TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation

Pengcheng Yin, Graham Neubig
Language Technologies Institute
Carnegie Mellon University
{pcyin,gneubig}@cs.cmu.edu

Abstract

We present TRANX, a transition-based neural semantic parser that maps natural language (NL) utterances into formal meaning representations (MRs). TRANX uses a transition system based on the *abstract syntax description language* for the target MR, which gives it two major advantages: (1) it is highly accurate, using information from the syntax of the target MR to constrain the output space and model the information flow, and (2) it is highly generalizable, and can easily be applied to new types of MR by just writing a new abstract syntax description corresponding to the allowable structures in the MR. Experiments on four different semantic parsing and code generation tasks show that our system is generalizable, extensible, and effective, registering strong results compared to existing neural semantic parsers.[1]

1 Introduction

Semantic parsing is the task of transducing natural language (NL) utterances into formal meaning representations (MRs). The target MRs can be defined according to a wide variety of formalisms. This include linguistically-motivated semantic representations that are designed to capture the meaning of any sentence such as λ-calculus (Zettlemoyer and Collins, 2005) or the abstract meaning representations (Banarescu et al., 2013). Alternatively, for more task-driven approaches to semantic parsing, it is common for meaning representations to represent executable programs such as SQL queries (Zhong et al., 2017), robotic commands (Artzi and Zettlemoyer, 2013), smart phone instructions (Quirk et al., 2015), and even general-purpose programming languages like Python (Yin and Neubig, 2017; Rabinovich et al., 2017) and Java (Ling et al., 2016).

Because of these varying formalisms for MRs, the design of semantic parsers, particularly neural network-based ones has generally focused on a small subset of tasks — in order to ensure the syntactic well-formedness of generated MRs, a parser is usually specifically designed to reflect the domain-dependent grammar of MRs in the structure of the model (Zhong et al., 2017; Xu et al., 2017). To alleviate this issue, there have been recent efforts in neural semantic parsing with general-purpose grammar models (Xiao et al., 2016; Dong and Lapata, 2018). Yin and Neubig (2017) put forward a neural sequence-to-sequence model that generates tree-structured MRs using a series of tree-construction actions, guided by the task-specific context free grammar provided to the model *a priori*. Rabinovich et al. (2017) propose the abstract syntax networks (ASNs), where domain-specific MRs are represented by abstract syntax trees (ASTs, Fig. 2 *Left*) specified under the abstract syntax description language (ASDL) framework (Wang et al., 1997). An ASN employs a modular architecture, generating an AST using specifically designed neural networks for each construct in the ASDL grammar.

Inspired by this existing research, we have developed TRANX, a **TRAN**sition-based abstract syntax**X** parser for semantic parsing and code generation. TRANX is designed with the following principles in mind:

- **Generalization ability** TRANX employs ASTs as a general-purpose intermediate meaning representation, and the task-dependent grammar is provided to the system as external knowledge to guide the parsing process, therefore decoupling the semantic parsing procedure with specificities of grammars.
- **Extensibility** TRANX uses a simple transition system to parse NL utterances into tree-

[1] Available at https://github.com/pcyin/tranX. An earilier version is used in Yin et al. (2018).

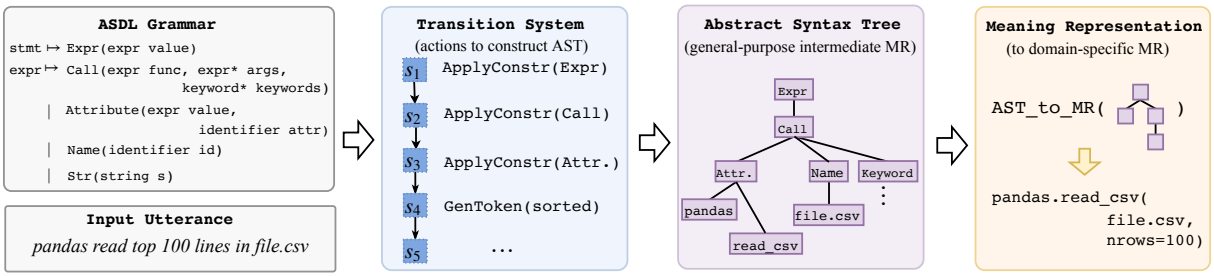


Figure 1: Workflow of TRANX

structured ASTs. The transition system is designed to be easy to extend, requiring minimal engineering to adapt to tasks that need to handle extra domain-specific information.

- **Effectiveness** We test TRANX on four semantic parsing (ATIS, GEO) and code generation (DJANGO, WIKISQL) tasks, and demonstrate that TRANX is capable of generalizing to different domains while registering strong performance, out-performing existing neural network-based approaches on three of the four datasets (GEO, ATIS, DJANGO).

2 Methodology

Given an NL utterance, TRANX parses the utterance into a formal meaning representation, typically represented as λ-calculus logical forms, domain-specific, or general-purpose programming languages (e.g., Python). In the following description we use Python code generation as a running example, where a programmer's natural language intents are mapped to Python source code. Fig. 1 depicts the workflow of TRANX. We will present more use cases of TRANX in § 3.

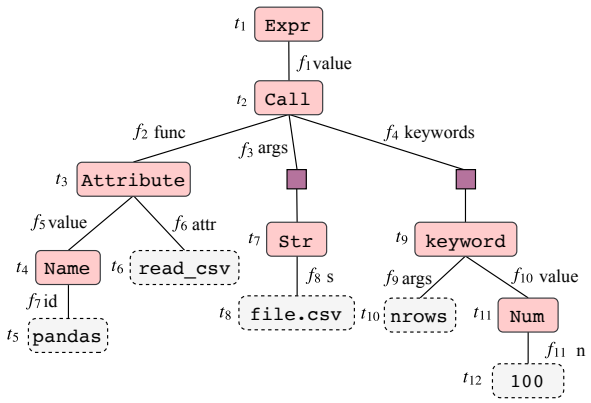
The core of TRANX is a transition system. Given an input NL utterance $\boldsymbol{x}$, TRANX employs the transition system to map the utterance $\boldsymbol{x}$ into an AST $\boldsymbol{z}$ using a series of tree-construction actions (§ 2.2). TRANX employs ASTs as the intermediate meaning representation to abstract over domain-specific structure of MRs. This parsing process is guided by the user-defined, domain-specific grammar specified under the ASDL formalism (§ 2.1). Given the generated AST $\boldsymbol{z}$, the parser calls the user-defined function, `AST_to_MR(·)`, to convert the intermediate AST into a domain-specific meaning representation $\boldsymbol{y}$, completing the parsing process. TRANX uses a probabilistic model $p(\boldsymbol{z}|\boldsymbol{x})$, parameterized by a neural network, to score each hypothesis AST (§ 2.3).

2.1 Modeling ASTs using ASDL Grammar

TRANX uses ASTs as the general-purpose, intermediate semantic representation for MRs. ASTs are commonly used to represent programming languages, and can also be used to represent other tree-structured MRs (*e.g.*, λ-calculus). The ASDL framework is a grammatical formalism to define ASTs. See Fig. 1 for an excerpt of the Python ASDL grammar. TRANX provides APIs to read such a grammar from human-readable text files.

An ASDL grammar has two basic constructs: *types* and *constructors*. A *composite* type is defined by the set of constructors under that type. For example, the `stmt` and `expr` composite types in Fig. 1 refer to Python statements and expressions, repectively, each defined by a series of constructors. A constructor specifies a language construct of a particular type using its *fields*. For instance, the `Call` constructor under the composite type `expr` denotes function call expressions, and has three fields: `func`, `args` and `keywords`. Each field in a constructor is also strongly typed, which specifies the type of value the field can hold. A field with a composite type can be instantiated by constructors of the same type. For example, the `func` field above can hold a constructor of type `expr`. There are also fields with *primitive* types, which store values. For example, the `id` field of `Name` constructor has a primitive type `identifier`, and is used to store identifier names. And the field `s` in the `Str` (string) constructor hold string literals. Finally, each field has a cardinality (single, optional ? and sequential $*$), denoting the number of values the field holds.

An AST is then composed of multiple constructors, where each node on the tree corresponds to a typed field in a constructor (except for the root node, which denotes the root constructor). Depending on the cardinality of the field, a node can hold one or multiple constructors as its values. For instance, the `func` field with single car-



t	n_{f_t}	Action
t_1	root	`Expr(expr value)`
t_2	f_1	`Call(expr func, expr* args, keyword* keywords)`
t_3	f_2	`Attribute(expr value, identifier attr)`
t_4	f_5	`Name(identifier id)`
t_5	f_7	GENTOKEN[pandas]
t_6	f_6	GENTOKEN[read_csv]
t_7	f_3	`Str(string s)`
t_8	f_8	GENTOKEN[file.csv]
t_9	f_8	GENTOKEN[</f>]
t_{10}	f_3	REDUCE (close the frontier field f_3)
t_{11}	f_4	`keyword(identifier arg, expr value)`
t_{12}	f_9	GENTOKEN[nrows]
t_{13}	f_{10}	`Num(object n)`
t_{14}	f_{11}	GENTOKEN[1000]
t_{15}	f_4	REDUCE (close the frontier field f_4)

Figure 2: ***Left*** The ASDL AST for the target Python code in Fig. 1. Field names are labeled on upper arcs, and indexed as f_i. Purple squares denote fields with *sequential* cardinality. Grey nodes denote primitive identifier fields. Fields are labeled with time steps at which they are generated. ***Right*** The action sequence used to construct the AST. Each action is labeled with its frontier field n_{f_t}. APPLYCONSTR actions are represented by their constructors.

dinality in the ASDL grammar in Fig. 1 is instantiated with one `Name` constructor, while the `args` field with sequential cardinality have multiple child constructors.

2.2 Transition System

Inspired by Yin and Neubig (2017) (hereafter YN17), we develop a transition system that decomposes the generation procedure of an AST into a sequence of tree-constructing *actions*. We now explain the transition system using our running example. Fig. 2 *Right* lists the sequence of actions used to construct the example AST. In high level, the generation process starts from an initial derivation AST with a single root node, and proceeds according to a top-down, left-to-right order traversal of the AST. At each time step, one of the following three types of actions is evoked to expand the opening *frontier field* n_{f_t} of the derivation:

APPLYCONSTR$[c]$ actions apply a constructor c to the opening composite frontier field which has the same type as c, populating the opening node using the fields in c. If the frontier field has sequential cardinality, the action appends the constructor to the list of constructors held by the field.

REDUCE actions mark the completion of the generation of child values for a field with optional (?) or multiple ($*$) cardinalities.

GENTOKEN$[v]$ actions populate a (empty) primitive frontier field with a token v. For example, the field f_7 on Fig. 2 has type `identifier`, and is instantiated using a single GENTOKEN action. For fields of `string` type, like f_8, whose value could consists of multiple tokens (only one shown here), it can be filled using a sequence of GENTOKEN actions, with a special GENTOKEN[`</f>`] action to terminate the generation of token values.

The generation completes once there is no frontier field on the derivation. TRANX then calls the user specified function `AST_to_MR(·)` to convert the generated intermediate AST $\boldsymbol{z}$ into the target domain-specific MR $\boldsymbol{y}$. TRANX provides various helper functions to ease the process of writing conversion functions. For example, our example conversion function to transform ASTs into Python source code contains only 32 lines of code. TRANX also ships with several built-in conversion functions to handle MRs commonly used in semantic parsing and code generation, like λ-calculus logical forms and SQL queries.

2.3 Computing Action Probabilities $p(\boldsymbol{z}|\boldsymbol{x})$

Given the transition system, the probability of an $\boldsymbol{z}$ is decomposed into the probabilities of the sequence of actions used to generate $\boldsymbol{z}$

$$p(\boldsymbol{z}|\boldsymbol{x}) = \prod_t p(a_t|a_{<t}, \boldsymbol{x}),$$

Following YN17, we parameterize the transition-based parser $p(\boldsymbol{z}|\boldsymbol{x})$ using a neural encoder-decoder network with augmented recurrent connections to reflect the topology of ASTs.

Encoder The encoder is a standard bidirectional Long Short-term Memory (LSTM) network, which encodes the input utterance $\boldsymbol{x}$ of n tokens, $\{x_i\}_{i=1}^n$ into vectorial representations $\{\mathbf{h}\}_{i=1}^n$.

Decoder The decoder is also an LSTM network, with its hidden state $\mathbf{s}_t$ at each time temp given by

$$\mathbf{s}_t = f_{\text{LSTM}}([\mathbf{a}_{t-1} : \tilde{\mathbf{s}}_{t-1} : \mathbf{p}_t], \mathbf{s}_{t-1}),$$

where f_{LSTM} is the LSTM transition function, and $[:]$ denotes vector concatenation. $\mathbf{a}_{t-1}$ is the em-

```
expr
 = Variable(var variable)
 | Entity(ent entity)
 | Number(num number)
 | Apply(pred predicate, expr* arguments)
 | Argmax(var variable, expr domain, expr body)
 | Argmin(var variable, expr domain, expr body)
 | Count(var variable, expr body)
 | Exists(var variable, expr body)
 | Lambda(var variable, var_type type, expr body)
 | Max(var variable, expr body)
 | Min(var variable, expr body)
 | Sum(var variable, expr domain, expr body)
 | The(var variable, expr body)
 | Not(expr argument)
 | And(expr* arguments)
 | Or(expr* arguments)
 | Compare(cmp_op op, expr left, expr right)

cmp_op = Equal | LessThan | GreaterThan
```

Figure 3: The λ-calculus ASDL grammar for GEO and ATIS, defined in Rabinovich et al. (2017)

bedding of the previous action. We maintain an embedding vector for each action. $\tilde{\mathbf{s}}_t$ is the attentional vector defined as in Luong et al. (2015)

$$\tilde{\mathbf{s}}_t = \tanh(\mathbf{W}_c[\mathbf{c}_t : \mathbf{s}_t]).$$

where $\mathbf{c}_t$ is the context vector retrieved from input encodings $\{\mathbf{h}_i\}_{i=1}^n$ using attention.

Parent Feeding $\mathbf{p}_t$ is a vector that encodes the information of the parent frontier field n_{f_t} on the derivation, which is a concatenation of two vectors: the embedding of the frontier field $\mathbf{n}_{f_t}$, and $\mathbf{s}_{p_t}$, the decoder's state at which the constructor of n_{f_t} is generated by the APPLYCONSTR action. Parent feeding reflects the topology of tree-structured ASTs, and gives better performance on generating complex MRs like Python code (§ 3).

Action Probabilities The probability of an APPLYCONSTR$[c]$ action with embedding $\mathbf{a}_c$ is[2]

$$p(a_t = \text{APPLYCONSTR}[c]|a_{<t}, \boldsymbol{x}) = \text{softmax}(\mathbf{a}_c^\intercal \mathbf{W}\tilde{\mathbf{s}}_t) \quad (1)$$

For GENTOKEN actions, we employ a hybrid approach of generation and copying, allowing for out-of-vocabulary variable names and literals (*e.g.*, "*file.csv*" in Fig. 1) in $\boldsymbol{x}$ to be directly copied to the derivation. Specifically, the action probability is defined to be the marginal probability

$$p(a_t = \text{GENTOKEN}[v]|a_{<t}, \boldsymbol{x}) = p(\text{gen}|a_t, \boldsymbol{x})p(v|\text{gen}, a_t, \boldsymbol{x}) + p(\text{copy}|a_t, \boldsymbol{x})p(v|\text{copy}, a_t, \boldsymbol{x})$$

[2] REDUCE is treated as a special APPLYCONSTR action.

```
stmt = Select(agg_op? agg, idx column_idx,
                           cond_expr* conditions)
cond_expr = Condition(cmp_op op, idx column_idx,
                                 string value)
agg_op = Max | Min | Count | Sum | Avg
cmp_op = Equal | GreaterThan | LessThan | Other
```

Figure 4: The ASDL grammar for WIKISQL

The binary probability $p(\text{gen}|\cdot)$ and $p(\text{copy}|\cdot)$ is given by $\text{softmax}(\mathbf{W}\tilde{\mathbf{s}}_t)$. The probability of generating v from a closed-set vocabulary, $p(v|\text{gen}, \cdot)$ is defined similarly as Eq. (1). The copy probability of copying the i-th word in $\boldsymbol{x}$ is defined using a pointer network (Vinyals et al., 2015)

$$p(x_i|\text{copy}, a_{<t}, \boldsymbol{x}) = \text{softmax}(\mathbf{h}_i^\intercal \mathbf{W}\tilde{\mathbf{s}}_t).$$

3 Experiments

3.1 Datasets

To demonstrate the generalization and extensibility of TRANX, we deploy our parser on four semantic parsing and code generation tasks.

3.1.1 Semantic Parsing

We evaluate on GEO and ATIS datasets. GEO is a collection of 880 U.S. geographical questions (*e.g.*, *"Which states border Texas?"*), and ATIS is a set of 5,410 inquiries of flight information (*e.g.*, *"Show me flights from Dallas to Baltimore"*). The MRs in the two datasets are defined in λ-calculus logical forms (*e.g.*, "`lambda` x `(and (state` x`) (next_to` x `texas))`" and "`lambda` x `(and (flight` x `dallas) (to` x `baltimore))`"). We use the pre-processed datasets released by Dong and Lapata (2016). We use the ASDL grammar defined in Rabinovich et al. (2017), as listed in Fig. 3.

3.1.2 Code Generation

We evaluate TRANX on both general-purpose (Python, DJANGO) and domain-specific (SQL, WIKISQL) code generation tasks. The DJANGO dataset (Oda et al., 2015) consists of 18,805 lines of Python source code extracted from the Django Web framework, with each line paired with an NL description. Code in this dataset covers various real-world use cases of Python, like string manipulation, I/O operation, exception handling, *etc.*

WIKISQL (Zhong et al., 2017) is a code generation task for *domain-specific* languages (*i.e.*, SQL). It consists of 80,654 examples of NL questions (*e.g.*, *"What position did Calvin Mccarty play?"*) and annotated SQL queries (*e.g.*, "`SELECT Position FROM Table WHERE`

Methods	GEO	ATIS
ZH15 (Zhao and Huang, 2015)	88.9	84.2
ZC07 (Zettlemoyer and Collins, 2007)	89.0	84.6
WKZ14 (Wang et al., 2014)	**90.4**	**91.3**
Neural Network-based Models		
SEQ2TREE (Dong and Lapata, 2016)	87.1	84.6
ASN (Rabinovich et al., 2017)	87.1	85.9
TRANX	88.2	86.2

Table 1: Semantic parsing accuracies on GEO and ATIS

Methods	ACC.
NMT (Neubig, 2015)	45.1
LPN (Ling et al., 2016)	62.3
YN17 (Yin and Neubig, 2017)	71.6
TRANX	**73.7**

Table 2: Code generation accuracies on DJANGO

Methods	ACC_{EM}	ACC_{EX}
Seq2SQL (Zhong et al., 2017)	48.3	59.4
SQLNet (Xu et al., 2017)	–	68.0
PT-MAML (Huang et al., 2018)	62.8	68.0
TypeSQL (Yu et al., 2018)	–	**73.5**
TRANX	**62.9**	71.7
PointSQL (Wang et al., 2017)†	61.5	66.8
TypeSQL+TC (Yu et al., 2018)†	–	**82.6**
STAMP (Sun et al., 2018)†	60.7	74.4
STAMP+RL (Sun et al., 2018)†	61.0	74.6
TRANX	**68.4**	78.6

Table 3: Exact match (EM) and execution (EX) accuracies on WIKISQL. †Methods that use the contents of input tables.

`Player = Calvin Mccarty`"). Different from other datasets, each example also has a table extracted from Wikipedia, and the SQL query is executed against the table to get an answer.

Extending TRANX for WIKISQL In order to achieve strong results, existing parsers, like most models in Tab. 3, use specifically designed architectures to reflect the syntactic structure of SQL queries. We show that the transition system used by TRANX can be easily extended for WIKISQL with minimal engineering, while registering strong performance. First, we use define a simple ASDL grammar following the syntax of SQL (Fig. 4). We then augment the transition system with a special GENTOKEN action, SELCOLUMN$[k]$. A SELCOLUMN$[k]$ action is used to populate a primitive `column_idx` field in `Select` and `Condition` constructors in the grammar by selecting the k-th column in the table. To compute the probability of SELCOLUMN$[k]$ actions, we use a pointer network over column encodings, where the column encodings are given by a bidirectional LSTM network over column names in an input table. This can be simply implemented by overriding the base `Parser` class in TRANX and modifying the functions that compute action probabilities.

3.2 Results

In this section we discuss our experimental results. All results are averaged over three runs with different random seeds.

Semantic Parsing Tab. 1 lists the results for semantic parsing tasks. We test TRANX with two configurations, with or without parent feeding (§ 2.3). Our system outperforms existing neural network-based approaches. This demonstrates the effectiveness of TRANX in closed-domain semantic parsing. Interestingly, we found the model without parent feeding achieves slightly better accuracy on GEO, probably because that its relative simple grammar does not require extra handling of parent information.

Code Generation Tab. 2 lists the results on DJANGO. TRANX achieves state-of-the-art results on DJANGO. We also find parent feeding yields +1 point gain in accuracy, suggesting the importance of modeling parental connections in ASTs with complex domain grammars (*e.g.*, Python).

Tab. 3 shows the results on WIKISQL. We first discuss our standard model which only uses information of column names and do not use the contents of input tables during inference, as listed in the top two blocks in Tab. 3. We find TRANX, although just with simple extensions to adapt to this dataset, achieves impressive results and outperforms many *task-specific* methods. This demonstrates that TRANX is easy to extend to incorporate task-specific information, while maintaining its effectiveness. We also extend TRANX with a very simple *answer pruning* strategy, where we execute the candidate SQL queries in the beam against the input table, and prune those that yield empty execution results. Results are listed in the bottom two-blocks in Tab. 3, where we compare with systems that also use the contents of tables. Surprisingly, this (frustratingly) simple extension yields significant improvements, outperforming many task-specific models that use specifically designed, heavily-engineered neural networks to incorporate information of table contents.

4 Conclusion

We present TRANX, a transition-based abstract syntax parser. TRANX is generalizable, extensible and effective, achieving strong results on semantic parsing and code generation tasks.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1815287. PY would like to thank Junxian He and Li Dong for helpful discussions.

References

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transaction of ACL*.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of LAW-ID@ACL*.

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

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

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

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of ACL*.

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

Graham Neubig. 2015. lamtram: A toolkit for language and translation modeling using neural networks. http://www.github.com/neubig/lamtram.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (T). In *Proceedings of ASE*.

Chris Quirk, Raymond J. Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of ACL*.

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

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. *CoRR*, abs/1804.08338.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Proceedings of NIPS*.

Adrienne Wang, Tom Kwiatkowski, and Luke Zettlemoyer. 2014. Morpho-syntactic lexical generalization for CCG semantic parsing. In *Proceedings of EMNLP*.

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

Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. 1997. The Zephyr abstract syntax description language. In *Proceedings of DSL*.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based structured prediction for semantic parsing. In *Proceedings of ACL*.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQL-Net: 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 *Proceedings of ACL*.

Pengcheng Yin, Chunting Zhou, Junxian He, and Graham Neubig. 2018. StructVAE: Tree-structured latent variable models for semi-supervised semantic parsing. In *Proceedings of ACL*.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir R. Radev. 2018. TypeSQL: Knowledge-based type-aware neural text-to-sql generation.

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

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

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

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.