Thesis Proposal

Towards Generalized Neural Semantic Parsing

Pengcheng Yin

April 27, 2020

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15123

Thesis Committee:

Graham Neubig (Chair)	Carnegie Mellon University
Tom Mitchell	Carnegie Mellon University
Yonatan Bisk	Carnegie Mellon University
Luke Zettlemoyer	University of Washington & Facebook AI Research

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: semantic parsing, code generation, program synthesis, structured prediction

Abstract

Semantic parsing, the task of translating user-issued natural language (NL) utterances (*e.g.*, *flights from Pittsburgh to New York*) into machine-executable meaning representations (MRs, *e.g.*, a database query), has become an important direction in developing natural language interfaces to computational systems. Recent years have witnessed the burgeoning of applying neural network-based semantic parsers in various tasks and domains. However, in order to achieve strong performance on a domain, most existing neural semantic parsing models require specifically designed components tailored to capture task-dependent knowledge structures (e.g., in-house database schemas) or domain-specific logical formalisms of MRs (e.g., lambda calculus or SQL queries). Such neural systems are also data-hungry, requiring non-trivial manual annotation effort by domain experts. These issues limit the scope of applications supported by a neural semantic parser, impeding the progress of applying the system to broader scenarios, especially those with diverse and complex structures of meaning representations.

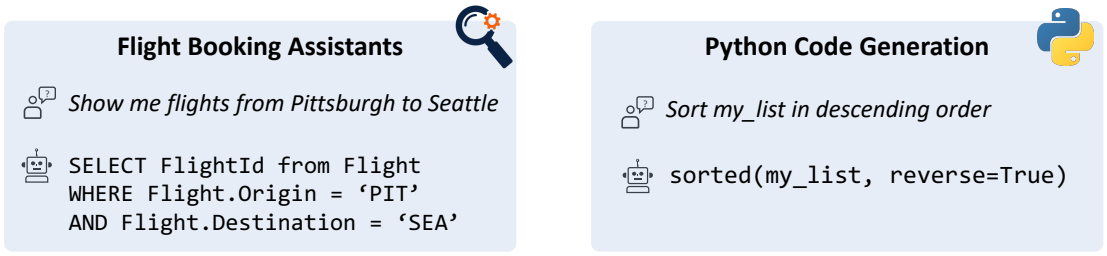
In this thesis, we explore developing neural semantic parsing models that are generally applicable to more domains with various types of knowledge structures and logical formalisms, while providing approaches to mitigate the cost of labeled data acquisition. The dissertation consists of three parts. The first part presents our series of work on developing general-purpose parsing models using abstract syntax trees as domain-agnostic intermediate meaning representations, which decouples the parsing algorithm with task-dependent grammar and logical formalisms. Next, in the second part, we investigate generalized approaches to encode and understand domain-dependent knowledge schema for semantic parsing, with a focus on interpreting domain knowledge represented in (semi-)structured databases. Finally, in the third part, we aim to improve the data efficiency of semantic parsers via semi-supervised learning, while developing machine-assisted approaches to accelerate training data acquisition.

Contents

Chapter 1

Introduction

Semantic parsing studies the task of transducing natural language (NL) utterances into structured formal meaning representations (MRs) [6]. Research in semantic parsing could be categorized into two major threads. The first thread of research considers parsing to general-purpose logical forms that represent the meaning of natural language sentences, like abstract meaning representations [8, 12, 82]. Another direction, which is the central topic of this thesis, is *task-oriented* semantic parsing, where a system accomplishes user-issued tasks by translating her natural language queries into machine-executable programs. Figure 1.1 illustrates two exemplar scenarios of task-oriented semantic parsing. The first example shows a semantic parser as a natural language interface to a flight booking system, where a user's utterance is converted to an SQL query executable on a database of flight information. The second example features code generation from natural language, where a programmer's intent is directly translated into source code written in general-purpose programming languages like Python.



(a) (b)

Figure 1.1: Example applications of task-oriented semantic parsing.

Recent years have witnessed the burgeoning of applying neural network-based semantic parsers in various tasks. Besides the above two applications, significant progress has been

made in other domains, including robot instruction following [83], issuing voice commands on smartphones [14, 100], and question answering over large-scale knowledge bases [62]. Despite such empirical success, advance in neural semantic parsing research is still obstructed by the following practical challenges:

Semantic Parsers are Highly Domain-specific In order to achieve strong performance on a target domain, most existing neural semantic parsers require specifically designed components to capture task-dependent knowledge schema of the domain, or special grammatical structures of the meaning representation. For instance, a parser that interacts with databases (*e.g.*, Figure 1.1(a)) needs to encode the structured relational schema of DB tables, while a Python code generation system (*e.g.*, Figure 1.1(b)) needs to model the syntax rules of the underlying programming language. Additionally, when producing meaning representations, a neural semantic parser often employs specifically-designed components tailored to the structure of task-dependent MRs. As an example, a parser over DBs would use dedicated neural modules to predict columns in the `SELECT` clause and conditional expressions in the `WHERE` clause when generating an SQL query [101, 131]. This issue becomes more problematic for applications with complex structures of MRs, like Python code generation (Figure 1.1(b)). Such strong reliance of a parser's neural architecture on the properties of its underlying domain renders designing neural semantic parsers a non-trivial domain-specific endeavor.

Neural Semantic Parsers are Data Hungry Another issue related to the domain-specific efforts required in developing neural semantic parsers is the cost of model training. Classical supervised learning of neural semantic parsers requires large amounts of training data consisting of parallel NL utterances and manually annotated MRs. However, understanding task-specific MRs requires strong domain knowledge, and its annotation process can be expensive, cumbersome, and time-consuming. Therefore, the limited availability of parallel data has become the bottleneck of existing supervised-based models.

The above challenges have significantly impeded the process of applying neural semantic parsing systems to a broader range of applications, especially those with complex structures of domain knowledge and formalisms of meaning representations.

1.1 Thesis Overview

In this thesis, we put forward a series of approaches to tackle the challenges mentioned above. First, to mitigate the domain-specific engineering efforts in designing neural semantic parsers for understanding task-dependent knowledge schemas and meaning representations, we develop generalizable knowledge representation and parsing models that apply to various types of knowledge schemas and grammatical formalisms. Next, to reduce the cost of acquiring labeled training data, we improve the data-efficiency of neural semantic parsing systems using semi-supervised learning, while expediting the data acquisition process with a semi-automatic mining algorithm.

The thesis consists of three parts. A detailed overview of each part is as follows:

Part I Generalized Syntactic Models for Neural Semantic Parsing In this part, we put forward generalized parsing models that abstract the domain-specific formalisms of meaning representations using abstract syntax trees (ASTs) as a general-purpose MR. Instead of designing specialized parsing modules to reflect the structure of domain grammars, these models employ a unified syntactic parsing model that transduces NL utterances into domain-general ASTs. The generation of ASTs is gauged by the underlying domain-specific grammar, which is defined as syntax rules in a context-free grammar, ensuring the syntactic well-formedness of generated ASTs. We first show that the proposed syntax-driven parsing model leads to significant improvements in code generation, where the grammar of MRs is larger and more complex than classical semantic parsing tasks (Chapter 2). To demonstrate the approach could be generalized to different paradigms of MRs and tasks, we further develop a general-purpose parsing framework (Chapter 3), which provides a unified interface to specify task-dependent grammars for the syntax-driven parsing model, while remaining to be flexible to incorporate extra domain-specific knowledge. We show the model achieves competitive performance on a variety of semantic parsing and code generation benchmarks.

Part II Generalized Data Understanding Models for Neural Semantic Parsing Another related issue that hinders the development of generalized neural semantic parsers is the fact that a parser needs to encode task-dependent knowledge schemas (*e.g.*, the relational database schema of a flight booking system in Figure 1.1), which are essential to understand user-issued utterances in the domain. This chapter introduces generalized approaches to encode and understand domain knowledge represented in (semi-)structured database tables. First, we propose

TaBert, a pre-trained encoding model for learning joint contextual representations of NL utterances and database tables (Chapter 4). TaBert is trained on massive parallel corpora of NL contexts and Web tables, and can be used as a drop-in replacement of a parser's original encoder for computing representations of utterances and DB tables. Next, to quickly adapt a semantic parser to understand the knowledge schemas presented in new domains, as the proposed work, we lift the requirement of annotated logical forms for training the schema understanding model by leveraging external broad-coverage semantic representations of NL utterances, as well as the content of databases as indirect supervision to guide the grounding of utterances to domain-specific schemas (Chapter 5).

Part III Data Efficient Approaches for Neural Semantic Parsing In the third part, we seek measures to mitigate the paucity of annotated training data. We first present an algorithm for semi-supervised learning of semantic parsers, where a parser is trained with both limited amount of labeled parallel data, as well as readily-available unlabeled natural language utterances. We propose a variational auto-encoding model that treats MRs not observed in the unlabeled data as tree-structured latent variables (Chapter 6). Next, we study the problem of training data collection in the context of code generation, with the aim of acquiring relatively large amounts of utterances and labeled programs with fewer annotation efforts required from domain experts. We resort to curated resources on community question answering websites (StackOverflow), and propose a semi-automatic mining algorithm for cost-effective extraction of parallel corpora (Chapter 7).

1.2 Summary of Progress

Part I and Part III are completed. The original syntax-driven neural semantic parser (Chapter 2) is published at ACL 2017, with its extended general-purpose parsing framework (Chapter 3) published at EMNLP 2018 (Demo Track). The TaBert representation learning model over utterances and DB contents (Chapter 4) is published at ACL 2020. The StructVAE semi-supervised learning method (Chapter 6) is published at ACL 2018. The semi-automatic data acquisition approach (Chapter 7) is published at MSR 2018. The unsupervised schema understanding model (Chapter 5) is proposed as future work.

The work presented here has also inspired other relevant research projects not included in the thesis. These include a post hoc reranking model for neural semantic parsing (ACL 2019 short paper), a neural tree-to-tree transduction model for learning edit patterns of natural

language and source code (ICLR 2019), and graph neural networks for variable re-naming in decompiled binaries (ASE 2019).

1.3 Proposed Timeline

May 2020	Thesis Proposal
May 2020 — Aug. 2020	Internship at Microsoft Semantic Machines
Sept. 2020 — Jan. 2021	Finish the Proposed Work
Jan 2021 — Apr. 2021	Job Hunting
Aug. 2021	Thesis Defense

Part I

Generalized Syntactic Models for Neural Semantic Parsing

Chapter 2

Syntactic Models for Code Generation (Completed)

In this chapter, we introduce a neural semantic parsing model that works with different grammatical formalisms of meaning representations. The model uses abstract syntax trees (ASTs) as general-purpose intermediate meaning representations. ASTs are abstract data structures that represent the syntactic and structural information of programs, without modeling domain-specific details of how the program is actually written in the target programming language (*e.g.*, whether to use semicolons as line delimiters). Under this model, domain-specific MRs are represented as ASTs to abstract away their task-dependent details. The parser also captures domain-specific grammar rules as syntactic prior knowledge to guide the generation of ASTs. In this chapter, we use code generation as a running example to demonstrate the proposed model could scale to generating MRs with complex syntactic structures (*e.g.*, Python code). Later on, in Chapter 3, we discuss how to generalize the approach to a parsing framework that handles a variety of semantic parsing tasks. This line of work first appears in:

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

2.1 Overview

Every programmer has experienced the situation where they know what they want to do, but do not have the ability to turn it into a concrete implementation. For example, a Python programmer may want to "*sort my_list in descending order*," but not be able to come up with the

proper syntax `sorted(my_list, reverse=True)` to realize his intention. To resolve this impasse, it is common for programmers to search the web in natural language (NL), find an answer, and modify it into the desired form [20, 21]. However, this is time-consuming, and thus the software engineering literature is ripe with methods to directly generate code from NL descriptions, mostly with hand-engineered methods highly tailored to specific programming languages [11, 39, 69].

In parallel, the NLP community has developed methods for data-driven semantic parsing, which attempt to map NL to structured logical forms executable by computers. While these methods have the advantage of being learnable from data, compared to the programming languages (PLs) in use by programmers, the *domain-specific* languages targeted by these works have a schema and syntax that is relatively simple.

Recently, Ling et al. [68] have proposed a data-driven code generation method for high-level, *general-purpose* PLs like Python and Java. This work treats code generation as a sequence-to-sequence modeling problem, and introduce methods to generate words from character-level models, and copy variable names from input descriptions. However, unlike most work in semantic parsing, it does not consider the fact that code has to be well-defined programs in the target syntax.

In this chapter, we propose a data-driven syntax-based neural network model tailored for generation of general-purpose PLs like Python. In order to capture the strong underlying syntax of the PL, we define a model that transduces an NL statement into an Abstract Syntax Tree (AST; Fig. 2.1(a), § 2.2) for the target PL. ASTs can be deterministically generated for all well-formed programs using standard parsers provided by the PL, and thus give us a way to obtain syntax information with minimal engineering. Once we generate an AST, we can use deterministic generation tools to convert the AST into surface code. We hypothesize that such a structured approach has two benefits.

First, we hypothesize that structure can be used to constrain our search space, ensuring generation of well-formed code. To this end, we propose a syntax-driven neural code generation model. The backbone of our approach is a *grammar model* (§ 2.3) which formalizes the generation story of a derivation AST into sequential application of *actions* that either apply production rules (§ 2.3.1), or emit terminal tokens (§ 2.3.2). The underlying syntax of the PL is therefore encoded in the grammar model *a priori* as the set of possible actions. Our approach frees the model from recovering the underlying grammar from limited training data, and instead enables the system to focus on learning the compositionality among existing grammar rules. Xiao et al. [129] have noted that this imposition of structure on neural models is useful

Production Rule	Role	Explanation
Call $\mapsto$ expr[*func*] expr*[*args*] keyword*[*keywords*]	Function Call	▷ *func*: the function to be invoked ▷ *args*: arguments list ▷ *keywords*: keyword arguments list
If $\mapsto$ expr[*test*] stmt*[*body*] stmt*[*orelse*]	If Statement	▷ *test*: condition expression ▷ *body*: statements inside the If clause ▷ *orelse*: elif or else statements
For $\mapsto$ expr[*target*] expr*[*iter*] stmt*[*body*] stmt*[*orelse*]	For Loop	▷ *target*: iteration variable ▷ *iter*: enumerable to iterate over ▷ *body*: loop body ▷ *orelse*: else statements
FunctionDef $\mapsto$ identifier[*name*] arguments*[*args*] stmt*[*body*]	Function Def.	▷ *name*: function name ▷ *args*: function arguments ▷ *body*: function body

Table 2.1: Example production rules for common Python statements [99]

for semantic parsing, and we expect this to be even more important for general-purpose PLs where the syntax trees are larger and more complex.

Second, we hypothesize that structural information helps to model information flow within the neural network, which naturally reflects the recursive structure of PLs. To test this, we extend a standard recurrent neural network (RNN) decoder to allow for additional neural connections which reflect the recursive structure of an AST (§ 2.3.3). As an example, when expanding the node $\star$ in Fig. 2.1(a), we make use of the information from both its parent and left sibling (the dashed rectangle). This enables us to locally pass information of relevant code segments via neural network connections, resulting in more confident predictions.

Experiments (§ 2.4) on two Python code generation tasks show 11.7% and 9.3% absolute improvements in accuracy against the state-of-the-art system [68]. Our model also gives competitive performance on a standard semantic parsing benchmark[1].

2.2 The Code Generation Problem

Given an NL description x, our task is to generate the code snippet c in a modern PL based on the intent of x. We attack this problem by first generating the underlying AST. We define a probabilistic grammar model of generating an AST y given x: $p(y|x)$. The best-possible AST $\hat{y}$

[1]Implementation available at https://github.com/neulab/NL2code

is then given by

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, p(y|x). \tag{2.1}$$

$\hat{y}$ is then deterministically converted to the corresponding surface code c.[2] While this paper uses examples from Python code, our method is PL-agnostic.

Before detailing our approach, we first present a brief introduction of the Python AST and its underlying grammar. The Python abstract grammar contains a set of production rules, and an AST is generated by applying several production rules composed of a head node and multiple child nodes. For instance, the first rule in Tab. 2.1 is used to generate the function call `sorted(·)` in Fig. 2.1(a). It consists of a head node of type `Call`, and three child nodes of type `expr`, `expr*` and `keyword*`, respectively. Labels of each node are noted within brackets. In an AST, non-terminal nodes sketch the general structure of the target code, while terminal nodes can be categorized into two types: *operation terminals* and *variable terminals*. Operation terminals correspond to basic arithmetic operations like `AddOp`.Variable terminal nodes store values for variables and constants of built-in data types[3]. For instance, all terminal nodes in Fig. 2.1(a) are variable terminal nodes.

2.3 Grammar Model

Before detailing our neural code generation method, we first introduce the grammar model at its core. Given an input description x, our probabilistic grammar model defines the generative story of a derivation AST. We factorize the generation process of an AST into sequential application of *actions* of two types:

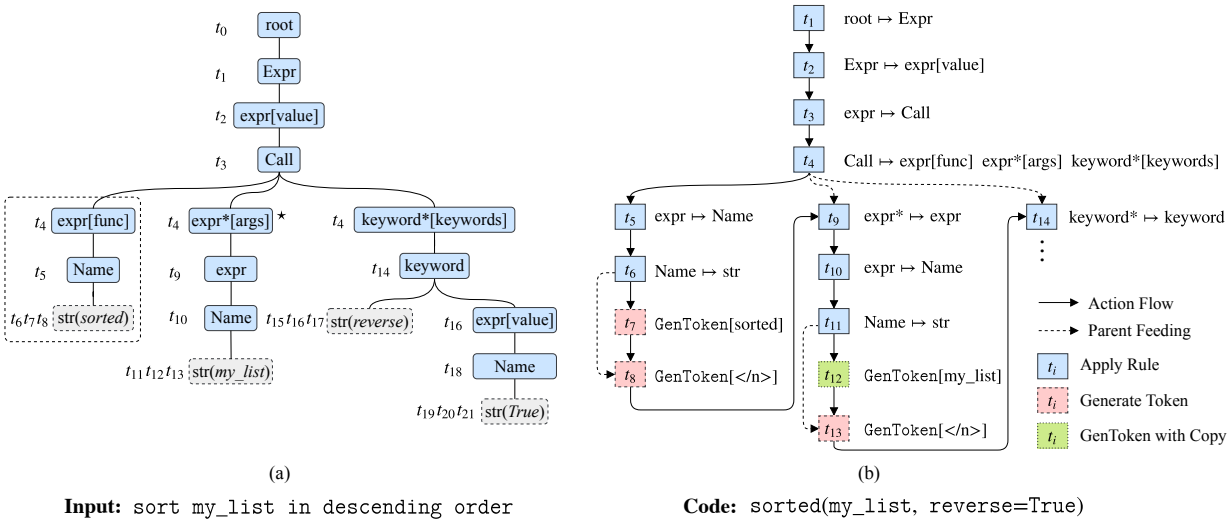
- ApplyRule$[r]$ applies a production rule r to the current derivation tree;
- GenToken$[v]$ populates a variable terminal node by appending a terminal token v.

Fig. 2.1(b) shows the generation process of the target AST in Fig. 2.1(a). Each node in Fig. 2.1(b) indicates an action. Action nodes are connected by solid arrows which depict the chronological order of the action flow. The generation proceeds in depth-first, left-to-right order (dotted arrows represent parent feeding, explained in § 2.3.3).

Formally, under our grammar model, the probability of generating an AST y is factorized as:

[2] We use `astor` library to convert ASTs into Python code.

[3] `bool`, `float`, `int`, `str`.



Input: `sort my_list in descending order`

Code: `sorted(my_list, reverse=True)`

Figure 2.1: (a) the Abstract Syntax Tree (AST) for the given example code. Dashed nodes denote terminals. Nodes are labeled with time steps during which they are generated. (b) the action sequence (up to t_{14}) used to generate the AST in (a)

$$p(y|x) = \prod_{t=1}^{T} p(a_t|x, a_{<t}), \tag{2.2}$$

where a_t is the action taken at time step t, and $a_{<t}$ is the sequence of actions before t. We will explain how to compute the action probabilities $p(a_t|\cdot)$ in Eq. (2.2) in § 2.3.3. Put simply, the generation process begins from a `root` node at t_0, and proceeds by the model choosing ApplyRule actions to generate the overall program structure from a closed set of grammar rules, then at leaves of the tree corresponding to variable terminals, the model switches to GenToken actions to generate variables or constants from the open set. We describe this process in detail below.

2.3.1 ApplyRule Actions

ApplyRule actions generate program structure, expanding the current node (the *frontier node* at time step t: n_{f_t}) in a depth-first, left-to-right traversal of the tree. Given a fixed set of production rules, ApplyRule chooses a rule r from the subset that has a head matching the type of n_{f_t}, and uses r to expand n_{f_t} by appending all child nodes specified by the selected production. As an example, in Fig. 2.1(b), the rule `Call` $\mapsto$ `expr...` expands the frontier node `Call` at time step t_4, and its three child nodes `expr`, `expr*` and `keyword*` are added to the derivation.

ApplyRule actions grow the derivation AST by appending nodes. When a variable terminal node (e.g., `str`) is added to the derivation and becomes the frontier node, the grammar model then switches to GenToken actions to populate the variable terminal with tokens.

Unary Closure Sometimes, generating an AST requires applying a chain of unary productions. For instance, it takes three time steps ($t_9 - t_{11}$) to generate the sub-structure `expr*` $\mapsto$ `expr` $\mapsto$ `Name` $\mapsto$ `str` in Fig. 2.1(a). This can be effectively reduced to one step of ApplyRule action by taking the closure of the chain of unary productions and merging them into a single rule: `expr*` $\mapsto^*$ `str`. Unary closures reduce the number of actions needed, but would potentially increase the size of the grammar. In our experiments we tested our model both with and without unary closures (§ 2.4).

2.3.2 GenToken Actions

Once we reach a frontier node n_{f_t} that corresponds to a variable type (e.g., `str`), GenToken actions are used to fill this node with values. For general-purpose PLs like Python, variables and constants have values with one or multiple tokens. For instance, a node that stores the name of a function (e.g., `sorted`) has a single token, while a node that denotes a string constant (e.g., `a='hello world'`) could have multiple tokens. Our model copes with both scenarios by firing GenToken actions at one or more time steps. At each time step, GenToken appends one terminal token to the current frontier variable node. A special `</n>` token is used to "close" the node. The grammar model then proceeds to the new frontier node.

Terminal tokens can be generated from a pre-defined vocabulary, or be directly copied from the input NL. This is motivated by the observation that the input description often contains out-of-vocabulary (OOV) variable names or literal values that are directly used in the target code. For instance, in our running example the variable name `my_list` can be directly copied from the the input at t_{12}. We give implementation details in § 2.3.3.

2.3.3 Estimating Action Probabilities

We estimate action probabilities in Eq. (2.2) using attentional neural encoder-decoder models with an information flow structured by the syntax trees.

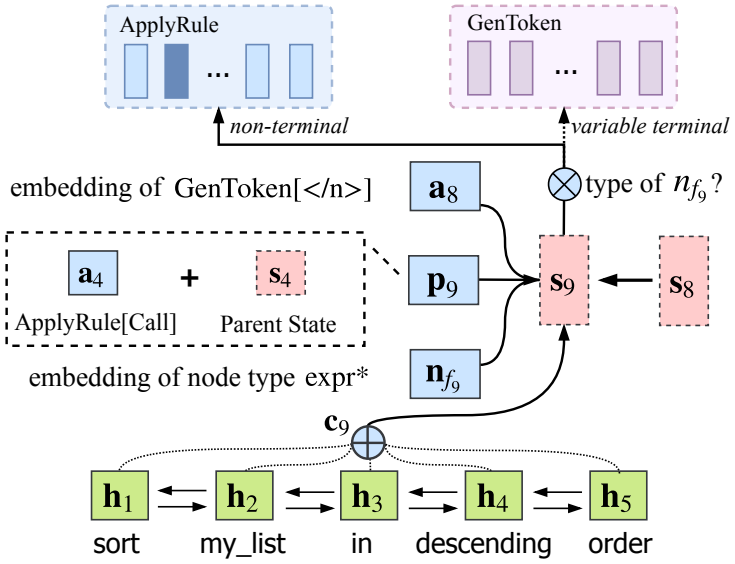


Figure 2.2: Illustration of a decoder time step ($t = 9$)

Encoder

For an NL description x consisting of n words $\{w_i\}_{i=1}^{n}$, the encoder computes a context sensitive embedding $\mathbf{h_i}$ for each w_i using a bidirectional Long Short-Term Memory (LSTM) network [43], similar to the setting in [9]. See supplementary materials for detailed equations.

Decoder

The decoder uses an RNN to model the sequential generation process of an AST defined as Eq. (2.2). Each action step in the grammar model naturally grounds to a time step in the decoder RNN. Therefore, the action sequence in Fig. 2.1(b) can be interpreted as unrolling RNN time steps, with solid arrows indicating RNN connections. The RNN maintains an internal state to track the generation process (§ 2.3.3), which will then be used to compute action probabilities $p(a_t|x, a_{<t})$ (§ 2.3.3).

Tracking Generation States

Our implementation of the decoder resembles a vanilla LSTM, with additional neural connections (parent feeding, Fig. 2.1(b)) to reflect the topological structure of an AST. The decoder's internal hidden state at time step t, $\mathbf{s}_t$, is given by:

$$\mathbf{s}_t = f_{\text{LSTM}}([\mathbf{a}_{t-1} : \mathbf{c}_t : \mathbf{p}_t : \mathbf{n}_{f_t}], \mathbf{s}_{t-1}), \tag{2.3}$$

where $f_{\text{LSTM}}(\cdot)$ is the LSTM update function. $[:]$ denotes vector concatenation. $\mathbf{s}_t$ will then be used to compute action probabilities $p(a_t|x, a_{<t})$ in Eq. (2.2). Here, $\mathbf{a}_{t-1}$ is the embedding of the previous action. $\mathbf{c}_t$ is a context vector retrieved from input encodings $\{\mathbf{h}_i\}$ via soft attention. $\mathbf{p}_t$ is a vector that encodes the information of the parent action. $\mathbf{n}_{f_t}$ denotes the node type embedding of the current frontier node n_{f_t}[4]. Intuitively, feeding the decoder the information of n_{f_t} helps the model to keep track of the frontier node to expand.

Action Embedding $\mathbf{a}_t$ We maintain two action embedding matrices, $\mathbf{W}_R$ and $\mathbf{W}_G$. Each row in $\mathbf{W}_R$ ($\mathbf{W}_G$) corresponds to an embedding vector for an action ApplyRule$[r]$ (GenToken$[v]$).

Context Vector $\mathbf{c}_t$ The decoder RNN uses soft attention to retrieve a context vector $\mathbf{c}_t$ from the input encodings $\{\mathbf{h}_i\}$ pertain to the prediction of the current action. We follow Bahdanau et al. [9] and use a Deep Neural Network (DNN) with a single hidden layer to compute attention weights.

Parent Feeding $\mathbf{p}_t$ Our decoder RNN uses additional neural connections to directly pass information from parent actions. For instance, when computing $\mathbf{s}_9$, the information from its parent action step t_4 will be used. Formally, we define the *parent action step* p_t as the time step at which the frontier node n_{f_t} is generated. As an example, for t_9, its parent action step p_9 is t_4, since n_{f_9} is the node $\star$, which is generated at t_4 by the ApplyRule$[\texttt{Call}\mapsto\ldots]$ action.

We model parent information $\mathbf{p}_t$ from two sources: (1) the hidden state of parent action $\mathbf{s}_{p_t}$, and (2) the embedding of parent action $\mathbf{a}_{p_t}$. $\mathbf{p}_t$ is the concatenation. The parent feeding schema enables the model to utilize the information of parent code segments to make more confident predictions. Similar approaches of injecting parent information were also explored in the Seq2Tree model in Dong and Lapata [31][5].

Calculating Action Probabilities

In this section we explain how action probabilities $p(a_t|x, a_{<t})$ are computed based on $\mathbf{s}_t$.

[4] We maintain an embedding for each node type.

[5] Seq2Tree generates tree-structured outputs by conditioning on the hidden states of parent non-terminals, while our parent feeding uses the states of parent actions.

ApplyRule The probability of applying rule r as the current action a_t is given by a softmax[6]:

$$p(a_t = \textsc{ApplyRule}[r]|x, a_{<t}) = \text{softmax}(\mathbf{W}_R \cdot g(\mathbf{s}_t))^\intercal \cdot \mathbf{e}(r) \tag{2.4}$$

where $g(\cdot)$ is a non-linearity $\tanh(\mathbf{W} \cdot \mathbf{s}_t + \mathbf{b})$, and $\mathbf{e}(r)$ the one-hot vector for rule r.

GenToken As in § 2.3.2, a token v can be generated from a predefined vocabulary or copied from the input, defined as the marginal probability:

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

The selection probabilities $p(\text{gen}|\cdot)$ and $p(\text{copy}|\cdot)$ are given by $\text{softmax}(\mathbf{W}_S \cdot \mathbf{s}_t)$. The probability of generating v from the vocabulary, $p(v|\text{gen}, x, a_{<t})$, is defined similarly as Eq. (2.4), except that we use the GenToken embedding matrix $\mathbf{W}_G$, and we concatenate the context vector $\mathbf{c}_t$ with $\mathbf{s}_t$ as input. To model the copy probability, we follow recent advances in modeling copying mechanism in neural networks [36, 48, 68], and use a pointer network [117] to compute the probability of copying the i-th word from the input by attending to input representations $\{\mathbf{h}_i\}$:

$$p(w_i|\text{copy}, x, a_{<t}) = \frac{\exp(\omega(\mathbf{h}_i, \mathbf{s}_t, \mathbf{c}_t))}{\sum_{i'=1}^{n} \exp(\omega(\mathbf{h}_{i'}, \mathbf{s}_t, \mathbf{c}_t))},$$

where $\omega(\cdot)$ is a DNN with a single hidden layer. Specifically, if w_i is an OOV word (e.g., the variable name `my_list`), which is represented by a special `<unk>` token during encoding, we then directly copy the actual word w_i from the input description to the derivation.

2.3.4 Training and Inference

Given a dataset of pairs of NL descriptions x_i and code snippets c_i, we parse c_i into its AST y_i and decompose y_i into a sequence of oracle actions, which explains the generation story of y_i under the grammar model. The model is then optimized by maximizing the log-likelihood of the oracle action sequence. At inference time, given an NL description, we use beam search to approximate the best AST $\hat{y}$ in Eq. (2.1). See supplementary materials for the pseudo-code of the inference algorithm.

[6] We do not show bias terms for all softmax equations.

2.4 Experimental Evaluation

2.4.1 Datasets and Metrics

Dataset	HS	Django	Ifttt
Train	533	16,000	77,495
Development	66	1,000	5,171
Test	66	1,805	758
Avg. tokens in description	39.1	14.3	7.4
Avg. characters in code	360.3	41.1	62.2
Avg. size of AST (# nodes)	136.6	17.2	7.0
Statistics of Grammar			
w/o unary closure			
# productions	100	222	1009
# node types	61	96	828
terminal vocabulary size	1361	6733	0
Avg. # actions per example	173.4	20.3	5.0
w/ unary closure			
# productions	100	237	–
# node types	57	92	–
Avg. # actions per example	141.7	16.4	–

Table 2.2: Statistics of datasets and associated grammars

HearthStone (HS) dataset [68] is a collection of Python classes that implement cards for the card game HearthStone. Each card comes with a set of fields (e.g., name, cost, and description), which we concatenate to create the input sequence. This dataset is relatively difficult: input descriptions are short, while the target code is in complex class structures, with each AST having 137 nodes on average.

Django dataset [90] is a collection of lines of code from the Django web framework, each with a manually annotated NL description. Compared with the HS dataset where card implementations are somewhat homogenous, examples in Django are more diverse, spanning a wide variety of real-world use cases like string manipulation, IO operations, and exception handling.

Ifttt dataset [100] is a domain-specific benchmark that provides an interesting side comparison. Different from HS and Django which are in a general-purpose PL, programs in Ifttt are written in a domain-specific language used by the IFTTT task automation App. Users of the App

write simple instructions (e.g., `If Instagram.AnyNewPhotoByYou Then Dropbox.AddFileFromURL`) with NL descriptions (e.g., "*Autosave your Instagram photos to Dropbox*"). Each statement inside the `If` or `Then` clause consists of a channel (e.g., `Dropbox`) and a function (e.g., `AddFileFromURL`)[7]. This simple structure results in much more concise ASTs (7 nodes on average). Because all examples are created by ordinary Apps users, the dataset is highly noisy, with input NL very loosely connected to target ASTs. The authors thus provide a high-quality filtered test set, where each example is verified by at least three annotators. We use this set for evaluation. Also note IFTTT's grammar has more productions (Tab. 2.2), but this does not imply that its grammar is more complex. This is because for HS and DJANGO terminal tokens are generated by GENTOKEN actions, but for IFTTT, all the code is generated directly by APPLYRULE actions.

Metrics As is standard in semantic parsing, we measure **accuracy**, the fraction of correctly generated examples. However, because generating an exact match for complex code structures is non-trivial, we follow Ling et al. [68], and use token-level **BLEU-4** with as a secondary metric, defined as the averaged BLEU scores over all examples.[8]

2.4.2 Setup

Preprocessing All input descriptions are tokenized using NLTK. We perform simple canonicalization for DJANGO, such as replacing quoted strings in the inputs with place holders. See supplementary materials for details. We extract unary closures whose frequency is larger than a threshold k ($k = 30$ for HS and 50 for DJANGO).

Configuration The size of all embeddings is 128, except for node type embeddings, which is 64. The dimensions of RNN states and hidden layers are 256 and 50, respectively. Since our datasets are relatively small for a data-hungry neural model, we impose strong regularization using recurrent dropouts [34] for all recurrent networks, together with standard dropout layers added to the inputs and outputs of the decoder RNN. We validate the dropout probability from $\{0, 0.2, 0.3, 0.4\}$. For decoding, we use a beam size of 15.

[7] Like Beltagy and Quirk [14], we strip function parameters since they are mostly specific to users.

[8] These two metrics are not ideal: accuracy only measures exact match and thus lacks the ability to give credit to semantically correct code that is different from the reference, while it is not clear whether BLEU provides an appropriate proxy for measuring semantics in the code generation task. A more intriguing metric would be directly measuring semantic/functional code equivalence, for which we present a pilot study at the end of this section (cf. Error Analysis). We leave exploring more sophisticated metrics (e.g. based on static code analysis) as future work.

	HS		Django	
	ACC	BLEU	ACC	BLEU
Retrieval System†	0.0	62.5	14.7	18.6
Phrasal Statistical MT†	0.0	34.1	31.5	47.6
Hierarchical Statistical MT†	0.0	43.2	9.5	35.9
NMT	1.5	60.4	45.1	63.4
Seq2Tree	1.5	53.4	28.9	44.6
Seq2Tree–UNK	13.6	62.8	39.4	58.2
LPN†	4.5	65.6	62.3	77.6
Our system	16.2	**75.8**	**71.6**	**84.5**
Ablation Study				
– frontier embed.	**16.7**	**75.8**	70.7	83.8
– parent feed.	10.6	75.7	71.5	84.3
– copy terminals	3.0	65.7	32.3	61.7
+ unary closure	–		70.3	83.3
– unary closure	10.1	74.8	–	

Table 2.3: Results on two Python code generation tasks. †Results previously reported in Ling et al. [68].

2.4.3 Results

Evaluation results for Python code generation tasks are listed in Tab. 2.3. Numbers for our systems are averaged over three runs. We compare primarily with two approaches: (1) Latent Predictor Network (LPN), a state-of-the-art sequence-to-sequence code generation model [68], and (2) Seq2Tree, a neural semantic parsing model [31]. Seq2Tree generates trees one node at a time, and the target grammar is not explicitly modeled a priori, but *implicitly* learned from data. We test both the original Seq2Tree model released by the authors and our revised one (Seq2Tree–UNK) that uses unknown word replacement to handle rare words [74]. For completeness, we also compare with a strong neural machine translation (NMT) system [86] using a standard encoder-decoder architecture with attention and unknown word replacement[9], and include numbers from other baselines used in Ling et al. [68]. On the HS dataset, which has

[9]For NMT, we also attempted to find the best-scoring syntactically correct predictions in the size-5 beam, but this did not yield a significant improvement over the NMT results in Tab. 2.3.

relatively large ASTs, we use unary closure for our model and SEQ2TREE, and for DJANGO we do not.

System Comparison As in Tab. 2.3, our model registers 11.7% and 9.3% absolute improvements over LPN in accuracy on HS and DJANGO. This boost in performance strongly indicates the importance of modeling grammar in code generation. For the baselines, we find LPN outperforms NMT and SEQ2TREE in most cases. We also note that SEQ2TREE achieves a decent accuracy of 13.6% on HS, which is due to the effect of unknown word replacement, since we only achieved 1.5% without it. A closer comparison with SEQ2TREE is insightful for understanding the advantage of our syntax-driven approach, since both SEQ2TREE and our system output ASTs: (1) SEQ2TREE predicts one node each time step, and requires additional "dummy" nodes to mark the boundary of a subtree. The sheer number of nodes in target ASTs makes the prediction process error-prone. In contrast, the APPLYRULE actions of our grammar model allows for generating multiple nodes at a single time step. Empirically, we found that in HS, SEQ2TREE takes more than 300 time steps on average to generate a target AST, while our model takes only 170 steps. (2) SEQ2TREE does not directly use productions in the grammar, which possibly leads to grammatically incorrect ASTs and thus empty code outputs. We observe that the ratio of grammatically incorrect ASTs predicted by SEQ2TREE on HS and DJANGO are 21.2% and 10.9%, respectively, while our system guarantees grammaticality.

Ablation Study We also ablated our best-performing models to analyze the contribution of each component. "–frontier embed." removes the frontier node embedding $\mathbf{n}_{f_t}$ from the decoder RNN inputs (Eq. (2.3)). This yields worse results on DJANGO while gives slight improvements in accuracy on HS. This is probably because that the grammar of HS has fewer node types, and thus the RNN is able to keep track of n_{f_t} without depending on its embedding. Next, "–parent feed." removes the parent feeding mechanism. The accuracy drops significantly on HS, with a marginal deterioration on DJANGO. This result is interesting because it suggests that parent feeding is more important when the ASTs are larger, which will be the case when handling more complicated code generation tasks like HS. Finally, removing the pointer network ("–copy terminals") in GENTOKEN actions gives poor results, indicating that it is important to directly copy variable names and values from the input.

The results with and without unary closure demonstrate that, interestingly, it is effective on HS but not on DJANGO. We conjecture that this is because on HS it significantly reduces the number of actions from 173 to 142 (c.f., Tab. 2.2), with the number of productions in the

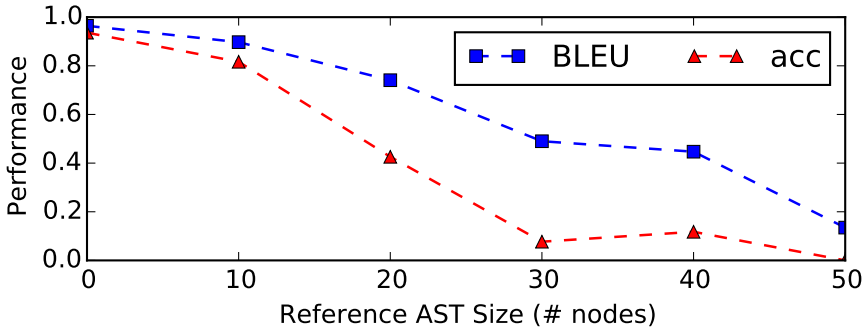


Figure 2.3: Performance w.r.t reference AST size on DJANGO

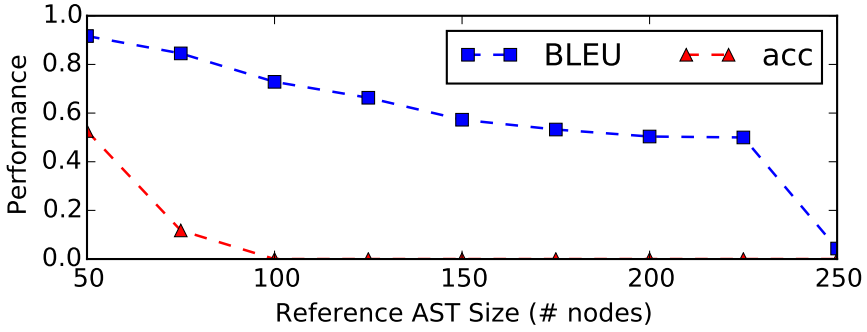


Figure 2.4: Performance w.r.t reference AST size on HS

grammar remaining unchanged. In contrast, DJANGO has a broader domain, and thus unary closure results in more productions in the grammar (237 for DJANGO vs. 100 for HS), increasing sparsity.

Performance by the size of AST We further investigate our model's performance w.r.t. the size of the gold-standard ASTs in Figs. 2.3 and 2.4. Not surprisingly, the performance drops when the size of the reference ASTs increases. Additionally, on the HS dataset, the BLEU score still remains at around 50 even when the size of ASTs grows to 200, indicating that our proposed syntax-driven approach is robust for long code segments.

Domain Specific Code Generation Although this is not the focus of our work, evaluation on IFTTT brings us closer to a standard semantic parsing setting, which helps to investigate similarities and differences between generation of more complicated general-purpose code and and more limited-domain simpler code. Tab. 2.4 shows the results, following the evaluation protocol in [14] for accuracies at both channel and full parse tree (channel + function) levels. Our full model performs on par with existing neural network-based methods, while outperforming other neural models in full tree accuracy (82.0%). This score is close to the best clas-

	Channel	Full Tree
Classical Methods		
posclass [100]	81.4	71.0
LR [14]	88.8	**82.5**
Neural Network Methods		
NMT	87.7	77.7
NN [14]	88.0	74.3
Seq2Tree [31]	89.7	78.4
Doubly-Recurrent NN [4]	**90.1**	78.2
Our system	90.0	82.0
– parent feed.	89.9	81.1
– frontier embed.	**90.1**	78.7

Table 2.4: Results on the noise-filtered Ifttt test set of ">3 agree with gold annotations" (averaged over three runs), our model performs competitively among neural models.

sical method (LR), which is based on a logistic regression model with rich hand-engineered features (e.g., brown clusters and paraphrase). Also note that the performance between NMT and other neural models is much closer compared with the results in Tab. 2.3. This suggests that general-purpose code generation is more challenging than the simpler Ifttt setting, and therefore modeling structural information is more helpful.

Case Studies We present output examples in Tab. 2.5. On HS, we observe that most of the time our model gives correct predictions by filling learned code templates from training data with arguments (e.g., cost) copied from input. This is in line with the findings in Ling et al. [68]. However, we do find interesting examples indicating that the model learns to generalize beyond trivial copying. For instance, the first example is one that our model predicted wrong — it generated code block A instead of the gold B (it also missed a function definition not shown here). However, we find that the block A actually conveys part of the input intent by destroying all, not some, of the minions. Since we are unable to find code block A in the training data, it is clear that the model has learned to generalize to some extent from multiple training card examples with similar semantics or structure.

The next two examples are from Django. The first one shows that the model learns the

input *<name> Brawl </name> <cost> 5 </cost> <desc> Destroy all minions except one (chosen randomly) </desc> <rarity> Epic </rarity> ...*

pred.
```
class Brawl(SpellCard):
  def __init__(self):
    super().__init__('Brawl', 5, CHARACTER_CLASS.WARRIOR, CARD_RARITY.EPIC)
  def use(self, player, game):
    super().use(player, game)
    targets = copy.copy(game.other_player.minions)
    targets.extend(player.minions)
    for minion in targets:                          A
      minion.die(self)
```

ref.
```
minions = copy.copy(player.minions)
minions.extend(game.other_player.minions)
if len(minions) > 1:                                B
  survivor = game.random_choice(minions)
  for minion in minions:
    if minion is not survivor: minion.die(self)
```

---

input *join app_config.path and string 'locale' into a file path, substitute it for localedir.*

pred.
```
localedir = os.path.join(app_config.path, 'locale') ✓
```

---

input *self.plural is an lambda function with an argument n, which returns result of boolean expression n not equal to integer 1*

pred.
```
self.plural = lambda n: len(n) ✗
```

ref.
```
self.plural = lambda n: int(n!=1)
```

Table 2.5: Predicted examples from HS (1st) and DJANGO. Copied contents (copy probability > 0.9) are highlighted.

usage of common API calls (e.g., `os.path.join`), and how to populate the arguments by copying from inputs. The second example illustrates the difficulty of generating code with complex nested structures like lambda functions, a scenario worth further investigation in future studies. More examples are attached in supplementary materials.

Error Analysis To understand the sources of errors and how good our evaluation metric (exact match) is, we randomly sampled and labeled 100 and 50 failed examples (with accuracy=0) from DJANGO and HS, respectively. We found that around 2% of these examples in the two datasets are actually semantically equivalent. These examples include: (1) using different parameter names when defining a function; (2) omitting (or adding) default values of parameters

in function calls. While the rarity of such examples suggests that our exact match metric is reasonable, more advanced evaluation metrics based on statistical code analysis are definitely intriguing future work.

For DJANGO, we found that 30% of failed cases were due to errors where the pointer network failed to appropriately copy a variable name into the correct position. 25% were because the generated code only partially implemented the required functionality. 10% and 5% of errors were due to malformed English inputs and pre-processing errors, respectively. The remaining 30% of examples were errors stemming from multiple sources, or errors that could not be easily categorized into the above. For HS, we found that all failed card examples were due to partial implementation errors, such as the one shown in Table 2.5.

2.5 Related Work

Code Generation and Analysis Most works on code generation focus on generating code for domain specific languages (DSLs) [58, 76, 104], with neural network-based approaches recently explored [10, 70, 92]. For general-purpose code generation, besides the general framework of Ling et al. [68], existing methods often use language and task-specific rules and strategies [61, 102]. A similar line is to use NL queries for code retrieval [3, 124]. The reverse task of generating NL summaries from source code has also been explored [46, 90]. Finally, our work falls into the broad field of probabilistic modeling of source code [75, 88]. Our approach of factoring an AST using probabilistic models is closely related to Allamanis et al. [3], which uses a factorized model to measure the semantic relatedness between NL and ASTs for code retrieval, while our model tackles the more challenging generation task.

Semantic Parsing Our work is related to the general topic of semantic parsing, which aims to transform NL descriptions into executable logical forms. The target logical forms can be viewed as DSLs. The parsing process is often guided by grammatical formalisms like combinatory categorical grammars [7, 59], dependency-based syntax [65, 93] or task-specific formalisms [27, 56, 78, 136]. Recently, there are efforts in designing neural network-based semantic parsers [31, 82, 85, 139]. Several approaches have be proposed to utilize grammar knowledge in a neural parser, such as augmenting the training data by generating examples guided by the grammar [48, 54]. Liang et al. [63] used a neural decoder which constrains the space of next valid tokens in the query language for question answering. Finally, the structured prediction approach proposed by Xiao et al. [129] is closely related to our model in using the underlying

grammar as prior knowledge to constrain the generation process of derivation trees, while our method is based on a unified grammar model which jointly captures production rule application and terminal symbol generation, and scales to general purpose code generation tasks.

Chapter 3

Generalized Parsing Framework (Completed)

The previous chapter introduced a syntax-driven neural parsing model for general-purpose code generation, and demonstrated its effectiveness in generating open-domain programs with complex grammars. In this chapter, we generalize the approach and build a semantic parsing framework TRANX, which is applicable to various parsing tasks with different logical formalisms of meaning representations. TRANX provides a convenient interface for users to quickly adapt the parsing model in Chapter 2 to the domain at hand, with the help of a programmable transition system based on the abstract syntax description language. This work is presented in:

- Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of EMNLP Demonstration Track*, 2018

3.1 Overview

Depending on the task considered, meaning representations in semantic parsing could 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 [148] or the abstract meaning representations [12]. Alternatively, for more task-driven approaches to semantic parsing, it is common for meaning representations to represent executable programs such as SQL queries [156], robotic commands [5], smart phone instructions [100], and even general-purpose programming languages like Python [101, 137] and Java [68].

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 [131, 156]. To alleviate this issue, there have been recent efforts in neural semantic parsing with general-purpose grammar models [32, 129]. Yin and Neubig [137] 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. [101] propose the abstract syntax networks (ASNs), where domain-specific MRs are represented by abstract syntax trees (ASTs, Figure 3.2 *Left*) specified under the abstract syntax description language (ASDL) framework [122]. 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 synta**X** parser for semantic parsing and code generation. TRANX is designed with the following principles in mind:

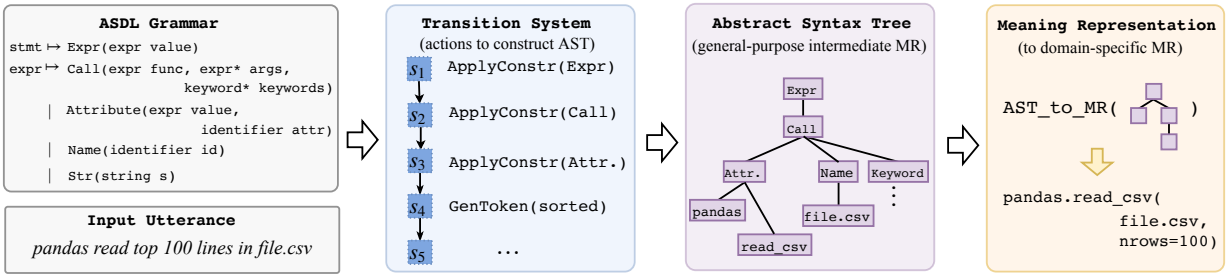


Figure 3.1: Workflow of TRANX

- **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 domain grammars.
- **Extensibility** TRANX uses a simple transition system to parse NL utterances into tree-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).

3.2 Methodology

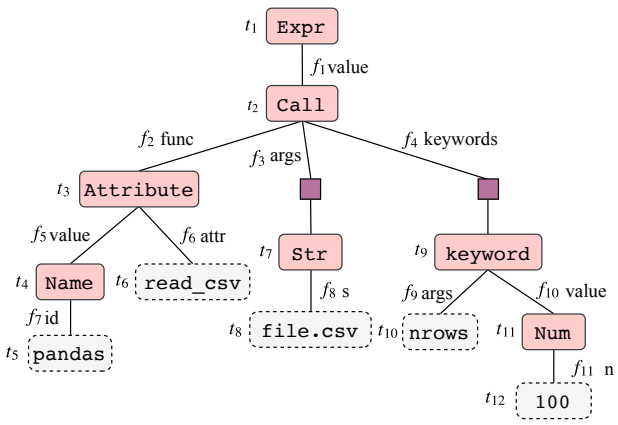
Given an NL utterance, TRANX parses the utterance into a formal meaning representation, typically represented as λ-calculus logical forms, domain-specific, or general-purpose programming languages (e.g., Python). In the following description we use Python code generation as a running example, where a programmer's natural language intents are mapped to Python source code. Figure 3.1 depicts the workflow of TRANX. We will present more use cases of TRANX in § 3.3.

The core of TRANX is a transition system. Given an input NL utterance x, TRANX employs the transition system to map the utterance x into an AST $\boldsymbol{z}$ using a series of tree-construction actions (§ 3.2.2). 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 (§ 3.2.1). 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 $p(\boldsymbol{z}|x)$, parameterized by a neural network, to score each hypothesis AST (§ 3.2.3).

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 Figure 3.1 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 Figure 3.1 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



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 3.2: ***Left*** The ASDL AST for the target Python code in Figure 3.1. 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.

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 cardinality in the ASDL grammar in Figure 3.1 is instantiated with one `Name` constructor, while the `args` field with sequential cardinality have multiple child constructors.

3.2.2 Transition System

Inspired by Yin and Neubig [137] (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. Figure 3.2 *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 Figure 3.2 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.

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

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

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

Following YN17, we parameterize the transition-based parser $p(\boldsymbol{z}|x)$ 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 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 embedding 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. [73]

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

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 (§ 3.3).

Action Probabilities The probability of an ApplyConstr$[c]$ action with embedding $\mathbf{a}_c$ is[1]

$$p(a_t = \textsc{ApplyConstr}[c]|a_{<t}, x) = \text{softmax}(\mathbf{a}_c^\intercal \mathbf{W} \tilde{\mathbf{s}}_t) \tag{3.1}$$

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 Figure 3.1) in x to be directly copied to the derivation. Specifically, the action probability is defined to be the marginal probability

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

[1] Reduce is treated as a special ApplyConstr action.

```
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 3.3: The λ-calculus ASDL grammar for GEO and ATIS, defined in Rabinovich et al. [101]

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 (3.1). The copy probability of copying the i-th word in x is defined using a pointer network [117]

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

3.3 Experiments

3.3.1 Datasets

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

```
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 3.4: The ASDL grammar for WikiSQL

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 [31]. We use the ASDL grammar defined in Rabinovich et al. [101], as listed in Figure 3.3.

Code Generation

We evaluate Tranx on both general-purpose (Python, Django) and domain-specific (SQL, WikiSQL) code generation tasks. The Django dataset [90] 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 [156] 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 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 Table 3.3, 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

Methods	GEO	ATIS
ZH15 [155]	88.9	84.2
ZC07 [149]	89.0	84.6
WKZ14 [118]	**90.4**	**91.3**
Neural Network-based Models		
SEQ2TREE [31]	87.1	84.6
ASN [101]	87.1	85.9
TRANX	88.2	86.2

Table 3.1: Semantic parsing accuracies on GEO and ATIS

Methods	ACC.
NMT [86]	45.1
LPN [68]	62.3
YN17 [137]	71.6
TRANX	**73.7**

Table 3.2: Code generation accuracies on DJANGO

define a simple ASDL grammar following the syntax of SQL (Figure 3.4). 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.

3.3.2 Results

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

Semantic Parsing Table 3.1 lists the results for semantic parsing tasks. We test TRANX with two configurations, with or without parent feeding (§ 3.2.3). Our system outperforms existing neural network-based approaches. This demonstrates the effectiveness of TRANX in closed-

Methods	Acc_{EM}	Acc_{EX}
Seq2SQL [156]	48.3	59.4
SQLNet [131]	–	68.0
PT-MAML [44]	62.8	68.0
TypeSQL [143]	–	**73.5**
TRANX	**62.9**	71.7
PointSQL [120]†	61.5	66.8
TypeSQL+TC [143]†	–	**82.6**
STAMP [114]†	60.7	74.4
STAMP+RL [114]†	61.0	74.6
TRANX	**68.4**	78.6

Table 3.3: Exact match (EM) and execution (EX) accuracies on WIKISQL. †Methods that use the contents of input tables.

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 Table 3.2 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).

Table 3.3 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 Table 3.3. 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 Table 3.3, 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.

Part II

Generalized Data Understanding Models for Neural Semantic Parsing

Chapter 4

Pretraining for Structured Data Understanding (Completed)

The systems presented in previous chapters concern generalization of the decoding models for handling syntactically diverse meaning representations, where the encoding model is assumed as a generic sequence-encoding network over input utterances. In many applications, however, the semantic parser needs to encode both user-issued utterances as well as domain-specific knowledge schemas necessary to understand the task. A representative example is semantic parsing over databases, which requires the parser to understand the structured information presented in database tables. To this end, we develop TaBert, a pre-trained Transformer model for learning joint representations of NL utterances and structured schemas of tabular data. This work will appear in:

- Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Annual Conference of the Association for Computational Linguistics (ACL)*, July 2020

4.1 Overview

Recent years have witnessed a rapid advance in the ability to understand and answer questions about free-form natural language (NL) text [103], largely due to large-scale, pretrained language models (LMs) like BERT [30]. These models allow us to capture the syntax and semantics of text via representations learned in a unsupervised manner, before fine-tuning the model to downstream tasks [35, 72, 77, 79, 94, 133]. It is also relatively easy to apply such pretrained LMs

to comprehension tasks that are modeled as text span selection problems, where the boundary of an answer span can be predicted using a simple classifier on top of the LM [49].

However, it is less clear how one could pretrain and fine-tune such models for other QA tasks that involve joint reasoning over both free-form NL text and *structured* data. One example task is semantic parsing for access to databases (DBs) [16, 136, 147], the task of transducing an NL utterance (*e.g.*,, *"Which country has the largest GDP?"*) into a structured query over DB tables (*e.g.*,, SQL querying a database of economics). A key challenge in this scenario is understanding the structured schema of DB tables (*e.g.*,, the name, data type, and stored values of columns), and more importantly, the alignment between the input text and the schema (*e.g.*,, the token *"GDP"* refers to the `Gross Domestic Product` column), which is essential for inferring the correct DB query [15].

Neural semantic parsers tailored to this task therefore attempt to learn joint representations of NL utterances and the (semi-)structured schema of DB tables (*e.g.*,, representations of its columns or cell values, as in Bogin et al. [18], Krishnamurthy et al. [57], Wang et al. [119], *inter alia*). However, this unique setting poses several challenges in applying pretrained LMs. First, information stored in DB tables exhibit strong underlying structure, while existing LMs (*e.g.*,, BERT) are solely trained for encoding free-form text. Second, a DB table could potentially have a large number of rows, and naively encoding all of them using a resource-heavy LM is computationally intractable. Finally, unlike most text-based QA tasks (*e.g.*,, SQuAD) which could be formulated as a generic answer span selection problem and solved by a pretrained model with additional classification layers, semantic parsing is highly domain-specific, and the architecture of a neural parser is strongly coupled with the structure of its underlying DB (*e.g.*,, systems for SQL-based and other types of DBs use different encoder models). In fact, existing systems have attempted to leverage BERT, but each with their own domain-specific, in-house strategies to encode the structured information in DB [37, 45, 150], and importantly, without pretraining representations on structured data. These challenges call for development of general-purpose pretraining approaches tailored to learning representations for both NL utterances and structured DB tables.

In this chapter we present TaBert, a pretraining approach for joint understanding of NL text and (semi-)structured tabular data (§ 4.3). TaBert is built on top of BERT, and jointly learns contextual representations for utterances and the structured schema of DB tables (*e.g.*,, a vector for each utterance token and table column). Specifically, TaBert linearizes the structure of tables to be compatible with a Transformer-based BERT model. To cope with large tables, we propose *content snapshots*, a method to encode a subset of table content most relevant to the

input utterance. This strategy is further combined with a *vertical attention* mechanism to share information among cell representations in different rows (§ 4.3.1). To capture the association between tabular data and related NL text, TaBert is pretrained on a parallel corpus of 26 million tables and NL paragraphs (§ 4.3.2).

TaBert can be plugged into a neural semantic parser as a general-purpose encoder to compute representations for utterances and tables. Our key insight is that although semantic parsers are highly domain-specific, most systems rely on representations of input utterances and the table schemas to facilitate subsequent generation of DB queries, and these representations can be provided by TaBert, regardless of the domain of the parsing task.

We apply TaBert to two different semantic parsing paradigms: (1) a classical supervised learning setting on the Spider text-to-SQL dataset [145], where TaBert is fine-tuned together with a task-specific parser using parallel NL utterances and labeled DB queries (§ 4.4.1); (2) a challenging weakly-supervised learning benchmark WikiTableQuestions [93], where a system has to infer latent DB queries from its execution results (§ 4.4.2). We demonstrate TaBert is effective in both scenarios, showing that it is a drop-in replacement of a parser's original encoder for computing contextual representations of NL utterances and DB tables. Specifically, systems augmented with TaBert register state-of-the-art performance on WikiTableQuestions, while performing competitively on Spider (§ 4.5).

4.2 Background

Semantic Parsing over Tables Semantic parsing tackles the task of translating an NL utterance $\boldsymbol{u}$ into a formal meaning representation (MR) $\boldsymbol{z}$. Specifically, we focus on parsing utterances to access database tables, where $\boldsymbol{z}$ is a structured query (*e.g.*,, an SQL query) executable on a set of relational DB tables $\mathcal{T} = \{T_t\}$. A relational table T is a listing of N rows $\{R_i\}_{i=1}^{N}$ of data, with each row R_i consisting of M cells $\{s_{\langle i,j \rangle}\}_{j=1}^{M}$, one for each column c_j. Each cell $s_{\langle i,j \rangle}$ contains a list of tokens.

Depending on the underlying data representation schema used by the DB, a table could either be fully structured with strongly-typed and normalized contents (*e.g.*,, a table column named `distance` has a unit of `kilometers`, with all of its cell values, like *200*, bearing the same unit), as is commonly the case for SQL-based DBs (§ 4.4.1). Alternatively, it could be semi-structured with unnormalized, textual cell values (*e.g.*,, *200 km*, § 4.4.2). The query language could also take a variety of forms, from general-purpose DB access languages like SQL to domain-specific ones tailored to a particular task.

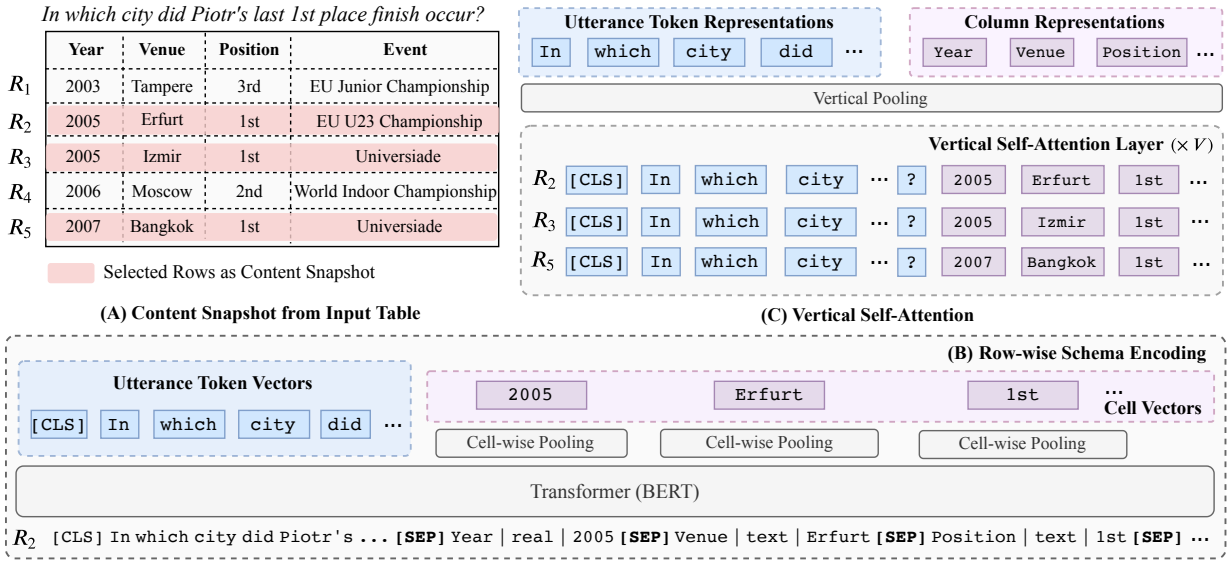


Figure 4.1: Overview of TaBert for learning representations of utterance and table schema with an example from WikiTableQuestions. **(A)** A content snapshot of the table is created based on the input NL utterance. **(B)** Each row in the snapshot is encoded by a Transformer (only R_2 is shown), producing row-wise encodings for utterance tokens and cells. **(C)** All row-wise encodings are aligned and processed by V vertical self-attention layers, generating utterance and column representations.

Given a utterance and the its associated tables, a neural semantic parser generates a DB query from the vector representations of the utterance tokens and the structured schema of tables. We refer schema as the set of columns in a table, and its representation as the list of vectors that represent its columns[1]. We will introduce how TaBert computes these representations in § 4.3.1.

Masked Language Models Given a sequence of NL tokens $\boldsymbol{x} = x_1, x_2, \ldots, x_n$, a masked language model (*e.g.*,, BERT) is an LM trained using the masked language modeling objective, which aims recover the original tokens in $\boldsymbol{x}$ from a "corrupted" context created by randomly masking out certain tokens in $\boldsymbol{x}$. Specifically, let $\boldsymbol{x}_m = \{x_{i_1}, \ldots, x_{i_m}\}$ be the subset of tokens in $\boldsymbol{x}$ selected to be masked out, and $\widetilde{\boldsymbol{x}}$ denote the masked sequence with tokens in $\boldsymbol{x}_m$ replaced by a `[MASK]` symbol. A masked LM defines a distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\widetilde{\boldsymbol{x}})$ over the target tokens $\boldsymbol{x}_m$ given the masked context $\widetilde{\boldsymbol{x}}$.

[1]Column representations for more complex schemas, *e.g.*,, those capturing inter-table dependency via primary and foreign keys, could be derived from these table-wise representations.

BERT parameterizes $p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\widetilde{\boldsymbol{x}})$ using a Transformer model. During the pretraining phase, BERT maximize $p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\widetilde{\boldsymbol{x}})$ on large-scale textual corpora. In the fine-tuning phase, the pre-trained model is used as an encoder to compute representations of input NL tokens, and its parameters are jointly tuned with other task-specific neural components.

4.3 TaBert: Learning Joint Representa- tions over Textual and Tabular Data

We first present how TaBert computes representations for NL utterances and tables schemas (§ 4.3.1), followed by the pretraining procedure (§ 4.3.2).

4.3.1 Computing Representations for NL Utterances and Table Schemas

Figure 4.1 presents a schematic overview of TaBert. Given a utterance $\boldsymbol{u}$ and a table T, TaBert first creates a content snapshot of T, which are sampled rows that summarize the most relevant information in T to the input utterance. The model then linearizes each row in the snapshot (and concatenate with the utterance) as input to a Transformer (*e.g.*,, BERT) model, which outputs row-wise encoding vectors of utterance tokens and cells. The encodings for all the rows in the snapshot are fed into a series of vertical self-attention layers, where a cell representation (or a utterance token representation) is computed by attending to vertically-aligned vectors of the same column (or the same NL token). Finally, representations for each utterance token and column are generated from a pooling layer.

Content Snapshot One major feature of TaBert is its use of the table *contents*, as opposed to just using the column names, in encoding the table schema. This is motivated by the fact that contents provide more detailed information about the semantics of a column than just the column's name, which might be ambiguous. For instance, the `Venue` column in Figure 4.1 which is used to answer the example question actually refers to *host cities*, and encoding the sampled cell values while creating its representation may help match the term "*city*" in the input utterance to this column.

However, a DB table could potentially have a large number of rows, with only few of them actually relevant to answering the input utterance. Encoding all of the contents using

[1]Example adapted from `stanford.io/38iZ8Pf`

a resource-heavy Transformer is both computationally intractable and likely not necessary. Thus, we instead use a *content snapshot* consisting of only a few rows that are most relevant to the input utterance, providing an efficient approach to calculate content-sensitive column representations from cell values.

We use a simple strategy to create content snapshots of K rows based on the relevance between the utterance and a row. For $K > 1$, we select the top-K rows in the input table that have the highest n-gram overlap ratio with the utterance. For $K = 1$, to include in the snapshot as much information relevant to the utterance as possible, we create a synthetic row by selecting the cell values from each column that have the highest n-gram overlap with the utterance.

Row Linearization TaBert creates a linearized sequence for each row in the content snapshot as input to the Transformer model. Figure 4.1(B) depicts the linearization for R_2, which consists of a concatenation of the utterance, columns and their cell values. Specifically, each cell is represented by the name and data type[2] of the column, together with its actual value, separated by a vertical bar. As an example, the cell $s_{\langle 2,1 \rangle}$ valued 2005 in R_2 in Figure 4.1 is encoded as

$$\underbrace{\texttt{Year}}_{\text{Column Name}} \mid \underbrace{\texttt{real}}_{\text{Column Type}} \mid \underbrace{\texttt{2005}}_{\text{Cell Value}} \tag{4.1}$$

The linearization of a row is then formed by concatenating the above string encodings of all the cells, separated by the `[SEP]` symbol. We then prefix the row linearization with utterance tokens as input sequence to the Transformer.

Existing works have applied different linearization strategies to encode table with Transformer [24, 45], while our row approach is specifically designed for encoding content snapshots. We present in § 4.5 results with different linearization choices.

Vertical Self-Attention Mechanism The base Transformer model in TaBert outputs vector encodings of utterance and cell tokens for each row. These row-level vectors are computed separately and therefore independent of each other. To allow for information flow across cell representations of different rows, we propose vertical self-attention, a self-attention mechanism that operates over vertically aligned vectors from different rows.

[2] We use two data types, `text`, and `real` for numbers, predicted by majority voting over the NER labels of cell tokens.

As in Figure 4.1(C), TaBert has V stacked vertical-level self-attention layers. To generate aligned inputs for vertical attention, we first compute a fixed-length initial vector for each cell at position $\langle i, j \rangle$, which is given by mean-pooling over the sequence of the Transformer's output vectors that correspond to its variable-length linearization as in E.q. (4.1). Next, the sequence of word vectors for the NL utterance (from the base Transformer model) are concatenated with the cell vectors as initial inputs to the vertical attention layer.

Each vertical attention layer has the same parameterization as the Transformer layer in Vaswani et al. [116], but operates on vertically aligned elements, *i.e.*,, utterance and cell vectors that correspond to the same question token and column, resp. This vertical self-attention mechanism enables the model to aggregate information from different rows in the content snapshot, allowing TaBert to capture cross-row dependencies on cell values.

Utterance and Column Representations A representation $\mathbf{c}_j$ is computed for each column c_j by mean-pooling over its vertically aligned cell vectors, $\{\mathbf{s}_{\langle i,j \rangle} : R_i \text{ in content snapshot}\}$, from the last vertical layer. A representation for each utterance token, $\mathbf{x}_j$, is computed similarly over the vertically aligned token vectors. These representations will be used by downstream neural semantic parsers. TaBert also outputs an optional fixed-length table representation $\mathbf{T}$ using the representation of the prefixed `[CLS]` symbol, which is useful for parsers that operates on multiple DB tables.

4.3.2 Pretraining Procedure

Training Data Since there is no large-scale, high-quality parallel corpus of NL text and structured tables, we instead use semi-structured tables that commonly exist on the Web as a surrogate data source. Specifically, we collect tables and their surrounding NL text from English Wikipedia and the WDC WebTable Corpus [60], a large-scale table collection from CommonCrawl. The raw data is extremely noisy, and we apply aggressive cleaning heuristics to filter out invalid examples (*e.g.*,, examples with HTML snippets). We will include more details of the data preprocessing in the final thesis. The pre-processed corpus contains 26.6 million parallel examples of tables and NL sentences. We perform sub-tokenization using the Wordpiece tokenizer shipped with BERT.

Unsupervised Learning Objectives We apply different objectives for learning representations of the NL context and structured tables. For NL contexts, we use the standard Masked

Language Modeling (MLM) objective [30], with a masking rate of 15% sub-tokens in an NL context.

For learning column representations, we design objectives motivated by the intuition that a column representation should contain both the general information of the column (*e.g.*,, its name and data type), and representative cell values relevant to the NL context. We use two objectives to capture such intuition. First, a **Masked Column Prediction (MCP)** objective encourages the model to recover the names and data types of masked columns. Specifically, we randomly select 20% of the columns in an input table, masking their names and data types in each row linearization (*e.g.*,, if the column `Year` in Figure 4.1 is selected, the tokens `Year` and `real` in E.q. (4.1) will be masked). Given the column representation $\mathbf{c}_j$, TaBert is trained to predict the bag of masked tokens from $\mathbf{c}_j$ using a multi-label classification objective. Intuitively, MCP encourages the model to recover column information from its contexts.

Next, we use an auxiliary **Cell Value Recovery (CVR)** objective to ensure information of representative cell values in content snapshots is retained after additional layers of vertical self-attention. Specifically, for each masked column c_j in the above MCP objective, CVR predicts the original tokens of each cell $s_{\langle i,j \rangle}$ (of c_j) in the content snapshot conditioned on its cell vector $\mathbf{s}_{\langle i,j \rangle}$[3]. For instance, for the example cell $s_{\langle 2,1 \rangle}$ in E.q. (4.1), we predict its value 2005 from $\mathbf{s}_{\langle 2,1 \rangle}$. Since a cell could have multiple value tokens, we apply the span-based prediction objective [49]. Specifically, to predict a cell token $s_{\langle i,j \rangle_k} \in s_{\langle i,j \rangle}$, its positional embedding $\mathbf{e}_k$ and the cell representation $\mathbf{s}_{\langle i,j \rangle}$ is fed into a two-layer network $f(\cdot)$ with GeLU activations [41]. The output of $f(\cdot)$ is then used to predict the original value token $s_{\langle i,j \rangle_k}$ from a softmax layer.

4.4 Example Application: Semantic Parsing over Tables

We apply TaBert for representation learning on two semantic parsing paradigms, a classical supervised text-to-SQL task over structured DBs (§ 4.4.1), and a weakly supervised parsing problem on semi-structured Web tables (§ 4.4.2).

4.4.1 Supervised Semantic Parsing

Benchmark Dataset Supervised learning is the typical scenario of learning a parser using parallel data of utterances and queries. We use Spider [145], a text-to-SQL dataset with 10,181

[3]The cell value tokens are not masked in the input sequence, since predicting masked cell values is challenging even with the presence of its surrounding context.

examples across 200 DBs. Each example consists of a utterance (*e.g.*,, *"What is the total number of languages used in Aruba?"*), a DB with one or more tables, and an annotated SQL query, which typically involves joining multiple tables to get the answer (*e.g.*,, `SELECT COUNT(*) FROM Country JOIN Lang ON Country.Code = Lang.CountryCode WHERE Name = 'Aruba'`).

Base Semantic Parser We aim to show TaBert could help improve upon an already strong parser. Unfortunately, as the time of writing, none of the top systems on Spider was publicly available. To establish a reasonable testbed, we developed our in-house system based on TranX [138], an open-source general-purpose semantic parser. TranX translates an NL utterance into an intermediate meaning representation guided by a user-defined grammar. The generated intermediate MR could then be deterministically converted to domain-specific query languages (*e.g.*,, SQL).

We use TaBert as encoder of utterances and table schemas. Specifically, for a given utterance $\boldsymbol{u}$ and a DB with a set of tables $\mathcal{T} = \{T_t\}$, we first pair $\boldsymbol{u}$ with each table T_t in $\mathcal{T}$ as inputs to TaBert, which generates $|\mathcal{T}|$ sets of table-specific representations of utterances and columns. At each time step, an LSTM decoder performs hierarchical attention [66] over the list of table-specific representations, constructing an MR based on the predefined grammar. Following the IRNet model [37] which achieved the best performance on Spider, we use SemQL, a simplified version of the SQL, as the underlying grammar. We will include more details of the system in the final thesis.

4.4.2 Weakly Supervised Semantic Parsing

Benchmark Dataset Weakly supervised semantic parsing considers the reinforcement learning task of inferring the correct query from its execution results (*i.e.*,, whether the answer is correct). Compared to supervised learning, weakly supervised parsing is significantly more challenging, as the parser does not have access to the labeled query, and has to explore the exponentially large search space of possible queries guided by the noisy binary reward signal of execution results.

WikiTableQuestions [93] is a popular environment for weakly supervised semantic parsing, which has 22,033 utterances and 2,108 semi-structured Web tables from Wikipedia. Compared to Spider, examples in this dataset does not involve joining multiple tables, but typically require compositional, multi-hop reasoning over a series of entries in the given table (*e.g.*,, to answer the example in Figure 4.1 the parser need to reason over the row set $\{R_2, R_3, R_5\}$,

locating the `Venue` field with the largest value of `Year`).

Base Semantic Parser MAPO [64] is a strong system for weakly supervised semantic parsing. It improves the sample efficiency of the REINFORCE algorithm by biasing the exploration of queries towards the high-rewarding ones already discovered by the model. MAPO uses a domain-specific query language tailored to answering compositional questions on single tables, and its utterances and column representations are derived from an LSTM encoder, which we replaced with our TaBert model. We will include implementation details of the system in the final thesis.

4.5 Experiments

In this section we evaluate TaBert on downstream tasks of semantic parsing to DB tables.

Pretraining Configuration We train two variants of the model, $\text{TaBert}_{\texttt{Base}}$ and $\text{TaBert}_{\texttt{Large}}$, with the underlying Transformer model initialized with the uncased versions of $\text{Bert}_{\texttt{Base}}$ and $\text{Bert}_{\texttt{Large}}$, resp[4]. During pretraining, for each table and its associated NL context in the corpus, we create a series of training instances of paired NL sentences (as synthetically generated utterances) and tables (as content snapshots) by (1) sliding a (non-overlapping) context window of sentences with a maximum length of 128 tokens, and (2) using the NL tokens in the window as the utterance, and paring it with randomly sampled rows from the table as content snapshots. TaBert is implemented in PyTorch using distributed training.

Comparing Models We mainly present results for two variants of TaBert by varying the size of content snapshots K. $\textbf{TaBert}(\mathbf{K = 3})$ uses three rows from input tables as content snapshots and three vertical self-attention layers. $\textbf{TaBert}(\mathbf{K = 1})$ uses one synthetically generated row as the content snapshot as described in § 4.3.1. Since this model does not have multi-row input, we do not use additional vertical attention layers (and the cell value recovery learning objective). Its column representation $\mathbf{c}_j$ is defined by mean-pooling over the Transformer's output encodings that correspond to the column name (*e.g.*,, the representation for the `Year` column in Figure 4.1 is derived from the vector of the `Year` token in E.q. (4.1)). We

[4]We also attempted to train TaBert on our collected corpus from scratch without initialization from BERT, but with inferior results, potentially due to the average lower quality of web-scraped tables compared to purely textual corpora. We leave improving the quality of training data as future work.

find this strategy gives better results compared with using the cell representation $\mathbf{s}_j$ as $\mathbf{c}_j$. We also compare with **BERT** using the same row linearization and content snapshot approach as TABERT(K = 1), which reduces to a TABERT(K = 1) model without pretraining on tabular corpora.

Evaluation Metrics As standard, we report execution accuracy on WIKITABLEQUESTIONS and exact-match accuracy of DB queries on SPIDER.

4.5.1 Main Results

Previous Systems on WikiTableQuestions		
Model	DEV	TEST
Pasupat and Liang [93]	37.0	37.1
Neelakantan et al. [85]	34.1	34.2
Ensemble 15 Models	37.5	37.7
Zhang et al. [152]	40.6	43.7
Dasigi et al. [29]	43.1	44.3
Agarwal et al. [1]	43.2	44.1
Ensemble 10 Models	–	46.9

Our System based on MAPO [64]				
	DEV	Best	TEST	Best
Base Parser	42.3 ±0.3	42.7	43.1 ±0.5	43.8
w/ $\text{BERT}_{\text{Base}}$ (K = 1)	49.6 ±0.5	50.4	49.4 ±0.5	49.2
− content snapshot	49.1 ±0.6	50.0	48.8 ±0.9	50.2
w/ $\text{TABERT}_{\text{Base}}$ (K = 1)	51.2 ±0.5	51.6	50.4 ±0.5	51.2
− content snapshot	49.9 ±0.4	50.3	49.4 ±0.4	50.0
w/ $\text{TABERT}_{\text{Base}}$ (K = 3)	51.6 ±0.5	52.4	51.4 ±0.3	51.3
w/ $\text{BERT}_{\text{Large}}$ (K = 1)	50.3 ±0.4	50.8	49.6 ±0.5	50.1
w/ $\text{TABERT}_{\text{Large}}$ (K = 1)	51.6 ±1.1	52.7	51.2 ±0.9	51.5
w/ $\text{TABERT}_{\text{Large}}$ (K = 3)	**52.2** ±0.7	**53.0**	**51.8** ±0.6	**52.3**

Table 4.1: Execution accuracies on WIKITABLEQUESTIONS. Models are evaluated with 10 random runs. We report mean, standard deviation and the best results. TEST↦BEST refers to the result from the run with the best performance on DEV. set.

Top-ranked Systems on Spider Leaderboard	
Model	Dev. Acc.
Global–GNN [17]	52.7
EditSQL + Bert [150]	57.6
RatSQL [119]	60.9
IRNet + Bert [37]	60.3
+ Memory + Coarse-to-Fine	61.9
IRNet V2 + Bert	63.9
RyanSQL + Bert (Anonymous)	**66.6**
Our System based on TranX [138]	
	Mean / Best
$w/$ $\text{Bert}_{\texttt{Base}}$ (K = 1)	61.8 ±0.8 / 62.4
− content snapshot	59.6 ±0.7 / 60.3
$w/$ $\text{TaBert}_{\texttt{Base}}$ (K = 1)	63.3 ±0.6 / 64.2
− content snapshot	60.4 ±1.3 / 61.8
$w/$ $\text{TaBert}_{\texttt{Base}}$ (K = 3)	63.3 ±0.7 / 64.1
$w/$ $\text{Bert}_{\texttt{Large}}$ (K = 1)	61.3 ±1.2 / 62.9
$w/$ $\text{TaBert}_{\texttt{Large}}$ (K = 1)	64.0 ±0.4 / 64.4
$w/$ $\text{TaBert}_{\texttt{Large}}$ (K = 3)	**64.5** ±0.6 / **65.2**

Table 4.2: Exact match accuracies on the public development set of Spider. Models are evaluated with 5 random runs.

Table 4.1 and Table 4.2 summarize the end-to-end evaluation results on WikiTableQuestions and Spider, respectively. First, comparing with existing strong semantic parsing systems, we found our parsers with TaBert as the utterance and table encoder perform competitively. On the test set of WikiTableQuestions, MAPO augmented with a $\text{TaBert}_{\texttt{Large}}$ model with three-row content snapshots, $\text{TaBert}_{\texttt{Large}}(\text{K} = 3)$, registers a single-model exact-match accuracy of 52.3%, surpassing the previously best ensemble system (46.9%) from Agarwal et al. [1] by 5.4% absolute.

On Spider, our semantic parser based on TranX and SemQL (§ 4.4.1) is conceptually similar to the base version of IRNet as both systems use the SemQL grammar, while our system has a simpler decoder. Interestingly, we observe that its performance with $\text{Bert}_{\texttt{Base}}$ (61.8%) matches the full BERT-augmented IRNet model with a stronger decoder using augmented memory and coarse-to-fine decoding (61.9%). This confirms that our base parser is an effective baseline.

Augmented with representations produced by TaBERT$_{\texttt{Large}}$(K = 3), our parser achieves up to 65.2% exact-match accuracy, a 2.8% increase over the base model using BERT$_{\texttt{Base}}$. Note that while other competitive systems on the leaderboard use BERT with more sophisticated semantic parsing models, our best Dev. result is already close to the score registered by the best submission (RyanSQL+Bert). This suggests that if they instead used TaBert as a featurizer, they would see further gains [5].

Comparing semantic parsers augmented with TaBert and Bert. We found TaBert is more effective across the board. Overall, the results on the two benchmarks demonstrate that pre-training on aligned textual and tabular data is necessary for joint understanding of NL utterances and tables, and TaBert works well with both structured (Spider) and semi-structured DBs (WikiTableQuestions), and agnostic of the task-specific structures of semantic parsers.

Effect of Content Snapshots In this chapter we propose using content snapshots to capture the information in input DB tables that is most relevant to answering the NL utterance. We therefore study the effectiveness of including content snapshots when generating schema representations. We include in Table 4.1 and Table 4.2 results of models without using content in row linearization ("−content snapshot"). Under this setting a column is represented as "`Column Name | Type`" without cell values (*c.f.*, E.q. (4.1)). We find that content snapshots are helpful for both Bert and TaBert, especially for TaBert. As discussed in § 4.3.1, encoding sampled values from columns in learning their representations helps the model infer alignments between entity and relational phrases in the utterance and the corresponding column. This is particularly helpful for identifying relevant columns from a DB table that is mentioned in the input utterance. As an example, empirically we observe on Spider our semantic parser with TaBert$_{\texttt{Base}}$ using just one row of content snapshots (K = 1) registers a higher accuracy of selecting the correct columns when generating SQL queries (*e.g.*,, columns in `SELECT` and `WHERE` clauses), compared to the TaBert$_{\texttt{Base}}$ model without encoding content information (87.4% v.s. 86.4%).

Additionally, comparing TaBert using one synthetic row (K = 1) and three rows from input tables (K = 3) as content snapshots, the latter generally performs better. Intuitively, encoding more table contents relevant to the input utterance could potentially help answer questions that involve reasoning over information across multiple rows in the table. Table 4.3 shows such an example, and to answer this question a parser need to subtract the values of `Year` in the rows for *"The Watermelon"* and *"The Bacchae"*. TaBert$_{\texttt{Large}}$ (K = 3) is able to capture

[5] Unfortunately, these systems have not released source code at this time, so we cannot perform these experiments.

$\boldsymbol{u}$: How many years before was the film Bacchae out before the Watermelon?

Input to TaBERT$_{\texttt{Large}}$ (K = 3) ▷ *Content Snapshot with Three Rows*

Film	Year	Function	Notes
The Bacchae	2002	Producer	Screen adaptation of...
The Trojan Women	2004	Producer/Actress	Documutary film...
The Watermelon	2008	Producer	Oddball romantic comedy...

Input to TaBERT$_{\texttt{Large}}$ (K = 1) ▷ *Content Snapshot with One Synthetic Row*

Film	Year	Function	Notes
The Watermelon	2013	Producer	Screen adaptation of...

Table 4.3: Content snapshots generated by two models for a WikiTableQuestions Dev. example. Matched tokens between the question and content snapshots are highlighted.

the two target rows in its content snapshot and generates the correct DB query, while the TaBERT$_{\texttt{Large}}$(K = 1) model with only one row as content snapshot fails to answer this example.

Effect of Row Linearization TaBERT uses row linearization to represent a table row as sequential input to Transformer. Table 4.4 (*Upper-Half*) presents results using various linearization methods. We find adding type information and content snapshots improves performance, as they provide more hints about the meaning of a column.

We also compare with existing linearization methods in literature using a TaBERT$_{\texttt{Base}}$ model (Table 4.4 *Lower-Half*). Hwang et al. [45] uses BERT to encode concatenated column names to learn column representations. In line with our previous discussion on the effectiveness content snapshots, this simple strategy without encoding cell contents underperforms (although with TaBERT$_{\texttt{Base}}$ pretrained on our tabular corpus the results become slightly better). Additionally, we remark that linearizing table contents has also be applied to other BERT-based tabular reasoning tasks. For instance, Chen et al. [24] propose a "natural" linearization approach for checking if an NL statement entails the factual information listed in a table using a binary classifier with representations from Bert, where a table is linearized by concatenating the semicolon-separated cell linearization for all rows. Each cell is represented by a phrase "`column name is cell value`". For completeness, we also tested this cell linearization approach, and find Bert$_{\texttt{Base}}$ get improved results. We leave pretraining TaBERT with this linearization strategy as promising future work.

Cell Linearization Template	WikiQ.	Spider
Pretrained TaBert$_{\text{Base}}$ Models (K = 1)		
Column Name	49.6 ±0.4	60.0 ±1.1
Column Name | Type† (−content snap.)	49.9 ±0.4	60.4 ±1.3
Column Name | Type | Cell Value†	51.2 ±0.5	63.3 ±0.6
Bert$_{\text{Base}}$ Models		
Column Name [45]	49.0 ±0.4	58.6 ±0.3
Column Name is Cell Value (Chen19)	50.2 ±0.4	63.1 ±0.7

Table 4.4: Performance of pretrained TaBert$_{\text{Base}}$ models and Bert$_{\text{Base}}$ on the Dev. sets with different linearization methods. Slot names are underlined. †Results copied from Table 4.1 and Table 4.2.

Impact of Pretraining Objectives TaBert uses two objectives (§ 4.3.2), a masked column prediction (MCP) and a cell value recovery (CVR) objective, to learn column representations that could capture both the general information of the column (via MCP) and its representative cell values related to the utterance (via CVR). Table 4.5 shows ablation results of pretraining TaBert with different objectives. We find TaBert trained with both MCP and the auxiliary CVR objectives gets a slight advantage, suggesting CVR could potentially lead to more representative column representations with additional cell information.

4.6 Related Works

Semantic Parsing over Tables Tables are important media of world knowledge. Semantic parsers have been adapted to operate over structured DB tables [32, 106, 121, 123, 131, 144], and open-domain, semi-structured Web tables [85, 93, 113]. To improve representations of utterances and tables for neural semantic parsing, existing systems have applied pretrained word embeddings (*e.g.*,., GloVe, as in Liang et al. [64], Sun et al. [114], Yu et al. [143], Zhong et al. [156]), and BERT-family models for learning joint contextual representations of utterances and tables, but with domain-specific approaches to encode the structured information in tables [37, 40, 45, 150]. TaBert advances this line of research by presenting a general-purpose, pretrained encoder over parallel corpora of Web tables and NL context. Another relevant direction is to augment representations of columns from an individual table with global information of

Learning Objective	WikiQ.	Spider
MCP only	51.6 $\pm$0.7	62.6 $\pm$0.7
MCP + CVR	51.6 $\pm$0.5	63.3 $\pm$0.7

Table 4.5: Performance of pretrained TaBert$_{\texttt{Base}}$(K = 3) on Dev. sets with different pretraining objectives.

its linked tables defined by the DB schema [17, 119]. TaBert could also potentially improve performance of these systems with improved table-level representations.

Knowledge-enhanced Pretraining Recent work has incorporated structured information from knowledge bases (KBs) into training contextual word representations, either by fusing vector representations of entities and relations on KBs into word representations of LMs [95, 153, 154], or by encouraging the LM to recover KB entities and relations from text [71, 115]. TaBert is broadly relevant to this line in that it also exposes an LM with structured data (*i.e.*,, tables), while aiming to learn joint representations for both textual and structured tabular data.

Chapter 5

Schema Understanding without Logical Forms (Proposed Work)

5.1 Overview

The previous chapter describes a pre-training approach for understanding the schema of database tables for semantic parsing. This method still requires supervised fine-tuning using annotated logical forms on a target database. In this chapter, we attempt to relax this requirement with a schema understanding model that grounds NL utterances to logical forms without the supervision of annotated examples.

Unsupervised schema understanding is motivated by two observations. First, there has been a proliferation in developing general-purpose question understanding formalisms that model the compositionality of operations involved in NL utterances without referring to the actual knowledge schema, which could be used as surrogate supervision to infer the grounded DB query. The recently proposed question-decomposition meaning representation (QDMR, [126]) is such an instance. Figure 5.1 shows the QDMR annotation of an example NL utterance. The QDMR shares a similar structure with the underlying SQL program, hence providing useful signals for the inference of the latter. Second, databases used in many applications tasks have relatively simple schema. For instance, the average number of tables for databases in Spider, a cross-domain semantic parsing benchmark, is only around 4. The limited scope of structured information presented in DB schemas could be used to prune the search space of allowable groundings of NL utterances. Figure 5.1 illustrates the possible groundings of arguments in the QDMR annotation of the example utterance. Once the grounding is disambiguated via a

u : *For each customer who has at least two orders, find the customer name and number of orders made.*

QDMR Analysis	**Schema Grounding**
$R_1 = \textsc{Select}(\text{entity} = \text{customers})$	customers↦`Table:Customers`
$R_2 = \textsc{Project}(R_1, \text{relation} = \text{orders})$	orders↦`Table:Orders`
$R_3 = \textsc{Group}(\text{reduction} = \texttt{Count}, \text{values} = R_2, \text{keys} = R_1)$	–
$R_4 = \textsc{Comparative}(\text{op} = \texttt{LargerThanEqual}, \text{values} = R_3, \text{keys} = R_1, \text{number} = 2)$	–
$R_5 = \textsc{Project}(R_4, \text{relation} = \text{customer names})$	customer names↦`Column:Customers.Name`
$R_6 = \textsc{Group}(\text{reduction} = \texttt{Count}, \text{values} = R_2, \text{keys} = R_4)$	–
$R_7 = \textsc{Union}(R_5, R_6)$	–


```
z : SELECT Customers.Name , Count (*) FROM Customers
    JOIN Orders ON Customers.Id = Orders.CustomerId
    GROUP BY Customers.Name HAVING Count (*) >= 2
```

Figure 5.1: Example utterance from the Spider dataset and its QDMR annotations with schema grounding results.

probabilistic model, the corresponding SQL query could be deterministically inferred.

5.2 Proposed Model

There has been existing work on unsupervised semantic parsing without annotation of logical forms. Poon [97] proposed a system that uses the dependency parse of NL utterances as linguistic scaffold to induce target database queries. Specifically, given the dependency tree T of an utterance, the grounding of nodes on T to DB constants are modeled as latent variables. The system is trained to maximizing the marginal probability of observing the unlabeled utterance. To infer the logical form of a testing utterance, posterior inference is performed to compute the possible grounding of nodes on its dependency parse, which is then deterministically converted to an SQL query.

Different from [97], which induces DB queries from dependency trees, our model employs QDMRs as general-purpose semantic analysis of utterances. Compared with dependency trees, QDMRs, like the one shown in Figure 5.1, have better alignment with the underlying database schema. We plan to design a generative model to capture the latent grounding of operators and

arguments in a QDMR to the corresponding DB constants.

Research Challenge A challenging issue in this line is to ground utterances with complex semantics, which do not enjoy straightforward alignments with the database schema. Consider the utterance *"Find the customer who placed multiple orders."*, it is non-trivial to automatically infer that *"multiple"* denotes the operation that filters the records based on the number of orders, which corresponds to the QDMR operation $\textsc{Comparative}(\text{op} = \texttt{gt}, \text{values} = \cdot, \text{keys} = \cdot, \text{number} = 1)$ (*c.f.*, R_4 in Figure 5.1). While some of such complex cases could be covered by heuristic rules (*e.g.*, *"multiple"*$\mapsto \textsc{Comparative}(\text{op} = \texttt{gt}, \text{values} = \cdot, \text{keys} = \cdot, \text{number} = 1)$) as the grounding of those terms is domain-general, there are examples which require domain-specific knowledge to infer the correct grounding (*e.g.*, *"Who is the youngest player?"*). We plan to cover those cases using domain-dependent mapping rules.

5.3 Proposed Experiment

Data We plan to use Spider [145] as the major benchmark. Spider is a cross-domain text-to-SQL dataset covering 200 databases. We will also evaluate on Atis, a popular semantic parsing corpus of flight booking and airport information queries.

Baselines The previous system of Poon [97] is a straight-forward baseline. We also developed a rule-based system, which grounds arguments of QDMR operations to the database schema (*e.g.*, Figure 5.1) using heuristic n-gram matching. This heuristic system achieved 17% exact match accuracy on Spider.

Part III

Data Efficient Approaches for Neural Semantic Parsing

Chapter 6

Semi-supervised Learning (Completed)

So far we have discussed modeling techniques for developing generalized neural semantic parsers. This chapter studies another important procedure in the life-cycle of a semantic parser — inferring model parameters on training data. Systems introduced in previous chapters mostly rely on parallel training corpora of utterances and annotated meaning representations for supervised learning. However, these neural models tend to be data-hungry, requiring large amounts of parallel data for learning, while collecting such corpora requires costly manual annotation by domain experts (*e.g.*, professional programmers). While Chapter 5 attempts to mitigate this issue using unsupervised learning, such unsupervised systems still could not rival their supervised counterparts. In this chapter, we study a data-efficient semi-supervised learning approach, where a parser is trained on both limited amount of annotated data, together with extra unlabeled natural language utterances. We show systems trained using semi-supervised learning outperform purely supervised ones. This work first appears in:

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

6.1 Overview

Recent advances in semantic parsing research are largely attributed to the success of neural network models [31, 47, 68, 129, 156]. However, these models are also extremely *data hungry*: optimization of such models requires large amounts of training data of parallel NL utterances and manually annotated MRs, the creation of which can be expensive, cumbersome, and time-consuming. Therefore, the limited availability of parallel data has become the bottle-

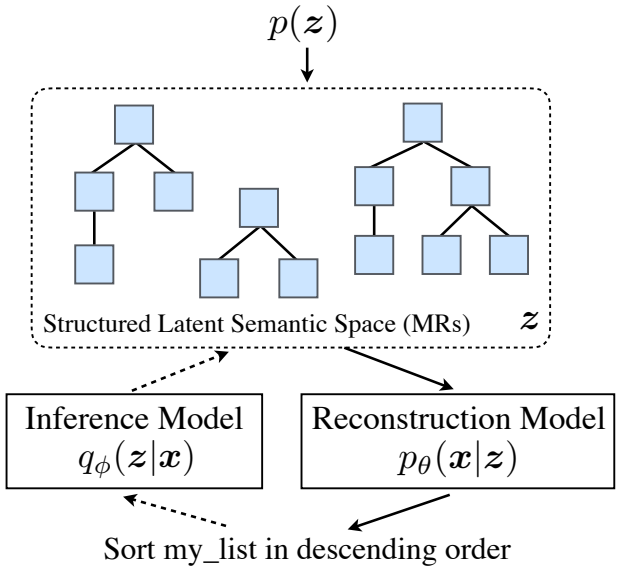


Figure 6.1: Graphical Representation of StructVAE

neck of existing, purely supervised-based models. These data requirements can be alleviated with *weakly-supervised* learning, where the denotations (*e.g.*, answers in question answering) of MRs (*e.g.*, logical form queries) are used as indirect supervision (Berant et al. [16], Clarke et al. [27], Liang et al. [65], *inter alia*), or *data-augmentation techniques* that automatically generate pseudo-parallel corpora using hand-crafted or induced grammars [48, 123].

In this work, we focus on *semi-supervised* learning, aiming to learn from both limited amounts of parallel NL-MR corpora, and *unlabeled* but readily-available NL utterances. We draw inspiration from recent success in applying variational auto-encoding (VAE) models in semi-supervised sequence-to-sequence learning [54, 80], and propose StructVAE — a principled deep generative approach for semi-supervised learning with tree-structured latent variables (Figure 6.1). StructVAE is based on a generative story where the surface NL utterances are generated from tree-structured latent MRs following the standard VAE architecture: (1) an off-the-shelf semantic parser functions as the *inference model*, parsing an observed NL utterance into latent meaning representations (§ 6.3.2); (2) a *reconstruction model* decodes the latent MR into the original observed utterance (§ 6.3.1). This formulation enables our model to perform both standard supervised learning by optimizing the inference model (*i.e.*, the parser) using parallel corpora, and unsupervised learning by maximizing the variational lower bound of the likelihood of the unlabeled utterances (§ 6.3.3).

In addition to these contributions to semi-supervised semantic parsing, StructVAE contributes to generative model research as a whole, providing a recipe for training VAEs with

structured latent variables. Such a structural latent space is contrast to existing VAE research using *flat* representations, such as continuous distributed representations [52], discrete symbols [80], or hybrids of the two [157].

We apply StructVAE to semantic parsing on the Atis domain and Python code generation. As an auxiliary contribution, we implement a transition-based semantic parser, which uses Abstract Syntax Trees (ASTs, § 6.3.2) as intermediate MRs and achieves strong results on the two tasks. We then apply this parser as the inference model for semi-supervised learning, and show that with extra unlabeled data, StructVAE outperforms its supervised counterpart. We also demonstrate that StructVAE is compatible with different structured latent representations, applying it to a simple sequence-to-sequence parser which uses λ-calculus logical forms as MRs.

6.2 Semi-supervised Semantic Parsing

In this section we introduce the objectives for semi-supervised semantic parsing, and present high-level intuition in applying VAEs for this task.

6.2.1 Supervised and Semi-supervised Training

Formally, semantic parsing is the task of mapping utterance x to a meaning representation $\boldsymbol{z}$. As noted above, there are many varieties of MRs that can be represented as either graph structures (*e.g.*, AMR) or tree structures (*e.g.*, λ-calculus and ASTs for programming languages). In this work we specifically focus on tree-structured MRs (see Figure 6.2 for a running example Python AST), although application of a similar framework to graph-structured representations is also feasible.

Traditionally, purely supervised semantic parsers train a probabilistic model $p_\phi(\boldsymbol{z}|x)$ using parallel data $\mathbb{L}$ of NL utterances and annotated MRs (*i.e.*, $\mathbb{L} = \{\langle x, \boldsymbol{z}\rangle\}$). As noted in the introduction, one major bottleneck in this approach is the lack of such parallel data. Hence, we turn to semi-supervised learning, where the model additionally has access to a relatively large amount of unlabeled NL utterances $\mathbb{U} = \{x\}$. Semi-supervised learning then aims to maximize the log-likelihood of examples in both $\mathbb{L}$ and $\mathbb{U}$:

$$\mathcal{J} = \underbrace{\sum_{\langle x,\boldsymbol{z}\rangle \in \mathbb{L}} \log p_\phi(\boldsymbol{z}|x)}_{\text{supervised obj. } \mathcal{J}_s} + \alpha \underbrace{\sum_{x \in \mathbb{U}} \log p(x)}_{\text{unsupervised obj. } \mathcal{J}_u} \tag{6.1}$$

The joint objective consists of two terms: (1) a supervised objective $\mathcal{J}_s$ that maximizes the conditional likelihood of annotated MRs, as in standard supervised training of semantic parsers; and (2) a unsupervised objective $\mathcal{J}_u$, which maximizes the marginal likelihood $p(x)$ of unlabeled NL utterances $\mathbb{U}$, controlled by a tuning parameter α. Intuitively, if the modeling of $p_\phi(\boldsymbol{z}|x)$ and $p(x)$ is coupled (*e.g.*, they share parameters), then optimizing the marginal likelihood $p(x)$ using the unsupervised objective $\mathcal{J}_u$ would help the learning of the semantic parser $p_\phi(\boldsymbol{z}|x)$ [158]. StructVAE uses the variational auto-encoding framework to jointly optimize $p_\phi(\boldsymbol{z}|x)$ and $p(x)$, as outlined in § 6.2.2 and detailed in § 6.3.

6.2.2 VAEs for Semi-supervised Learning

From Eq. (6.1), our semi-supervised model must be able to calculate the probability $p(x)$ of unlabeled NL utterances. To model $p(x)$, we use VAEs, which provide a principled framework for generative models using neural networks [52]. As shown in Figure 6.1, VAEs define a *generative story* (bold arrows in Figure 6.1, explained in § 6.3.1) to model $p(x)$, where a latent MR $\boldsymbol{z}$ is sampled from a prior, and then passed to the *reconstruction* model to decode into the surface utterance x. There is also an *inference* model $q_\phi(\boldsymbol{z}|x)$ that allows us to infer the most probable latent MR $\boldsymbol{z}$ given the input x (dashed arrows in Figure 6.1, explained in § 6.3.2). In our case, the inference process is equivalent to the task of semantic parsing if we set $q_\phi(\cdot) \triangleq p_\phi(\cdot)$. VAEs also provide a framework to compute an approximation of $p(x)$ using the inference and reconstruction models, allowing us to effectively optimize the unsupervised and supervised objectives in Eq. (6.1) in a joint fashion (Kingma et al. [53], explained in § 6.3.3).

6.3 StructVAE: VAEs with Tree-structured Latent Variables

6.3.1 Generative Story

StructVAE follows the standard VAE architecture, and defines a generative story that explains how an NL utterance is generated: a latent meaning representation $\boldsymbol{z}$ is sampled from a prior distribution $p(\boldsymbol{z})$ over MRs, which encodes the latent semantics of the utterance. A *reconstruction* model $p_\theta(x|\boldsymbol{z})$ then decodes the sampled MR $\boldsymbol{z}$ into the observed NL utterance x.

Both the prior $p(\boldsymbol{z})$ and the reconstruction model $p(x|\boldsymbol{z})$ takes tree-structured MRs as inputs. To model such inputs with rich internal structures, we follow Konstas et al. [55], and model the distribution over a sequential surface representation of $\boldsymbol{z}$, $\boldsymbol{z}^s$ instead. Specifically,

we have $p(\boldsymbol{z}) \triangleq p(\boldsymbol{z}^s)$ and $p_\theta(x|\boldsymbol{z}) \triangleq p_\theta(x|\boldsymbol{z}^s)$[1]. For code generation, $\boldsymbol{z}^s$ is simply the surface source code of the AST $\boldsymbol{z}$. For semantic parsing, $\boldsymbol{z}^s$ is the linearized s-expression of the logical form. Linearization allows us to use standard sequence-to-sequence networks to model $p(\boldsymbol{z})$ and $p_\theta(x|\boldsymbol{z})$. As we will explain in § 6.4.3, we find these two components perform well with linearization.

Specifically, the prior is parameterized by a Long Short-Term Memory (LSTM) language model over $\boldsymbol{z}^s$. The reconstruction model is an attentional sequence-to-sequence network [73], augmented with a copying mechanism [36], allowing an out-of-vocabulary (OOV) entity in $\boldsymbol{z}^s$ to be copied to x (*e.g.*, the variable name `my_list` in Figure 6.1 and its AST in Figure 6.2).

6.3.2 Inference Model

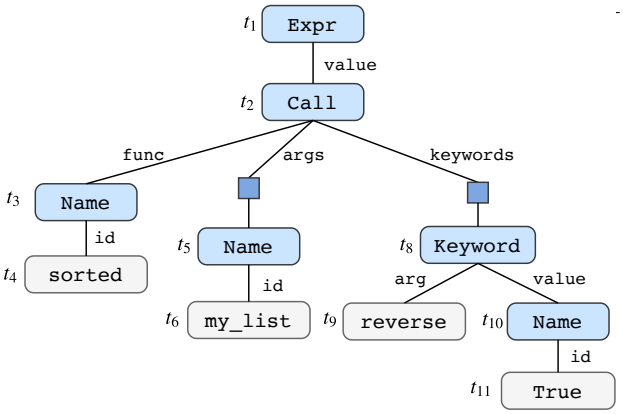
StructVAE models the semantic parser $p_\phi(\boldsymbol{z}|x)$ as the inference model $q_\phi(\boldsymbol{z}|x)$ in VAE (§ 6.2.2), which maps NL utterances x into tree-structured meaning representations $\boldsymbol{z}$. $q_\phi(\boldsymbol{z}|x)$ can be any trainable semantic parser, with the corresponding MRs forming the structured latent semantic space. In this work, we primarily use a semantic parser based on the Abstract Syntax Description Language (ASDL) framework [122] as the inference model. The parser encodes x into ASTs (Figure 6.2). ASTs are the native meaning representation scheme of source code in modern programming languages, and can also be adapted to represent other semantic structures, like λ-calculus logical forms (see § 6.4.2 for details). We remark that StructVAE works with other semantic parsers with different meaning representations as well (*e.g.*, using λ-calculus logical forms for semantic parsing on Atis, explained in § 6.4.3).

Our inference model is a transition-based parser inspired by recent work in neural semantic parsing and code generation. The transition system is an adaptation of Yin and Neubig [137] (hereafter YN17), which decomposes the generation process of an AST into sequential applications of tree-construction actions following the ASDL grammar, thus ensuring the syntactic well-formedness of generated ASTs. Different from YN17, where ASTs are represented as a Context Free Grammar learned from a parsed corpus, we follow Rabinovich et al. [101] and use ASTs defined under the ASDL formalism (§ 6.3.2).

Generating ASTs with ASDL Grammar

First, we present a brief introduction to ASDL. An AST can be generated by applying typed *constructors* in an ASDL grammar, such as those in Figure 6.3 for the Python ASDL grammar.

[1]Linearizion is used by the prior and the reconstruction model only, and not by the inference model.



t	Frontier Field	Action
t_1	`stmt root`	`Expr(expr value)`
t_2	`expr value`	`Call(expr func, expr* args, keyword* keywords)`
t_3	`expr func`	`Name(identifier id)`
t_4	`identifier id`	GenToken$[sorted]$
t_5	`expr* args`	`Name(identifier id)`
t_6	`identifier id`	GenToken$[my_list]$
t_7	`expr* args`	Reduce (close the frontier field)
t_8	`keyword* keywords`	`keyword(identifier arg, expr value)`
t_9	`identifier arg`	GenToken$[reverse]$
t_{10}	`expr value`	`Name(identifier id)`
t_{11}	`identifier id`	GenToken$[True]$
t_{12}	`keyword* keywords`	Reduce (close the frontier field)

Figure 6.2: ***Left*** An example ASDL AST with its surface source code. Field names are labeled on upper arcs. Blue squares denote fields with *sequential* cardinality. Grey nodes denote primitive identifier fields, with annotated values. Fields are labeled with time steps at which they are generated. ***Right*** Action sequences used to construct the example AST. Frontier fields are denoted by their signature `(type name)`. Each constructor in the Action column refers to an ApplyConstr action.

Each constructor specifies a language construct, and is assigned to a particular *composite type*. For example, the constructor `Call` has type `expr` (expression), and it denotes function calls. Constructors are associated with multiple *fields*. For instance, the `Call` constructor and has three fields: *func*, *args* and *keywords*. Like constructors, fields are also strongly typed. For example, the *func* field of `Call` has `expr` type. Fields with composite types are instantiated by constructors of the same type, while fields with *primitive* types store values (*e.g.*, identifier names or string literals). Each field also has a cardinality (single, optional ?, and sequential $*$), specifying the number of values the field has.

Each node in an AST corresponds to a typed field in a constructor (except for the root node). Depending on the cardinality of the field, an AST node can be instantiated with one or multiple constructors. For instance, the *func* field in the example AST has single cardinality, and is instantiated with a `Name` constructor; while the *args* field with sequential cardinality could have multiple constructors (only one shown in this example).

Our parser employs a transition system to generate an AST using three types of actions. Figure 6.2 (Right) lists the sequence of actions used to generate the example AST. The generation process starts from an initial derivation with only a root node of type `stmt` (statement), and

```
stmt ↦ FunctionDef(identifier name,
                                arguments args, stmt* body)
     |  ClassDef(identifier name, expr* bases, stmt* body)
     |  Expr(expr value)
     |  Return(expr? value)

expr ↦ Call(expr func, expr* args, keyword* keywords)
     |  Name(identifier id)
     |  Str(string s)
```

Figure 6.3: Excerpt of the python abstract syntax grammar [99]

proceeds according to the top-down, left-to-right traversal of the AST. At each time step, the parser applies an action to the *frontier field* of the derivation:

ApplyConstr$[c]$ actions apply a constructor c to the frontier composite field, expanding the derivation using the fields of c. For fields with single or optional cardinality, an ApplyConstr action instantiates the empty frontier field using the constructor, while for fields with sequential cardinality, it appends the constructor to the frontier field. For example, at t_2 the `Call` constructor is applied to the *value* field of `Expr`, and the derivation is expanded using its three child fields.

Reduce actions complete generation of a field with optional or multiple cardinalities. For instance, the *args* field is instantiated by `Name` at t_5, and then closed by a Reduce action at t_7.

GenToken$[v]$ actions populate an empty primitive frontier field with token v. A primitive field whose value is a single token (*e.g.*, identifier fields) can be populated with a single GenToken action. Fields of `string` type can be instantiated using multiple such actions, with a final GenToken`[</f>]` action to terminate the generation of field values.

Modeling $q_\phi(\boldsymbol{z}|x)$

The probability of generating an AST $\boldsymbol{z}$ is naturally decomposed into the probabilities of the actions $\{a_t\}$ used to construct $\boldsymbol{z}$:

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

Following YN17, we parameterize $q_\phi(\boldsymbol{z}|x)$ using a sequence-to-sequence network with auxiliary recurrent connections following the topology of the AST. Interested readers are referred to Chapter 3 for implementation details of the neural network architecture.

6.3.3 Semi-supervised Learning

In this section we explain how to optimize the semi-supervised learning objective Eq. (6.1) in StructVAE.

Supervised Learning For the supervised learning objective, we modify $\mathcal{J}_s$, and use the labeled data to optimize both the inference model (the semantic parser) and the reconstruction model:

$$\mathcal{J}_s \triangleq \sum_{(x,\boldsymbol{z})\in\mathbb{L}} \big(\log q_\phi(\boldsymbol{z}|x) + \log p_\theta(x|\boldsymbol{z})\big) \tag{6.2}$$

Unsupervised Learning To optimize the unsupervised learning objective $\mathcal{J}_u$ in Eq. (6.1), we maximize the variational lower-bound of $\log p(x)$:

$$\log p(x) \geq \mathbb{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|x)}\big(\log p_\theta(x|\boldsymbol{z})\big) - \lambda \cdot \mathrm{KL}[q_\phi(\boldsymbol{z}|x)||p(\boldsymbol{z})] = \mathcal{L} \tag{6.3}$$

where $\mathrm{KL}[q_\phi||p]$ is the Kullback-Leibler (KL) divergence. Following common practice in optimizing VAEs, we introduce λ as a tuning parameter of the KL divergence to control the impact of the prior [19, 80].

To optimize the parameters of our model in the face of non-differentiable discrete latent variables, we follow Miao and Blunsom [80], and approximate $\frac{\partial \mathcal{L}}{\partial \phi}$ using the score function estimator (a.k.a. REINFORCE, Williams [125]):

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \phi} &= \frac{\partial}{\partial \phi}\mathbb{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|x)} \\
&\qquad \underbrace{\Big(\log p_\theta(x|\boldsymbol{z}) - \lambda\big(\log q_\phi(\boldsymbol{z}|x) - \log p(\boldsymbol{z})\big)\Big)}_{\text{learning signal}} \\
&= \frac{\partial}{\partial \phi}\mathbb{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|x)} l'(x,\boldsymbol{z}) \\
&\approx \frac{1}{|\mathcal{S}(x)|}\sum_{\boldsymbol{z}_i\in\mathcal{S}(x)} l'(x,\boldsymbol{z}_i)\frac{\partial \log q_\phi(\boldsymbol{z}_i|x)}{\partial \phi}
\end{aligned} \tag{6.4}$$

where we approximate the gradient using a set of samples $\mathcal{S}(x)$ drawn from $q_\phi(\cdot|x)$. To ensure the quality of sampled latent MRs, we follow Guu et al. [38] and use beam search. The term $l'(x,\boldsymbol{z})$ is defined as the *learning signal* [80]. The learning signal weights the gradient for each latent sample $\boldsymbol{z}$. In REINFORCE, to cope with the high variance of the learning signal, it is common to use a baseline $b(x)$ to stabilize learning, and re-define the learning signal as

$$l(x,\boldsymbol{z}) \triangleq l'(x,\boldsymbol{z}) - b(x). \tag{6.5}$$

Specifically, in StructVAE, we define

$$b(x) = a \cdot \log p(x) + c, \tag{6.6}$$

where $\log p(x)$ is a pre-trained LSTM language model. This is motivated by the empirical observation that $\log p(x)$ correlates well with the reconstruction score $\log p_\theta(x|\boldsymbol{z})$, hence with $l'(x, \boldsymbol{z})$.

Finally, for the reconstruction model, its gradient can be easily computed:

$$\frac{\partial \mathcal{L}}{\partial \theta} \approx \frac{1}{|\mathcal{S}(x)|} \sum_{\boldsymbol{z}_i \in \mathcal{S}(x)} \frac{\partial \log p_\theta(x|\boldsymbol{z}_i)}{\partial \theta}.$$

Discussion Perhaps the most intriguing question here is why semi-supervised learning could improve semantic parsing performance. While the underlying theoretical exposition still remains an active research problem [107], in this chapter we try to empirically test some likely hypotheses. In Eq. (6.4), the gradient received by the inference model from each latent sample $\boldsymbol{z}$ is weighed by the learning signal $l(x, \boldsymbol{z})$. $l(x, \boldsymbol{z})$ can be viewed as the reward function in REINFORCE learning. It can also be viewed as weights associated with pseudo-training examples $\{\langle x, \boldsymbol{z}\rangle : \boldsymbol{z} \in \mathcal{S}(x)\}$ sampled from the inference model. Intuitively, a sample $\boldsymbol{z}$ with higher rewards should: (1) have $\boldsymbol{z}$ adequately encode the input, leading to high reconstruction score $\log p_\theta(x|\boldsymbol{z})$; and (2) have $\boldsymbol{z}$ be succinct and natural, yielding high prior probability. Let $\boldsymbol{z}^*$ denote the gold-standard MR of x. Consider the ideal case where $\boldsymbol{z}^* \in \mathcal{S}(x)$ and $l(x, \boldsymbol{z}^*)$ is positive, while $l(x, \boldsymbol{z}')$ is negative for other imperfect samples $\boldsymbol{z}' \in \mathcal{S}(x)$, $\boldsymbol{z}' \neq \boldsymbol{z}^*$. In this ideal case, $\langle x, \boldsymbol{z}^*\rangle$ would serve as a positive training example and other samples $\langle x, \boldsymbol{z}'\rangle$ would be treated as negative examples. Therefore, the inference model would receive informative gradient updates, and learn to discriminate between gold and imperfect MRs. This intuition is similar in spirit to recent efforts in interpreting gradient update rules in reinforcement learning [38]. We will present more empirical statistics and observations in § 6.4.3.

6.4 Experiments

6.4.1 Datasets

In our semi-supervised semantic parsing experiments, it is of interest how StructVAE could further improve upon a supervised parser with extra unlabeled data. We evaluate on two datasets:

Semantic Parsing We use the ATIS dataset, a collection of 5,410 telephone inquiries of flight booking (*e.g.*, *"Show me flights from ci0 to ci1"*). The target MRs are defined using λ-calculus logical forms (*e.g.*, "`lambda $0 e (and (flight $0) (from $ci0) (to $ci1))`"). We use the pre-processed dataset released by Dong and Lapata [31], where entities (*e.g.*, cities) are canonicalized using typed slots (*e.g.*, `ci0`). To predict λ-calculus logical forms using our transition-based parser, we use the ASDL grammar defined by Rabinovich et al. [101] to convert between logical forms and ASTs (see Chapter 3 for details).

Code Generation The DJANGO dataset [90] contains 18,805 lines of Python source code extracted from the Django web framework. Each line of code is annotated with an NL utterance. Source code in the DJANGO dataset exhibits a wide variety of real-world use cases of Python, including IO operation, data structure manipulation, class/function definition, *etc.* We use the pre-processed version released by Yin and Neubig [137] and use the `astor` package to convert ASDL ASTs into Python source code.

6.4.2 Setup

Labeled and Unlabeled Data STRUCTVAE requires access to extra unlabeled NL utterances for semi-supervised learning. However, the datasets we use do not accompany with such data. We therefore simulate the semi-supervised learning scenario by randomly sub-sampling K examples from the training split of each dataset as the labeled set $\mathbb{L}$. To make the most use of the NL utterances in the dataset, we construct the unlabeled set $\mathbb{U}$ using all NL utterances in the training set[2,3].

Training Procedure Optimizing the unsupervised learning objective Eq. (6.3) requires sampling structured MRs from the inference model $q_\phi(\boldsymbol{z}|x)$. Due to the complexity of the semantic parsing problem, we cannot expect any valid samples from randomly initialized $q_\phi(\boldsymbol{z}|x)$. We therefore pre-train the inference and reconstruction models using the supervised objective Eq. (6.2) until convergence, and then optimize using the semi-supervised learning objec-

[2]We also tried constructing $\mathbb{U}$ using the disjoint portion of the NL utterances not presented in the labeled set $\mathbb{L}$, but found this yields slightly worse performance, probably due to lacking enough unlabeled data. Interpreting these results would be an interesting avenue for future work.

[3]While it might be relatively easy to acquire additional unlabeled utterances in practical settings (*e.g.*, through query logs of a search engine), unfortunately most academic semantic parsing datasets, like the ones used in this work, do not feature large sets of in-domain unlabeled data. We therefore perform simulated experiments instead.

$\lvert\mathbb{L}\rvert$	Sup.	SelfTrain	StructVAE
500	63.2	65.3	**66.0**
1,000	74.6	74.2	**75.7**
2,000	80.4	**83.3**	82.4
3,000	82.8	**83.6**	**83.6**
4,434 (All)	**85.3**	–	84.5

Previous Methods	Acc.
ZC07 [149]	84.6
WKZ14 [118]	**91.3**
Seq2Tree [31]†	84.6
ASN [101]†	85.3
+ supervised attention	85.9

Table 6.1: Performance on Atis w.r.t. the size of labeled training data $\mathbb{L}$. †Existing neural network-based methods

tive Eq. (6.1). Throughout all experiments we set α (Eq. (6.1)) and λ (Eq. (6.3)) to 0.1. The sample size $|\mathcal{S}(x)|$ is 5. We observe that the variance of the learning signal could still be high when low-quality samples are drawn from the inference model $q_\phi(\boldsymbol{z}|x)$. We therefore clip all learning signals lower than $k = -20.0$. Early-stopping is used to avoid over-fitting. We also pre-train the prior $p(\boldsymbol{z})$ (§ 6.3.3) and the baseline function Eq. (6.6).

Metric As standard in semantic parsing research, we evaluate by exact-match **accuracy**.

6.4.3 Main Results

Table 6.1 and Table 6.2 list the results on Atis and Django, resp, with varying amounts of labeled data $\mathbb{L}$. We also present results of training the transition-based parser using only the supervised objective (**Sup.**, Eq. (6.2)). We also compare StructVAE with self-training (**SelfTrain**), a semi-supervised learning baseline which uses the supervised parser to predict MRs for unlabeled utterances in $\mathbb{U} - \mathbb{L}$, and adds the predicted examples to the training set to fine-tune the supervised model. Results for StructVAE are averaged over four runs to

$\lvert\mathbb{L}\rvert$	SUP.	SELFTRAIN	STRUCTVAE
1,000	49.9	49.5	**52.0**
2,000	56.6	55.8	**59.0**
3,000	61.0	61.4	**62.4**
5,000	63.2	64.5	**65.6**
8,000	70.3	69.6	**71.5**
12,000	71.1	71.6	**72.0**
16,000 (All)	**73.7**	–	72.3

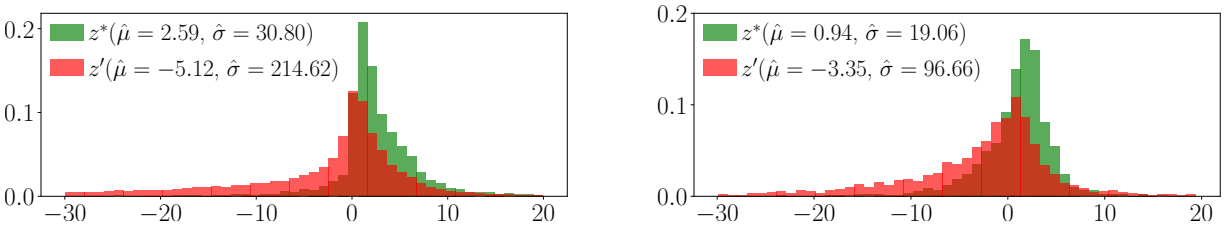
Previous Method	ACC.
YN17 [137]	71.6

Table 6.2: Performance on DJANGO w.r.t. the size of labeled training data $\mathbb{L}$

account for the additional fluctuation caused by REINFORCE training.

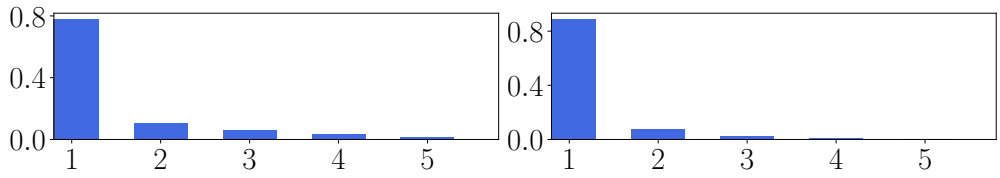
Supervised System Comparison First, to highlight the effectiveness of our transition parser based on ASDL grammar (hence the reliability of our supervised baseline), we compare the supervised version of our parser with existing parsing models. On ATIS, our supervised parser trained on the full data is competitive with existing neural network based models, surpassing the SEQ2TREE model, and on par with the Abstract Syntax Network (ASN) without using extra supervision. On DJANGO, our model significantly outperforms the YN17 system, probably because the transition system used by our parser is defined natively to construct ASDL ASTs, reducing the number of actions for generating each example. On DJANGO, the average number of actions is 14.3, compared with 20.3 reported in YN17.

Semi-supervised Learning Next, we discuss our main comparison between STRUCTVAE with the supervised version of the parser (recall that the supervised parser is used as the inference model in STRUCTVAE, § 6.3.2). First, comparing our proposed STRUCTVAE with the supervised parser when there are extra unlabeled data (*i.e.*, $|\mathbb{L}| < 4,434$ for ATIS and $|\mathbb{L}| < 16,000$ for DJANGO), semi-supervised learning with STRUCTVAE consistently achieves better performance. Notably, on DJANGO, our model registers results as competitive as previous state-of-the-art method (YN17) using only *half* the training data (71.5 when $|\mathbb{L}| = 8000$ v.s. 71.6 for



(a) Django (b) Atis

Figure 6.4: Histograms of learning signals on Django ($|\mathbb{L}| = 5000$) and Atis ($|\mathbb{L}| = 2000$). Difference in sample means is statistically significant ($p < 0.05$).



(a) Django (b) Atis

Figure 6.5: Distribution of the rank of $l(x, \boldsymbol{z}^*)$ in sampled set

YN17). This demonstrates that StructVAE is capable of learning from unlabeled NL utterances by inferring high quality, structurally rich latent meaning representations, further improving the performance of its supervised counterpart that is already competitive. Second, comparing StructVAE with self-training, we find StructVAE outperforms SelfTrain in eight out of ten settings, while SelfTrain under-performs the supervised parser in four out of ten settings. This shows self-training does not necessarily yield stable gains while StructVAE does. Intuitively, StructVAE would perform better since it benefits from the additional signal of the quality of MRs from the reconstruction model (§ 6.3.3), for which we present more analysis in our next set of experiments.

For the sake of completeness, we also report the results of StructVAE when $\mathbb{L}$ is the full training set. Note that in this scenario there is no extra unlabeled data disjoint with the labeled set, and not surprisingly, StructVAE does not outperform the supervised parser. In addition to the supervised objective Eq. (6.2) used by the supervised parser, StructVAE has the extra unsupervised objective Eq. (6.3), which uses sampled (probably incorrect) MRs to update the model. When there is no extra unlabeled data, those sampled (incorrect) MRs add noise to the optimization process, causing StructVAE to under-perform.

Study of Learning Signals As discussed in § 6.3.3, in semi-supervised learning, the gradient received by the inference model from each sampled latent MR is weighted by the learning signal. Empirically, we would expect that on average, the learning signals of gold-standard samples $\boldsymbol{z}^*$, $l(x, \boldsymbol{z}^*)$, are positive, larger than those of other (imperfect) samples $\boldsymbol{z}'$, $l(x, \boldsymbol{z}')$. We therefore study the statistics of $l(x, \boldsymbol{z}^*)$ and $l(x, \boldsymbol{z}')$ for all utterances $x \in \mathbb{U} - \mathbb{L}$, *i.e.*, the set of utterances which are not included in the labeled set.[4] The statistics are obtained by performing inference using trained models. Figures 6.4a and 6.4b depict the histograms of learning signals on DJANGO and ATIS, resp. We observe that the learning signals for gold samples concentrate on positive intervals. We also show the mean and variance of the learning signals. On average, we have $l(x, \boldsymbol{z}^*)$ being positive and $l(x, \boldsymbol{z})$ negative. Also note that the distribution of $l(x, \boldsymbol{z}^*)$ has smaller variance and is more concentrated. Therefore the inference model receives informative gradient updates to discriminate between gold and imperfect samples. Next, we plot the distribution of the rank of $l(x, \boldsymbol{z}^*)$, among the learning signals of all samples of x, $\{l(x, \boldsymbol{z}_i) : \boldsymbol{z}_i \in \mathcal{S}(x)\}$. Results are shown in Figure 6.5. We observe that the gold samples $\boldsymbol{z}^*$ have the largest learning signals in around 80% cases. We also find that when $\boldsymbol{z}^*$ has the largest learning signal, its average difference with the learning signal of the highest-scoring incorrect sample is 1.27 and 0.96 on DJANGO and ATIS, respectively.

Finally, to study the relative contribution of the reconstruction score $\log p(x|\boldsymbol{z})$ and the prior $\log p(\boldsymbol{z})$ to the learning signal, we present examples of inferred latent MRs during training (Table 6.3). Examples 1&2 show that the reconstruction score serves as an informative quality measure of the latent MR, assigning the correct samples $\boldsymbol{z}_1^s$ with high $\log p(x|\boldsymbol{z})$, leading to positive learning signals. This is in line with our assumption that a good latent MR should adequately encode the semantics of the utterance. Example 3 shows that the prior is also effective in identifying "unnatural" MRs (*e.g.*, it is rare to add a function and a string literal, as in $\boldsymbol{z}_2^s$). These results also suggest that the prior and the reconstruction model perform well with linearization of MRs. Finally, note that in Examples 2&3 the learning signals for the correct samples $\boldsymbol{z}_1^s$ are positive even if their inference scores $q(\boldsymbol{z}|x)$ are lower than those of $\boldsymbol{z}_2^s$. This result further demonstrates that learning signals provide informative gradient weights for optimizing the inference model.

Generalizing to Other Latent MRs Our main results are obtained using a strong AST-based semantic parser as the inference model, with copy-augmented reconstruction model and an LSTM language model as the prior. However, there are many other ways to represent and infer

[4] We focus on cases where $\boldsymbol{z}^*$ is in the sample set $\mathcal{S}(x)$.

NL	*join p and cmd into a file path, substitute it for f*
z_1^s	`f = os.path.join(p, cmd)` ✓
	$\log q(\boldsymbol{z}\|x) = -1.00$ $\log p(x\|\boldsymbol{z}) = -2.00$
	$\log p(\boldsymbol{z}) = -24.33$ $l(x, \boldsymbol{z}) = 9.14$
z_2^s	`p = path.join(p, cmd)` ✗
	$\log q(\boldsymbol{z}\|x) = -8.12$ $\log p(x\|\boldsymbol{z}) = -20.96$
	$\log p(\boldsymbol{z}) = -27.89$ $l(x, \boldsymbol{z}) = -9.47$
NL	*append i-th element of existing to child_loggers*
z_1^s	`child_loggers.append(existing[i])` ✓
	$\log q(\boldsymbol{z}\|x) = -2.38$ $\log p(x\|\boldsymbol{z}) = -9.66$
	$\log p(\boldsymbol{z}) = -13.52$ $l(x, \boldsymbol{z}) = 1.32$
z_2^s	`child_loggers.append(existing[existing])`✗
	$\log q(\boldsymbol{z}\|x) = -1.83$ $\log p(x\|\boldsymbol{z}) = -16.11$
	$\log p(\boldsymbol{z}) = -12.43$ $l(x, \boldsymbol{z}) = -5.08$
NL	*split string pks by ',', substitute the result for primary_keys*
z_1^s	`primary_keys = pks.split(',')` ✓
	$\log q(\boldsymbol{z}\|x) = -2.38$ $\log p(x\|\boldsymbol{z}) = -11.39$
	$\log p(\boldsymbol{z}) = -10.24$ $l(x, \boldsymbol{z}) = 2.05$
z_2^s	`primary_keys = pks.split + ','` ✗
	$\log q(\boldsymbol{z}\|x) = -0.84$ $\log p(x\|\boldsymbol{z}) = -14.87$
	$\log p(\boldsymbol{z}) = -20.41$ $l(x, \boldsymbol{z}) = -2.60$

Table 6.3: Inferred latent MRs on DJANGO ($|\mathbb{L}| = 5000$). For simplicity we show the surface representation of MRs ($\boldsymbol{z}^s$, source code) instead.

$\lvert\mathbb{L}\rvert$	Supervised	StructVAE-Seq
500	47.3	**55.6**
1,000	62.5	**73.1**
2,000	73.9	**74.8**
3,000	80.6	**81.3**
4,434 (All)	**84.6**	84.2

Table 6.4: Performance of the StructVAE-Seq on Atis w.r.t. the size of labeled training data $\mathbb{L}$

Atis				Django			
$\lvert\mathbb{L}\rvert$	Sup.	mlp	lm	$\lvert\mathbb{L}\rvert$	Sup.	mlp	lm
500	63.2	*61.5*†	**66.0**	1,000	49.9	*47.0*†	**52.0**
1,000	74.6	**76.3**	75.7	5,000	63.2	*62.5*†	**65.6**
2,000	80.4	**82.9**	82.4	8,000	70.3	*67.6*†	**71.5**
3,000	82.8	*81.4*†	**83.6**	12,000	71.1	71.6	**72.0**

Table 6.5: Comparison of StructVAE with different baseline functions $b(x)$, *italic*†: semi-supervised learning with the MLP baseline is worse than supervised results.

structure in semantic parsing [22, 111], and thus it is of interest whether our basic StructVAE framework generalizes to other semantic representations. To examine this, we test StructVAE using λ-calculus logical forms as latent MRs for semantic parsing on the Atis domain. We use standard sequence-to-sequence networks with attention [73] as inference and reconstruction models. The inference model is trained to construct a tree-structured logical form using the transition actions defined in Cheng et al. [25]. We use a classical tri-gram Kneser-Ney language model as the prior. Table 6.4 lists the results for this StructVAE-Seq model.

We can see that even with this very different model structure StructVAE still provides significant gains, demonstrating its compatibility with different inference/reconstruction networks and priors. Interestingly, compared with the results in Table 6.1, we found that the gains are especially larger with few labeled examples — StructVAE-Seq achieves improvements of 8-10 points when $\lvert\mathbb{L}\rvert < 1000$. These results suggest that semi-supervision is especially useful in improving a mediocre parser in low resource settings.

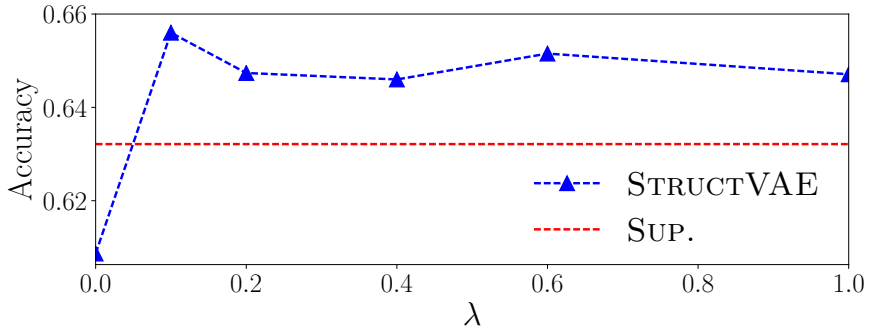


Figure 6.6: Performance on Django ($|\mathbb{L}| = 5000$) w.r.t. the KL weight λ

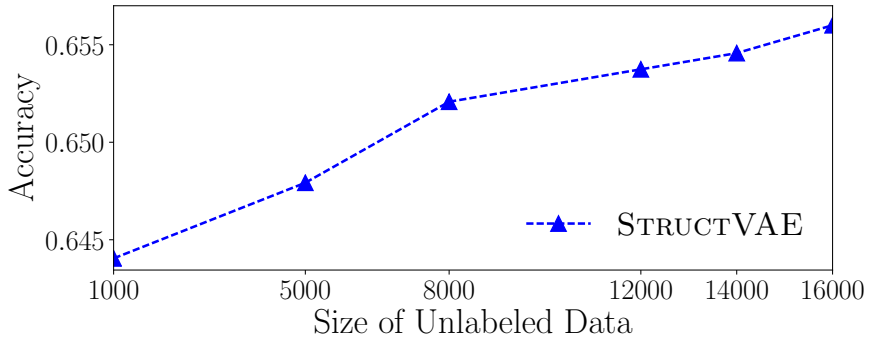


Figure 6.7: Performance on Django ($|\mathbb{L}| = 5000$) w.r.t. the size of unlabeled data $\mathbb{U}$

Impact of Baseline Functions In § 6.3.3 we discussed our design of the baseline function $b(x)$ incorporated in the learning signal (Eq. (6.4)) to stabilize learning, which is based on a language model (LM) over utterances (Eq. (6.6)). We compare this baseline with a commonly used one in REINFORCE training: the multi-layer perceptron (MLP). The MLP takes as input the last hidden state of the utterance given by the encoding LSTM of the inference model. Table 6.5 lists the results over sampled settings. We found that although StructVAE with the MLP baseline sometimes registers better performance on Atis, in most settings it is worse than our LM baseline, and could be even worse than the supervised parser. On the other hand, our LM baseline correlates well with the learning signal, yielding stable improvements over the supervised parser. This suggests the importance of using carefully designed baselines in REINFORCE learning, especially when the reward signal has large range (*e.g.*, log-likelihoods).

Impact of the Prior $p(\boldsymbol{z})$ Figure 6.6 depicts the performance of StructVAE as a function of the KL term weight λ in Eq. (6.3). When StructVAE degenerates to a vanilla auto-encoder without the prior distribution (*i.e.*, $\lambda = 0$), it under-performs the supervised baseline. This is in line with our observation in Table 6.3 showing that the prior helps identify unnatural samples.

The performance of the model also drops when $\lambda > 0.1$, suggesting that empirically controlling the influence of the prior to the inference model is important.

Impact of Unlabeled Data Size Figure 6.7 illustrates the accuracies w.r.t. the size of unlabeled data. StructVAE yields consistent gains as the size of the unlabeled data increases.

6.5 Related Works

Semi-supervised Learning for NLP Semi-supervised learning comes with a long history [158], with applications in NLP from early work of self-training [135], and graph-based methods [28], to recent advances in auto-encoders [26, 108, 151] and deep generative methods [130]. Our work follows the line of neural variational inference for text processing [81], and resembles Miao and Blunsom [80], which uses VAEs to model summaries as discrete latent variables for semi-supervised summarization, while we extend the VAE architecture for more complex, tree-structured latent variables.

Semantic Parsing Most existing works alleviate issues of limited parallel data through weakly-supervised learning, using the denotations of MRs as indirect supervision [56, 85, 93, 105, 139]. For semi-supervised learning of semantic parsing, Kate and Mooney [50] first explore using transductive SVMs to learn from a semantic parser's predictions. Konstas et al. [55] apply self-training to bootstrap an existing parser for AMR parsing. Kociský et al. [54] employ VAEs for semantic parsing, but in contrast to StructVAE's structured representation of MRs, they model NL utterances as flat latent variables, and learn from unlabeled MR data. There have also been efforts in unsupervised semantic parsing, which exploits external linguistic analysis of utterances (*e.g.*, dependency trees) and the schema of target knowledge bases to infer the latent MRs [96, 98]. Another line of research is domain adaptation, which seeks to transfer a semantic parser learned from a source domain to the target domain of interest, therefore alleviating the need of parallel data from the target domain [33, 42, 112].

Chapter 7

Speed-up Data Acquisition (Completed)

Chapter 6 highlights the research issue of labor-intensive data annotation. Data acquisition is particularly costly for applications where the target meaning representations have complex grammars, such as Python code generation, as annotating such MRs requires professionally trained experts like programmers. In this chapter, we seek to speed-up this process using a machine-assisted approach, where a probabilistic model first collects high-quality candidate parallel examples, which are further screened and edited by domain experts. We focus on the scenario of collecting NL intents of programmers and their code implementation in Python, and resort to the curated resource on Stack Overflow to bootstrap the data collection process. This work first appears in:

- Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of MSR*, 2018

7.1 Overview

In order to be effective, statistical semantic parsers, like the code generation model presented in Chapter 2, require access to *high volume, high quality, parallel data* between natural language and code. While one can hope to mine such data from Big Code repositories like Stack Overflow (SO), straightforward mining approaches may also extract quite a bit of noise. We illustrate the challenges associated with mining aligned (parallel) pairs of NL and code from SO with the example of a Python question in Figure 7.1. Given a NL query (or intent), *e.g.*, "removing duplicates in lists", and the goal of finding its matching source code snippets among the different answers, prior work used either a straightforward mining approach that simply picks

all code blocks that appear in the answers [3], or one that picks all code blocks from answers that are highly ranked or *accepted* [46, 127].[1] However, it is not necessarily the case that *every* code block accurately reflects the intent. Nor is it that the *entire* code block is answering the question; some parts may simply describe the context, such as variable definitions (Context 1) or import statements (Context 2), while other parts might be entirely irrelevant (*e.g.*, the latter part of the first code block).

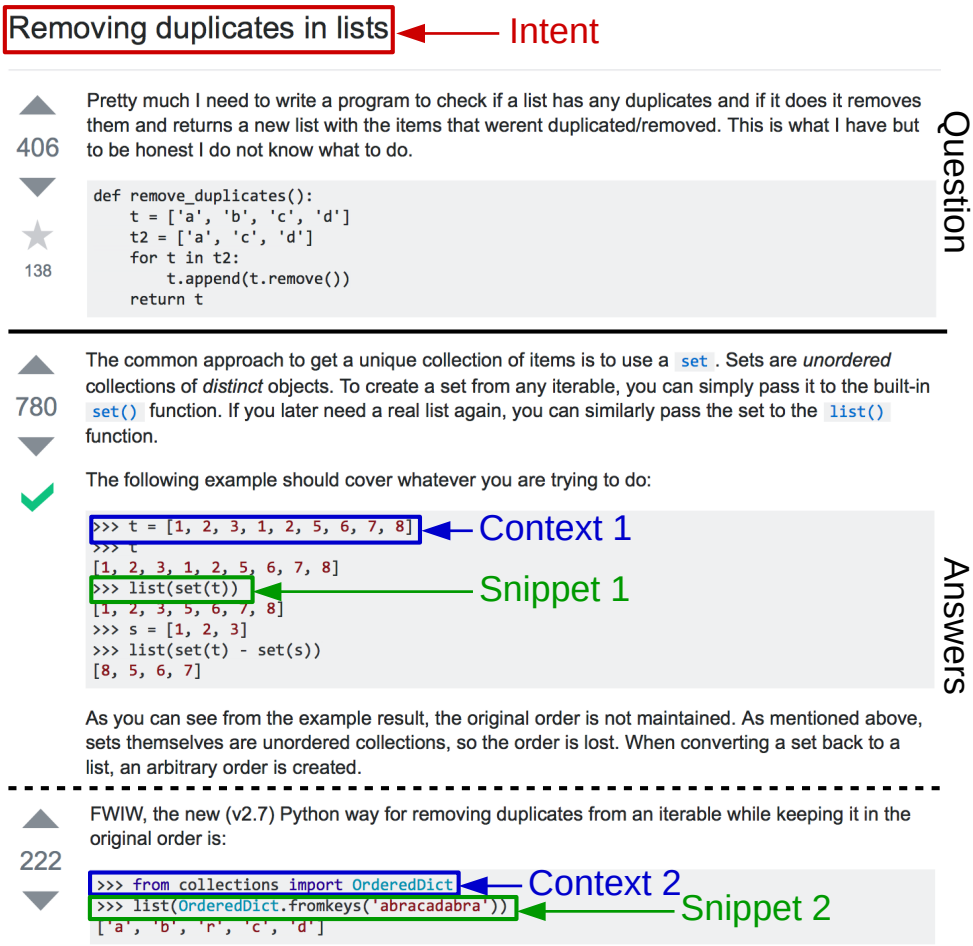


Figure 7.1: Excerpt from a SO post showing two answers, and the corresponding NL intent and code pairs.

There is an inherent trade-off here between scale and data quality. On the one hand, when mining pairs of NL and code from SO, one could devise filters using features of the SO questions, answers, and the specific programming language (*e.g.*, only consider accepted answers with a single code block or with high vote counts, or filtering out `print` statements in Python, much

[1]There is at most one accepted answer per question; see green check symbol in Fig 7.1.

like one thrust of prior work [46, 127]); fine-tuning heuristics may achieve high pair quality, but this inherently reduces the size of the mined data set and it may also be very language-specific. On the other hand, extracting all available code blocks, much like the other thrust of prior work [3], scales better but adds noise (and still cannot handle cases where the "best" code snippets are smaller than a full code block). Ideally, a mining approach to extract parallel pairs would handle these tricky cases and would operate at scale, extracting *many high-quality pairs*. To date, none of the prior work approaches satisfies both requirements of high quality and large quantity.

In this chapter, we propose a novel technique that fills this gap (see Figure 7.2 for an overview). Our key idea is to treat the problem as a classification problem: given an NL intent (*e.g.*, the SO question title) and *all* contiguous code fragments extracted from all answers of that question as candidate matches (for each answer code block, we consider all line-contiguous fragments as candidates, *e.g.*, for a 3-line code block 1-2-3, we consider fragments consisting of lines 1, 2, 3, 1-2, 2-3, and 1-2-3), we use a data-driven classifier to decide if a candidate aligns well with the NL intent. Our model uses two kinds of information to evaluate candidates: (1) *structural features*, which are hand-crafted but largely language-independent, and try to estimate whether a candidate code fragment is valid syntactically, and (2) *correspondence features*, automatically learned, which try to estimate whether the NL and code correspond to each other semantically. Specifically, for the latter we use a model inspired by recent developments in neural network models for machine translation [9], which can calculate bidirectional conditional probabilities of the code given the NL and vice-versa. We evaluate our method on two small labeled data sets of Python and Java code that we created from SO. We show that our approach can extract significantly more, and significantly more accurate code snippets in both languages than previous baseline approaches. We also demonstrate that the classifier is still effective even when trained on Python then used to extract snippets for Java, and vice-versa, which demonstrates potential for generalizability to other programming languages without laborious annotation of correct NL-code pairs.

Our approach strikes a good balance between training effort, scale, and accuracy: the correspondence features can be trained without human intervention on readily available data from SO; the structural features are simple and easy to apply to new programming languages; and the classifier requires minimal amounts of manually labeled data (we only used 152 Python and 102 Java manually-annotated SO question threads in total). Even so, compared to the heuristic techniques from prior work [3, 46, 127], our approach is able to extract up to an order of magnitude more aligned pairs with no loss in accuracy, or reduce errors by more than half while

holding the number of extracted pairs constant.

Specifically, we make the following contributions:

- We propose a novel technique for extracting aligned NL-code pairs from SO posts, based on a classifier that combines snippet structural features, readily extractable, with bidirectional conditional probabilities, estimated using a state-of-the-art neural network model for machine translation.
- We propose a protocol and tooling infrastructure for generating labeled training data.
- We evaluate our technique on two data sets for Python and Java and discuss performance, potential for generalizability to other languages, and lessons learned.
- All annotated data, the code for the annotation interface and the mining algorithm are available at `http://conala-corpus.github.io`.

7.2 Problem Setting

Stack Overflow (SO) is the most popular Q&A site for programming related questions, home to millions of users. An example of the SO interface is shown in Figure 7.1, with a question (in the upper half) and a number of answers by different SO users. Questions can be about anything programming-related, including features of the programming language or best practices. Notably, many questions are of the *"how to"* variety, *i.e.*, questions that ask how to achieve a particular goal such as *"sorting a list"*, *"merging two dictionaries"*, or *"removing duplicates in lists"* (as shown in the example); for example, around 36% of the Python-tagged questions are in this category, as discussed later in § 7.3.2. These *how-to* questions are the type that we focus on in this work, since they are likely to have corresponding snippets and they mimic NL-to-code (or vice versa) queries that users might naturally make in the applications we seek to enable, *e.g.*, code retrieval and synthesis.

Specifically, we focus on extracting triples of three specific elements of the content included in SO posts:

- **Intent:** A description in English of what the questioner wants to do; usually corresponds to some portion of the post title.
- **Context:** A piece of code that does not implement the intent, but is necessary setup, *e.g.*, import statements, variable definitions.
- **Snippet:** A piece of code that actually implements the intent.

An example of these three elements is shown in Figure 7.1. Several interesting points can be gleamed from this example. *First,* and most important, we can see that not all snippets in the post are implementing the original poster's intent: only two of four highlighted are actual examples of how to remove duplicates in lists, the other two highlighted are context, and others still are examples of interpreter output. If one is to train, *e.g.*, a data-driven system for code synthesis from NL, or code retrieval using NL, only the snippets, or portions of snippets, that actually implement the user intent should be used. Thus, we need a mining approach that can distinguish which segments of code are actually legitimate implementations, and which can be ignored. *Second,* we can see that there are often several alternative implementations with different trade-offs (*e.g.*, the first example is simpler in that it doesn't require additional modules to be imported first). One would like to be able to extract all of these alternatives, *e.g.*, to present them to users in the case of code retrieval[2] or, in the case of code summarization, see if any occur in the code one is attempting to summarize.

These aspects are challenging even for human annotators, as we illustrate next.

7.3 Manual Annotation

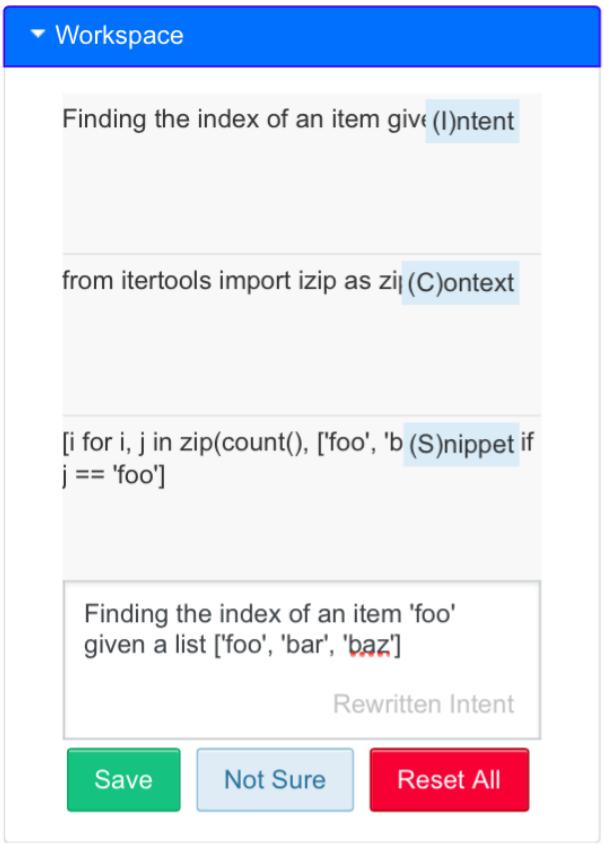
To better understand the challenges with automatically mining aligned NL-code snippet pairs from SO posts, we manually annotated a set of labeled NL-code pairs. These also serve as the gold-standard data set for training and evaluation. Here we describe our annotation method and criteria, salient statistics about the data collected, and challenges faced during annotation.

For each target programming language, we first obtained all questions from the official SO data dump[3] dated March 2017 by filtering questions tagged with that language. We then generated the set of questions to annotate by: (1) including all top-100 questions ranked by view count; and (2) sampling 1,000 questions from the probability distribution generated by their view counts on SO; we choose this method assuming that more highly-viewed questions are more important to consider as we are more likely to come across them in actual applications. While each question may have any number of answers, we choose to only annotate the top-3 highest-scoring answers to prevent annotators from potentially spending a long time on a single question.

[2]Ideally one would also like to present a description of the trade-offs, but mining this information is a challenge beyond the scope of this work.

[3]Available online at https://archive.org/details/stackexchange

7.3.1 Annotation Protocol and Interface



Consistently annotating the intent, context, and snippet for a variety of posts is not an easy task, and in order to do so we developed and iteratively refined a web annotation interface and a protocol with detailed annotation criteria and instructions.

The annotation interface allows users to select and label parts of SO posts as (I)intent, (C)ontext, and (S)nippet using shortcut keys, as well as rewrite the intent to better match the code (*e.g.*, adding variable names from the snippet into the original intent), in consideration of potential future applications that may require more precisely aligned NL-code data; in the following experiments we solely consider the intent and snippet, and reserve examination of the context and re-written intent for future work. Multiple NL-code pairs that are part of the same post can be annotated this way. There is also a "not applicable" button that allows users to skip posts that are not of the "how to" variety, and a "not sure" button, which can be used when the annotator is uncertain.

The annotation criteria were developed by having all authors attempt to perform annotations of sample data, gradually adding notes of the difficult-to-annotate cases to a shared document. We completed several pilot annotations for a sample of Python questions, iteratively discussing among the research team the annotation criteria and the difficult-to-annotate cases after each, before finalizing the annotation protocol. We repeated the process for Java posts. Once we converged on the final annotation standards in both languages, we discarded all pilot annotations, and one of the authors (a graduate-level NLP researcher and experienced programmer) re-annotated the entire data set according to this protocol.

While we cannot reflect all difficult cases here for lack of space, below is a representative sample from the Python instructions:

- **Intents:** Annotate the command form when possible (*e.g.*, "how do I merge dictionaries"

Table 7.1: Details of the labeled data set.

Lang.	#Annot.	#Ques.	#Answer Posts	#Code Blocks	Avg. Code Length	%Full Blocks	%Annot. with Context
Python	527	142	412	736	13.2	30.7%	36.8%
Java	330	100	297	434	30.6	53.6%	42.4%

will be annotated as "merge dictionaries"). Extraneous words such as "in Python" can be ignored. Intents will almost always be in the title of the post, but intents expressed elsewhere that are different from the title can also be annotated.

- **Context:** Contexts are a set of statements that do not directly reflect the annotated intent, but may be necessary in order to get the code to run, and include import statements, variable definitions, and anything else that is necessary to make sure that the code executes. When no context exists in the post this field can be left blank.
- **Snippet:** Try to annotate full lines when possible. Some special tokens such as "»>", "`print`", and "`In[...]`" that appear at the beginning of lines due to copy-pasting can be included. When the required code is encapsulated in a function, the function definition can be skipped.
- **Re-written intent:** Try to be accurate, but try to make the minimal number of changes to the original intent. Try to reflect all of the free variables in the snippet to be conducive to future automatic matching of these free variables to the corresponding position in code. When referencing string literals or numbers, try to write exactly as written in the code, and surround variables with a grave accent "`".

7.3.2 Annotation Outcome

We annotated a total of 418 Python questions and 200 Java questions. Of those, 152 in Python and 102 in Java were judged as annotatable (*i.e.*, the *"how-to"* style questions described in § 7.2), resulting in 577 Python and 354 Java annotations. We then removed the annotations marked as "not sure" and all unparsable code snippets.[4] In the end we generated 527 Python and 330 Java annotations, respectively. Table 7.1 lists basic statistics of the annotations. Notably, compared to Python, Java code snippets are longer (13.2 vs. 30.6 tokens per snippet), and more likely to be

[4]We use the built-in `ast` parser module for Python, and `JavaParser` for Java.

full code blocks (30.7% vs. 53.6%). That is, in close to 70% of cases for Python, the code snippet best-aligned with the NL intent expressed in the question title was *not* a full code block (SO uses special HTML tags to highlight code blocks, recall the example in Figure 7.1) from one of the answers, but rather a subset of it; similarly, the best-aligned Java snippets were *not* full code blocks in almost half the cases. This confirms the importance of mining code snippets beyond the level of entire code blocks, a limitation of prior approaches.

Overall, we found the annotation process to be non-trivial, which raises several noteworthy threats to validity: (1) it can be difficult for annotators to distinguish between incorrect solutions and unusual or bad solutions that are nonetheless correct; (2) in cases where a single SO question elicits many correct answers with many implementations and code blocks, annotators may not always label all of them; (3) long and complex solutions may be mis-annotated; and (4) inline code blocks are harder to recognize than stand-alone code blocks, increasing the risk of annotators missing some. We made a best effort to minimize the impact of these threats by carefully designing and iteratively refining our annotation protocol.

7.4 Mining Method

In this section, we describe our mining method (see Figure 7.2 for an overview). As mentioned in § 7.2, we frame the problem as a classification problem. First, for every "how to" SO question we consider its title as the intent and extract all contiguous lines from across all code blocks in the question's answers (including those we might manually annotate as context; inline code snippets are excluded) as candidate implementations of the intent, as long as we could parse the candidate snippets.[4] There are some cases where the title is not strictly equal to the intent, which go beyond the scope of this chapter; for the purpose of learning the model we assume the title is representative. This step generates, for every SO question considered, a set of pairs (intent I, candidate snippet S). For example, the second answer in Figure 7.1, containing a three-line-long code block, would generate six line-contiguous candidate snippets, corresponding to lines 1, 2, 3, 1-2, 2-3, and 1-2-3. Our candidate snippet generation approach, though clearly not the only possible approach (1) is simple and language-independent, (2) is informed by our manual annotations, and (3) it gives good coverage of all possible candidate snippets.

Then, our task is, given a candidate pair (I, S), to assign a label y representing whether or not the snippet S reflects the intent I; we define y to equal 1 if the pair matches and -1 otherwise. Our general approach to making this binary decision is to use machine learning to train a classifier that predicts, for every pair (I, S), the probability that S accurately implements I, *i.e.*,

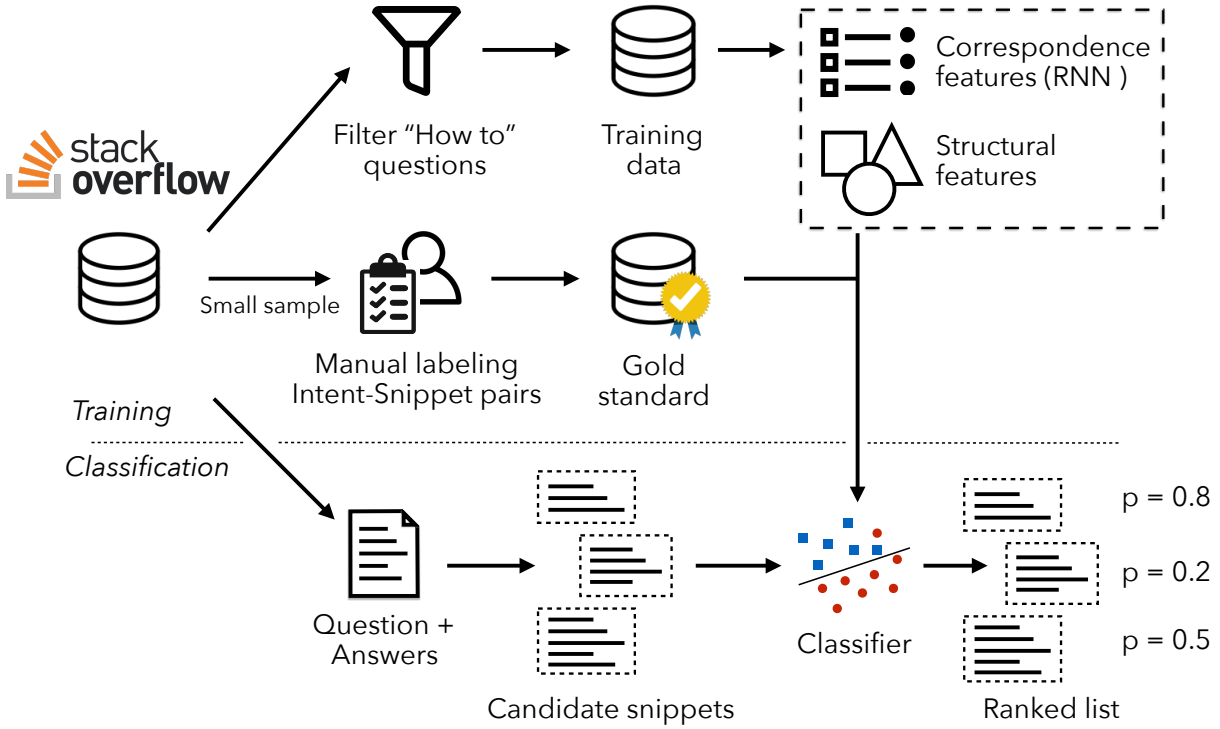


Figure 7.2: Overview of our approach.

$P(y = 1|I, S)$, based on a number of *features* (Sections § 7.4.1 and § 7.4.2). As is usual in supervised learning, our system first requires an offline *training* phase that learns the parameters (*i.e.*, feature weights) of the classifier, for which we use the annotated data described above (§ 7.3). This way, we can apply our system to an SO page of interest, and compute $P(y = 1|I, S)$ for each possible intent/candidate snippet pair mined from the SO page. We choose logistic regression as our classifier, as implemented in the `scikit-learn` Python package.

As human annotation to generate training data is costly, our goal is to keep the amount of manually labeled training data to a minimum, such that scaling our approach to other programming languages in the future can be feasible. Therefore, to ease the burden on the classifier in the face of limited training data, we combined two types of features: hand-crafted *structural* features of the code snippets (§ 7.4.1) and machine learned *correspondence* features that predict whether intents and code snippets correspond to each-other semantically (§ 7.4.2). Our intuition, again informed by the manual annotation, was that "good" and "bad" pairs can often be distinguished based on simple hand-crafted features; these features could eventually be learned (as opposed to hand-crafted), but this would require more labeled training data, which is relatively expensive to create.

7.4.1 Hand-crafted Code Structure Features

The structural features are intended to distinguish whether we can reasonably expect that a particular piece of code implements an intent. We aimed for these features to be both informative and generally applicable to a wide range of programming languages. These features include the following:

- **FullBlock, StartOfBlock, EndOfBlock:** A code block may represent a single cohesive solution. By taking only a piece of a code block, we may risk acquiring only a partial solution, and thus we use a binary feature to inform the classifier of whether it is looking at a whole code block or not. On the other hand, as shown in Figure 7.1, many code blocks contain some amount of context before the snippet, or other extraneous information, *e.g.*, print statements. To consider these, we also add binary features indicating that a snippet is at the start or end of its code block.
- **ContainsImport, StartsWithAssignment, IsValue:** Additionally, some statements are highly indicative of a statement being context or extraneous. For example, import statements are highly indicative of a particular line being context instead of the snippet itself, and thus we add a binary feature indicating whether an import statement is included. Similarly, variable assignments are often context, not the implementation itself, and thus we add another feature indicating whether the snippet starts with a variable assignment. Finally, we observed that in SO (particularly for Python), it was common to have single lines in the code block that consisted of only a variable or value, often as an attempt to print these values to the interactive terminal.
- **AcceptedAns, PostRank1, PostRank2, PostRank3:** The quality of the post itself is also indicative of whether the answer is likely to be valid or not. Thus, we add several features indicating whether the snippet appeared in a post that was the accepted answer or not, and also the rank of the post within the various answers for a particular question.
- **OnlyBlock:** Posts with only a single code block are more likely to have that snippet be a complete implementation of the intent, so we added another feature indicating when the extracted snippet is the only one in the post.
- **NumLinesX:** Snippets implementing the intent also tend to be concise, so we added features indicating the number of lines in the snippet, bucketed into $X = 1$, 2, 3, 4-5, 6-10, 11-15, >15.
- **Combination Features:** Some features can be logically combined to express more com-

plex concepts. E.g., AcceptedAns + OnlyBlock + WholeBlock can express the strategy of selecting whole blocks from accepted answers with only one block, as used in previous work [46, 127]. We use this feature and two other combination features: specifically ¬StartWithAssign + EndOfBlock and ¬StartWithAssign + NumLines1.

7.4.2 Unsupervised Correspondence Features

While all of the features in the previous section help us determine which code snippets are likely to implement *some* intent, they say nothing about whether the code snippet actually implements *the particular* intent I that is currently under consideration. Of course considering this correspondence is crucial to accurately mining intent-snippet pairs, but how to evaluate this correspondence computationally is non-trivial, as there are very few hard and fast rules that indicate whether an intent and snippet are expressing similar meaning. Thus, in an attempt to capture this correspondence, we take an indirect approach that uses a potentially-noisy (*i.e.*, not manually validated) but easy-to-construct data set to train a probabilistic model to approximately capture these correspondences, then incorporate the predictions of this noisily trained model as features into our classifier.

Training data of correspondence features: Apart from our manually-annotated data set, we collected a relatively large set of intent-snippet pairs using simple heuristic rules for learning the correspondence features. The data set is created by pairing the question titles and code blocks from all SO posts, where (1) the code block comes from an SO answer that was accepted by the original poster, and (2) there is only one code block in this answer. Of course, many of these code blocks will be noisy in the sense that they contain extraneous information (such as extra import statements or variable definitions, *etc.*), or not directly implement the intent at all, but they will still be of use for learning which NL expressions in the intent tend to occur with which types of source code.

Learning a model of correspondence: Given the training data above, we need to create a model of the correspondence between the intent I and snippet S. To this end, we build a statistical model of the bi-directional probability of the intent given the snippet $P(I \mid S)$, and the probability of the snippet given the intent $P(S \mid I)$. Specifically, we estimate $P(I \mid S)$ and $P(S \mid I)$ using attentional neural machine translation models [9] trained one the corpus described above.

Incorporating correspondence probabilities as features: For each intent I and candidate snippet S, we calculate the probabilities $P(S \mid I)$ and $P(I \mid S)$, and add them as features to our classifier, as we did with the hand-crafted structural features in § 7.4.1.

- **SGivenI, IGivenS:** Our first set of features are the logarithm of the probabilities mentioned above: $\log P(S \mid I)$ and $\log P(I \mid S)$.[5] Intuitively, these probabilities will be indicative of S and I being a good match because if they are not, the probabilities will be low. If the snippet and the intent are not a match at all, both features will have a low value. If the snippet and intent are partial matches, but either the snippet S or intent I contain extraneous information that cannot be predicted from the counterpart, then SGivenI and IGivenS will have low values respectively.
- **ProbMax, ProbMin:** We also represent the max and min of $\log P(S \mid I)$ and $\log P(I \mid S)$. In particular, the ProbMin feature is intuitively helpful because pairs where the probability in *either* direction is low are likely not good pairs, and this feature will be low in the case where either probability is low.
- **NormalizedSGivenI, NormalizedIGivenS:** In addition, intuitively we might want the *best* matching NL-code pairs within a particular SO page. In order to capture this intuition, we also normalize the scores over all posts within a particular page so that their mean is zero and standard deviation is one (often called the z-score). In this way, the pairs with the best scores within a page will get a score that is significantly higher than zero, while the less good scores will get a score close to or below zero.

7.5 Evaluation

In this section we evaluate our proposed mining approach. We first describe the experimental setting in § 7.5.1 before addressing the following research questions:

1. How does our mining method compare with existing approaches across different programming languages? (§ 7.5.2)
2. How do the structural and correspondence features impact the system's performance? (§ 7.5.2)
3. Given that annotation of data for each language is laborious, is it possible to use a classifier

[5] We take the logarithm of the probabilities because the actual probability values tend to become very small for very long sequences (*e.g.*, 10^{-50} to 10^{-100}), while the logarithm is in a more manageable range (*e.g.*, -50 to -100).

learned on one programming language to perform mining on other languages? (§ 7.5.3)

4. What are the qualitative features of the NL-code pairs that our method succeeds or fails at extracting? (§ 7.5.4)

We show that our method clearly outperforms existing approaches and shows potential for reuse *without retraining*, we uncover trade-offs between performance and training complexity, and we discuss limitations, which can inform future work.

7.5.1 Experimental Settings

We conduct experimental evaluation on two programming languages: Python and Java. These languages were chosen due to their large differences in syntax and verbosity, which have been shown to effect characteristics of code snippets on SO [132].

Learning unsupervised features: We start by filtering the SO questions in the Stack Exchange data dump[3] by tag (Python and Java), and we use an existing classifier [46] to identify the *how-to* style questions. The classifier is a support vector machine trained by bootstrapping from 100 labeled questions, and achieves over 75% accuracy as reported in [46]. We then extract intent/snippet pairs from all these questions as described in § 7.4.2, collecting 33,946 pairs for Python and 37,882 for Java. Next we split the data set into training and validation sets with a ratio of 9:1, keeping the 90% for training. Statistics of the data set are listed in Table 7.2.[6]

We implement our neural correspondence model using the DyNet neural network toolkit [87]. The dimensionality of word embedding and RNN hidden states is 256 and 512. We use dropout [110], a standard method to prevent overfitting, on the input of the last softmax layer over target words ($p = 0.5$), and recurrent dropout [34] on RNNs ($p = 0.2$). We train the network using the widely used optimization method Adam [51]. To evaluate the neural network, we use the remaining 10% of pairs left aside for testing, retaining the model with the highest likelihood on the validation set.

Evaluating the mining model: For the logistic regression classifier, which uses the structural and correspondence features described above, the latter computed by the previous neural network, we use our annotated intent/snippet data (§ 7.3.2)[7] during **5-fold cross validation**.

[6]Note that this data may contain some of the posts included in the cross-validation test set with which we evaluate our model later. However, even if it does, we are not using the annotations themselves in the training of the correspondence features, so this does not pose a problem with our experimental setting.

[7]Recall that our annotated data contains only how-to style questions, and therefore question filtering is not required. When applying our mining method to the full SO data, we could use the how-to question classifier in [46].

Table 7.2: Details of the NL-code data used for learning unsupervised correspondence features.

Lang.	Training Data (NL/Code Pairs)	Validation Data	Intents		Code	
			Avg. Length	Vocabulary Size	Avg. Length	Vocabulary Size
Python	33,946	3,773	11.9	12,746	65.4	30,286
Java	37,882	4,208	11.6	13,775	65.7	29,526

Recall, our code mining model takes as input a SO question (*i.e.*, intent reflected by the question title) with its answers, and outputs a ranked list of candidate intent/snippet pairs (with probability scores). For evaluation, we first rank all candidate intent/snippet pairs for all questions, and then compare the ranked list with gold-standard annotations. We present the results using standard precision-recall (PR) and Receiver Operating Characteristic (ROC) curves. In short, a PR curve shows the precision w.r.t. recall for the top-k predictions in the ranked list, with k from 1 to the number of candidates. A ROC curve plots the true positive rates w.r.t. false positive rates in similar fashion. We also compute the Area Under the Curve (AUC) scores for all ROC curves.

Baselines: As baselines for our model (denoted as FULL), we implement three approaches reflecting prior work and sensible heuristics:

ACCEPTONLY is the state-of-the art from prior work [46, 127]; it selects the whole code snippet in the *accepted* answers containing exactly one code snippet.

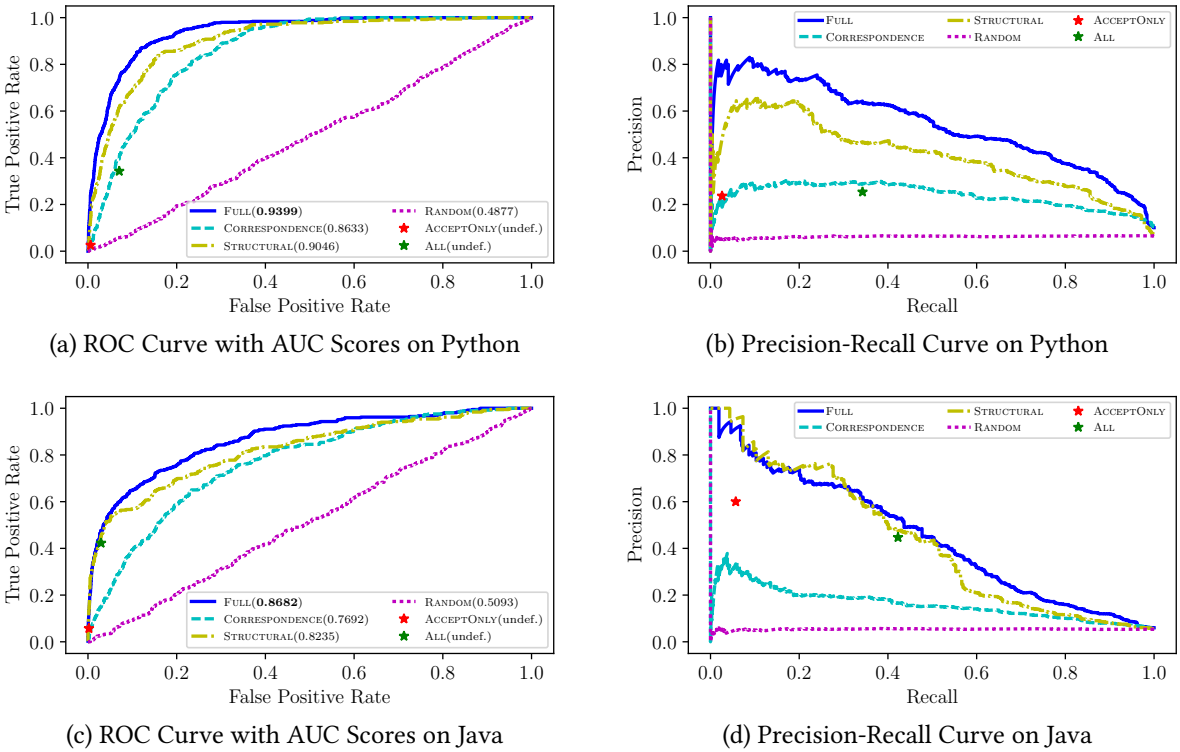
ALL denotes the baseline method that exhaustively selects all full code blocks in the top-3 answers in a post.

RANDOM is the baseline that randomly selects from all consecutive code segment candidates. Similarly to our model, we enforce the constraint that all mined code snippets given by the baseline approaches should be parseable.

Additionally, to study the impact of hand-crafted **STRUCTURAL** versus learned **CORRESPONDENCE** features, we also trained versions of our model with either of the two types of features only.

7.5.2 Results and Discussion

Our main results are depicted in Figure 7.3. First, we can see that the precision of the RANDOM baseline is only 0.10 for Python and 0.06 for Java. This indicates that only one in 10-17 candidate



(a) ROC Curve with AUC Scores on Python

(b) Precision-Recall Curve on Python

(c) ROC Curve with AUC Scores on Java

(d) Precision-Recall Curve on Java

Figure 7.3: Evaluation Results on Mining Python (a)(b) and Java (c)(d)

code snippets is judged to validly correspond to the intent, reflecting the difficulty of the task. The AcceptOnly and All baselines perform significantly better, with precision of 0.5 or 0.6 at recall 0.05-0.1 and 0.3-0.4 respectively, indicating that previous heuristic methods using full code blocks are significantly better than random, but still have a long way to go to extract broad-coverage and accurate NL-code pairs (particularly in the case of Python).[8]

Next, turning to the full system, we can see that the method with the full feature set significantly outperforms all baselines (Figures 7.3b and 7.3d): much better recall (precision) at the same level of precision (recall) as the heuristic approaches. The increase in precision suggests the importance of intelligently selecting NL-code pairs using informative features, and the increase in recall suggests the importance of considering segments of code within code blocks, instead of simply selecting the full code block as in prior work.

Comparing different types of features (Structural v.s. Correspondence), we find that with structural features alone our model already significantly outperforms baseline approaches;

[8]Interestingly, AcceptOnly and All have similar precision, which might be due to two facts. First, we enforce all candidate snippets to be syntactically correct, which rules out erroneous candidates like input/output examples. Second, we use the top 3 answers for each question, which usually have relatively high quality.

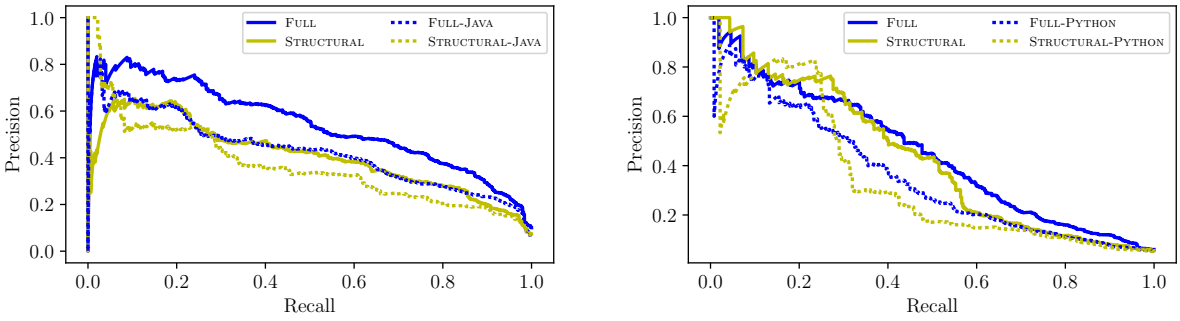
and these features are particularly effective for Java. On the other hand, interestingly the correspondence features alone provide less competitive results. Still, the structural and correspondence features seem to be complementary, with the combination of the two feature sets further significantly improving performance, particularly on Python. A closer examination of the results generated the following insights.

Why do correspondence features underperform? While these features effectively filter *totally* unrelated snippets, they still have a difficult time excluding related contextual statements, *e.g.*, imports, assignments. This is because (1) the snippets used for training correspondence features are full code blocks (as in §§ 7.4.2), usually starting with `import` statements; and (2) the library names in `import` statements often have strong correspondence with the intents (*e.g.*, *"How to get current time in Python?" and* `import datetime`), yielding high correspondence probabilities.

What are the trends and error cases for structural features? Like the baseline methods, Structural tends to give priority to full code blocks; out of the top-100 ranked candidates for Structural, all were full code blocks (in contrast to only 21 for Correspondence). Because selecting code blocks is a reasonably strong baseline, and because the model has access to other strongly-indicative binary features that can be used to further prioritize its choices, it is able to achieve reasonable precision-recall scores only utilizing these features. However, unsurprisingly, it lacks fine granularity in terms of pinpointing exact code segments that correspond to the intents; when it tries to select partial code segments, the results are likely to be irrelevant to the intent. As an example, we find that Structural tends to select the last line of code at each code block, since the learned weights for LineNum=1 and EndsCodeBlock features are high, even though these often consist of auxiliary `print` statement or even simply `pass` (for Python).

What is the effect of the combination? When combining Structural and Correspondence features together, the full model has the ability to use the knowledge of the Structural model extract full code blocks or ignore imports, leading to high performance in the beginning stages. Then, in the latter and more difficult cases, it is able to more effectively cherry-pick smaller snippets based on their correspondence properties, which is reflected in the increased accuracy on the right side of the ROC and precision-recall curves.

How do the trends differ between programming languages? Compared with the baseline approaches AcceptOnly and All, our full model performs significantly better on Python. We hypothesize that this is because learning correspondences between intent/snippet pairs for Java is more challenging. Empirically, Python code snippets are much shorter, and the average number of tokens for predicted code snippets on Python and Java is 11.6 and 42.4, respectively.



(a) Java ↦ Python (b) Python ↦ Java

Figure 7.4: Precision-Recall Curves for Transfer Learning on Java ↦ Python (a) and Python ↦ Java (b)

Meanwhile, since Java code snippets are more verbose and contain significantly more boilerplate (*e.g.*, class/function definitions, type declaration, exception handling, *etc.*), estimating correspondence scores using neural networks is more challenging.

Also note that the Structural model performs much better on Java than on Python. This is due to the fact that Java annotations are more likely to be full code blocks (see Table 7.1), which can be easily captured by our designed features like FullBlock. Nevertheless, adding correspondence features is clearly helpful for the harder cases for both programming languages. For instance, from the ROC curve in Figure 7.3c, our full model achieves higher true positive rates compared with Structural, registering higher AUC scores.

7.5.3 Must We Annotate Each Language?

As discussed in §§ 7.3, collecting high-quality intent/snippet annotations to train the code mining model for a programming language can be costly and time-consuming. An intriguing research question is how we could *transfer* the learned code mining model from one programming language and use it for mining intent/snippet data for another language. To test this, we train a code mining model using the annotated intent/snippet data on language A, and evaluate using the annotated data on language B.[9] This is feasible since almost all of the features used in our system is language-agnostic.[10] Also note values of a specific feature might have different ranges for different languages. As an example, the average value of SGivenI feature for Python and

[9]We still train the correspondence model using the target language unlabeled data.

[10]The only one that was not applicable to both languages was the SingleValue feature for Python, which helps rule out code that contains only a single value. We omit this feature in the cross-lingual experiments.

Java is -23.43 and -47.64, respectively. To mitigate this issue, we normalize all feature values to zero mean and unit variance before training the logistic regression classifier.

Figures 7.4a and 7.4b show the precision-recall curves for applying Java (Python) mining model on Python (Java) data. We report results for both the STRUCTURAL model and our full model, and compare with the original models trained on the target programming language. Unsurprisingly, the original full model tuned on the target language still performs the best. Nevertheless, we observe that the performance gap between the original full model and the transferred one is surprisingly small. Notably, we find that overall the transferred full model (FULL-JAVA) performs second best on Python, even outperforming the original STRUCTURAL model. These results are encouraging, in that they suggest that it is likely feasible to train a single code mining classifier and then apply it to different programming languages, even those for which we do not have any annotated intent/snippet data.

7.5.4 Successful and Failed Examples

As illustration, we showcase successful and failed examples of our proposed approach, for Python in Table 7.3 and for Java in Table 7.4. Given a SO question (intent), we show the top-3 most probable code snippets. First, we find our model can correctly identify code snippets for various types of intents, even when the target snippets are not full code blocks. I_1 and I_6 demonstrate that our model can leave contextual information like variable definitions in the original SO posts and only retain the actual implementation of the intent.[11] I_2, I_3 and I_7 are more interesting: in the original SO post, there could be multiple possible solutions in the same code block (I_2 and I_7), or the gold-standard snippets are located inside larger code structures like a for loop (S_2 for I_3). Our model learns to "break down" the solutions in single code block into multiple snippets, and extract the actual implementation from large code chunks.

We also identify four sources of errors:

- *Incomplete code:* Some code snippets are incomplete, and the model fails to include intermediate statements (*e.g.*, definitions of custom variables or functions) that are part of the implementation. For instance, S_3 for I_3 misses the definition of the `keys_to_keep`, which is the set of keys excluding the key to remove.
- *Including auxiliary info:* Sometimes the model fails to exclude auxiliary code segments like the extra context definition (*e.g.*, S_1 for I_8) and `print` function. This is especially true for Java, where full code blocks are likely to be correct snippets, and the model tends

[11] We refer readers to the original SO page for reference.

Table 7.3: Examples of Mined Python Code

I_1: *Remove specific characters from a string in python*

URL: https://stackoverflow.com/q/3939361/

Top Predictions:

S_1
```
string.replace('1', '')
```
✓

S_2
```
line = line.translate(None, '!@#$')
```
✓

S_3
```
line = re.sub('[!@#$]', '', line)
```
✓

I_2: *Get Last Day of the Month in Python*

URL: https://stackoverflow.com/q/42950/

Top Predictions:

S_1
```
calendar.monthrange(year, month)[1]
```
✓

S_2
```
calendar.monthrange(2100, 2)
```
✓

S_3
```
(datetime.date(2000, 2, 1) - datetime.
    timedelta(days=1))
```
✓

I_3: *Delete a dictionary item if the key exists*

URL: https://stackoverflow.com/q/15411107/

Top Predictions:

S_1
```
mydict.pop('key', None)
```
✓

S_2
```
del mydict[key]
```
✓

S_3
```
new_dict = {k: mydict[k] for k in
    keys_to_keep}
```
✗

I_4: *Python: take the content of a list and append it to another list*

URL: https://stackoverflow.com/q/8177079/

Top Predictions:

S_1
```
list2.append(list1)
```
✗

S_2
```
list2.extend(list1)
```
✓

S_3
```
list1.extend(mylog)
```
✓

I_5: *Converting integer to string in Python?*

URL: https://stackoverflow.com/q/961632/

Top Predictions:

S_1
```
int('10')
```
✗

S_2
```
str(10); int('10')
```
✗

S_3
```
a.__str__()
```
✓

Table 7.4: Examples of Mined Java Code

I_6: *How to convert List<Integer> to int[] in Java?*

URL: https://stackoverflow.com/q/960431/

Top Predictions:

S_1
```
int[] array = list.stream().mapToInt(i
    -> i).toArray();
```
✓

S_2
```
int[] intArray2 = ArrayUtils.toPrimitive(
    myList.toArray(NO_INTS));
```
✗

S_3
```
int[] intArray = ArrayUtils.toPrimitive(
    myList.toArray(new Integer[myList.
    size()]));
```
✓

I_7: *How do I compare strings in Java?*

URL: https://stackoverflow.com/q/513832/

Top Predictions:

S_1
```
new String("test").equals("test");
```
✓

S_2
```
Objects.equals(null, "test");
```
✓

S_3
```
nullString1.equals(nullString2);
```
✓

I_8: *How do I set the colour of a label (coloured text) in Java?*

URL: https://stackoverflow.com/q/2966334/

Top Predictions:

S_1
```
JLabel title = new JLabel("I love
    stackoverflow!", JLabel.CENTER);
title.setForeground(Color.white);
```
✗

S_2
```
frame.add(new JLabel("<html>Text color: <
    font color='red'>red</font></html>"))
    ;
```
✓

S_3
```
label.setForeground(Color.red);
```
✓

I_9: *Generating a Random Number between 1 and 10 Java*

URL: https://stackoverflow.com/q/20389890/

Top Prediction: (only show one for space reason)

S_1
```
public static int randInt(int min, int
    max) {
    Random rand = new Random();
    int randomNum = rand.nextInt((max -
        min) + 1) + min;
    return randomNum; }
```
✗ (annotation error)

to bias towards larger code chunks.

- *Spurious cases:* We identify two "spurious" cases where our correspondence feature often do not suffice. (1) *Counter examples*: the S_1 for I_4 is mentioned in the original post as a counter example, but the values of correspondence features are still high since `append()` is highly related to *"append it to another list"* in the intent. (2) *Related implementation*: I_5 shows an example where the model has difficulty distinguishing between the actual snippets and related implementations.
- *Annotation error:* We find cases where our annotation is incomplete. For instance, S_1 for I_9 should be correct. As discussed in § 7.3, guaranteeing coverage in the annotation process is non-trivial, and we leave this as a challenge for future work.

7.6 Related Work

A number of previous works have proposed methods for mining intent-snippet pairs for purposes of code summarization, search, or synthesis. We can view these methods from several perspectives:

Data Sources: First, what data sources do they use to mine their data? Our work falls in the line of mining intent-snippet pairs from SO (*e.g.*, [46, 127, 134, 146]), while there has been research on mining from other data sources such as API documentation [13, 23, 84], code comments [128], specialized sites [100], parameter/method/class names [2, 109], and developer mailing lists [91]. It is likely that it could be adapted to work with other sources, requiring only changes in the definition of our structural features to incorporate insights into the data source at hand.

Methodologies: Second, what is the methodology used therein, and can it scale to our task of gathering large-scale data across a number of languages and domains? Several prior work approaches used heuristics to extract aligned intent-snippet pairs [23, 127, 146]). Our approach also contains an heuristic component. However, as evidenced by our experiments here, our method is more effective at extracting accurate intent-snippet pairs.

Some work on code search has been performed by retrieving candidate code snippets given an intent based on weighted keyword matches and other features [89, 124]. These methods similarly aim to learn correspondences between natural language queries and returned code, but they are tailored specifically for performing code search, apply a more rudimentary feature set (*e.g.*, they do not employ neural network-based correspondence features) than we do, and

will generally not handle sub-code-block sized contexts, which proved important in our work.

We note that concurrent to this work, [134] also explored the problem of mining intent/code pairs from SO, identifying candidate code blocks of an intent using information from both the contextual texts and the code in an SO answer. Our approach, however, considers more fine-grained, sub-code-block sized candidates, aiming to recover code solutions that *exactly* answer the intent.

Finally, some work has asked programmers to manually write NL descriptions for code [67, 90], or vice-versa [123]. This allows for the generation of high-quality data, but is time consuming and does not scale beyond limited domains.

7.7 Threats to Validity

Besides threats related to the manual labeling (§ 7.3.2), we note the following overall threats to the validity of our approach:

Annotation Error: Our code mining approach is based on learning from a small amount of annotated data, and errors in annotation may impact the performance of the system (see Sections § 7.3 and § 7.5.4).

Data Set Volume: Our annotated data set contains mainly high-ranked SO questions, and is relatively small (with a few hundreds of examples for each language), which could penitentially hinder the generalization ability of the system on lower-ranked questions. Meanwhile, we used cross-validation for evaluation, while evaluating our mining method on full-scale SO data would be ideal but challenging.

Chapter 8

Conclusion

This thesis put forward a series of approaches to enable neural semantic parsers to handle diverse domain knowledge schema and varying complexity of meaning representations. We first proposed a general-purpose decoding model for constructing MRs using domain grammar as syntactic prior, and demonstrate it could handle a variety of domain-specific MRs, while being capable of scaling to generation of complex open-domain programs. Next, in the second part of the thesis, we studied the issue of cross-domain schema understanding in the context of semantic parsing over relational databases. We explored pre-training over massive corpora of Web tables as a universal receipy for learning representations of utterances and domain-specific database schemas. To mitigate the paucity of annotated database queries on new domains, we propose a unsupervised schema understanding model that grounds utterances to DB queries using external broad-coverage linguistic annotations as semantic scaffolds, which is further guided by the schema of domain databases. In the last part, we advanced this line of research on data-efficient semantic parsing with a semi-supervised learning method that outperforms purely supervised systems with additional unlabeled utterances. Finally, to speed-up the data acquisition process of utterances annotated with MRs, we proposed a machine-assisted mining pipeline for efficient collection of parallel training corpora.

In the final thesis, we will highlight our summary of contributions in this chapter, followed by discussions of future avenues.

Bibliography

[1] Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. Learning to generalize from sparse and underspecified rewards. In *ICML*, 2019. **??**, 4.5.1

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 38–49. ACM, 2015. 7.6

[3] Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of ICML*, 2015. 2.5, 7.1, 7.1

[4] David Alvarez-Melis and Tommi S. Jaakkola. Tree-structured decoding with doubly recurrent neural networks. In *Proceedings of ICLR*, 2017. **??**

[5] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transaction of ACL*, 2013. 3.1

[6] Yoav Artzi, Nicholas FitzGerald, and Luke Zettlemoyer. Semantic parsing with combinatory categorial grammars, 2013. 1

[7] Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. Broad-coverage CCG semantic parsing with AMR. In *Proceedings of EMNLP*, 2015. 2.5

[8] Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. Broad-coverage ccg semantic parsing with amr. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1699–1710, Lisbon, Portugal, September 2015. Association for Computational Linguistics. URL http://aclweb.org/anthology/D15-1198. 1

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*, 2015. 2.3.3, 2.3.3, 7.1, 7.4.2

[10] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016. 2.5

[11] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Trans. Software*

Eng., 11(11), 1985. 2.1

[12] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation for sembanking. In *Proceedings of LAW-ID@ACL*, 2013. 1, 3.1

[13] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*, 2017. 7.6

[14] I Beltagy and Chris Quirk. Improved semantic parsers for if-then statements. In *Proceedings of ACL*, 2016. 1, 7, 2.4.3, **??**, **??**

[15] Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *Proceedings of ACL*, 2014. 4.1

[16] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of EMNLP*, 2013. 4.1, 6.1

[17] Ben Bogin, Matt Gardner, and Jonathan Berant. Global reasoning over database structures for text-to-sql parsing. *ArXiv*, abs/1908.11214, 2019. **??**, 4.6

[18] Ben Bogin, Matthew Gardner, and Jonathan Berant. Representing schema structure with graph neural networks for text-to-sql parsing. In *Proceedings of ACL*, 2019. 4.1

[19] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Józefowicz, and Samy Bengio. Generating sentences from a continuous space. In *Proceedings of the SIGNLL*, 2016. 6.3.3

[20] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of CHI*, 2009. 2.1

[21] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of CHI*, 2010. 2.1

[22] Bob Carpenter. *Type-logical Semantics*. 1998. ISBN 0-262-03248-1. 6.4.3

[23] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009. 7.6

[24] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li,

Xiyou Zhou, and William Yang Wang. TabFact: A large-scale dataset for table-based fact verification. *ArXiv*, abs/1909.02164, 2019. 4.3.1, 4.5.1

[25] Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. Learning structured natural language representations for semantic parsing. In *Proceedings of ACL*, 2017. 6.4.3

[26] Yong Cheng, Wei Xu, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. Semi-supervised learning for neural machine translation. In *Proceedings of ACL*, 2016. 6.5

[27] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of CoNLL*, 2010. 2.5, 6.1

[28] Dipanjan Das and Noah A. Smith. Semi-supervised frame-semantic parsing for unknown predicates. In *Proceedings of HLT*, 2011. 6.5

[29] Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke S. Zettlemoyer, and Eduard H. Hovy. Iterative search for weakly supervised semantic parsing. In *Proceedings of NAACL-HLT*, 2019. **??**

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, 2019. 4.1, 4.3.2

[31] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of ACL*, 2016. 2.3.3, 2.4.3, **??**, 2.5, 3.3.1, **??**, 6.1, 6.4.1, **??**

[32] Li Dong and Mirella Lapata. coarse-to-fine decoding for neural semantic parsing. In *Proceedings of ACL*, 2018. 3.1, 4.6

[33] Xing Fan, Emilio Monti, Lambert Mathias, and Markus Dreyer. Transfer learning for neural semantic parsing. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*, 2017. 6.5

[34] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of NIPS*, 2016. 2.4.2, 7.5.1

[35] Yoav Goldberg. Assessing bert's syntactic abilities. *ArXiv*, abs/1901.05287, 2019. 4.1

[36] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of ACL*, 2016. 2.3.3, 6.3.1

[37] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation. In *Proceedings of ACL*, 2019. 4.1, 4.4.1, **??**, 4.6

[38] Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of ACL*, 2017. 6.3.3, 6.3.3

[39] Tihomir Gvero. *Search Techniques for Code Generation*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2015. 2.1

[40] Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. X-sql: reinforce schema representation with context. *ArXiv*, abs/1908.08113, 2019. 4.6

[41] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *ArXiv*, abs/1606.08415, 2016. 4.3.2

[42] Jonathan Herzig and Jonathan Berant. Decoupling structure and lexicon for zero-shot semantic parsing. *arXiv preprint arXiv:1804.07918*, 2018. 6.5

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997. 2.3.3

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

[45] Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. A comprehensive exploration on wikisql with table-aware word contextualization. *ArXiv*, abs/1902.01069, 2019. 4.1, 4.3.1, 4.5.1, **??**, 4.6

[46] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of ACL*, 2016. 2.5, 7.1, 7.1, 7.4.1, 7.5.1, 7, 7.5.1, 7.6

[47] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of ACL*, 2017. 6.1

[48] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *Proceedings of ACL*, 2016. 2.3.3, 2.5, 6.1

[49] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke S. Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. In *Proceedings of EMNLP*, 2019. 4.1, 4.3.2

[50] Rohit J. Kate and Raymond J. Mooney. Semi-supervised learning for semantic parsing

using support vector machines. In *Proceedings of NAACL-HLT*, 2007. 6.5

[51] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 7.5.1

[52] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. 6.1, 6.2.2

[53] Diederik P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Proceedings of NIPS*, 2014. 6.2.2

[54] Tomás Kociský, Gábor Melis, Edward Grefenstette, Chris Dyer, Wang Ling, Phil Blunsom, and Karl Moritz Hermann. Semantic parsing with semi-supervised sequential autoencoders. In *Proceedings of EMNLP*, 2016. 2.5, 6.1, 6.5

[55] Ioannis Konstas, Srinivasan Iyer, Mark Yatskar, Yejin Choi, and Luke Zettlemoyer. Neural amr: Sequence-to-sequence models for parsing and generation. In *Proceedings of ACL*, 2017. 6.3.1, 6.5

[56] Jayant Krishnamurthy, Oyvind Tafjord, and Aniruddha Kembhavi. Semantic parsing to probabilistic programs for situated question answering. In *Proceedings of EMNLP*, 2016. 2.5, 6.5

[57] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of EMNLP*, 2017. 4.1

[58] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *Proceedings of NAACL*, 2013. 2.5

[59] Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke S. Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the EMNLP*, 2013. 2.5

[60] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. A large public corpus of web tables containing time and context metadata. In *Proceedings of WWW*, 2016. 4.3.2

[61] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. From natural language specifications to program input parsers. In *Proceedings of ACL*, 2013. 2.5

[62] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *CoRR*, abs/1611.00020, 2016. 1

[63] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. Neural symbolic

machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of ACL*, 2017. 2.5

[64] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. In *Proceedings of NIPS*. 2018. 4.4.2, **??**, 4.6

[65] Percy Liang, Michael I Jordan, and Dan Klein. Learning dependency-based compositional semantics. In *Proceedings of ACL*, 2011. 2.5, 6.1

[66] Jindrich Libovický and Jindrich Helcl. Attention strategies for multi-source sequence-to-sequence learning. In *Proceedings of ACL*, 2017. 4.4.1

[67] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. 2018. 7.6

[68] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of ACL*, 2016. 2.1, 2.1, 2.3.3, 2.4.1, 2.3, 2.4.3, 2.4.3, 2.5, 3.1, **??**, 6.1

[69] Greg Little and Robert C. Miller. Keyword programming in java. *Autom. Softw. Eng.*, 16 (1), 2009. 2.1

[70] Chang Liu, Xinyun Chen, Eui Chul Richard Shin, Mingcheng Chen, and Dawn Xiaodong Song. Latent attention for if-then program synthesis. In *Proceedings of NIPS*, 2016. 2.5

[71] Weijie Liu, Peng Zhou, Zhe Zhao, Zhiruo Wang, Qi Ju, Haotang Deng, and Ping Wang. K-bert: Enabling language representation with knowledge graph. *ArXiv*, abs/1909.07606, 2019. 4.6

[72] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke S. Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692, 2019. 4.1

[73] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of EMNLP*, 2015. 3.2.3, 6.3.1, 6.4.3

[74] Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. In *Proceedings of ACL*, 2015. 2.4.3

[75] Chris J Maddison and Daniel Tarlow. Structured generative models of natural source code. In *Proceedings of ICML*, 2014. 2.5

[76] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating programming by example and natural language programming. In *Proceedings of AAAI*, 2013. 2.5

[77] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. In *Proceedings of NIPS*, 2017. 4.1

[78] Hongyuan Mei, Mohit Bansal, and Matthew R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of AAAI*, 2016. 2.5

[79] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional LSTM. In *Proceedings of CONLL*. 4.1

[80] Yishu Miao and Phil Blunsom. Language as a latent variable: Discrete generative models for sentence compression. In *Proceedings of EMNLP*, 2016. 6.1, 6.3.3, 6.3.3, 6.5

[81] Yishu Miao, Lei Yu, and Phil Blunsom. Neural variational inference for text processing. In *Proceedings of ICML*, 2016. 6.5

[82] Dipendra K. Misra and Yoav Artzi. Neural shift-reduce CCG semantic parsing. In *Proceedings of EMNLP*, 2016. 1, 2.5

[83] Dipendra Kumar Misra, Kejia Tao, Percy Liang, and Ashutosh Saxena. Environment-driven lexicon induction for high-level instructions. In *Proceedings of ACL*, 2015. 1

[84] Dana Movshovitz-Attias and William W Cohen. Natural language models for predicting programming comments. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 35–40. ACL, 2013. 7.6

[85] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of ICLR*, 2016. 2.5, **??**, 4.6, 6.5

[86] Graham Neubig. lamtram: A toolkit for language and translation modeling using neural networks. http://www.github.com/neubig/lamtram, 2015. 2.4.3, **??**

[87] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dy-

namic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017. 7.5.1

[88] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of ACM SIGSOFT*, 2013. 2.5

[89] Haoran Niu, Iman Keivanloo, and Ying Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, pages 1–33, 2016. 7.6

[90] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (T). In *Proceedings of ASE*, 2015. 2.4.1, 2.5, 3.3.1, 6.4.1, 7.6

[91] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *International Conference on Program Comprehension (ICPC)*, pages 63–72. IEEE, 2012. 7.6

[92] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016. 2.5

[93] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Proceedings of ACL*, 2015. 2.5, 4.1, 4.4.2, **??**, 4.6, 6.5

[94] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke S. Zettlemoyer. Deep contextualized word representations. In *Proceedings of NAACL*, 2018. 4.1

[95] Matthew E. Peters, Mark Neumann, IV RobertLLogan, Roy Schwartz, Vidur Joshi, Sameer Singh, and Noah A. Smith. Knowledge enhanced contextual word representations. In *Proceedings of EMNLP*, 2019. 4.6

[96] Hoifung Poon. Grounded unsupervised semantic parsing. In *Proceedings of ACL*, 2013. 6.5

[97] Hoifung Poon. Grounded unsupervised semantic parsing. In *ACL*, 2013. 5.2, 5.3

[98] Hoifung Poon and Pedro Domingos. Unsupervised semantic parsing. In *Proceedings of EMNLP*, 2009. 6.5

[99] Python Software Foundation. Python abstract grammar. https://docs.python.org/2/library/ast.html, 2016. 2.1, 6.3

[100] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic

parsers for if-this-then-that recipes. In *Proceedings of ACL*, 2015. 1, 2.4.1, ??, 3.1, 7.6

[101] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of ACL*, 2017. 1, 3.1, 3.3, 3.3.1, ??, 6.3.2, 6.4.1, ??

[102] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis. In *Proceedings of ICSE*, 2016. 2.5

[103] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *Proceedings of EMNLP*, 2016. 4.1

[104] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of IJCAI*, 2015. 2.5

[105] Siva Reddy, Mirella Lapata, and Mark Steedman. Large-scale semantic parsing without question-answer pairs. *Transactions of ACL*, 2014. 6.5

[106] Tianze Shi, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Oleksandr Polozov, and Weizhu Chen. Incsql: Training incremental text-to-sql parsers with non-deterministic oracles. *ArXiv*, abs/1809.05054, 2018. 4.6

[107] Aarti Singh, Robert D. Nowak, and Xiaojin Zhu. Unlabeled data: Now it helps, now it doesn't. In *Proceedings of NIPS*, 2008. 6.3.3

[108] Richard Socher, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of EMNLP*, 2011. 6.5

[109] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *International Conference on Program Comprehension (ICPC)*, pages 71–80. IEEE, 2011. 7.6

[110] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 7.5.1

[111] Mark Steedman. *The Syntactic Process*. 2000. ISBN 0-262-19420-1. 6.4.3

[112] Yu Su and Xifeng Yan. Cross-domain semantic parsing via paraphrasing. In *Proceedings of EMNLP*, 2017. 6.5

[113] Huan Sun, Hao Ma, Xiaodong He, Wen tau Yih, Yu Su, and Xifeng Yan. Table cell search for question answering. In *Proceedings of WWW*, 2016. 4.6

[114] Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. Semantic parsing with syntax- and table-aware SQL generation. In *Proceedings of EMNLP*, 2018. **??**, **??**, 4.6

[115] Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. Ernie: Enhanced representation through knowledge integration. *ArXiv*, abs/1904.09223, 2019. 4.6

[116] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of NIPS*, 2017. 4.3.1

[117] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of NIPS*, 2015. 2.3.3, 3.2.3

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

[119] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Margot Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *ArXiv*, abs/1911.04942, 2019. 4.1, **??**, 4.6

[120] Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. Pointing out SQL queries from text. Technical report, November 2017. URL https://www.microsoft.com/en-us/research/publication/pointing-sql-queries-text/. **??**

[121] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Xin Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding. *ArXiv*, abs/1807.03100, 2018. 4.6

[122] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of DSL*, 1997. 3.1, 6.3.2

[123] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Proceedings of ACL*, 2015. 4.6, 6.1, 7.6

[124] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. Building Bing Developer Assistant. Technical report, MSR-TR-2015-36, Microsoft Research, 2015. URL https://www.microsoft.com/en-us/research/publication/building-bing-developer-assistant/. 2.5, 7.6

[125] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist

reinforcement learning. *Machine Learning*, 1992. 6.3.3

[126] Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198, 2020. 5.1

[127] Edmund Wong, Jinqiu Yang, and Lin Tan. AutoComment: Mining question and answer sites for automatic comment generation. In *International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE, 2013. 7.1, 7.1, 7.4.1, 7.5.1, 7.6

[128] Edmund Wong, Taiyue Liu, and Lin Tan. CloCom: Mining existing source code for automatic comment generation. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389. IEEE, 2015. 7.6

[129] Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based structured prediction for semantic parsing. In *Proceedings of ACL*, 2016. 2.1, 2.5, 3.1, 6.1

[130] Weidi Xu, Haoze Sun, Chao Deng, and Ying Tan. Variational autoencoder for semi-supervised text classification. In *Proceedings of AAAI*, 2017. 6.5

[131] Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating structured queries from natural language without reinforcement learning. *arXiv*, abs/1711.04436, 2017. 1, 3.1, **??**, 4.6

[132] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: an analysis of Stack Overflow code snippets. In *Working Conference on Mining Software Repositories (MSR)*, pages 391–402. ACM, 2016. 7.5.1

[133] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Proceedings of NIPS*, 2019. 4.1

[134] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *WWW 2018: The 2018 Web Conference*, 2018. 7.6

[135] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of ACL*, 1995. 6.5

[136] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *Proceedings of ACL*, 2015. 2.5, 4.1

[137] Pengcheng Yin and Graham Neubig. a syntactic neural model for general-purpose code generation. In *Proceedings of ACL*, 2017. 3.1, 3.2.2, ??, 6.3.2, 6.4.1, ??

[138] Pengcheng Yin and Graham Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of EMNLP Demonstration Track*, 2018. 4.4.1, ??

[139] Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. Neural enquirer: Learning to query tables in natural language. In *Proceedings of IJCAI*, 2016. 2.5, 6.5

[140] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of MSR*, 2018.

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

[142] Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Annual Conference of the Association for Computational Linguistics (ACL)*, July 2020.

[143] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir R. Radev. TypeSQL: Knowledge-based type-aware neural text-to-sql generation. In *Proceedings of NAACL-HLT*, 2018. ??, ??, 4.6

[144] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir R. Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. In *Proceedings of EMNLP*, 2018. 4.6

[145] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of EMNLP*, 2018. 4.1, 4.4.1, 5.3

[146] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 38–42. IEEE Press, 2012. 7.6

[147] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of AAAI*, 1996. 4.1

[148] Luke Zettlemoyer and Michael Collins. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of UAI*, 2005. 3.1

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

[150] Rui Zhang, Tao Yu, He Yang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir R. Radev. Editing-based sql query generation for cross-domain context-dependent questions. *ArXiv*, abs/1909.00786, 2019. 4.1, **??**, 4.6

[151] Xiao Zhang, Yong Jiang, Hao Peng, Kewei Tu, and Dan Goldwasser. Semi-supervised structured prediction with neural crf autoencoder. In *Proceedings of EMNLP*, 2017. 6.5

[152] Yuchen Zhang, Panupong Pasupat, and Percy Liang. Macro grammars and holistic triggering for efficient semantic parsing. In *Proceedings of EMNLP*, 2017. **??**

[153] Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. Ernie: Enhanced language representation with informative entities. In *Proceedings of ACL*, 2019. 4.6

[154] Zhuosheng Zhang, Yu-Wei Wu, Hai Zhao, Zuchao Li, Shuailiang Zhang, Xi Zhou, and Xiaodong Zhou. Semantics-aware bert for language understanding. *ArXiv*, abs/1909.02209, 2019. 4.6

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

[156] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv*, abs/1709.00103, 2017. 3.1, 3.3.1, **??**, 4.6, 6.1

[157] Chunting Zhou and Graham Neubig. Multi-space variational encoder-decoders for semi-supervised labeled sequence transduction. In *Proceedings of ACL*, 2017. 6.1

[158] Xiaojin Zhu. Semi-supervised learning literature survey. Technical Report 1530, Computer Sciences, University of Wisconsin-Madison, 2005. 6.2.1, 6.5