult to apply when the space of outputs is large and the reward signal is sparse, and recent work has found that maximizing marginal loglikelihood is more effective in these circumstances (Guu et al., 2017). Our approach makes it tractable to maximize marginal loglikelihood with a neural model by using DPD to enumerate correct logical forms beforehand. This up-front enumeration, combined with the local normalization of the neural model, makes it possible to restrict the beam search to correct logical forms in the gradient computation, which enables training with a small beam size.

## 4 Evaluation

We evaluate our parser on the WIKITABLEQUESTIONS data set by comparing it to prior work and ablating several components to understand their contributions.

### 4.1 Experimental Setup

We used the standard train/test splits of WIKITABLEQUESTIONS. The training set consists of 14,152 examples and the test set consists of 4,344 examples. The training set comes divided into 5 cross-validation folds for development using an 80/20 split. All data sets are constructed so that the development and test tables are not present in the training set. We report question answering accuracy measured using the official evaluation script, which performs some simple normalization of numbers, dates, and strings before comparing predictions and answers. When generating answers from a model's predictions, we skip logical forms that do not execute (which may occur for some baseline models) or answer with the empty string (which is never correct). All reported accuracy numbers are an average of 5 parsers, each trained on one training fold, using the respective development set to perform early stopping.

We trained our parser with 20 epochs of stochastic gradient descent. We used 200-dimensional word embeddings for the question and entity tokens, mapping all tokens that occurred $< 3$ times in the training questions to `UNK`. (We tried using a larger vocabulary that included frequent tokens in tables, but this caused the parser to seriously overfit.) The hidden and output dimensions of the forward/backward encoder LSTMs were set to 100, such that the concatenated representations were also 200-dimensional. The decoder LSTM uses 100-dimensional action embeddings and has a 200-dimensional hidden state and output. The action selection MLP has a hidden layer dimension of 100. We used a dropout probability of $0.5$ on the output of both the encoder and decoder LSTMs, as well as on the hidden layer of the action selection MLP. All parameters are initialized using Glorot initialization (Glorot and Bengio, 2010). The learning rate for SGD is initialized to 0.1 with a decay of 0.01. At test time, we decode with a beam size of 10.

Our model is implemented as a probabilistic neural program (Murray and Krishnamurthy, 2016). This Scala library combines ideas from dynamic neural network frameworks (Neubig et al., 2017) and probabilistic programming (Goodman and Stuhlmüller, 2014) to simplify the implementation of complex neural structured prediction models. This library enables a user to specify the structure of the model in terms of discrete nondeterministic choices – as in probabilistic programming – where a neural network is used to score each choice. We implement our parser by defining $P(\ell|q, T; \theta)$, from which the library automatically implements both inference and training. In particular, the beam search and the corresponding backpropagation bookkeeping to implement the objective in Section 3.4 are both automatically handled by the library. Code and supplementary material for this paper are available at:
`http://allenai.org/paper-appendix/emnlp2017-wt/`

### 4.2 Results

Table 1 compares the accuracy of our semantic parser to prior work on WIKITABLEQUESTIONS. We distinguish between single models and ensembles, as we expect ensembling to improve accuracy, but not all prior work has used it. Prior work on this data set includes a loglinear semantic parser (Pasupat and Liang, 2015), that same parser with a neural, paraphrase-based reranker (Haug et al., 2017), and a neural programmer that answers questions by predicting a sequence of table operations (Neelakantan et al., 2017). We find that our parser outperforms the best prior result on this data set by 4.6%, despite that prior result using a 15-model ensemble. An ensemble of 5 parsers improves accuracy by an additional 2.6% for a total improvement of 7.2%. This ensemble was constructed by averaging the logical form probabilities of parsers trained on each of the 5 cross-validation folds. Note that this ensemble is trained on the entire training set – the development data from one fold is training data for the others – so we therefore cannot report its development accuracy. We investigate the sources of this accuracy improvement in the remainder of this section via ablation experiments.

### 4.3 Type Constraints

Our second experiment measures the importance of type constraints on the decoder by comparing it to sequence-to-sequence (seq2seq) and sequence-to-tree (seq2tree) models. The seq2seq model generates the logical form a token at a time, e.g., $[($, $($, `reverse`, ...], and has been used in several

| Model | Ensemble Size | Dev. | Test |
|---|---|---|---|
| Neelakantan et al. (2017) | 1 | 34.1 | 34.2 |
| Haug et al. (2017) | 1 | - | 34.8 |
| Pasupat and Liang (2015) | 1 | 37.0 | 37.1 |
| Neelakantan et al. (2017) | 15 | 37.5 | 37.7 |
| Haug et al. (2017) | 15 | - | 38.7 |
| Our Parser | 1 | 42.7 | 43.3 |
| Our Parser | 5 | - | **45.9** |

Table 1: Development and test set accuracy of our semantic parser compared to prior work on WIKITABLEQUESTIONS.

recent neural semantic parsers (Jia and Liang, 2016; Dong and Lapata, 2016). The seq2tree model improves on the seq2seq model by including an action for generating matched parentheses, then recursively generating the subtree within (Dong and Lapata, 2016). These baseline models use the same network architecture (including entity embedding and linking) and training regime as our parser, but assign every constant the same type and have a different grammar in the decoder. These models were implemented by preprocessing logical forms and applying a different type system.

Table 2 compares the accuracy of our parser to both the seq2seq and seq2tree baselines. Both of these models perform considerably worse than our parser, demonstrating the importance of type constraints during decoding. Interestingly, we found that both baselines typically generate well-formed logical forms: only 7.4% of seq2seq and 6.6% of seq2tree's predicted logical forms failed to execute. Type constraints prevent these errors from occurring in our parser, though the relatively small number of such errors does not does not seem to fully explain the 9% accuracy improvement. We hypothesize that the additional improvement occurs because type constraints also increase the effective capacity of the model, as both the seq2seq and seq2tree models must use some of their capacity to learn the type constraints on logical forms.

### 4.4 Entity Embedding and Linking

Our next experiment measures the contribution of the entity embedding and linking module. We trained several ablated versions of our parser, removing both the embedding similarity and featurized classifier from the entity linking module. Table 3 shows the accuracy of the resulting models.

| Model | Dev. Accuracy |
|---|---|
| seq2seq | 31.3 |
| seq2tree | 31.6 |
| Our Parser | 42.7 |

Table 2: Development accuracy of our semantic parser compared to sequence-to-sequence and sequence-to-tree models.

| Model | Dev. Accuracy |
|---|---|
| Full model | 42.7 |
| token features, no similarity | 28.1 |
| all features, no similarity | 37.8 |
| similarity only, no features | 27.5 |

Table 3: Development accuracy of ablated parser variants trained without parts of the entity linking module.

The results demonstrate that the entity linking features are important, particularly the more complex features beyond simple token matching. In our experience, the "related column" features are especially important for this data set, as columns that appear in the logical form are often not mentioned in the text, but rather implied by a mention of a cell from the column. Embedding similarity alone is not very effective, but it does improve accuracy when combined with the featurized classifier. We found that word embeddings enabled the parser to overfit, which may be due to the relatively small size of the training set, or because we did not use pretrained embeddings. Incorporating pretrained embeddings is an area for future work.

We also examined the effect of the entity embeddings computed using each entity's knowledge graph context by replacing them with one-hot vectors for the entity's type. The accuracy of this parser dropped from 42.7% to 41.8%, demonstrating that the knowledge graph embeddings help.

### 4.5 DPD Training

Our final experiment examines the impact on accuracy of varying the number of logical forms $m$ used when training with dynamic programming on denotations. Table 4 shows the development accuracy of several parsers trained with varying $m$. These results demonstrate that using more logical forms generally leads to higher accuracy.

| # of logical forms | 1 | 5 | 10 | 50 | 100 |
|---|---|---|---|---|---|
| Dev. Accuracy | 39.7 | 41.9 | 41.6 | 43.1 | 42.7 |

Table 4: Development accuracy of our semantic parser when trained with varying numbers of logical forms produced by dynamic programming on denotations.

### 4.6 Error Analysis

To better understand the mistakes made by our system, we analyzed a randomly selected set of 100 questions that were answered incorrectly. We identified three major classes of error:

**Parser errors (41%):** These are examples where a correct logical form is available, but the parser does not select it. A large number of these errors (15%) occur on questions that require selecting an answer from a given list of options, as in *Who had more silvers, Colombia or The Bahamas?* In such cases, the type of the predicted answer is often wrong. Another common subclass is entity linking errors due to missing background knowledge (13%), e.g., understanding that *largest* implicitly refers to the *Area* column.

**Representation failures (25%):** The knowledge graph representation makes certain assumptions about the table structure and cell values which are sometimes wrong. One common problem is that the graph lacks some cell parts necessary to answer the question (15%). For example, answering a question asking for a state may require splitting cell values in the *Location* column into city and state names. Another common problem is unusual table structures (10%), such as a table listing the number of Olympic medals won by each country that has a final row for the totals. These structures often cause quantifiers such as `argmax` to select the wrong row.

**Unsupported operations (11%):** These are examples where the logical form language lacks a necessary function. Examples of missing functions are finding consecutive sets of values, computing percentages and performing string operations on cell values.

## 5 Conclusion

We present a new semantic parsing model for answering compositional questions against semi-structured Wikipedia tables. Our semantic parser extends recent neural semantic parsers by enforcing type constraints during logical form generation, and by including an explicit entity embedding and linking module that enables it to identify entity mentions while generalizing across tables. An evaluation on WIKITABLEQUESTIONS demonstrates that our parser achieves state-of-the-art results, and furthermore that both type constraints and entity linking make significant contributions to accuracy. Analyzing the errors made by our parser suggests that improving entity linking and using the table structure are two directions for future work.

## References


Jacob Andreas, Andreas Vlachos, and Stephen Clark. 2013. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*.

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

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*.

Qingqing Cai and Alexander Yates. 2013. Large-scale semantic parsing via schema matching and lexicon extension. In *In Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. 1994. Expanding the scope of the ATIS task: The ATIS-3 corpus. In *Proceedings of the Workshop on Human Language Technology*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. pages 249–256.

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`. Accessed: 2017-4-13.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Association for Computational Linguistics (ACL)*.

Till Haug, Octavian-Eugen Ganea, and Paulina Grnarova. 2017. Neural multi-step reasoning for question answering on semi-structured tables http://arxiv.org/abs/1702.06589.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

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

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Percy Liang, Michael I Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Nicholas Locascio, Karthik Narasimhan, Eduardo De Leon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*.

Kenton W. Murray and Jayant Krishnamurthy. 2016. Probabilistic neural programs http://arxiv.org/abs/1612.00712.

Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei. 2017. Learning a natural language interface with neural programmer. In *ICLR*.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. 2017. DyNet: The dynamic neural network toolkit https://arxiv.org/abs/1701.03980.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*.

Panupong Pasupat and Percy Liang. 2016. Inferring logical forms from denotations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. The MIT Press, 1st edition.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8(3):229–256.

Yuk Wah Wong and Raymond J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL*.

Yuk Wah Wong and Raymond J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*.

Scott Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *Proceedings of the Joint Conference of the 53rd Annual Meeting of the ACL and the 7th International Joint Conference on Natural Language Processing of the AFNLP*.

Wen-tau Yih, Matthew Richardson, Christopher Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*.

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

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*.

Luke S Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL*.

Kai Zhao and Liang Huang. 2015. Type-driven incremental semantic parsing with polymorphism. In *HLT-NAACL*.