Learning Semantic Parsers from Denotations with Latent Structured Alignments and Abstract Programs

Bailin Wang, Ivan Titov and **Mirella Lapata**
Institute for Language, Cognition and Computation
School of Informatics, University of Edinburgh
`bailin.wang@ed.ac.uk, {ititov,mlap}@inf.ed.ac.uk`

Abstract

Semantic parsing aims to map natural language utterances onto machine interpretable meaning representations, aka programs whose execution against a real-world environment produces a denotation. Weakly-supervised semantic parsers are trained on utterance-denotation pairs treating programs as latent. The task is challenging due to the large search space and spuriousness of programs which may execute to the correct answer but do not generalize to unseen examples. Our goal is to instill an inductive bias in the parser to help it distinguish between spurious and correct programs. We capitalize on the intuition that correct programs would likely respect certain structural constraints were they to be aligned to the question (e.g., program fragments are unlikely to align to overlapping text spans) and propose to model alignments as structured latent variables. In order to make the latent-alignment framework tractable, we decompose the parsing task into (1) predicting a partial "abstract program" and (2) refining it while modeling structured alignments with differential dynamic programming. We obtain state-of-the-art performance on the WIKITABLEQUESTIONS and WIKISQL datasets. When compared to a standard attention baseline, we observe that the proposed structured-alignment mechanism is highly beneficial.

1 Introduction

Semantic parsing is the task of translating natural language to machine interpretable meaning representations. Typically, it requires mapping a natural language utterance onto a program, which is executed against a knowledge base to obtain an answer or a denotation. Most previous work (Zettlemoyer and Collins, 2005; Wong and Mooney, 2007; Lu et al., 2008; Jia and Liang, 2016) has focused on the supervised setting where a model is learned from question-program pairs. Weakly

Figure 1: After generating an abstract program for a question, our parser finds alignments between slots (with prefix #) and question spans. Based on the alignment, it instantiates each slot and a complete program is executed against a table to obtain a denotation.

supervised semantic parsing (Berant et al., 2013; Liang et al., 2011) reduces the burden of annotating programs by learning from questions paired with their answers (or denotations).

Two major challenges arise when learning from denotations: 1) training of the semantic parser requires exploring a large search space of possible programs to find those which are *consistent*, and execute to correct denotations; 2) the parser should be robust to *spurious* programs which accidentally execute to correct denotations, but do not reflect the semantics of the question. In this paper, we propose a weakly-supervised neural semantic parser that features structured latent alignments to bias learning towards *correct* programs which are consistent but not spurious.

Our intuition is that correct programs should re-

spect certain constraints were they to be aligned to the question text, while spurious and inconsistent programs do not. For instance, in Figure 1, the answer to the question ("0") can be obtained by executing the correct program which selects the number of Turkey's silver medals. However, the same answer can be also obtained by the spurious programs shown in the figure.[1] The spurious programs differ from the correct one in that they repeatedly use the column "silver". Whereas, in the question, the word "silver" only refers to the target column containing the answer; it also mistakenly triggers the appearance of the column "silver" in the row selection condition. This constraint, i.e., that a text span within a question cannot trigger two semantically distinct operations (e.g., selecting target rows and target columns) can provide a useful inductive bias. We propose to capture structural constraints by modeling the alignments between programs and questions *explicitly* as structured latent variables.

Considering the large search space of possible programs, an alignment model that takes into account the full range of correspondences between program operations and question spans would be very expensive. To make the process tractable, we introduce a two-stage approach that features *abstract* programs. Specifically, we decompose semantic parsing into two steps: 1) a natural language utterance is first mapped to an abstract program which is a composition of high-level operations; and 2) the abstract program is then instantiated with low-level operations that usually involve relations and entities specific to the knowledge base at hand. This decomposition is motivated by the observation that only a small number of sensible abstract programs can be instantiated into consistent programs. Similar ideas of using abstract meaning representations have been explored with fully-supervised semantic parsers (Dong and Lapata, 2018; Catherine Finegan-Dollak and Radev, 2018) and in other related tasks (Goldman et al., 2018; Herzig and Berant, 2018; Nye et al., 2019).

For a knowledge base in tabular format, we abstract two basic operations of row selection and column selection from programs: these are handled in the second (instantiation) stage. As shown in Figure 1, the question is first mapped to the abstract program "select (#*row_slot*, #*column_slot*)"

whose two slots are subsequently instantiated with filter conditions (row slot) and a column name (column slot). During the instantiation of abstract programs, each slot should refer to the question to obtain its specific semantics. In Figure 1, *row_slot* should attend to "nation of Turkey" while *column_slot* needs to attend to "silver medals". The structural constraint discussed above now corresponds to assuming that each span in a question can be aligned to a *unique* row or column slot. Under this assumption, the instantiation of spurious programs will be discouraged. The uniqueness constraint would be violated by both spurious programs in Figure 1, since "column:silver" appears in the program twice but can be only aligned to the span "silver medals" once.

The first stage (i.e., mapping a question onto an abstract program) is handled with a sequence-to-sequence model. The second stage (i.e., program instantiation) is approached with local classifiers: one per slot in the abstract program. The classifiers are conditionally independent given the abstract program and a latent alignment. Instead of marginalizing out alignments, which would be intractable, we use structured attention (Kim et al., 2017), i.e., we compute the marginal probabilities for individual span-slot alignment edges and use them to weight the input to the classifiers. As we discuss below, the marginals in our constrained model are computed with dynamic programming.

We perform experiments on two open-domain question answering datasets in the setting of learning from denotations. Our model achieves an execution accuracy of 44.5% in WIKITABLEQUES-TIONS and 79.3% in WIKISQL, which both surpass previous state-of-the-art methods in the same weakly-supervised setting. In WIKISQL, our parser is better than recent supervised parsers that are trained on question-program pairs. Our contributions can be summarized as follows:

- we introduce an alignment model as a means of differentiating between correct and spurious programs;

- we propose a neural semantic parser that performs tractable alignments by first mapping questions to abstract programs;

- we achieve state-of-the-art performance on two semantic parsing benchmarks.[2]

---

[1] The first program can be paraphrased as: find the row with the largest number of silver medals and then select the number of silver medals from the previous row.

Although we use structured alignments to mostly enforce the uniqueness constraint described above, other types of inductive biases can be useful and could be encoded in our two-stage framework. For example, we could replace the uniqueness constraint with modeling the number of slots aligned to a span, or favor sparse alignment distributions. Crucially, the two-stage framework makes it easier to inject prior knowledge about datasets and formalisms while maintaining efficiency.

2 Background

Given knowledge base t, our task is to map a natural utterance x to program z, which is then executed against a knowledge base to obtain denotation $[[z]]_t = d$. We train our parser only based on d without access to correct programs z^*. Our experiments focus on two benchmarks, namely WIKITABLEQUESTIONS (Pasupat and Liang, 2015) and WIKISQL (Zhong et al., 2017) where each question is paired with a Wikipedia table and a denotation. Figure 1 shows a simplified example taken from WIKITABLEQUESTIONS.

2.1 Grammars

Executable programs z that can query tables are defined according to a language. Specifically, the search space of programs is constrained by grammar rules so that it can be explored efficiently. We adopt the variable-free language of Liang et al. (2018) and define an *abstract* grammar and an *instantiation* grammar which decompose the generation of a program in two stages.[3]

The first stage involves the generation of an abstract version of a program which, in the second stage, gets instantiated. Abstract programs only consider compositions of high-level functions, such as superlatives and aggregation, while low-level functions and arguments, such as filter conditions and entities, are taken into account in the next step. In our table-based datasets, abstract programs do not include two basic operations of querying tables: row selection and column selection. These operations are handled at the instantiation stage. In Figure 1 the abstract program has two slots for row and column selection, which are filled with the conditions "column:nation = Turkey" and "column:silver" at the instantiation

---

Abstract Program:
select (first (#rows_slot) #column_slot)

Derivation Tree:

STRING → *select* (ROW, COLUMN)

ROW → *first* (LIST[ROW]) COLUMN → *#column_slot*
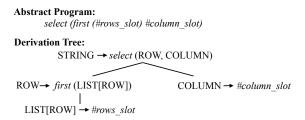
LIST[ROW] → *#rows_slot*

Figure 2: An abstract program and its derivation tree. Capitalized words indicate types of function arguments and their return value.

stage. The two stages can be easily merged into one step when conducting symbolic combinatorial search. The motivation for the decomposition is to facilitate the learning of our neural semantic parser and the handling of structured alignments.

Abstract Grammar Our abstract grammar has five basic types: ROW, COLUMN, STRING, NUMBER, and DATE; COLUMN is further sub-typed into STRING_COLUMN, NUMBER_COLUMN, and DATE_COLUMN; other basic types are augmented with LIST to represent a list of elements like LIST[ROW]. Arguments and return values of functions are typed using these basic types.

Function composition can be defined recursively based on a set of production rules, each corresponding to a function type. For instance, function ROW → *first*(LIST[ROW]) selects the first row from a list of rows and corresponds to production rule "ROW → *first*".

The abstract grammar has two additional types for slots (aka terminal rules) which correspond to row and column selection:

$$\text{LIST[ROW]} \to \textit{\#row\_slot}$$
$$\text{COLUMN} \to \textit{\#column\_slot}$$

An example of an abstract program and its derivation tree is shown in Figure 2. We linearize the derivation by traversing it in a left-to-right depth-first manner. We represent the tree in Figure 2 as a sequence of production rules: "ROOT → STRING, STRING → *select*, ROW → *first*", LIST[ROW] → *#row_slot*, COLUMN → *#column_slot*". The first action is always to select the return type for the root node.

Given a specific table t, the abstract grammar $\mathcal{H}_t$ will depend on its column types. For example, if the table does not have number cells, "max/min" operations will not be executable.

Instantiation Grammar A column slot is directly instantiated by selecting a column; a row

slot is filled with one or multiple conditions (COND) which are joined together with conjunction (*OR*) and disjunction (*AND*) operators:

COND → COLUMN OPERATOR VALUE
#row_slot → COND (*AND* COND)*
#row_slot → COND (*OR* COND)*

where OPERATOR $\in [>, <, =, \geq, \leq]$ and VALUE is a string, a number, or a date. A special condition *#row_slot* → *all_rows* is defined to signify that a program queries all rows.

2.2 Search for Consistent Programs

A problematic aspect of learning from denotations is that, since annotated programs are not available (e.g., for WIKITABLEQUESTIONS), we have no means to directly evaluate a proposed grammar. As an evaluation proxy, we measure the coverage of our grammar in terms of consistent programs. Specifically, we exhaustively search for all consistent programs for each question in the training set. While the space of programs is exponential, we observed that abstract programs which are instantiated into correct programs are not very complex in terms of the number of production rules used to generate them. As a result, we impose restrictions on the number of production rules which can abstract programs, and in this way the search process becomes tractable.[4]

We find that 83.6% of questions in WIKITABLEQUESTIONS are covered by at least one consistent program. However, each question eventually has 200 consistent programs on average and most of them are spurious. Treating them as ground truth poses a great challenge for learning a semantic parser. The coverage for WIKISQL is 96.6% and each question generates 84 consistent programs.

Another important observation is that there is only a limited number of abstract programs that can be instantiated into consistent programs. The number of such abstract programs is 23 for WIKITABLEQUESTIONS and 6 for WIKISQL, suggesting that there are a few patterns underlying several utterances. This motivates us to design a semantic parser that first maps utterances to abstract programs. For the sake of generality, we do not restrict our parser to abstract programs in the training set. We elaborate on this below.

---

[4] Details are provided in the Appendix.

3 Model

After obtaining consistent programs z for each question via offline search, we next show how to learn a parser that can generalize to unseen questions and tables.

3.1 Training and Inference

Our learning objective $\mathcal{J}$ is to maximize the log-likelihood of the marginal probability of all consistent programs, which are generated by mapping an utterance x to an interim abstract program h:

$$\mathcal{J} = \log\{\sum_{h \in \mathcal{H}_t} p(h|x,t) \sum_{[[z]]=d} p(z|x,t,h)\}. \quad (1)$$

During training, our model only needs to focus on abstract programs that have successful instantiations of consistent programs and it does not have to explore the whole space of possible programs.

At test time, the parser chooses the program $\hat{z}$ with the highest probability:

$$\hat{h}, \hat{z} = \underset{h \in \mathcal{H}_t, \, z}{\arg\max} \, p(h|x,t)p(z|x,t,h). \quad (2)$$

For efficiency, we only choose the top-k abstract programs to instantiate through beam search. $\hat{z}$ is then executed to obtain its denotation as the final prediction.

Next, we will explain the basic components of our neural parser. Basically, our model first encodes a question and a table with an input encoder; it then generates abstract programs with a seq2seq model; and finally, these abstract programs are instantiated based on a structured alignment model.

3.2 Input Encoder

Each word in an utterance is mapped to a distributed representation through an embedding layer. Following previous work (Neelakantan et al., 2017; Liang et al., 2018), we also add an indicator feature specifying whether the word appears in the table. This feature is mapped to a learnable vector. Additionally, in WIKITABLEQUESTIONS, we use POS tags from the CoreNLP annotations released with the dataset and map them to vector representations. The final representation for a word is the concatenation of the vectors above. A bidirectional LSTM (Hochreiter and Schmidhuber, 1997) is then used to obtain a contextual representation $\boldsymbol{l}_i$ for the i_{th} word.

A table is represented by a set of columns. Each column is encoded by averaging the embeddings

of words under its column name. We also have a column type feature (i.e., number, date, or string) and an indicator feature signaling whether at least one entity in the column appears in the utterance.

3.3 Generating Abstract Programs

Instead of extracting abstract programs as templates, similarly to Xu et al. (2017) and Catherine Finegan-Dollak and Radev (2018), we generate them with a seq2seq model. Although template-based approaches would be more efficient in practice, a seq2seq model is more general since it could generate unseen abstract programs which fixed templates could not otherwise handle.

Our goal here is to generate a sequence of production rules that lead to abstract programs. During decoding, the hidden state g_j of the j-th timestep is computed based on the previous production rule, which is mapped to an embedding a_{j-1}. We also incorporate an attention mechanism (Luong et al., 2015) to compute a contextual vector b_j. Finally, a score vector s_j is computed by feeding the concatenation of the hidden state and context vector to a multilayer perceptron (MLP):

$$
\begin{aligned}
g_j &= \text{LSTM}(g_{j-1}, a_{j-1}) \\
b_j &= \text{Attention}(g_j, l) \\
s_j &= \text{MLP}_1([g_j; b_j]) \\
p(a_j|x, t, a_{<j}) &= \text{softmax}_{a_j}(s_j)
\end{aligned}
\tag{3}
$$

where the probability of production rule a_j is computed by the softmax function. According to our abstract grammar, only a subset of production rules will be valid at the j-th time step. For instance, in Figure 2, production rule "STRING → *select*" will only expand to rules whose left-hand side is ROW, which is the type of the first argument of *select*. In this case, the next production rule is "ROW → *first*". We thus restrict the normalization of softmax to only focus on these valid production rules. The probability of generating an abstract program $p(h|x, t)$ is simply the product of the probability of predicting each production rule $\prod_j p(a_j|x, t, a_{<j})$.

After an abstract program is generated, we need to instantiate slots in abstract programs. Our model first encodes the abstract program using a bi-directional LSTM. As a result, the representation of a slot is contextually aware of the entire abstract program (Dong and Lapata, 2018).

3.4 Instantiating Abstract Programs

To instantiate an abstract program, each slot must obtain its specific semantics from the question. We model this process by an alignment model which learns the correspondence between slots and question spans. Formally, we use a binary alignment matrix A with size $m \times n \times n$, where m is the number of slots and n is the number of tokens. In Figure 1, the alignment matrix will only have $A_{0,6,8} = 1$ and $A_{1,2,3} = 1$ which indicates that the first slot is aligned with "nation of Turkey", and the second slot is aligned with "silver medals". The second and third dimension of the matrix represent the start and end position of a span.

We model alignments as discrete latent variables and condition the instantiation process on the alignments as follows:

$$
\begin{aligned}
\sum_{[z]=d} p(z|x, t, h) &= \\
\sum_A p(A|x, t, h) &\sum_{[z]=d} p(z|x, t, h, A).
\end{aligned}
\tag{4}
$$

We will first discuss the instantiation model $p(z|x, t, h, A)$ and then elaborate on how to avoid marginalization in the next section. Each slot in an abstract program can be instantiated by a set of candidates following the instantiation grammar. For efficiency, we use local classifiers to model the instantiation of each slot independently:

$$
p(z|x, t, h, A) = \prod_{s \in S} p(s \to c|x, t, h, A), \tag{5}
$$

where S is the set of slots and c is a candidate following our instantiation grammar. "$s \to c$" represents the instantiation of slot s into candidate c.

Recall that there are two types of slots, one for rows and one for columns. All column names in the table are potential instantiations of column slots. We represent each column slot candidate by the average of the embeddings of words in the column name. Based on our instantiation grammar in Section 2.1, candidates for row slots are represented as follows: 1) each condition is represented with the concatenation of the representations of a column, an operator, and a value. For instance, condition "string_column:nation = Turkey" in Figure 1 is represented by vector representations of the column 'nation', the operator '=', and the entity 'Turkey'; 2) multiple conditions are encoded by averaging the representations of all conditions and adding a vector representation of *AND*/*OR* to indicate the relation between them.

For each slot, the probability of generating a candidate is computed with softmax normalization on a score function:

$$p(s \rightarrow c | x, t, h, \boldsymbol{A}) \propto \exp\{\mathrm{MLP}([\boldsymbol{s}; \boldsymbol{c}])\}, \quad (6)$$

where $\boldsymbol{s}$ is the representation of the span that slot s is aligned with, and $\boldsymbol{c}$ is the representation of candidate c. The representations $\boldsymbol{s}$ and $\boldsymbol{c}$ are concatenated and fed to a MLP. We use the same MLP architecture but different parameters for column and row slots.

3.5 Structured Attention

We first formally define a few structural constraints over alignments and then explain how to incorporate them efficiently into our parser.

The intuition behind our alignment model is that row and column selection operations represent distinct semantics, and should therefore be expressed by distinct natural language expressions. Hence, we propose the following constraints:

Unique Span   In most cases, the semantics of a row selection or a column selection is expressed uniquely with a single contiguous span:

$$\forall k \in [1, |S|], \quad \sum_{i,j} \boldsymbol{A}_{k,i,j} = 1, \quad (7)$$

where $|S|$ is the number of slots.

No Overlap   Spans aligned to different slots should not overlap. Formally, at most one span that contains word i can be aligned to a slot:

$$\forall i \in [1, n], \quad \sum_{k,j} \boldsymbol{A}_{k,i,j} \leq 1. \quad (8)$$

As an example, the alignments in Figure 1 follow the above constraints. Intuitively, the one-to-one mapping constraint aims to assign distinct and non-overlapping spans to slots of abstract programs. To further bias the alignments and improve efficiency, we impose additional restrictions: (1) a row slot must be aligned to a span that contains an entity since conditions that instantiate the slot would require entities for filtering; (2) a column slot must be aligned to a span with length 1 since most column names only have one word.

Marginalizing out all $\boldsymbol{A}$ in Equation (4) would be very expensive considering the exponential number of possible alignments. We approximate

the marginalization by moving the outside expectation directly inside over $\boldsymbol{A}$. As a result, we instead optimize the following objective:

$$\mathcal{J} \approx \log \Big\{ \sum_{h \in \mathcal{H}_t} p(h|x, t) \sum_{[[z]] = d} p(z|x, t, h, \mathbb{E}[\boldsymbol{A}]) \Big\}, \quad (9)$$

where $\mathbb{E}[\boldsymbol{A}]$ are the marginals of $\boldsymbol{A}$ with respect to $p(\boldsymbol{A}|x, t, h)$.

The idea of using differentiable surrogates for discrete latent variables has been used in many other works like differentiable data structures (Grefenstette et al., 2015; Graves et al., 2014) and attention-based networks (Bahdanau et al., 2015; Kim et al., 2017). Using marginals $\mathbb{E}[\boldsymbol{A}]$ can be viewed as *structured attention* between slots and question spans.

The marginal probability of the alignment matrix $\boldsymbol{A}$ can be computed efficiently using dynamic programming (see Täckström et al. 2015 for details). An alignment is encoded into a path in a weighted lattice where each vertex has $2^{|S|}$ states to keep track of the set of covered slots. The marginal probability of edges in this lattice can be computed by the forward-backward algorithm (Wainwright et al., 2008). The lattice weights, represented by a scoring matrix $\boldsymbol{M} \in \mathbb{R}^{m \times n \times n}$ for all possible slot-span pairs, are computed using the following scoring function:

$$\boldsymbol{M}_{k,i,j} = \mathrm{MLP}_2([\boldsymbol{r}(k); \mathbf{span}[i:j]]), \quad (10)$$

where $\boldsymbol{r}(k)$ represents the k_{th} slot and $\mathbf{span}[i:j]$ represents the span from word i to j. Recall that we obtain $\boldsymbol{r}(k)$ by encoding a generated abstract program. A span is represented by averaging the representations of the words therein. These two representations are concatenated and fed to a MLP to obtain a score. Since $\mathbb{E}[\boldsymbol{A}]$ is not discrete anymore, the aligned representation of slot $\boldsymbol{s}$ in Equation (6) becomes the weighted average of representations of all spans in the set.

4 Experiments

We evaluated our model on two semantic parsing benchmarks, WIKITABLEQUESTIONS and WIK-ISQL. We compare against two common baselines to demonstrate the effectiveness of using abstract programs and alignment. We also conduct detailed analysis which shows that structured attention is highly beneficial, enabling our parser to differentiate between correct and spurious programs. Finally, we break down the errors of our parser so as

to examine whether structured attention is better at instantiating abstract programs.

4.1 Experimental Setup

Datasets WIKITABLEQUESTIONS contains 2,018 tables and 18,496 utterance-denotation pairs. The dataset is challenging as 1) the tables cover a wide range of domains and unseen tables appear at test time; and 2) the questions involve a variety of operations such as superlatives, comparisons, and aggregation (Pasupat and Liang, 2015). WIKISQL has 24,241 tables and 80,654 utterance-denotation pairs. The questions are logically simpler and only involve aggregation, column selection, and conditions. The original dataset is annotated with SQL queries, but we only use the execution result for training. In both datasets, tables are extracted from Wikipedia and cover a wide range of domains.

Entity extraction is important during parsing since entities are used as values in filter conditions during instantiation. String entities are extracted by string matching utterance spans and table cells. In WIKITABLEQUESTIONS, numbers and dates are extracted from the CoreNLP annotations released with the dataset. WIKISQL does not have entities for dates, and we use string-based normalization to deal with numbers.

Implementation We obtained word embeddings by a linear projection of GloVe pre-trained embeddings (Pennington et al., 2014) which were fixed during training. Attention scores were computed based on the dot product between two vectors. Each MLP is a one-hidden-layer perceptron with ReLU as the activation function. Dropout (Srivastava et al., 2014) was applied to prevent overfitting. All models were trained with Adam (Kingma and Ba, 2015). Implementations of abstract and instantiation grammars were based on AllenNLP (Gardner et al., 2017).[5]

4.2 Baselines

Aside from comparing our model against previously published approaches, we also implemented the following baselines:

Typed Seq2Seq Programs were generated using a sequence-to-sequence model with attention (Dong and Lapata, 2016). Similarly to Krishnamurthy et al. (2017), we constrained the decod-

---

[5]Please refer to the Appendix for the full list of hyperparameters used in our experiments.

Supervised by Denotations	Dev.	Test
Pasupat and Liang (2015)	37.0	37.1
Neelakantan et al. (2017)	34.1	34.2
Haug et al. (2018)	—	34.8
Zhang et al. (2017)	40.4	43.7
Liang et al. (2018)	42.3	43.1
Dasigi et al. (2019)	42.1	43.9
Agarwal et al. (2019)	43.2	44.1
Typed Seq2Seq	37.3	38.3
Abstract Programs		
f.w. standard attention	39.4	41.4
f.w. structured attention	**43.7**	**44.5**

Table 1: Results on WIKITABLEQUESTIONS. *f.w.* stands for slots filled with.

ing process so that only well-formed programs are predicted. This baseline can be viewed as merging the two stages of our model into one stage where generation of abstract programs and their instantiations are performed with a shared decoder.

Standard Attention The aligned representation of slot s in Equation (6) is computed by a standard attention mechanism: $s = \text{Attention}(r(s), l)$ where $r(s)$ is the representation of slot s from abstract programs. Each slot is aligned independently with attention, and there are no global structural constraints on alignments.

4.3 Main Results

For all experiments, we report the mean accuracy of 5 runs. Results on WIKITABLEQUESTIONS are shown in Table 1. The structured-attention model achieves the best performance, compared against the two baselines and previous approaches. The standard attention baseline with abstract programs is superior to the typed Seq2Seq model, demonstrating the effectiveness of decomposing semantic parsing into two stages. Results on WIKISQL are shown in Table 2. The structured-attention model is again superior to our two baseline models. Interestingly, its performance surpasses previously reported weakly-supervised models (Liang et al., 2018; Agarwal et al., 2019) and is on par even with fully supervised ones (Dong and Lapata, 2018).

The gap between the standard attention baseline and the typed Seq2Seq model is not very large on WIKISQL, compared to WIKITABLEQUESTIONS. Recall from Section 2.2 that WIK-

Supervised by Programs	Dev.	Test
Zhong et al. (2017)	60.8	59.4
Wang et al. (2017)	67.1	66.8
Xu et al. (2017)	69.8	68.0
Huang et al. (2018)	68.3	68.0
Yu et al. (2018)	74.5	73.5
Sun et al. (2018)	75.1	74.6
Dong and Lapata (2018)	79.0	78.5
Shi et al. (2018)	**84.0**	**83.7**
Supervised by Denotations	**Dev.**	**Test**
Liang et al. (2018)	72.2	72.1
Agarwal et al. (2019)	74.9	74.8
Typed Seq2Seq	74.5	74.7
Abstract Programs		
f.w. standard attention	75.2	75.3
f.w. structured attention	79.4	79.3

Table 2: Results on WIKISQL. *f.w.*: slots filled with.

Models	WTQ	WIKISQL
Liang et al. (2018)	46.3 (10)	74.2 (5)
Agarwal et al. (2019)	46.9 (10)	76.9 (5)
Our model	**47.3** (10)	**81.7** (5)

Table 3: Results of ensembled models on the test set; ensemble sizes are shown within parentheses.

Error Types	standard	structured
Abstraction Error	19.2	20.0
Instantiation Error	41.5	36.2
Coverage Error	39.2	43.8

Table 4: Proportion of errors on the development set in WIKITABLEQUESTIONS.

ISQL only has 6 abstract programs that can be successfully instantiated. For this reason, our decomposition alone may not be very beneficial if coupled with standard attention. In contrast, our structured-attention model consistently performs much better than both baselines.

We report scores of ensemble systems in Table 3. We use the best model which relies on abstract programs and structured attention as a base model in our ensemble. Our ensemble system achieves better performance than Liang et al. (2018) and Agarwal et al. (2019), while using the same ensemble size.

4.4 Analysis of Spuriousness

To understand how well structured attention can help a parser differentiate between correct and spurious programs, we analyzed the posterior distribution of consistent programs given a denotation: $p(z|x, t, d)$ where $[[z]] = d$.

WIKISQL includes gold-standard SQL annotations, which we do not use in our experiments but exploit here for analysis. Specifically, we converted the annotations released with WIKISQL to programs licensed by our grammar. We then computed the log-probability of these programs according to the posterior distribution as a measure of how well a parser can identify them amongst all consistent programs $\log \sum_{z^*} p(z^*|x, t, d)$, where z^* denotes correct programs. The average log-

probability assigned to correct programs by structured and standard attention is -0.37 and -0.85, respectively. This gap confirms that structured attention can bias our parser towards correct programs during learning.

4.5 Error Analysis

We further manually inspected the output of our structured-attention model and the standard attention baseline in WIKITABLEQUESTIONS. Specifically, we randomly sampled 130 error cases independently from both models and classified them into three categories.

Abstraction Errors If a parser fails to generate an abstract program, then it is impossible for it to instantiate a consistent complete program.

Instantiation Errors These errors arise when abstract programs are correctly generated, but are mistakenly instantiated either by incorrect column names or filter conditions.

Coverage Errors These errors arise from implicit assumptions made by our parser: a) there is a long tail of unsupported operations that are not covered by our abstract programs; b) if entities are not correctly identified and linked, abstract programs cannot be correctly instantiated.

Table 4 shows the proportion of errors attested by the two attention models. We observe that structured attention suffers less from instantiation errors compared against the standard attention baseline, which points to the benefits of the structured alignment model.

5 Related Work

Neural Semantic Parsing We follow the line of work that applies sequence-to-sequence models (Sutskever et al., 2014) to semantic parsing (Jia and Liang, 2016; Dong and Lapata, 2016). Our work also relates to models which enforce type constraints (Yin and Neubig, 2017; Rabinovich et al., 2017; Krishnamurthy et al., 2017) so as to restrict the vast search space of potential programs. We use both methods as baselines to show that the structured bias introduced by our model can help our parser handle spurious programs in the setting of learning from denotations. Note that our alignment model can also be applied in the supervised case in order to help the parser rule out incorrect programs.

Earlier work has used lexicon mappings (Zettlemoyer and Collins, 2007; Wong and Mooney, 2007; Lu et al., 2008; Kwiatkowski et al., 2010) to model correspondences between programs and natural language. However, these methods cannot generalize to unseen tables where new relations and entities appear. To address this issue, Pasupat and Liang (2015) propose a floating parser which allows partial programs to be generated without being anchored to question tokens. In the same spirit, we use a sequence-to-sequence model to generate abstract programs while relying on explicit alignments to instantiate them. Besides semantic parsing, treating alignments as discrete latent variables has proved effective in other tasks like sequence transduction (Yu et al., 2016) and AMR parsing (Lyu and Titov, 2018).

Learning from Denotations To improve the efficiency of searching for consistent programs, Zhang et al. (2017) use a macro grammar induced from cached consistent programs. Unlike Zhang et al. (2017) who abstract entities and relations from logical forms, we take a step further and abstract the computation of row and column selection. Our work also differs from Pasupat and Liang (2016) who resort to manual annotations to alleviate spuriousness. Instead, we equip our parser with an inductive bias to rule out spurious programs during training. Recently, reinforcement learning based methods address the computational challenge by using a memory buffer (Liang et al., 2018) which stores consistent programs and an auxiliary reward function (Agarwal et al., 2019) which provides feedback to deal with spurious programs. Guu et al. (2017) employ vari-

ous strategies to encourage even distributions over consistent programs in cases where the parser has been misled by spurious programs. Dasigi et al. (2019) use coverage of lexicon-like rules to guide the search of consistent programs.

6 Conclusions

In this paper, we proposed a neural semantic parser that learns from denotations using abstract programs and latent structured alignments. Our parser achieves state-of-the-art performance on two benchmarks, WIKITABLEQUESTIONS and WIKISQL. Empirical analysis shows that the inductive bias introduced by the alignment model helps our parser differentiate between correct and spurious programs. Alignments can exhibit different properties (e.g., monotonicity or bijectivity), depending on the meaning representation language (e.g., logical forms or SQL), the definition of abstract programs, and the domain at hand. We believe that these properties can be often captured within a probabilistic alignment model and hence provide a useful inductive bias to the parser.

References

Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. 2019. Learning to generalize from sparse and underspecified rewards. In *Proc. of ICML*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR*.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proc. of EMNLP*.

Li Zhang Karthik Ramanathan Sesh Sadasivam Rui Zhang Catherine Finegan-Dollak, Jonathan K. Kummerfeld and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In *Proc. of ACL*.

Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke Zettlemoyer, and Eduard Hovy. 2019. Iterative

search for weakly supervised semantic parsing. In *Proc. of NAACL*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proc. of ACL*.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proc. of ACL*.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. Allennlp: A deep semantic natural language processing platform.

Omer Goldman, Veronica Latcinnik, Ehud Nave, Amir Globerson, and Jonathan Berant. 2018. Weakly supervised semantic parsing with abstract examples. In *Proc. of ACL*.

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proc. of NeurIPS*, pages 1828–1836.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proc. of ACL*.

Till Haug, Octavian-Eugen Ganea, and Paulina Grnarova. 2018. Neural multi-step reasoning for question answering on semi-structured tables. *ECIR*.

Jonathan Herzig and Jonathan Berant. 2018. Decoupling structure and lexicon for zero-shot semantic parsing. In *Proc. of EMNLP*.

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

Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen tau Yih, and Xiaodong He. 2018. Natural language to structured query generation via meta-learning. In *Proc. of NAACL*.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proc. of ACL*.

Yoon Kim, Carl Denton, Luong Hoang, and Alexander M Rush. 2017. Structured attention networks. In *Proc. of ICLR*.

Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proc. of ICLR*.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proc. of EMNLP*.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proc. of EMNLP*.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Proc. of NeurIPS*.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Proc. of ACL*.

Wei Lu, Hwee Tou Ng, Wee Sun Lee, and Luke S Zettlemoyer. 2008. A generative model for parsing natural language to meaning representations. In *Proc. of EMNLP*.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proc. of EMNLP*.

Chunchuan Lyu and Ivan Titov. 2018. Amr parsing as graph prediction with latent alignment. In *Proc. of ACL*.

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

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. *Proc. of ICML*.

P. Pasupat and P. Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proc. of ACL*.

Panupong Pasupat and Percy Liang. 2016. Inferring logical forms from denotations. In *Proc. of ACL*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proc. of EMNLP*.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proc. of ACL*.

Tianze Shi, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Oleksandr Polozov, and Weizhu Chen. 2018. Incsql: Training incremental text-to-sql parsers with non-deterministic oracles. *arXiv preprint arXiv:1809.05054*.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 15(1):1929–1958.

Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic parsing with syntax- and table-aware sql generation. In *Proc. of ACL*.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Proc. of NeurIPS*.

Oscar Täckström, Kuzman Ganchev, and Dipanjan Das. 2015. Efficient inference and structured learning for semantic role labeling. *TACL*.

Martin J Wainwright, Michael I Jordan, et al. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305.

Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. 2017. Pointing out SQL queries from text. Technical report, MSR.

Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proc. of ACL*.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proc. of ACL*.

Lei Yu, Jan Buys, and Phil Blunsom. 2016. Online segment to segment neural transduction. In *Proc. of EMNLP*.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *Proc. of NAACL*.

Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In *Proc. of EMNLP/CoNLL*.

Luke S Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proc. of UAI*.

Yuchen Zhang, Panupong Pasupat, and Percy Liang. 2017. Macro grammars and holistic triggering for efficient semantic parsing. In *Proc. of EMNLP*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Grammars

We created our grammars following Zhang et al. (2017) and Liang et al. (2018). Compared with Liang et al. (2018), we additionally support disjunction(*OR*) and conjunction(*AND*). Some functions are pruned based on their effect on coverage, which is the proportion of questions that obtain at least one consistent program. "same_as" function (Liang et al., 2018) is excluded since it introduces too many spurious programs while contributing little to the coverage. For the same reason, conjunction(*AND*) is not used in WIKITABLEQUESTIONS and disjunction(*OR*) is not used in WIKISQL.

We also include non-terminals of function types in production rules (Krishnamurthy et al., 2017). For instance, function "ROW → *first*(LIST[ROW])" selects the first row from a list of rows and will lead to the production rule "ROW → *first*". In the paper, we eliminate the function type for simplicity. Practically, we use two production rules to represent the function: ROW → <ROW: LIST[ROW]> and <ROW:LIST[ROW]> → *first*, where < ROW: LIST[ROW] > is an abstract function type.

B Search for Consistent Programs

We enumerate all possible programs in two stages using the abstract and instantiation grammars. To constrain the space and make the search process tractable, we restrict the maximal number of production rules for generating abstract programs during the first stage. It is based on the observation that the abstract programs which can be successfully instantiated into *correct* programs are usually not very complex. In other words, the *consistent* programs that are instantiated by long abstract programs are very likely to be spurious. For instance, programs like "select(previous(next(previous(argmax [all_rows], column:silver) column:bronze)" are unlikely to have a corresponding question. Specifically, we set the maximal number of production rules for generating abstract programs to 6 and 9, which leads to search time of around 7 and 10 hours for WIKISQL and WIKITABLEQUESTIONS respectively, using a single CPU. Note that this needs to be done only once.

C Hyperparameters

Models used in WIKITABLEQUESTIONS and WIKISQL share similar hyperparameters which are listed in Table 5. Our input embeddings are obtained by a linear projection from the fixed pre-trained embedding (Pennington et al., 2014). Word Indicator refers to the indicator feature of whether a word appears in the table; Column Indicator refers to the indicator feature of whether at least one entity in a column appears in the question. All MLPs mentioned in the paper have the

Hyperparameters	WTQ	WikiSQL
Input Fixed Embedding Size	300	300
Input Linear Projection Size	256	256
POS Embedding Size	64	-
Production Rule Embedding Size	436	328
Column Type Embedding Size	16	16
Word Indictor Embedding Size	16	16
Column Indictor Embedding Size	16	16
Operator Embedding Size	128	128
Encoder Hidden Size	256	256
Encoder Dropout	0.45	0.35
Decoder Hidden Size	218	164
AP Encoder Hidden Size	218	164
AP Encoder Dropout	0.25	0.25
MLP Hidden Size	436	328
MLP Dropout	0.25	0.2

Table 5: Hyperparameters for WTQ (WIKITABLE-QUESTIONS) and WIKISQL. AP Encoder is the encoder representing the abstract programs we generate.

.

same hidden size and dropout rate. During decoding, we choose the top-6 abstract programs to instantiate via beam search.

D Alignments

If a row slot is instantiated with the special condition 'all_rows', then it is possible that the semantics of this slot is implicit. For instance, the question "which driver completed the least number of laps? " should first be mapped to the abstract program "select (argmin (#*row_slot*, #*column_slot*), #*column_slot*)" which is then instantiated to the correct program "select (argmin (all_rows, column:laps) column:driver)". The row slot in the abstract program is instantiated with 'all_rows', but this is not explicitly expressed in the question.

To make it compatible with our constraints in Equation (6) and (7), we add a special token "ALL_ROW" at the end of each question. If a row slot is aligned with this token, then it is expected to be instantiated with the special condition 'all_rows'. Specifically, this special token is mapped to a learnable vector during instantiations. Our alignment needs to learn to align this special token with a row slot if this row slot should be instantiated with the condition 'all_rows'.