Learning Structured Neural Semantic Parsers

Pengcheng Yin

CMU-LTI-21-014

August 16, 2021

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15123
www.lti.cs.cmu.edu

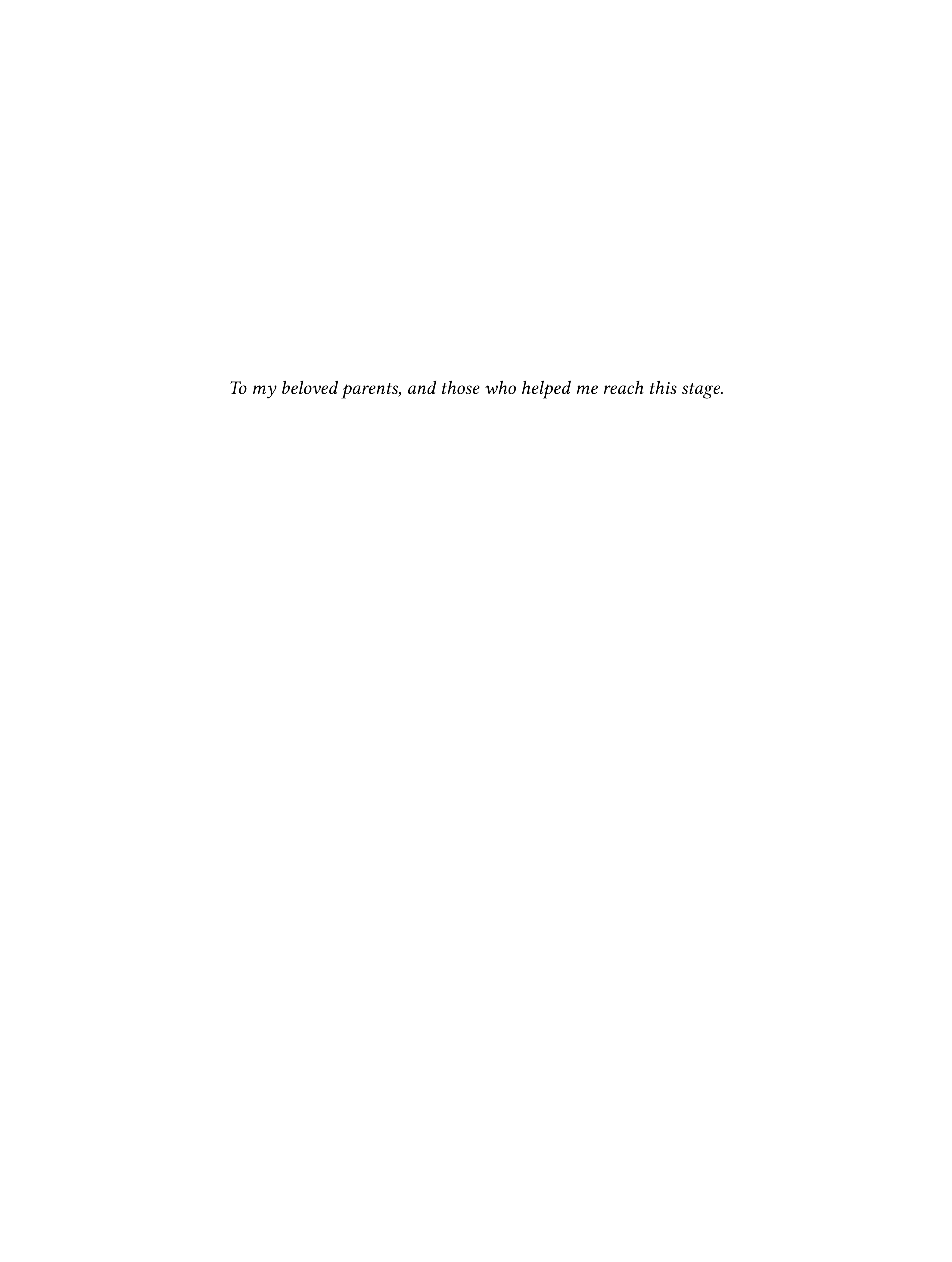
Thesis Committee:

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.

Copyright © 2021 Pengcheng Yin

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



To my beloved parents, and those who helped me reach this stage.

Abstract

Semantic parsing, the task of translating user-issued natural language (NL) utterances (*e.g.*, *Flights from Pittsburgh to New York*) into formal meaning representations (MRs, *e.g.*, an SQL database query or a Python program), 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, meaning representations typically exhibit strong syntactic structure, and are defined following domain-specific structured knowledge schemas (*e.g.*, a database schema or Python API specification), which is not easily captured by standard neural sequence transduction models. Neural semantic parsers 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 structure of meaning representations.

In this thesis, we explore developing neural semantic parsing models that could better capture the *structure* in various types of logical formalisms and knowledge schemas, while providing approaches to mitigate the cost of labeled data acquisition. The dissertation consists of three parts. The first part introduces a general-purpose parsing model with built-in syntactic knowledge of the grammatical structure of meaning representations. Next, in the second part, we investigate approaches to encode structured information in domain knowledge schemas (*e.g.*, database tables) useful to understand user-issued utterances. Specifically, we focus on grounding elements in the schema (*e.g.*, columns like `departure_city` in database tables, or functions like `GetFlight(from=GetCityByName(·))` in API specifications) to their corresponding NL constituents (*e.g.*, *from Pittsburgh*) in utterances. 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.

Acknowledgments

Five years ago, when I first arrived in Pittsburgh on a gloomy night after a long-haul flight, with two large trunks packed with everything including pillows and quilts, I had no idea about the life ahead of me. Five years later, after spending a wonderful and rewarding time at CMU, I have a clearer vision of pursuing a career in research, and feel grateful and humbled by the help I received along the journey.

This dissertation would not have been possible without the continued support and guidance from my advisor, Graham Neubig. His enthusiasm and commitment towards high-quality research and thoughtfulness for students have always inspired me to improve myself and achieve more. I wish I could follow his passion and tireless attitude towards work and become a good mentor in the future. I would also like to thank my thesis committee members, Tom Mitchell, Luke Zettlemoyer, and Yonatan Bisk, for their insightful feedback.

I am also grateful for the companionship and support I received from members of the NeuLab: Junjie Hu, Pengfei Liu, Shuyan Zhou, Cindy Wang, Junxian He, Frank Xu, Zhengbao Jiang, Zecong Hu, Chunting Zhou, Mengzhou Xia, Hao Zhu, Paul Mitchel, Patrick Fernandes, Danish, Hiroaki Hayashi, Shruti Rijhwani, Aditi Chaudhary, Lucio Dery, John Wieting, Kayo Yin. Junjie has been my roommate for the past five years, who taught me stuff from multilingual machine translation to cooking beef steaks and stock market 101. Junxian, Cindy, and Chunting have been especially close colleagues for our collaboration during my early years in CMU, and also for sampling nice restaurants in Pittsburgh for Friday gatherings. I am also grateful to Junxian, who taught me most of my knowledge in probabilistic graphical models and Chinese talent shows, and Zecong, who generously offered his office space during my MLT and taught me how to write Pythonic code. I also had great pleasure working with some of our lab members: Shuyan, Alex, Zhengbao, Frank, and Hao. I wish I could have contributed more and became a better mentor in the future. I am also thankful to Pengfei for sharing his insights about the field and advice on mentorship. I wish I could have learned these earlier. Finally, I want to thank my friends in and outside of LTI for our research discussions, game nights, and holiday gatherings, including Zihang Dai, Zhiting Hu, Zhilin Yang, Qizhe Xie, Guokun Lai, Jingzhou Liu, Xin Qian, Jiarui Xu, Haohan Wang, Diyi Yang, Xuezhe Ma, Hector Liu, Di Wang, Keyang Xu, Lu Jiang, Wei-Cheng Chang, Tao Yu, and also Bhuwan Dhingra for sharing this beautiful LaTeX template.

During my time at CMU, I spent three wonderful summers interning at Microsoft Research,

Facebook AI and Semantic Machines. I am grateful to my awesome mentors: Miltos Allamanis, Marc Brockschmidt, Scott Yih, Sebastian Riedel, Jacob Andreas, Yu Su, Hao Fang, Sam Thomson, Antonios Platanios, Adam Pauls. They provided invaluable guidance during those internships. I am especially thankful to Miltos and Marc — my mentors for the first internship in my graduate study. Since I was working on something not related to NLP, they taught me everything from designing experiments to framing a compelling story when writing papers. This internship experience has broadened my horizon in learning for code, and has a tremendous influence on my future research topics. The work on pre-trained language models in Chapter 5 is a collaboration with Scott and Sebastian at Facebook AI. I am grateful to Sebastian for his support to this project and helping me with idea brainstorming, and Scott for his deep insights in semantic parsing and feedback regarding modeling details. The supervised attention project in Chapter 6 is joint work with researchers at Microsoft Semantic Machines. I want to thank Adam for establishing continued collaboration after the internship. I am deeply grateful to Jacob, whose insights and research taste in language understanding have inspired me to keep pushing forward and think more. I also thank Avi Sil from IBM research for collaborating on the zero-short semantic parsing project in Chapter 7.

Doing a Ph.D. is a tremendous endeavor. The last two years amid the COVID-19 pandemic have been especially challenging for me. I owe my advisor and those who offered help immense gratitude for your understanding, patience, and support in this difficult time. I wish I could provide the same assistance to my friends and future students when they are in need.

I am also deeply grateful to my family members, my parents Changzi Yin and Yuqin Zhao, for raising me with your unconditional love, understanding, and support. I would also like to thank my girlfriend and Coke for your companionship along the way.

Finally, I thank you for stopping by and reading my thesis. Most of its contents would probably become outdated within just a few years. Still, I hope this thesis could help you get a grasp of semantic parsing research between 2016 $\sim$ 2021.

Contents

1 Introduction **1**

1.1 Thesis Overview . . . 3

2 Background **7**

2.1 Logical Formalisms and Parsing Algorithms . . . 7

2.2 Learning Paradigms . . . 15

2.3 Summary . . . 22

I Structured Program Generation Models **25**

3 Syntactic Models for Code Generation **27**

3.1 Overview . . . 27

3.2 The Code Generation Problem . . . 29

3.3 Grammar Model . . . 30

3.4 Experimental Evaluation . . . 35

3.5 Related Work . . . 42

3.6 Summary . . . 42

4 Generalized Parsing Framework **45**

4.1 Overview . . . 45

4.2 Methodology . . . 46

4.3 Experiments . . . 51

4.4 Summary . . . 54

II Understanding Structured Domain Schemas **57**

5 Pretraining for Structured Data Understanding **59**

5.1 Overview 59

5.2 Background 61

5.3 TaBert: Learning Joint Representa- tions over Textual and Tabular Data 63

5.4 Example Application: Semantic Parsing over Tables 66

5.5 Experiments 68

5.6 Related Works 73

5.7 Summary 74

6 Ground Utterances to Schema with Structured Inductive Bias **75**

6.1 Overview 76

6.2 Span-level Supervised Attention 77

6.3 Experiments 79

6.4 Summary 84

7 Ground Language to Schema without Labeled Data **85**

7.1 Overview 86

7.2 Zero-shot Semantic Parsing via Data Synthesis 87

7.3 Bridging the Gaps between Canonical and Natural Data 89

7.4 Experiments 91

7.5 Related Work 99

7.6 Summary 99

III Data Efficient Approaches **101**

8 Semi-supervised Learning **103**

8.1 Overview 103

8.2 Semi-supervised Semantic Parsing 105

8.3 StructVAE: VAEs with Tree-structured Latent Variables 106

8.4 Experiments 109

8.5 Related Works 117

8.6 Summary 117

9 Speeding Up Data Acquisition **119**
9.1 Overview . . . 119
9.2 Problem Setting . . . 122
9.3 Manual Annotation . . . 123
9.4 Mining Method . . . 126
9.5 Evaluation . . . 130
9.6 Related Work . . . 138
9.7 Threats to Validity . . . 139
9.8 Summary . . . 139

10 Conclusions and Future Directions **141**
10.1 Summary of Contributions . . . 141
10.2 Open Problems and Future Directions . . . 144

A Appendix for Chapter 7 **149**
A.1 Synchronous Grammar . . . 149

Bibliography **155**

Chapter 1

Introduction

Semantic parsing studies the task of transducing natural language (NL) utterances into structured formal meaning representations (MRs, *e.g.*, first-order logic or computer programs) [8]. 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 λ-calculus [177, 178] and abstract meaning representations [9, 12, 137]. 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 [184, 235]. Fig. 1.1 illustrates two representative scenarios of task-oriented semantic parsing. The first example shows a semantic parser as an interface to a flight booking system, where a user's utterance is converted to an SQL query executable on a database of flight information [81]. The second example features code generation from a programmer's natural language intent, where the intent is directly translated into source code written in general-purpose programming languages like Python (*e.g.*, GitHub Copilot [30]). Indeed, task-oriented semantic parsers play a key role in building natural language interfaces to computational systems, with other applications like conversational virtual assistants [38, 43, 65, 166], robot instruction following [7, 19], question answering over knowledge bases [18, 27], and semantic search [15]. Those systems also power some of the most popular commercial AI products, like Wolfram Alpha, Amazon Alexa, Apple Siri and Google Assistant.

Recent years have witnessed the proliferation of development in neural network-based semantic parsers [50, 51, 64, 84, 157, 188, 202, 207, 244, *inter alia*]. The simplest form of such neural parsers is based on attentional sequence-to-sequence models [10, 123], where utterances and meaning representations (*e.g.*, those in Fig. 1.1) are treated as tokenized sequences, and the model predicts MR tokens given the input utterance [50, 81, 84]. Despite the effectiveness of

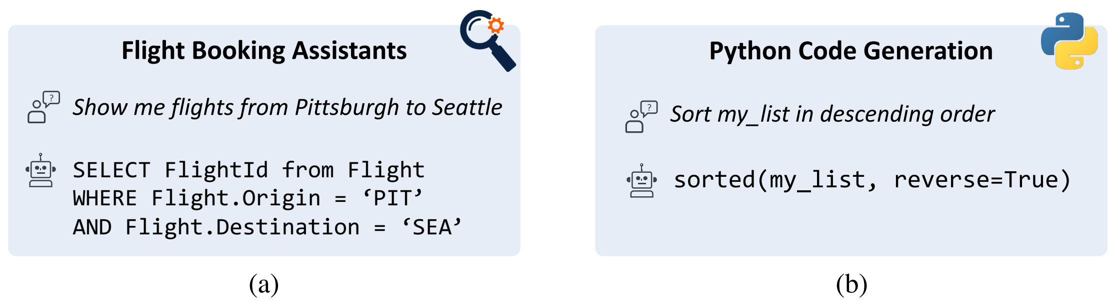


Figure 1.1: Example applications of task-oriented semantic parsing.

those semantic parsers based on vanilla neural sequence transduction models, they fail to capture the underlying *structure* of the task. Specifically, in this thesis, we consider the following two types of structure.

Syntactic Structure in Meaning Representations Meaning representations exhibit rich syntactic structure. For example, the SQL query and Python code in Fig. 1.1 are defined following the grammar specifications of their programming languages (*e.g.*, Fig. 1.2, Part I). Generating MRs using vanilla neural sequence decoders without modeling the syntactic knowledge could potentially yield grammatically incorrect outputs. Such models could also be data hungry, requiring more data to learn the underlying grammatical structure of MRs [84]. Therefore, to capture the grammar knowledge and ensure the syntactic correctness of generation results, a neural semantic parser often employs specifically-designed components tailored to the structure of task-dependent MRs. As an example, a parser that generates SQL queries over databases would use dedicated neural modules to predict columns in the `SELECT` clause and conditional expressions in the `WHERE` clause when generating an SQL query [207, 244]. However, modeling syntactic structure using dedicated neural components would easily become intractable for applications with complex MRs, like Python code generation (Fig. 1.1b). Such strong reliance of a parser's neural architecture on the underlying syntactic structure of MRs renders designing neural semantic parsers a non-trivial domain-specific endeavor.

Structure in Domain Knowledge Schemas To understand user-issued utterances in a particular domain, semantic parsers often need to encode necessary task-dependent knowledge schemas. Such domain knowledge is also typically defined in a structured fashion. As illustrated in part II of Fig. 1.2, the text-to-SQL parser in Fig. 1.1a will need to process a relational database schema of flight information, consisting of structured DB tables like `Flight`. Additionally, the code generation system in Fig. 1.1b could potentially rely on an API specification defining signatures of domain functions (*e.g.*, `sorted(arr, reverse=bool)`) and their ex-

emplar natural language intents (*e.g.*, *sort an array* arr). Such schematic specification of custom functions and their NL query patterns is adopted by commercial semantic parsing systems to allow developers add new functionalities.[1] When encoding those domain knowledge schemas, a key issue is to ground NL constituents in utterances (*e.g.*, "*from Pittsburgh*" in Fig. 1.1a, or "*in descending order*" in Fig. 1.1b) to their corresponding elements in the schema (*e.g.*, the table column `Origin`, or the named argument `reverse=True`), which is beneficial for semantic parsers to infer the meaning representations related to these NL constituents [107, 157, 188, 227].

Neural Semantic Parsers are also Data Hungry Besides the lack of modeling structure in meaning representations and domain schemas, another issue pertaining to the design of neural semantic parsers is the cost of data acquisition. Classical supervised learning of neural semantic parsers requires large amounts of parallel training data consisting of NL utterances with manually annotated MRs [84]. However, understanding task-specific MRs requires strong domain knowledge, and its annotation can be expensive, cumbersome, and time-consuming [18]. Therefore, the limited availability of parallel data has become the bottleneck of existing supervised-based models.

The rich structure in meaning representations and domain knowledge schemas are not easily captured by semantic parsers based on standard neural sequence-to-sequence models. This issue, together with the cost of data annotation, has significantly impeded the process of applying neural semantic parsing systems to a broader range of applications, especially those with complex structure of domain knowledge and formalisms of meaning representations.

1.1 Thesis Overview

In this thesis, we put forward a series of methods to tackle the challenges in modeling structure in meaning representations and domain knowledge schemas, as well as to mitigate the cost of annotating parallel training data. Fig. 1.2 presents an illustrative overview of the thesis. Specifically, we first develop syntax-driven parsing models that generate MRs following their grammatical structure, reducing the output space of valid MRs while ensuring their syntactic correctness (Part I). Next, we explore modeling approaches that could encode and understand structured knowledge schemas of target domains (Part II). Finally, to reduce the cost of acquiring labeled training data, we improve the data-efficiency of neural semantic parsers using

[1] For example, Google Assistant supports this feature using custom intents.

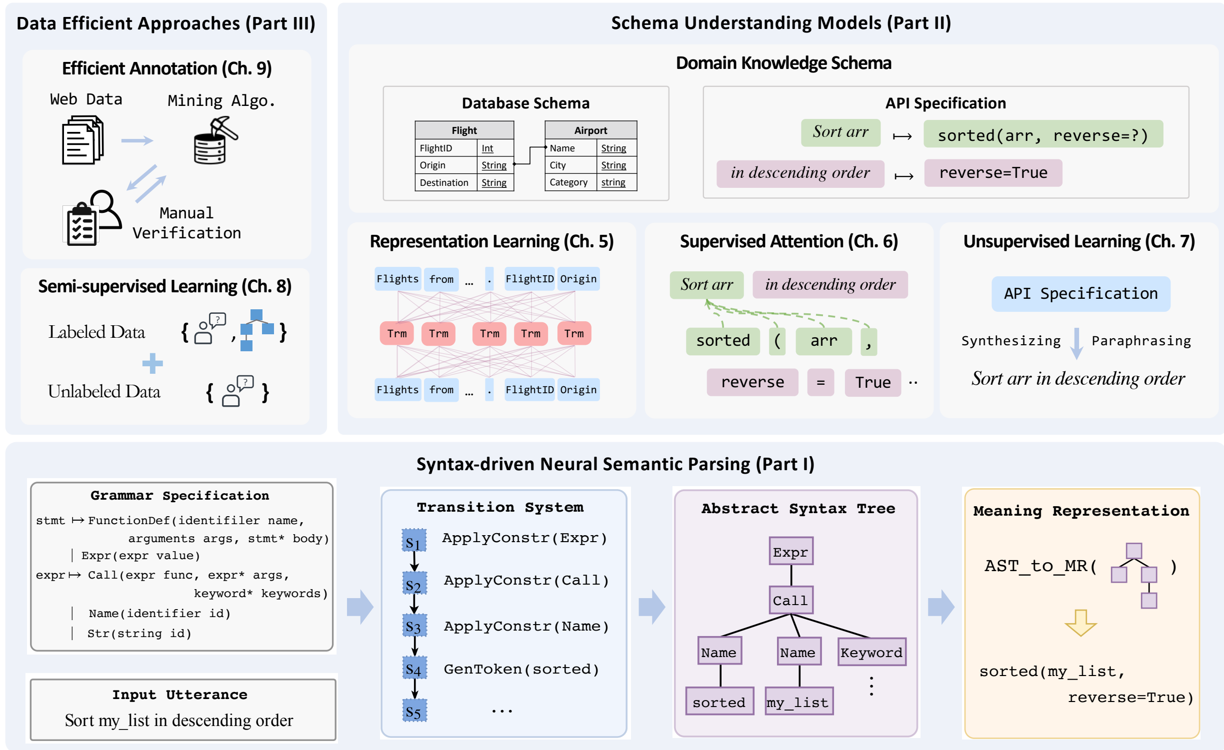


Figure 1.2: Roadmap of the thesis covering the entire life cycle of neural semantic parsing. First, Parallel training data of utterances and MRs is collected using an efficient machine-in-the-loop pipeline (Chapter 9). Next, schema understanding models aim to encode domain knowledge schemas useful to interpret the semantics of utterances, such as database tables and API specifications (Part II). Those domain schemas can be encoded using pre-trained encoders (Chapter 5), or used to improve prediction of meaning representations by modeling the alignments between NL phrases and MR spans in API specifications (Chapter 6). The schemas could also be used to synthesize parallel training data in an unsupervised fashion, which helps parsers implicitly learn schema information when trained on the synthetic data (Chapter 7). Given the domain knowledge and an input utterance, a syntactic driven parsing model translates the utterance into an MR represented by its abstract syntax tree, using a transition system that encodes the domain grammar as prior syntactic knowledge (Part I, Chapters 3 and 4). Finally, the performance of a semantic parser could be further improved using semi-supervised learning with additional unlabeled utterances (Chapter 8).

semi-supervised learning, while expediting the data acquisition process with a semi-automatic machine-in-the-loop annotation framework (Part III). The thesis consists of three parts. A detailed overview of each part is as follows:

Part I Structured Program Generation Models In this part, we build general-purpose structured semantic parsing networks that capture the syntax of meaning representations as prior knowledge. As illustrated in Fig. 1.2(Part I), instead of designing specialized parsing modules to reflect the structure of domain grammars, we put forward generalized parsing models that abstract the domain-specific syntactic formalisms of meaning representations using abstract syntax trees (ASTs), which serve as a general-purpose syntax-agnostic form of meaning representation. Specifically, we design a unified parsing model that transduces NL utterances into domain-general ASTs. This process is gauged by the underlying domain-specific 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 3). To demonstrate the approach could be generalized to different paradigms of MRs and tasks, we further develop a general-purpose parsing framework (Chapter 4), which provides a unified interface to specify task-dependent grammars for the syntax-driven parsing model, while remaining flexible enough 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 Understanding Structured Domain Schemas In this part, we explore methods to encode structured domain knowledge schemas essential to understand the semantics of user-issued utterances. We start with a Transformer model [185] for learning joint contextual representations of utterances and (semi-)structured database tables (Chapter 5). This model is pre-trained on massive parallel corpora of Web tables and their NL contexts, which implicitly captures the general alignments between NL constituents in utterances (*e.g.*, *from Pittsburgh*, Fig. 1.1a) and schema elements like table columns (*e.g.*, `Origin`) using self-attention. Next, in Chapter 6 we generalize the notion of knowledge schema from database tables to specifications of domain function signatures (*e.g.*, `sorted(`[`arr`]`, reverse=`[`bool`]`)`, Fig. 1.1b) and their exemplar NL intents (*e.g.*, *Sort an array* [*arr*]), and attempt to explicitly model the grounding of phrases in utterances to those schema elements, such as functions (*e.g.*, "*Sort an array* [*arr*]"↔`sorted(`[`arr`]`, `[`?`]`)`) and their arguments (*e.g.*, "*in descending order*"↔`reverse=True`). We propose a supervised attention mechanism, encouraging the model to predict MR segments (*e.g.*, `reverse=True`) using aligned spans in the utterance (*e.g.*, *in descending order*), as shown in Fig. 1.2(Ch. 6). Finally, to quickly adapt a semantic parser to understand knowledge schemas presented in emerging new domains, in Chapter 7 we lift the requirement of having annotated

MRs for training the schema understanding model using zero-shot data synthesis, where we automatically generate compositional utterances (*e.g.*, *Sort arr in descending order*) labeled with MRs from a grammar specification that defines canonical NL intents and their MR implementation (*e.g.*, "*Sort arr*"↔`sorted(arr, ?)`, "*in descending order*"↔`reverse=True`, as in Fig. 1.2). We also study approaches to bridge the gap between those synthetic utterances and real-world user-issued ones using grammar engineering and paraphrasing by pre-trained language models. The resulting model achieves competitive performance without using annotated training data.

Part III Data Efficient Approaches 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 8). 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 (Stack Overflow), and propose a machine-in-the-loop approach for cost-effective collection of parallel corpora, where annotators work with a mining model that proposes candidate examples to verify and revise, whose performance is iteratively improved using the newly annotated data (Chapter 9).

* * *

To summarize, this thesis put forward neural semantic parsers that better capture the structure in meaning representations and knowledge schemas, as well as data annotation and learning paradigms that are more cost effective. Before diving into the details of those proposed approaches, we first present a survey of related research topics in Chapter 2.

Chapter 2

Background

In this chapter, we present a holistic overview of research in semantic parsing related to this thesis. We survey existing approaches from two perspectives. Specifically, §2.1 describes different formalisms of meaning representations and their representative semantic parsing algorithms, which is closely related to our proposed parsing and schema understanding models in Parts I and II. Next, §2.2 outlines learning paradigms to train semantic parsers with various types of supervision, which inspire our data efficient learning methods in Part III. When discussing those related works, we also try to draw the connections between existing approaches and our contributions made in this thesis.

2.1 Logical Formalisms and Parsing Algorithms

Central to a semantic parser is the logical formalism that the parser adopts to represent meaning of natural language utterances. We start with a brief review of different kind of logical formalisms for meaning representation, together with the parsing algorithms that convert utterances into MRs.

2.1.1 Classical Representations: λ-calculus and Others

Lambda Calculus Lambda (λ)-calculus is a general-purpose formal symbolic system to represent computation, and has been adopted as a commonly-used formalism to represent the meaning of NL sentences for semantic parsing [28]. Fig. 2.1 gives example MRs defined in λ-calculus. The basic constructs of λ-calculus expressions are constants, like entities (*e.g.*, `pittsburgh`) and functions (*e.g.*, `flight`, `from`). The simplest form of functions map entities to binary

$\boldsymbol{u}_1$: *Show me flights from Pittsburgh to Seattle.*

$\boldsymbol{z}_1$: $\lambda x.\texttt{flight}(x) \land \texttt{from}(x,\texttt{pittsburgh}) \land \texttt{to}(x,\texttt{seattle})$

$\boldsymbol{u}_2$: *What is the largest state?*

$\boldsymbol{z}_2$: $\text{argmax}\ (\lambda x.\texttt{state}(x), \lambda x.\texttt{size}(x))$

Figure 2.1: Example utterances with λ-calculus meaning representations. Examples are adapted from Zettlemoyer and Collins [236, 237]

boolean values (*e.g.*, $\texttt{flight}(x)$ in $\boldsymbol{z}_1$ returns `True` if x is a flight, or `False` otherwise) or numbers (*e.g.*, $\texttt{size}(x)$ in $\boldsymbol{z}_2$). More complex functions can be composed via logical connectors like conjunction ($\land$). A λ-expression, like those in Fig. 2.1, defines a function that takes an input x and returns the output value following computation specified by the function body. Some functions also accept λ-expressions as input arguments. For instance, the `argmax` operator in Fig. 2.1 returns the set of entities that are states (state(x) evaluates to `True`) and have the largest area (size(x) is the highest).

Combinatory Categorial Grammar Combinatory Categorial Grammar (CCG, [177, 178]) is a commonly used grammar formalism to parse utterances into λ-calculus logical forms [236]. A CCG consists of a **lexicon** specifying the mapping of NL constituents (*e.g.*, the noun *Pittsburgh* and the preposition *from*) to elements in logical forms (*e.g.*, the constant entity `pittsburgh` and the lambda function $\lambda x \lambda y.\texttt{from}(x,y)$). Those logical elements for lower-level constituents can be composed recursively to form more expressive expressions. For example, the constant denoting *Pittsburgh* could substitue the variable y in the lambda function that represents *from*, yielding a new lambda function $\lambda x.\texttt{from}(x,\texttt{pittsburgh})$ for the phrase *from Pittsburgh*. A full introduction of CCG is beyond the scope of this thesis, and interested readers are referred to Artzi et al. [8] for a comprehensive review.

Semantic Parsing using CCGs Statistical semantic parsers typically use probabilistic CCGs. Similarly to probabilistic context-free grammars (PCFGs), given an utterance $\boldsymbol{u}$, probabilistic CCGs define the conditional distribution over MRs $\boldsymbol{z}$ and the derivation d recording the composition of $\boldsymbol{z}$ conditioned on $\boldsymbol{u}$, *i.e.*, $p(\boldsymbol{z}, d|\boldsymbol{u})$. $p(\boldsymbol{z}, d|\boldsymbol{u})$ can be parameterized using log-linear models, with features defined over triples $\langle \boldsymbol{z}, d, \boldsymbol{u} \rangle$ [41]. Since the features only depend on local segments of $\langle \boldsymbol{z}, d, \boldsymbol{u} \rangle$ (*e.g.*, a binary feature denoting if `from` in $\boldsymbol{z}$ co-occurrs with *from* in $\boldsymbol{u}$ in training data), inference can be performed using dynamic programming, similar to CKY parsing with PCFGs. As discussed above, central to semantic parsing with (probabilistic) CCGs is the

lexicon, which specifies the alignments between NL constituents and logical predicates. A core research issue is therefore learning the lexicon together with the model parameters (*e.g.*, the feature vector in log-linear models). Early work assumed access to derivations d, which are costly to label [41]. A more practical solution is learning lexicons only using utterances and MRs $\langle \boldsymbol{u}, \boldsymbol{z} \rangle$, where derivations are modeled as latent variables. Along this line, Zettlemoyer and Collins [236] proposed a hard EM algorithm, which interleaves between lexicon induction, and parameter update using the newly induced lexicon. In the lexicon induction step, a seed lexicon is first constructed by over-generating all the possible (and redundant) lexical entries pairing NL phrases with logical entries, a more compact lexicon is then induced from the highest-scoring derivations inferred by the current model parameter. Zettlemoyer and Collins [237] further introduced more relaxed CCG combination rules to parse utterances with free-form language styles, such as flexible word ordering, together with an online learning algorithm that is more efficient (update model parameters per example instead of per batch). Later work focused on learning more compact lexicons, creating lexical entries by splitting MRs top-down instead of exhaustive pairing [98], or factorization of lexical entries to allow for more information sharing [99].

Neural Approaches Like other sequence generation tasks in NLP, neural sequence-to-sequence models [10] have been applied to semantic parsing around 2016, where meaning representations like λ-calculus logical forms are decoded from recurrent neural networks as tokenized sequences [50, 81, 84]. To capture structure in MRs, Dong and Lapata [50] propose a sequence-to-tree model that generates logical forms (*e.g.*, λ-calculus) following their tree-structured hierarchy, where the prediction of child nodes is dependent on the information (*e.g.*, RNN hidden states) from their parent nodes. Besides being tree-structured, another strong prior for MRs is that they are defined following a pre-specified grammar. To model the grammatical information in decoding, Xiao et al. [202] put forward a syntax-based decoder, which generates MRs by predicting the sequence of production rules used to construct an MR following the domain grammar. This approaches guarantees the syntactic correctness of model outputs, since the space of valid continuating predictions is constrained by the grammar. In this sense, it is related to our code generation model in Part I which also generates MRs using productions, while we consider generating more complex programs in general-purpose programming languages, with a model output space augmented with other actions besides productions to support generation of programs in different languages and formalisms (see §2.1.2, more in Part I). Apart from constraining the decoding space with grammar rules, other approaches also explored us-

ing typing information for constrained decoding [96], reminiscent of the pre-neural type-driven parsing algorithms like Zhao and Huang [243].

Language Compositionality and Compositional Generalization As an interesting side note, natural utterances can be highly compositional, and human can easily understand the meaning of a compositional utterance (*e.g.*, *Flights from Pittsburgh to the host city of ACL 2022*) by interpreting the meaning of its simpler segments (*e.g.*, "*the host city of ACL 2022*", and "*Flights from Pittsburgh to* ?"). Indeed, compositionality is believed to be the key characteristic of human minds to attain strong generalization ability from limited data [103]. Classical semantic parsing methods based on CCG could also generalize well to inputs with novel compositional patterns, as such models are equipped with explicit inductive biases about language compositionality, which first maps NL phrases into atomic logical constituents according to the lexicon, and then compositionally builds up the meaning representation of a larger span from the MRs of its child spans. However, end-to-end neural semantic parsers based on sequence-to-sequence networks without such inductive bias could fail spectacularly when extrapolating to compositionally novel examples [101]. In Chapter 6, we will attempt to tackle the problem of improving **compositional generalization** of neural sequence transduction models using supervised attention [133], which injects lexicon-level NL-MR alignments into recurrent decoding networks. Please refer to Chapter 6 for a detailed review of compositional generalization and our proposed method.

Other Variants

As a general-purpose formalism of meaning representation, λ-calculus is adopted by many task-oriented semantic parsers. However, certain applications might not require the full generality of λ-calculus, and it makes sense to develop task-oriented MRs based on λ-calculus. Here we briefly review some representative examples.

Lambda Dependency-based Compositional Semantics (λ-DCS) λ-DCS is a formal language built for semantic parsing over structured knowledge bases [112], where the parser translates NL questions into executable database queries (*e.g.*, SQL or SPARQL queries. See Fig. 1.1a for an example). Utterances in this domain usually concern querying over a set of entities (*e.g.*, flights). λ-DCS is a simplified version of λ-calculus, which models sets of entities as first-class citizens. For instance, the λ-DCS representation for $\boldsymbol{z}_1$ in Fig. 2.1 is defined as

$$\texttt{from.Pittsburgh} \sqcap \texttt{to.Seattle}$$

Here, `from` is a binary predicate denoting the set of pairs of flights and their departure cities. When it is joined with a unary predicate `Pittsburgh` representing the single entity, the resulting expression `from.Pittsburgh` denotes the set of flights from Pittsburgh. The logical form `to.Seattle` is defined similarly. Similar to the dependency-based compositional semantics formalism [113], λ-DCS eliminates explicit usage of variables in λ-calculus, making the representation more conscience.

Query Graphs Query graphs are another form of meaning representations targeted for knowledge-based question answering (KBQA) [163, 216]. The intuition behind the design of query graphs is that information stored in structured knowledge bases is typically represented as graphs (*e.g.*, Pittsburgh ←from— UA123 —to→ Seattle; Pittsburgh ←from— UA456 —to→ Seattle). Therefore question answering over such KBs can be reformulated as a sub-graph matching problem, where answers are obtained by matching the KB graph with the query graph parsed from utterances (*e.g.*, Pittsburgh ←from— ? —to→ Seattle, [247]). Query graphs can be viewed as a graphic representation of λ-calculus, and are also similar to λ-DCS since they both capture joining operations over sets of entities.

2.1.2 Programming Languages as Meaning Representations

Besides representing utterance meaning using specialized formalisms like λ-calculus, another stream of research attempts to translate utterances into executable computer programs in modern programming languages (PLs), which is also a major contribution of this thesis. In this section, we survey two distinctive examples of using PLs as MRs — natural language interface to databases (NLIDBs), where utterances are parsed to executable DB queries like SQL (Fig. 1.1a), and code generation, where a programmer's NL intent is translated to code in general-purpose programming languages (*e.g.*, Python, Fig. 1.1b).

Natural Language Interface to Databases (NLIDBs)

Early versions of NLIDBs relied on rule-based pattern-matching (*e.g.*, *from $city_name*↦`Departure_City =$city_name`) or hand-crafted grammars (similar to CCGs in §2.1.1) to convert utterances to database queries. Most early works were from the database research community, with the goal of developing systems with high precision but could only handle queries with simple intents (*e.g.*, [196, 201], refer to Androutsopoulos et al. [6] for a comprehensive survey). Later on, NLP researchers attempted to build systems using statistical machine learning, aiming to generalize NLIDB from querying domain-specific relational databases to answering complex, compo-

sitional NL questions over large-scale open-domain databases of world knowledge (*e.g.*, Freebase, as in [17, 18, 27, 100]). Therefore, many works in this line fall to the topic of knowledge-based question answering. In short, these models typically employed a feature-based ranking approach, where multiple hypothesized DB queries are first generated from input NL questions using CKY-style parsing [13, 18], rule-based pattern matching [54], or query graph generation (§2.1.1). Next, a ranking model scores the generated hypotheses using features extracted from the question and the hypothesis (*e.g.*, a binary indicator feature that returns 1 *iff* the question token *from* is mapped to the DB column `Departure_City`), and returns the highest-scoring query. A core step in this process is entity and schema linking, where entity mentions (*e.g.*, *Pittsburgh*) and relational phrases (*e.g.*, *from*) in utterances are grounded to the corresponding DB entities and relations [107].

Intermediate Meaning Representations Mapping utterances to structured database queries is not a straight-forward task, as NL utterances and DB queries exhibit different syntax and structures. Therefore, most NLIDB systems capture utterance semantics using intermediate form of meaning representations, and then convert the intermediate MR into DB queries. Early rule-based systems rely on hand-crafted intermediate semantic representations [6]. λ-calculus (and its variants) is widely used as a form of intermediate MR [18, 27] owing to its matured parsing algorithms (*e.g.*, CCG or λ-DCS, as in §2.1.1). The query graphs outlined in §2.1.1 are another popular choice due to their homogeneous structure with bankend graph KBs. Other approaches also use simplified query languages as intermediate MRs. For example, Guo et al. [64] proposed a syntactically simplified version of SQL queries without `FROM` and `JOIN` clauses, which could be deterministically inferred for databases with simple schemas [229]. However, even parsing to intermediate MRs could still be non-trivial. As discussed in Yin [217], free-form natural language utterances are highly flexible, and the same intents could be expressed in different styles (*e.g.*, "*What did Barack Obama do before he was a president?*", and "*Barack Obama's occupation before he took office*"). On the other hand, information stored in databases are highly structured, typically following a pre-defined schema (*e.g.*, a table recording histories of job titles and start/end dates). To close the gaps between the flexible NL space and the rigid DB schema, Kwiatkowski et al. [100] proposed decoupling schema matching from semantic parsing, where utterances are first parsed to an intermediate MR that is not grounded to the DB, and a separate matching model is learned to link entity mentions and relations in the MR to DB entries. Other works also attempted to close the gap by augmenting the structured DB schema with semi-structured natural language assertions (*e.g.*, ⟨*Barack Obama*, *was*, *a senator*, *before he became*

president⟩) that can be more easily matched with utterances [54, 221].

Neural Models Recent advances in NLIDBs have been largely attributed to the success of neural network-based models. The simplest form of such models uses standard attentional neural sequence-to-sequence networks, where tokenized utterances are encoded using a bidirectional LSTM network, and SQL queries are predicted using another LSTM decoder network [81]. Later works have been mainly focused on improving the encoder and decoder networks.

To improve the encoding network, the field has explored incorporating contextual information in the DB schema useful to infer queries. Such contextual information could be broadly divided into two categories: (a) information of the database schema itself, such as the names of tables and the columns of each table; and (b) alignments between utterance tokens and elements in the schema (*i.e.*, schema linking, such as "*What is the* $\underline{\textit{GDP}}_{\textit{column:}\texttt{gross_domestic_product}}$ *of* $\underline{\textit{United Sates}}_{\textit{cell_value:}\texttt{USA}}$*?*"). To represent information of tables in a schema, a commonly used approach is to flatten the structured schema of table names and their header information (columns) as a sequence of tokens consumable by a sequence encoding network [78, 227, 244]. To model schema linking, a simple solution is to augment the encodings of utterance tokens with information about their aligned schema elements, such as the entity type of the token [227], and features indicating whether it is a mention of cell values or columns [64, 111]. Besides these simple methods by flattening structured information in schemas or feature augmentation, there are also more systematic solutions for schema encoding while preserving its structured information. Specifically, Bogin et al. [21] used graph neural networks (GNNs) to capture the topology between multiple tables in a schema, and Wang et al. [188] augmented self-attention in Transformers with biased attention weights between aligned tokens in utterances and schemas (*e.g.*, utterance tokens that are mentions of columns), and related schema elements (*e.g.*, columns connected via primary-foreign key mappings), reminiscent of the relational self-attention mechanism in text [167] and source code encoding [71]. This thesis also studies understanding schema information of DB tables in Part II. Specifically, Chapter 5 will introduce a pre-trained Transformer for learning representations of (semi-)structured database tables. Different from the biased attention approach in Wang et al. [188], our model uses standard self-attention mechanisms to encode tables, while pre-trained on large-scale tabular data to learn how to represent such structured information. Recently, pre-training Transformers for table understanding has gained increasing popularity [47, 48, 75, 168, 232, 233, *inter alia*]. We will discuss more possible future directions in this line in Chapter 10.

To improve the decoding module that predicts database queries, a major research focus

is to capture the syntactic structures of DB queries, which could reduce the output space of possible queries while guarantee the syntactic correctness of generated results [244]. Many works along this line were based on our model in Part I, which generates the abstract syntax trees of DB queries using a decoder that predicts tree construction production rules guided by the grammar. The generated syntax trees can be deterministically converted to the target query. Other work generalized this idea of grammar-constrained decoding to token level, and design grammars that constrain the space of valid contiuating tokens when predicting a query [116, 207, 228]. Grammar-constrained generation could be further combined with coarse-to-fine decoding [51], factorizing the generation of queries into sketch (*e.g.*, `SELECT` `?` `WHERE` `Column?` `Op?` `Value?`) and arguments prediction (filling in `?`, as in [207, 228]).

General-purpose Code Generation

Programming is the act of turning the programmer's intention into source code. 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 (Fig. 1.1b). 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 [23, 24]. 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, 67, 118].

Research in code generation aims to develop models to automatically translate a programmer's natural language intent into source code written in programming languages. Early works in this line have focused on generating code in domain-specific languages (DSLs), like regular expressions [97], string manipulations formulas in spreadsheets [127, 162], or instructions for task automation [16, 156]. Later research attempts to generalize code generation to high-level, general-purpose programming languages, like Python and Java (Fig. 1.1b), while limited to specific application scenarios [106] or languages [160]. Ling et al. [117] proposed a language-agnostic data-driven code generation method, which 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 Part I of this thesis, we study syntax-driven code generation models, which capture the grammar of target programming languages as prior knowledge, ensuring the syntactic correctness of generated code. Instead of predicting code tokens using sequential decoding networks (*e.g.*, LSTMs), we use abstract syntax trees of programs as intermediate meaning representations, and design structured decoding models to generate ASTs following the grammar of the PL. Once the AST is generated, it can be deterministically converted to surface code snippets using libraries provided by the PL.

Since our first work in this area, code generation has become a popular research topic, with extensions to handle programmatic context (*e.g.*, previously defined class members and functions [82]), model idiomatic implementations [79, 170], leverage prior knowledge in API documentations [203], and augment generation with code retrieval [68, 69]. The idea of syntax-driven program generation has also been applied to some domain-specific semantic parsing scenarios, like natural language interfaces to database systems (§2.1.2), as well as other program synthesis tasks, such as code translation [32] and editing [213, 224].

2.2 Learning Paradigms

In the previous section, we discussed formalisms of logical meaning representations and their representative semantic parsing models. These parsers can be trained with various types of supervision signals, for which we present a systematic overview in this section.

2.2.1 Classical Supervised Learning

In the standard supervised learning setting, a parametric semantic parser $p_\theta(\boldsymbol{z}|\boldsymbol{u})$ is trained on parallel corpora of utterances ($\boldsymbol{u}$) annotated with meaning representations ($\boldsymbol{z}$). The parameters can be estimated using maximum likelihood estimation (MLE), with the following objective to maximize the log likelihood of generating gold-standard MRs in the labeled training data $\mathbb{L}$.

$$\mathcal{J}_{\text{MLE}} = \sum_{\langle \boldsymbol{u},\boldsymbol{z}\rangle \in \mathbb{L}} \log p(\boldsymbol{z}|\boldsymbol{u}) \tag{2.1}$$

For neural semantic parsers, optimizing Eq. (2.1) is straight-forward by performing stochastic gradient descent to minimize a cross-entropy loss. However, for CKY-style parsers based on grammar formalisms like probabilistic CCGs, optimization could involve latent variables not observed in $\mathbb{L}$, such as the derivation of MRs. This requires specialized learning methods for lexicon induction. Interested readers are referred to §2.1.1 for more details.

Cost in Annotating Training Data

To collect training data of NL utterances annotated with meaning representations, most approaches hired domain-experts that are proficient in the target logical formalisms. For tasks that use code in programming languages (*e.g.*, SQL, Python) as logical representations, finding annotators (*e.g.*, professional programmers) is relatively easy due to the popularity of those PLs. For example, the Spider text-to-SQL dataset [229] was created by 11 college students in computer science. The Python code generation dataset CoNaLa that will be discussed in Chapter 9 was constructed by a group of professional programmers annotating questions and Python implementations on Stack Overflow. However, data annotation is a laborious and costly endeavor. Yu et al. [229] reported $1,000$ man hours spent in curating $10K$ text-to-SQL examples, while the CoNaLa dataset costs roughly 1 US dollar to annotate only one example (Chapter 9). Data annotation is perhaps even more challenging for tasks using domain-specific meaning representations like λ-calculus, due to the difficulty in finding annotators who are fluent in the those MRs. In such cases, parallel data was typically hand-annotated by authors of the system [236], or converted semi-automatically from other versions of the dataset where MRs are defined in more commonly-used programming languages [237]. Therefore, semantic parsing datasets annotated with MRs are often limited in size due to the significant amount of cost and expertise required, with most corpora consisting of hundreds [235] or thousands [18, 46] of examples, and few with more than $10K$ instances [166, 229, 244]. The cost of data acquisition is a hurdle to deploying semantic parsers to emerging domains where labeled data is scarce, and becomes even more problematic with the dominance of neural semantic parsers, which are especially data-hungry [84].

Efficient Annotation Methods

Machine-in-the-loop To more efficiently annotate MRs while lowering the cost, the field has explored machine-in-the-loop approaches where human annotators are assisted by automated systems. This method was adopted in the creation of the ATIS dataset back in 1990s [46], where new utterances are first interpreted by a parser before verified by annotators. Iyer et al. [81] further put forward an online learning framework for data collection with a semantic parser in the loop, where annotators accept or revise the parser's hypothesized MRs as new training instances. The parser is then updated after collecting a batch of new examples. In Chapter 9, we will discuss generalizing this idea to mine parallel corpora of NL questions and Python code.

Alternative Annotation Targets Besides accelerating annotation using feedback from a model, another line of research considers alternative annotation targets that are easier to label than meaning representations. Instead of hand-annotating MRs, one intuitive alternative is to automatically synthesize MRs since they are defined following a grammar, and then design annotation tasks that match those synthetic MRs with NL utterances. This idea is proposed by the seminal OVERNIGHT work [195], which synthesizes parallel training data by using a synchronous grammar to align elements in MRs and their canonical NL expressions (*e.g.*, `Filter(paper,venue=`?`)` $\leftrightarrow$ *papers in* ? and `acl`$\leftrightarrow$*ACL*), and generates examples of compositional utterances (*e.g.*, *Papers in ACL*) with MRs (*e.g.*, `Filter(paper,venue=acl)`) from the grammar. The synthesized canonical utterances are then paraphrased by annotators, a much easier task than writing MRs. Follow-up work has focused on further mitigating the cost in writing paraphrases of canonical utterances. Instead of manually creating paraphrases, Herzig and Berant [73] cast this step as a paraphrase identification problem, where annotators pair canonical utterances with the real ones, assuming access to the set of (unlabeled) real utterances. In Chapter 7, we will explore further lifting the requirement of manually annotating paraphrases using automated paraphrase generation models, which enables training downstream semantic parsers using synthesized data without manual labeling.

In the following section, we will discuss more related works using alternative annotation targets. Those approaches do not consider building datasets with annotated MRs at all, and instead leverage other forms of supervision (*e.g.*, the execution results of utterances) to train semantic parsers.

2.2.2 Weakly-supervised Learning from Executions

Given the cost of annotating NL utterances with MRs, a large body of work explores training semantic parsers using alternative weak supervision signals that are cheaper to acquire. Weakly-supervised learning methods leverage the denotations (execution results) of utterances as indirect supervision [18, 42].[1] Under this paradigm, meaning representations are modeled as latent variables. Given an NL utterance $\boldsymbol{u}$, a parser predicts one or multiple (latent) MRs of $\boldsymbol{u}$, executes the hypothesized MRs, and collect training signals by comparing the execution results

[1]The term "weakly-supervised learning" was broadly used to refer to the general idea of training semantic parsers using indirect supervision other than annotated logical forms. Besides denotations, other works have also considered signals such as syntactic parse [95]. In this section, we follow later development of the field and dedicate this term for denotation-based methods. Other approaches, like Krishnamurthy and Mitchell [95], are categorized as unsupervised learning and discussed in §2.2.4.

with the annotated ones (*e.g.*, using the binary rewards of execution correctness). There are two commonly used optimization methods for weakly supervised learning, namely maximum marginal likelihood estimation and reinforcement learning.

Maximum Marginal Likelihood Estimation Maximum marginal likelihood (MML) maximizes the probability of observing the correct denotation by marginalizing over latent choices of MRs [18, 179]. Specifically, let $\boldsymbol{y}$ be the target denotation of $\boldsymbol{u}$, the learning objective of MML is

$$\begin{aligned}\mathcal{J}_{\text{MML}} &= \sum_{\langle \boldsymbol{u}, \boldsymbol{y} \rangle} \log p(\boldsymbol{y}|\boldsymbol{u}) = \sum_{\langle \boldsymbol{u}, \boldsymbol{y} \rangle} \log \sum_{\boldsymbol{z}} p(\boldsymbol{y}, \boldsymbol{z}|\boldsymbol{u}) \\ &= \sum_{\langle \boldsymbol{u}, \boldsymbol{y} \rangle} \log \sum_{\boldsymbol{z}} p(\boldsymbol{y}|\boldsymbol{z}) p(\boldsymbol{z}|\boldsymbol{u}) \\ &= \sum_{\langle \boldsymbol{u}, \boldsymbol{y} \rangle} \log \sum_{\boldsymbol{z} \in \mathcal{Z}(\boldsymbol{y})} p(\boldsymbol{z}|\boldsymbol{u})\end{aligned} \tag{2.2}$$

Where $\mathcal{Z}(\boldsymbol{y})$ denotes the set of MRs that execute to the denotation $\boldsymbol{y}$, which can be approximated using beam search [18, 66].

Reinforcement Learning Reinforcement learning (RL) models semantic parsers as RL agents, and maximizes the expected rewards that the agent receives when generating the correct denotation [110].

$$\mathcal{J}_{\text{RL}} = \mathbb{E}_{\boldsymbol{u} \sim p(\boldsymbol{u})} \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{u})} R(\boldsymbol{z}) = \sum_{\langle \boldsymbol{u}, \boldsymbol{y} \rangle} \sum_{\boldsymbol{z}} R(\boldsymbol{z}) p(\boldsymbol{z}|\boldsymbol{u}) = \sum_{\langle \boldsymbol{u}, \boldsymbol{y} \rangle} \sum_{\boldsymbol{z} \in \mathcal{Z}(\boldsymbol{y})} p(\boldsymbol{z}|\boldsymbol{u}) \tag{2.3}$$

where $R(\boldsymbol{z})$ is the binary reward function that returns 1 *iff* $\boldsymbol{z}$ executes to $\boldsymbol{y}$. The gradient of Eq. (2.3) with respect to model parameters could be estimated using policy gradient with the REINFORCE algorithm [198], as in [110]. One key challenge of optimizing Eq. (2.3) is that the binary reward $R(\boldsymbol{z})$ is sparse, and also *delayed* since the reward is only calculated after a full episode (*i.e.*, after the parser finishes predicting an MR $\boldsymbol{z}$). This could lead to large variance in the policy gradient estimator. To reduce the variance during gradient estimation, Liang et al. [111] proposed a low-variance gradient estimator by separately (a) sampling from a memory buffer caching high-reward MRs discovered so far, and (b) sampling additional on-policy examples from the model.

Weakly-supervised learning is highly non-trivial since MRs are not observed from data, and the model is trained using the noisy binary reward signal of execution correctness. Learning using such sparse and under-specified rewards could lead to two issues. First, the search space of MRs is combinatorial and exponentially exploding, and a parser need to explore a large

search space to discover high-reward programs. Second, since the reward is binary and under-specified, there are *spurious* MRs, which are programs that happen to execute to the correct answer while semantically incorrect (*e.g.*, $\boldsymbol{z} =$ `multiply(2,2)` for $\boldsymbol{u}$: *What is two plus two*).

Exploration Efficiency of Latent MRs To improve the efficiency in discovering high-reward latent programs amid the combinatorial search space, the field has focused on methods that reduce the size of the search space by pruning invalid partial hypotheses. The key insight is to leverage extra MR semantics (*e.g.*, typing and execution information) to constrain valid continuations during auto-regressive decoding of MRs. Liang et al. [110] used execution results of partially generated programs to prune the set of valid functions and variables to be predicted in the next time step. Krishnamurthy et al. [96] leveraged the type information, constraining the choices of next MR tokens to have compatible types with the current hypothesis. Other work aims to guide the search process using extra supervision that captures the pattern of correct MRs, such as abstract sketches of MRs without variable instantiations [60], and efficient online learning algorithms to predict MRs using sketches [240].

Spurious MRs Another long-standing issue of learning with weak supervision is the existence of spurious MRs, which are programs that happen to execute to the correct denotation, but are semantically wrong [151]. The fundamental reason of spuriousness is that the reward function used in Eq. (2.3) is under-specified, which lacks the granularity to distinguish semantically correct MRs with the spurious ones. These spurious high-reward MRs will bring noise to the optimization process. Pasupat and Liang [151] first used crowdsourcing to rule out spurious MRs. Besides manual annotation, another line of research attempts to augment the under-specified reward function to better capture the similarity between utterances and hypothesized MRs. Intuitively, assume an oracle similarity metric $s(\boldsymbol{u}, \boldsymbol{z})$ between utterances $\boldsymbol{u}$ (*e.g.*, *What is two plus two*) and MRs $\boldsymbol{z}$, if $\boldsymbol{z}$ is semantically correct (*e.g.*, `plus(2,2)`), $s(\boldsymbol{u}, \boldsymbol{z})$ should be high, while $s(\boldsymbol{u}, \boldsymbol{z}')$ will be lower for spurious predictions $\boldsymbol{z}'$ (*e.g.*, `multiply(2,2)`). Such similarity metric could be captured by predefined lexical alignments (*e.g.*, `plus` $\in \boldsymbol{z} \leftrightarrow$ *plus* $\in \boldsymbol{u}$) and injected into the existing reward function via policy shaping [60, 136]. Later work generalized this idea and model $s(\boldsymbol{u}, \boldsymbol{z})$ as the "back-translation" probability $p(\boldsymbol{u}|\boldsymbol{z})$, measuring the likelihood of reconstructing the original utterance given information in predicted MRs [35]. Besides reward shaping, another strategy is to directly learn reward functions that favor non-spurious hypotheses that generalize well on validation data using meta learning [1].

We remark that the two issues of exploration efficiency and spuriousness are often interconnected. Intuitively, methods that explore the space of high-reward MRs more efficiently

could potentially discover more spurious MRs, as only *very few* (most of the time, only one) MRs among all the those reward-earning ones are indeed semantically correct. Therefore, it is important to carefully control the exploration process to bias towards potentially correct samples [111], or using special gradient update rules to ensure gradients from possibly spurious samples will not dominate optimization [66, 136]. Additionally, some efficient exploration approaches could also tackle the issue of spurious programs as well. For instance, the sketch-based exploration methods introduced above reduce spuriousness using idiomatic MR sketches that generalize well across similar utterances [60, 240].

2.2.3 Semi-supervised Learning

Instead of relying on weak supervision signals that are noisy. Semi-supervised learning reduces the requirement of annotated parallel data using additional unlabeled NL utterances. Specifically, besides the labeled data $\mathbb{L}$ used in the supervised learning objective Eq. (2.1), we assume the model additionally has access to a relatively large amount of unlabeled NL utterances $\mathbb{U} = \{\boldsymbol{u}\}$, semi-supervised learning then aims to maximize the log-likelihood of examples in both $\mathbb{L}$ and $\mathbb{U}$:

$$\mathcal{J}_{\text{semi_supervised}} = \underbrace{\sum_{\langle \boldsymbol{u},\boldsymbol{z}\rangle \in \mathbb{L}} \log p_\phi(\boldsymbol{z}|\boldsymbol{u})}_{\text{supervised obj. } \mathcal{J}_s} + \alpha \cdot \underbrace{\sum_{\boldsymbol{u}\in\mathbb{U}} \log p(\boldsymbol{u})}_{\text{unsupervised obj. } \mathcal{J}_u} \tag{2.4}$$

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 $p(\boldsymbol{u})$ of unlabeled NL utterances $\mathbb{U}$, controlled by a tuning parameter α. Intuitively, if the modeling of $p_\phi(\boldsymbol{z}|\boldsymbol{u})$ and $p(\boldsymbol{u})$ is coupled (*e.g.*, they share parameters), then optimizing the marginal likelihood $p(\boldsymbol{u})$ using the unsupervised objective $\mathcal{J}_u$ would help the learning of the semantic parser $p_\phi(\boldsymbol{z}|\boldsymbol{u})$ [246].

For semi-supervised learning of semantic parsing, Kate and Mooney [86] first explored using transductive SVMs to learn from a semantic parser's predictions. Konstas et al. [92] applied self-training to bootstrap an existing parser for AMR parsing. Kociský et al. [91] employed variational auto-encoders (VAEs) for semi-supervised semantic parsing, where NL utterances as modeled as discrete latent variables, and the model uses extra synthetically generated MRs for learning.

In Chapter 8, we put forward a novel variational auto-encoding architecture for semi-supervised learning. Different from Kociský et al. [91], we model the unobserved MRs of unlabeled utter-

ances as tree-structured latent variables (*e.g.*, representing MRs using abstract syntax trees), generalizing VAEs to handle latent variables with rich internal structures.

2.2.4 Unsupervised Learning

To further reduce the amount of supervision required, unsupervised learning methods do not use any annotated training data. The majority of work in this line only uses unlabeled utterances ($\mathbb{U}$ in Eq. (2.4)) for learning. The intuition is to leverage *distant* supervision signals, such as lexicons that define the natural language expressions for entities and relations in a knowledge base, as well as existing linguistic analysis of utterances (*e.g.*, dependency parse) as scaffolds to infer logical forms that contain relevant entity and relation mentions and also share similar syntactic structures. Krishnamurthy and Mitchell [95] captured the above intuition as factors in a probabilistic graphical model, and learned parameters of semantic parsers that agree with these constraints. Poon [154] further generalized this idea to handle complex logical forms with multi-hop relations, using a hidden Markov model that assigns latent semantic states to nodes (*e.g.*, the token *Pittsburgh* has state `entity`) and edges (*e.g.*, the dependency edge from $\xrightarrow{\text{pobj}}$ Pittsburgh has state `flight.from_city`) on dependency parse trees of NL utterances (Fig. 1.1a), which are used to generate MRs. This method requires predefined lexicons to capture alignments between NL phrases and logical predicates [61], and also unlabeled utterances for learning. The design of semantic states is also strongly dependent on the target domain and its database schema, and it is unclear if this method could be applied to other scenarios with different domain schemas [94].

Recently, Xu et al. [205] proposed an unsupervised semantic parsing approach that alleviates the requirement of unlabeled utterances. Similar to the OverNight framework introduced in §2.2.1, this approach uses synthesized training data of canonical utterances and MRs generated from a domain grammar capturing NL-MR alignments, while the crowdsourced paraphrases of the canonical utterances are replaced with automatically generated ones using paraphrase generation models. In Chapter 7, we extend this unsupervised learning framework and show that the auto-paraphrased synthetic data with diverse compositional patterns could help a semantic parser capture the grounding of NL phrases to relations and entities in the domain schema. We also present solutions to bridge the gap between synthetic utterances and real-world user-issued ones, improving the linguistic diversity and logical pattern coverage of the synthetic data.

Tasks	Code Generation	Natural Language Database Interface	
		Fixed Schema	Multiple Schemas
Datasets	CoNaLa [Chapter 9], Codex HumanEval [30]	Geo [235], ATIS [46], JOBs [184], Scholar [81]	WikiSQL [244], WikiTableQuestions [150], SQA [83], Spider [229]
Modeling Approaches			
General Topics			
∟ Model Program Structure	Part I	Part I	[64, 188, 244]
∟ Compositional Generalization	?	Chapter 6	?
∟ Pre-training on NL-MR Data	[30, 203]	?	?
Database-specific			
∟ Utterance-Schema Linking	—	[Chapter 6; 107]	
∟ Encode Multi-table DB Structure	—	—	[21, 188]
∟ Pre-training on Tabular Data	—	—	[Chapter 5; 75, 232]
Cost-effective Learning Methods			
Weakly-supervised (§2.2.2)	?	[190]	[111, 150]
Semi-supervised (§2.2.3)	Chapter 8	Chapter 8	?
Unsupervised (§2.2.4)	?	[154]	?
Data Synthesis (§2.2.4)	?	[Chapter 7; 205]	?

Table 2.1: Systematic overview of semantic parsing tasks, datasets, and their applicable modeling techniques and learning methods. "?" denotes unexplored territories.

2.3 Summary

So far we have discussed a variety of tasks, benchmarks, modeling techniques and learning paradigms in semantic parsing. Indeed, semantic parsing is a highly domain-specific task, and each application scenario has its own focused research problems and suitable solutions. It could be a bit difficult to get a grasp of those different applications and their applicable methods at first glance. We therefore present a holistic overview of the task hierarchy discussed in this chapter and their corresponding research topics and solutions. Such an overview could also help us understand the status quo of the field and find potential future directions in those unexplored areas (denoted as ? in Tab. 2.1). As an example, most cost-effective learning methods, such as unsupervised learning and training using synthetically generated data, have only considered semantic parsing tasks with fixed database schemas, since some of those approaches, like the data synthesis model based on OverNight, would require domain-dependent grammar rules to capture the grounding of NL expressions to schema-specific logical forms. A potential

future avenue would be making those methods more generalizable to cross-domain scenarios with multiple (and held-out) database schemas. On the other head, some directions are broadly applicable to many tasks, like modeling program structure during decoding. Advance in those areas could potentially have a higher impact on a variety of application scenarios.

Finally, we remark that the literature review presented in this chapter is far from complete. It does not cover many other important directions in the field that are not directly related to this thesis, such as visual question answering [128, 215], robot command-and-control with natural language [7, 125, 131, 172], interactive semantic parsing [53, 211, 212], as well as semantic parsing for task-oriented dialogue systems [43, 65, 166]. We refer readers to the above cited references for more information about those topics.

Part I

Structured Program Generation Models

Chapter 3

Syntactic Models for Code Generation

In this chapter, we introduce a general-purpose neural semantic parsing model that captures the grammatical structure of MRs as prior syntactic knowledge. 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. Throughout this chapter, we will 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 4, 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

3.1 Overview

We aim to develop models that could translate a programmer's natural language intent into code written in general-purpose programming languages like Python and Java. Compared to classical semantic parsing tasks using meaning representations defined in domain-specific languages (*e.g.*, λ-calculus, §2.1.1), code generation targets programming languages with more complex schema and syntactic structures.

As introduced in §2.1.2, prior work in this line [117] 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

Production Rule	Role	Explanation
Call ↦ expr[*func*] expr*[*args*] keyword*[*keywords*]	Function Call	▷ *func*: the function to be invoked ▷ *args*: arguments list ▷ *keywords*: keyword arguments list
If ↦ expr[*test*] stmt*[*body*] stmt*[*orelse*]	If Statement	▷ *test*: condition expression ▷ *body*: statements inside the If clause ▷ *orelse*: elif or else statements
For ↦ 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 ↦ identifier[*name*] arguments*[*args*] stmt*[*body*]	Function Def.	▷ *name*: function name ▷ *args*: function arguments ▷ *body*: function body

Table 3.1: Example production rules for common Python statements [155]

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. 3.1(a), §3.2) 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* (§3.3) which formalizes the generation story of a derivation AST into sequential application of *actions* that either apply production rules (§3.3.1), or emit terminal tokens (§3.3.2). 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. [202] have noted that this imposition of structure on neural models is useful 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 (§3.3.3). As an example, when expanding the node $\star$ in Fig. 3.1(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 (§3.4) on two Python code generation tasks show 11.7% and 9.3% absolute improvements in accuracy against the state-of-the-art system [117]. Our model also gives competitive performance on a standard semantic parsing benchmark[1].

3.2 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 $\hat{y}$ is then given by

$$\hat{y} = \operatorname*{argmax}_{y} p(y|x). \tag{3.1}$$

$\hat{y}$ is then deterministically converted to the corresponding surface code c.[2] While this chapter 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. 3.1 is used to generate the function call `sorted(·)` in Fig. 3.1(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. 3.1(a) are variable terminal nodes.

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

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

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

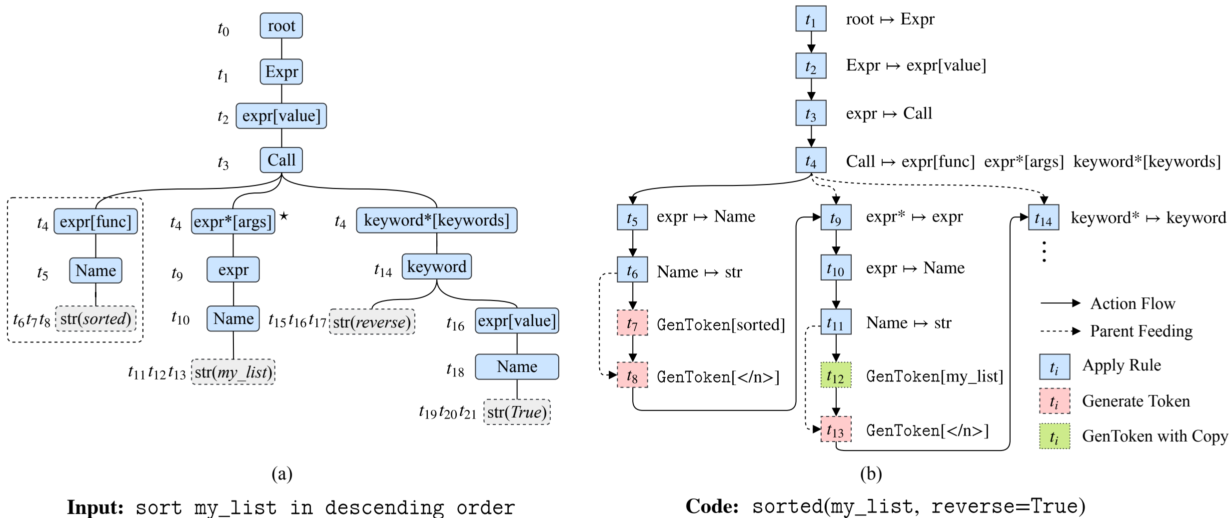


Figure 3.1: (a) the Abstract Syntax Tree (AST) for the given example code. Dashed nodes denote terminals. Nodes are labeled with time steps during which they are generated. (b) the action sequence (up to t_{14}) used to generate the AST in (a)

3.3 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. 3.1(b) shows the generation process of the target AST in Fig. 3.1(a). Each node in Fig. 3.1(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 §3.3.3).

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

$$p(y|x) = \prod_{t=1}^{T} p(a_t|x, a_{<t}), \tag{3.2}$$

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. (3.2) in §3.3.3. 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.

3.3.1 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. 3.1(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. 3.1(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 (§3.4).

3.3.2 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 §3.3.3.

3.3.3 Estimating Action Probabilities

We estimate action probabilities in Eq. (3.2) using attentional neural encoder-decoder models with an information flow structured by the syntax trees.

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 [76], similar to the setting in [10]. See supplementary materials for detailed equations.

Decoder

The decoder uses an RNN to model the sequential generation process of an AST defined as Eq. (3.2). Each action step in the grammar model naturally grounds to a time step in the decoder RNN. Therefore, the action sequence in Fig. 3.1(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 (§3.3.3), which will then be used to compute action probabilities $p(a_t|x, a_{<t})$ (§3.3.3).

Tracking Generation States

Our implementation of the decoder resembles a vanilla LSTM, with additional neural connections (parent feeding, Fig. 3.1(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:

$$\mathbf{s}_t = f_{\mathrm{LSTM}}([\mathbf{a}_{t-1} : \mathbf{c}_t : \mathbf{p}_t : \mathbf{n}_{f_t}], \mathbf{s}_{t-1}), \tag{3.3}$$

where $f_{\mathrm{LSTM}}(\cdot)$ 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. (3.2). Here, $\mathbf{a}_{t-1}$ is the embedding of

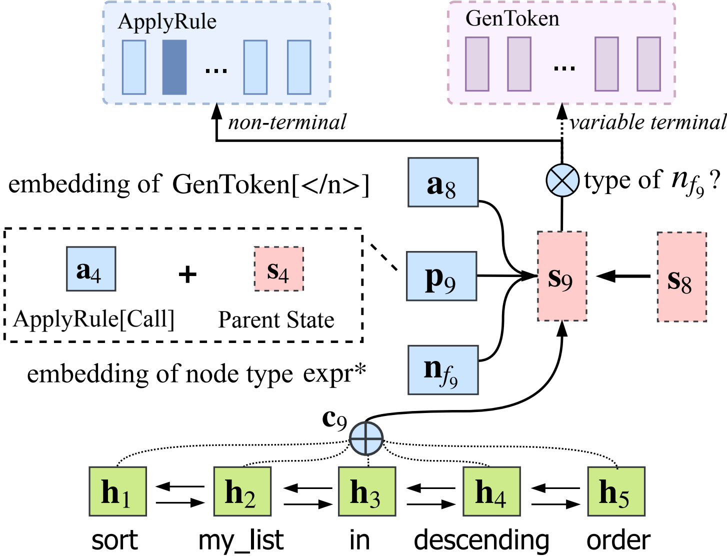


Figure 3.2: Illustration of a decoder time step ($t = 9$)

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. [10] 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

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

confident predictions. Similar approaches of injecting parent information were also explored in the SEQ2TREE model in Dong and Lapata [50][5].

Calculating Action Probabilities

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

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

$$p(a_t = \text{APPLYRULE}[r]|x, a_{<t}) = \text{softmax}(\mathbf{W}_R \cdot g(\mathbf{s}_t))^\intercal \cdot \mathbf{e}(r) \tag{3.4}$$

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

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

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. (3.4), 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 [62, 84, 117], and use a pointer network [186] 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.

3.3.4 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

[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.

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

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. (3.1). See supplementary materials for the pseudo-code of the inference algorithm.

3.4 Experimental Evaluation

3.4.1 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 3.2: Statistics of datasets and associated grammars

HearthStone (HS) dataset [117] 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 [146] 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 imple-

mentations 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 [156] 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. 3.2), 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. [117], and use token-level **BLEU-4** with as a secondary metric, defined as the averaged BLEU scores over all examples.[8]

3.4.2 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

[7] Like Beltagy and Quirk [16], 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 3.3: Results on two Python code generation tasks. †Results previously reported in Ling et al. [117].

datasets are relatively small for a data-hungry neural model, we impose strong regularization using recurrent dropouts [57] 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.

3.4.3 Results

Evaluation results for Python code generation tasks are listed in Tab. 3.3. 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 [117], and (2) SEQ2TREE, a neural semantic parsing model [50]. 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 [124]. For completeness, we also compare with a strong neural machine translation (NMT) system [140] using

a standard encoder-decoder architecture with attention and unknown word replacement[9], and include numbers from other baselines used in Ling et al. [117]. On the HS dataset, which has relatively large ASTs, we use unary closure for our model and Seq2Tree, and for Django we do not.

System Comparison As in Tab. 3.3, 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. (3.3)). 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.

[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. 3.3.

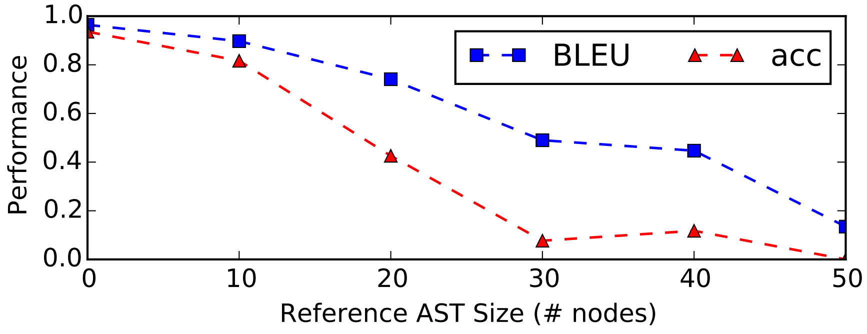


Figure 3.3: Performance w.r.t reference AST size on DJANGO

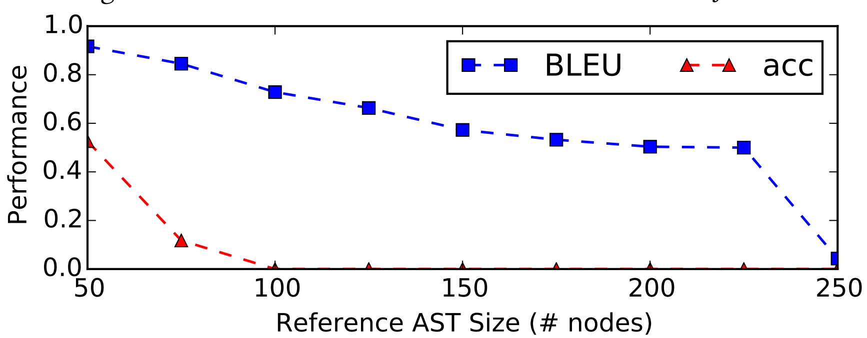


Figure 3.4: Performance w.r.t reference AST size on HS

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. 3.2), with the number of productions in the 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. 3.3 and 3.4. 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. 3.4 shows the results, following the evaluation protocol in [16] for accuracies at both channel and full parse tree (channel + function) levels.

	Channel	Full Tree
Classical Methods		
posclass [156]	81.4	71.0
LR [16]	88.8	**82.5**
Neural Network Methods		
NMT	87.7	77.7
NN [16]	88.0	74.3
Seq2Tree [50]	89.7	78.4
Doubly-Recurrent NN [5]	**90.1**	78.2
Our system	90.0	82.0
– parent feed.	89.9	81.1
– frontier embed.	**90.1**	78.7

Table 3.4: 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.

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 classical 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. 3.3. 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. 3.5. 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. [117]. 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.

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

The next two examples are from DJANGO. The first one shows that the model learns the 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 3.5.

3.5 Related Work

Code Generation Interested readers are referred to §2.1.2 for a review of the code generation task. To draw connections with the broader literature in machine learning for code, our work also falls into the field of probabilistic modeling of source code [126, 142]. Our approach of factoring an AST using probabilistic models is closely related to Allamanis et al. [4], 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.

Neural Semantic Parsing with Syntactic Information Prior to this work, several existing approaches in semantic parsing have be proposed to utilize grammar knowledge in a neural parser, such as augmenting the training data by generating examples guided by the grammar [84, 91]. Liang et al. [110] 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. [202] 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.

3.6 Summary

In this chapter we propose a syntax-driven neural code generation approach. Instead of directly predicting surface-level code tokens as in prior works, this model generates abstract syntax

trees of programs guided by the grammar of the target programming language as prior syntactic knowledge. ASTs are generated by sequentially applying tree-construction actions from a grammar model. Experiments on both code generation and semantic parsing tasks demonstrate the effectiveness of our proposed approach.

Chapter 4

Generalized Parsing Framework

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 3 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

4.1 Overview

As discussed in §2.1, meaning representations in semantic parsing could be defined according to a wide variety of formalisms. 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 [207, 244]. To alleviate this issue, there have been recent efforts in neural semantic parsing with general-purpose grammar models [51, 202]. In Chapter 3, we 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.* Concurrently, Rabinovich et al. [157] proposed the abstract syntax networks (ASNs),

where domain-specific MRs are represented by abstract syntax trees (ASTs, Fig. 4.2 *Left*) specified under the abstract syntax description language (ASDL) framework [194]. 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:

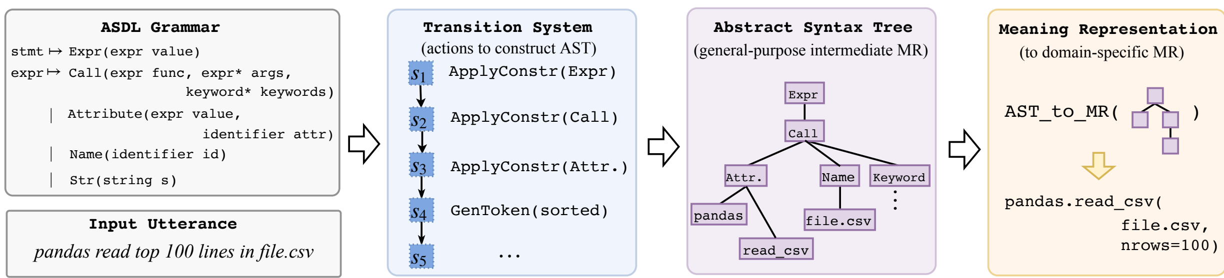
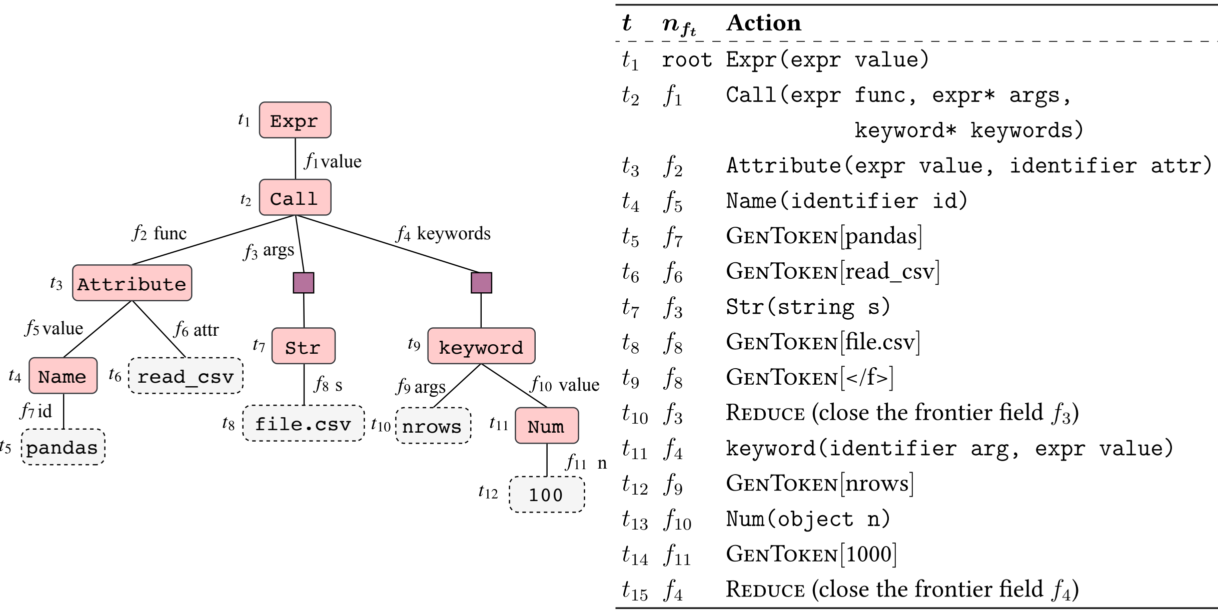


Figure 4.1: Workflow of TRANX

- **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).

4.2 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



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 4.2: ***Left*** The ASDL AST for the target Python code in Fig. 4.1. 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.

running example, where a programmer's natural language intents are mapped to Python source code. Fig. 4.1 depicts the workflow of Tranx. We will present more use cases of Tranx in §4.3.

The core of Tranx is a transition system. Given an input NL utterance $\boldsymbol{u}$, Tranx employs the transition system to map the utterance $\boldsymbol{u}$ into an AST $\boldsymbol{z}$ using a series of tree-construction actions (§4.2.2). 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 (§4.2.1). Given the generated AST $\boldsymbol{z}$, the parser calls the user-defined function, `AST_to_MR(·)`, to convert the intermediate AST into a domain-specific meaning representation $\boldsymbol{y}$, completing the parsing process. Tranx uses a probabilistic model $p(\boldsymbol{z}|\boldsymbol{u})$, parameterized by a neural network, to score each hypothesis AST (§4.2.3).

4.2.1 Modeling ASTs using ASDL Grammar

Tranx uses ASTs as the general-purpose, intermediate semantic representation for MRs. ASTs are commonly used to represent programming languages, and can also be used to represent

other tree-structured MRs (*e.g.*, λ-calculus). The ASDL framework is a grammatical formalism to define ASTs. See Fig. 4.1 for an excerpt of the Python ASDL grammar. TRANX provides APIs to read such a grammar from human-readable text files.

An ASDL grammar has two basic constructs: *types* and *constructors*. A *composite* type is defined by the set of constructors under that type. For example, the `stmt` and `expr` composite types in Fig. 4.1 refer to Python statements and expressions, repectively, each defined by a series of constructors. A constructor specifies a language construct of a particular type using its *fields*. For instance, the `Call` constructor under the composite type `expr` denotes function call expressions, and has three fields: `func`, `args` and `keywords`. Each field in a constructor is also strongly typed, which specifies the type of value the field can hold. A field with a composite type can be instantiated by constructors of the same type. For example, the `func` field above can hold a constructor of type `expr`. There are also fields with *primitive* types, which store values. For example, the `id` field of `Name` constructor has a primitive type `identifier`, and is used to store identifier names. And the field `s` in the `Str` (string) constructor hold string literals. Finally, each field has a cardinality (single, optional $?$ and sequential $*$), denoting the number of values the field holds.

An AST is then composed of multiple constructors, where each node on the tree corresponds to a typed field in a constructor (except for the root node, which denotes the root constructor). Depending on the cardinality of the field, a node can hold one or multiple constructors as its values. For instance, the `func` field with single cardinality in the ASDL grammar in Fig. 4.1 is instantiated with one `Name` constructor, while the `args` field with sequential cardinality have multiple child constructors.

4.2.2 Transition System

Inspired by our grammar-constrained decoding model in Chapter 3, we develop a transition system that decomposes the generation procedure of an AST into a sequence of tree-constructing *actions*. We now explain the transition system using our running example. Fig. 4.2 *Right* lists the sequence of actions used to construct the example AST. In high level, the generation process starts from an initial derivation AST with a single root node, and proceeds according to a top-down, left-to-right order traversal of the AST. At each time step, one of the following three types of actions is evoked to expand the opening *frontier field* n_{f_t} of the derivation:

APPLYCONSTR$[c]$ actions apply a constructor c to the opening composite frontier field which has the same type as c, populating the opening node using the fields in c. If the frontier

field has sequential cardinality, the action appends the constructor to the list of constructors held by the field.

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

GenToken$[v]$ actions populate a (empty) primitive frontier field with a token v. For example, the field f_7 on Fig. 4.2 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.

4.2.3 Computing Action Probabilities $p(\boldsymbol{z}|\boldsymbol{u})$

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

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

Following Chapter 3, we parameterize the transition-based parser $p(\boldsymbol{z}|\boldsymbol{u})$ using a neural encoder-decoder network with augmented recurrent connections to reflect the topology of ASTs.

Encoder The encoder is a standard bidirectional Long Short-term Memory (LSTM) network, which encodes the input utterance $\boldsymbol{u}$ 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

```
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 4.3: The λ-calculus ASDL grammar for Geo and Atis, defined in Rabinovich et al. [157]

attentional vector defined as in Luong et al. [123]

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

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 (§4.3).

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

$$p(a_t = \textsc{ApplyConstr}[c] | a_{<t}, \boldsymbol{u}) = \text{softmax}(\mathbf{a}_c^\intercal \mathbf{W} \tilde{\mathbf{s}}_t) \tag{4.1}$$

For GenToken actions, we employ a hybrid approach of generation and copying, allowing for out-of-vocabulary variable names and literals (*e.g.*, "*file.csv*" in Fig. 4.1) in $\boldsymbol{u}$ to be directly copied to the derivation. Specifically, the action probability is defined to be the marginal probability

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

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 (4.1). The copy probability of copying the i-th word in $\boldsymbol{u}$ is defined using a pointer network [186]

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

4.3 Experiments

4.3.1 Datasets

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

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 [50]. We use the ASDL grammar defined in Rabinovich et al. [157], as listed in Fig. 4.3.

Code Generation

We evaluate Tranx on both general-purpose (Python, Django) and domain-specific (SQL, WikiSQL) code generation tasks. The Django dataset [146] consists of 18,805 lines of Python source

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

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

Figure 4.4: The ASDL grammar for WikiSQL

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 [244] 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 Tab. 4.3, use specifically designed architectures to reflect the syntactic structure of SQL queries. We show that the transition system used by Tranx can be easily extended for WikiSQL with minimal engineering, while registering strong performance. First, we use define a simple ASDL grammar following the syntax of SQL (Fig. 4.4). 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.

4.3.2 Results

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

Methods	**Geo**	**Atis**
ZH15 [243]	88.9	84.2
ZC07 [237]	89.0	84.6
WKZ14 [187]	**90.4**	**91.3**
Neural Network-based Models		
Seq2Tree [50]	87.1	84.6
ASN [157]	87.1	85.9
Tranx	88.2	86.2

Table 4.1: Semantic parsing accuracies on Geo and Atis

Methods	Acc.
nmt [140]	45.1
lpn [117]	62.3
Yin and Neubig [218] (Chapter 7)	71.6
Tranx	**73.7**

Table 4.2: Code generation accuracies on Django

Semantic Parsing Tab. 4.1 lists the results for semantic parsing tasks. We test Tranx with two configurations, with or without parent feeding (§4.2.3). Our system outperforms existing neural network-based approaches. This demonstrates the effectiveness of Tranx in closed-domain semantic parsing. Interestingly, we found the model without parent feeding achieves slightly better accuracy on Geo, probably because that its relative simple grammar does not require extra handling of parent information.

Code Generation Tab. 4.2 lists the results on Django. Tranx achieves state-of-the-art results on Django. We also find parent feeding yields +1 point gain in accuracy, suggesting the importance of modeling parental connections in ASTs with complex domain grammars (*e.g.*, Python).

Tab. 4.3 shows the results on WikiSQL. We first discuss our standard model which only uses information of column names and do not use the contents of input tables during inference, as listed in the top two blocks in Tab. 4.3. 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* strat-

Methods	Acc$_{EM}$	Acc$_{EX}$
Seq2SQL [244]	48.3	59.4
SQLNet [207]	–	68.0
PT-MAML [77]	62.8	68.0
TypeSQL [227]	–	**73.5**
TRANX	**62.9**	71.7
PointSQL [192]†	61.5	66.8
TypeSQL+TC [227]†	–	**82.6**
STAMP [182]†	60.7	74.4
STAMP+RL [182]†	61.0	74.6
TRANX	**68.4**	78.6

Table 4.3: Exact match (EM) and execution (EX) accuracies on WikiSQL. †Methods that use the contents of input tables.

egy, where we execute the candidate SQL queries in the beam against the input table, and prune those that yield empty execution results. Results are listed in the bottom two-blocks in Tab. 4.3, where we compare with systems that also use the contents of tables. Surprisingly, this (frustratingly) simple extension yields significant improvements, outperforming many task-specific models that use specifically designed, heavily-engineered neural networks to incorporate information of table contents.

4.4 Summary

In this chapter we present TRANX, a transition-based abstract syntax parser. TRANX is generalizable, extensible and effective. It provides a convenient interface for users to specify the grammar of their task-specific programming languages, and 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. We test TRANX on four semantic parsing and code generation tasks, and demonstrate the system achieves competitive results on all those benchmarks. Since the release of TRANX and our prior code generation model in Chapter 3, there has been extensive follow-up work on grammar-constrained neural semantic parsing, with applications to text-to-SQL parsing [188, 193, 207, 228, *inter alia*] and general-purpose program synthesis [69, 79, 82, 170, *inter alia*]. This framework has also been extended to related fields like program translations [32]

and modeling code edits [213, 224]. We believe our TRANX model, as a general-purpose syntax-driven structured decoding framework, can be generalized to other structured prediction tasks whose constraints can be specified as context-free grammars. We provide more possible future directions in Chapter 10.

Part II

Understanding Structured Domain Schemas

Chapter 5

Pretraining for Structured Data Understanding

In previous chapters, we introduce structured decoding models that generate meaning representations following their syntactic structures, while the encoding model is assumed as a generic sequence-encoding network over input utterances. In many applications, however, in order to understand user-issued utterances, a semantic parser needs to process necessary domain-specific knowledge schemas that are rich in structures. A representative example is semantic parsing over databases, which requires the model 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 is published 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

5.1 Overview

Recent years have witnessed a rapid advance in the ability to understand and answer questions about free-form natural language (NL) text [161], largely due to large-scale, pretrained language models (LMs) like BERT [49]. 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 [59, 122, 130, 132, 152, 209]. 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 [85].

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) [18, 216, 235, refer to §2.1.2 for a survey], 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). As discussed in Chapter 1 and §2.1.2, 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 [17].

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. [21], Krishnamurthy et al. [96], Wang et al. [188], *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 structured information in DB [64, 78, 238], 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 (§5.3). 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 (§5.3.1). 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 (§5.3.2).

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 [229], where TaBert is fine-tuned together with a task-specific parser using parallel NL utterances and labeled DB queries (§5.4.1); (2) a challenging weakly-supervised learning benchmark WikiTableQuestions [150], where a system has to infer latent DB queries from its execution results (§5.4.2). 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 (§5.5).

5.2 Background

Semantic Parsing over Tables We focus on parsing utterances $\boldsymbol{u}$ to access database tables, where the meaning representation $\boldsymbol{z}$ 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. Refer to §2.1.2 for a review of existing works in this line.

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 (§5.4.1). Alternatively, it could be semi-structured with unnormalized, textual cell values (*e.g.*, *200 km*, §5.4.2). 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.

Given a utterance and the its associated tables, a neural semantic parser generates a DB

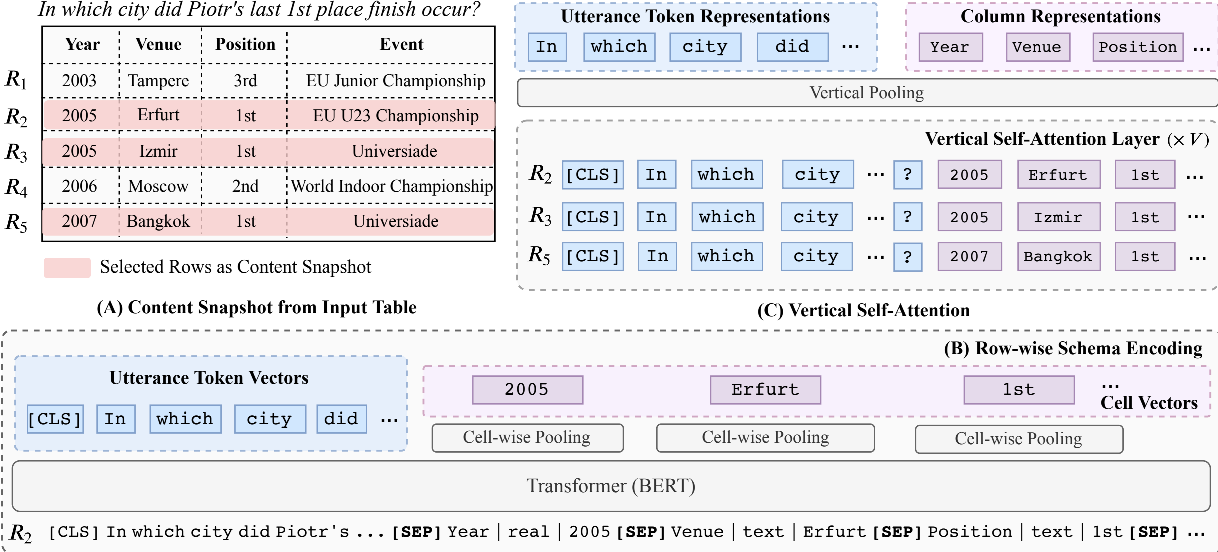


Figure 5.1: Overview of TaBert for learning representations of utterance and table schema with an example from WikiTableQuestions. **(A)** A content snapshot of the table is created based on the input NL utterance. **(B)** Each row in the snapshot is encoded by a Transformer (only R_2 is shown), producing 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.

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 §5.3.1.

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 $\widetilde{\boldsymbol{x}}$.

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

[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.

parameters are jointly tuned with other task-specific neural components.

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

We first present how TaBert computes representations for NL utterances and tables schemas (§5.3.1), followed by the pretraining procedure (§5.3.2).

5.3.1 Computing Representations for NL Utterances and Table Schemas

Fig. 5.1 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 Fig. 5.1 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 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.

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

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. Fig. 5.1(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 Fig. 5.1 is encoded as

$$\underbrace{\texttt{Year}}_{\text{Column Name}} \mid \underbrace{\texttt{real}}_{\text{Column Type}} \mid \underbrace{\texttt{2005}}_{\text{Cell Value}} \tag{5.1}$$

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 [31, 78], while our row approach is specifically designed for encoding content snapshots. We present in §5.5 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.

As in Fig. 5.1(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. (5.1). 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.

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

Each vertical attention layer has the same parameterization as the Transformer layer in Vaswani et al. [185], 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.

5.3.2 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 [105], 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 [49], 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 Fig. 5.1 is selected, the tokens `Year` and `real` in E.q. (5.1) 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. (5.1), 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 [85]. 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 [72]. The output of $f(\cdot)$ is then used to predict the original value token $s_{\langle i,j \rangle_k}$ from a softmax layer.

5.4 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 (§5.4.1), and a weakly supervised parsing problem on semi-structured Web tables (§5.4.2).

5.4.1 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 [229], a text-to-SQL dataset with 10,181 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 the TranX system proposed in Chapter 4, an open-source general-purpose semantic parser.

[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.

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 [114] over the list of table-specific representations, constructing an MR based on the predefined grammar. Following the IRNet model [64] 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.

5.4.2 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 [150] 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 Fig. 5.1 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 [111] 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.

5.5 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. **TaBert**$(\mathbf{K}=\mathbf{3})$ uses three rows from input tables as content snapshots and three vertical self-attention layers. **TaBert**$(\mathbf{K}=\mathbf{1})$ uses one synthetically generated row as the content snapshot as described in §5.3.1. 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 Fig. 5.1 is derived from the vector of the `Year` token in E.q. (5.1)). We 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$(\mathrm{K}=1)$, which reduces to a TaBert$(\mathrm{K}=1)$ 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.

5.5.1 Main Results

Tab. 5.1 and Tab. 5.2 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

[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.

Previous Systems on WikiTableQuestions		
Model	Dev	Test
Pasupat and Liang [150]	37.0	37.1
Neelakantan et al. [139]	34.1	34.2
Ensemble 15 Models	37.5	37.7
Zhang et al. [240]	40.6	43.7
Dasigi et al. [45]	43.1	44.3
Agarwal et al. [1]	43.2	44.1
Ensemble 10 Models	–	46.9

Our System based on MAPO [111]				
	Dev	Best	Test	Best
Base Parser	42.3 ± 0.3	42.7	43.1 ± 0.5	43.8
$w/$ BERT$_{\text{Base}}$ $(\text{K}=1)$	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
$w/$ TaBERT$_{\text{Base}}$ $(\text{K}=1)$	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
$w/$ TaBERT$_{\text{Base}}$ $(\text{K}=3)$	51.6 ± 0.5	52.4	51.4 ± 0.3	51.3
$w/$ BERT$_{\text{Large}}$ $(\text{K}=1)$	50.3 ± 0.4	50.8	49.6 ± 0.5	50.1
$w/$ TaBERT$_{\text{Large}}$ $(\text{K}=1)$	51.6 ± 1.1	52.7	51.2 ± 0.9	51.5
$w/$ TaBERT$_{\text{Large}}$ $(\text{K}=3)$	**52.2** ± 0.7	**53.0**	**51.8** ± 0.6	**52.3**

Table 5.1: Execution accuracies on WikiTableQuestions. Models are evaluated with 10 random runs. We report mean, standard deviation and the best results. Test$\mapsto$Best refers to the result from the run with the best performance on Dev. set.

the test set of WikiTableQuestions, MAPO augmented with a TaBERT$_{\text{Large}}$ model with three-row content snapshots, TaBERT$_{\text{Large}}(\text{K}=3)$, 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 (§5.4.1) 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 BERT$_{\text{Base}}$ (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$_{\text{Large}}(\text{K}=3)$, our parser achieves up to

Top-ranked Systems on Spider Leaderboard		
Model		DEV. ACC.
Global–GNN [20]		52.7
EditSQL + BERT [238]		57.6
RatSQL [188]		60.9
IRNet + BERT [64]		60.3
+ Memory + Coarse-to-Fine		61.9
IRNet V2 + BERT		63.9
RyanSQL + BERT (Anonymous)		**66.6**
Our System based on TranX (Chapter 4 [219])		
	Mean	Best
w/ BERT$_{\text{Base}}$ (K = 1)	61.8 ±0.8	62.4
− content snapshot	59.6 ±0.7	60.3
w/ TABERT$_{\text{Base}}$ (K = 1)	63.3 ±0.6	64.2
− content snapshot	60.4 ±1.3	61.8
w/ TABERT$_{\text{Base}}$ (K = 3)	63.3 ±0.7	64.1
w/ BERT$_{\text{Large}}$ (K = 1)	61.3 ±1.2	62.9
w/ TABERT$_{\text{Large}}$ (K = 1)	64.0 ±0.4	64.4
w/ TABERT$_{\text{Large}}$ (K = 3)	**64.5** ±0.6	**65.2**

Table 5.2: Exact match accuracies on the public development set of SPIDER. Models are evaluated with 5 random runs.

65.2% exact-match accuracy, a 2.8% increase over the base model using BERT$_{\text{Base}}$. 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.

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 pretraining 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.

u: *How many years before was the film Bacchae out before the Watermelon?*			
Input to TaBert$_{\texttt{Large}}$ $(\mathrm{K}=3)$		▷ *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$_{\texttt{Large}}$ $(\mathrm{K}=1)$		▷ *Content Snapshot with One Synthetic Row*	
Film	**Year**	**Function**	**Notes**
The Watermelon	2013	Producer	Screen adaptation of...

Table 5.3: Content snapshots generated by two models for a WikiTableQuestions Dev. example. Matched tokens between the question and content snapshots are highlighted.

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 Tab. 5.1 and Tab. 5.2 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. (5.1)). We find that content snapshots are helpful for both Bert and TaBert, especially for TaBert. As discussed in §5.3.1, 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 $(\mathrm{K}=1)$ 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 $(\mathrm{K}=1)$ and three rows from input tables $(\mathrm{K}=3)$ 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. Tab. 5.3 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$_{\texttt{Large}}$ $(\mathrm{K}=3)$ is able to capture the two target rows in its content snapshot and generates the correct DB query, while the TaBert$_{\texttt{Large}}$$(\mathrm{K}=1)$ model with only one row as content snapshot fails to answer this example.

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 [78]	49.0 ±0.4	58.6 ±0.3
Column Name is Cell Value (Chen19)	50.2 ±0.4	63.1 ±0.7

Table 5.4: 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 Tab. 5.1 and Tab. 5.2.

Effect of Row Linearization TaBERT uses row linearization to represent a table row as sequential input to Transformer. Tab. 5.4 (*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$_{\text{Base}}$ model (Tab. 5.4 *Lower-Half*). Hwang et al. [78] 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$_{\text{Base}}$ 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. [31] 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$_{\text{Base}}$ get improved results. We leave pretraining TaBERT with this linearization strategy as promising future work.

Impact of Pretraining Objectives TaBERT uses two objectives (§5.3.2), 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). Tab. 5.5 shows ablation results of pretraining

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 5.5: Performance of pretrained TaBERT$_{\text{Base}}$($K = 3$) on Dev. sets with different pretraining objectives.

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.

5.6 Related Works

Semantic Parsing over Tables Tables are important media of world knowledge. Semantic parsers have been adapted to operate over structured DB tables [51, 169, 193, 195, 207, 228], and open-domain, semi-structured Web tables [139, 150, 181]. 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. [111], Sun et al. [182], Yu et al. [227], Zhong et al. [244]), 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 [64, 70, 78, 238]. 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 its linked tables defined by the DB schema [20, 188]. 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 [153, 241, 242], or by encouraging the LM to recover KB entities and relations from text [121, 183]. 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.

5.7 Summary

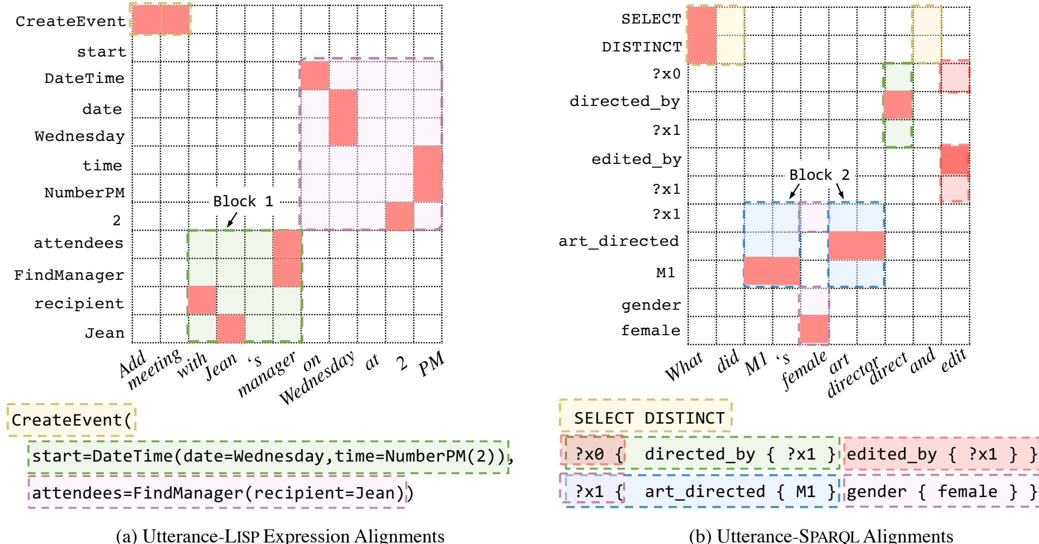
In this chapter we present TaBert, a pretrained encoder for joint understanding of textual and tabular data. We show that semantic parsers using TaBert as a general-purpose feature representation layer achieved strong results on two benchmarks. Future directions would include evaluating TaBert on other table-based reasoning tasks, such as fact checking [31] or text generation from tables [149]. We will also consider exploring other table representation strategies (*e.g.*, [58]), improving the quality of pretraining data, as well as designing novel pretraining objectives. Since this work and the concurrent paper by [75], pre-training for semantic parsing has become a rapily growing research area [47, 48, 168, 232, 233]. In Chapter 10 we will discuss more promising future directions in this line.

Chapter 6

Ground Utterances to Schema with Structured Inductive Bias

The previous chapter describes a pre-training approach for joint understanding of utterances and (semi-)structured schemas of database tables, where we use self-attention over tokens in utterances (*e.g.*, *GDP of France*) and flattened table headers (*e.g.*, `Country [SEP] Gross_Domestic _Product`) to capture the alignments between the two modalities. This approach enables self-supervised pre-training over large-scale weakly aligned corpora of text and tables to implicitly learn such alignments. However, learning could be data-hungry, as unconstrained self-attention does not *explicitly* model the association between NL phrases (*e.g.*, *GDP*) and their corresponding schema elements (*e.g.*, the column `Gross_Domestic_Product`). In this chapter, we attempt to augment attentional sequence transduction networks (*e.g.*, LSTMs, Transformers) with explicit inductive bias of NL-schema alignments. Intuitively, when an autoregressive decoder generates a logical form constituent (*e.g.*, `Gross_Domestic_Product`), we regularize its attention distribution over utterance tokens to focus on the tied NL phrase (*e.g.*, *GDP*). In this chapter, we also generalize the notion of knowledge schemas from database tables to specifications of domain functions (*e.g.*, `FindManager(` ? `)`) and their canonical NL intents (*e.g.*, *Who's ?'s manager*), a commonly used approach to configurate dialogue systems. We show this approach improves generalization ability of semantic parsers to utterances with compositionally novel contexts (*e.g.*, *Add meeting with Jean's manager*). This work is published in:

- Pengcheng Yin, Hao Fang, Graham Neubig, Adam Pauls, Emmanouil Antonios Platanios, Yu Su, Sam Thomson, and Jacob Andreas. Compositional generalization for neural semantic parsing via span-level supervised attention. In *Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, June 2021



(a) Utterance-Lisp Expression Alignments

(b) Utterance-SPARQL Alignments

Figure 6.1: Token and span level alignments (shown in $A^{\intercal}_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$) between utterances and programs in Lisp-style expressions (a) and SPARQL queries (b). Token alignments are marked in ■. Span-level alignments are marked using dashed bounding boxes (alignment to program sketch tokens are marked in ⬚). Programs in matrices are simplified for presentation. We use simplified SPARQL representation [56] grouping relations (*e.g.*, `directed_by` and `edited_by`) by subjects (*e.g.*, `?x0`).

6.1 Overview

As in many language understanding problems, a central challenge in semantic parsing is compositional generalization [55, 87]. Consider a personal digital assistant for which developers have assembled separate collections of annotated utterances for user requests involving their calendars (*e.g.*, *Schedule a meeting with Jean*) and their contact books (*e.g.*, *Who is Jean's manager?*). An effective model should learn from this data how to additionally handle requests like *Schedule a meeting with Jean's manager*, composing skills from the calendar and contacts domains, with little or no supervision for such combinations.

Neural sequence-to-sequence models, which provide the foundation for state-of-the-art semantic parsers (*e.g.*, [50], Chapter 3), tend to perform poorly at out-of-distribution generalization of this kind [56, 101, 180]. Methods have been proposed to bridge the generalization gap using meta-learning [102, 189] or specialized model architectures [33, 74, 109, 120, 164]. These have registered impressive performance on small synthetic benchmark datasets, but it has proven difficult to effectively combine them with large-scale pre-training [108, 159] and natural data [56].

In contrast to this extensive literature on data transformations and model architectures, the design of **loss functions** to encourage compositional generalization has been under-explored. This chapter investigates **attention supervision** losses that encourage attention matrices in neural sequence models to resemble the output of word alignment algorithms [119, 133] as a source of inductive bias for compositional tasks. Previous work has found that aligning program tokens (*e.g.*, `FindManager` in Fig. 6.1) to natural language tokens (*manager*) improves model performance [74, 147, 157]. However, the token-level alignments derived from off-the-shelf aligners are often noisy, and the correspondence between natural language and program tokens is not always a many-to-one map of the kind returned by standard alignment algorithms. Programs also have explicit hierarchical structure that is not captured by existing attention regularizers. Here we investigate the use of **span-level alignments**, identifying sub-programs that should be predicted as a unit and aligning all tokens in a sub-program to a corresponding natural language span.

We present a simple algorithm to derive span-level alignments from token-level alignments. Our approach is compatible with multiple models (RNNs, transformers, and structured tree-based decoders), pretrained or not. In experiments, span-based attention supervision consistently improves over token-level objectives, achieving strong results on three semantic parsing datasets featuring diverse formalisms and tests of generalization.

6.2 Span-level Supervised Attention

Neural Semantic Parsers In this chapter, we consider neural parsers using token-based attentive decoders, in which the meaning representation (program) $\boldsymbol{z}$ is predicted as a sequence of consecutive tokens $\{z_{j=1}^{|\boldsymbol{z}|}\}$ by attending to tokens in the utterance $\boldsymbol{u} = \{u_{i=1}^{|\boldsymbol{u}|}\}$. Examples include sequence-to-sequence models based on recurrent networks [50, 84] or transformers [159, 185], as well as structured parsing methods that predict a program following its syntactic structure ([51], see Chapter 2 and §6.3 for more details).

Supervised Attention Existing *token-level* supervised attention approaches assume access to an alignment matrix $A_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$ with entries $a_{i,j}$, where $a_{i,j} = 1$ *iff* the i-th source (utterance) token u_i is aligned to the j-th target (program) token z_j. $A_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$ can be inferred using latent variable models [25, 52, 145]. During training, when the decoder predicts a target token z_j, supervised attention encourages the target-to-source attention distribution $p_{\text{att}}(u_i|z_j)$ to match the prior alignment distribution $p_{\text{prior}}(u_i|z_j) = \frac{a_{i,j}}{\sum_k a_{k,j}}$, which is normalized by the number of

source tokens aligned to z_j. We use a squared error loss [119]:

$$\mathcal{L}_{\text{sup_att}} = \frac{1}{|\boldsymbol{z}|} \sum_{j=1}^{|\boldsymbol{z}|} \sum_{i=1}^{|\boldsymbol{u}|} \left(p_{\text{att}}(u_i|z_j) - p_{\text{prior}}(u_i|z_j)\right)^2. \tag{6.1}$$

Previous work has also used a cross entropy loss [147, 157].

Sub-program-to-Span Alignment We present a simple heuristic algorithm to extract span-level alignments between programs and utterances from existing token-level results (Algo. 1). Fig. 6.1 illustrates example span-level alignments for two types of programs (Lisp and simplified Sparql). Similarly to Dong and Lapata [51], we assume each program can be decomposed into a top-level **sketch** and a set of **sub-programs**.[1] For the Lisp expression in Fig. 6.1a, the sketch contains the top-level function call (`CreateEvent(`[?]`,`[?]`)`) and sub-programs are named arguments paired with values (`attendees=FindManager`...). For the Sparql expression in Fig. 6.1b, sketches include the query form (*e.g.*, `SELECT DISTINCT`) and sub-programs hold individual `subject-relation-object` assertions (*e.g.*, `?x0 edited_by ?x1`).

In this chapter, we use these program decompositions to guide span-level alignment. The underlying intuition is that *every* token in a sketch or sub-program will get aligned to the *same* set of utterance tokens. Algo. 1 extracts such set of utterance spans aligned to a sub-program $\boldsymbol{z}^{\text{s}}$ from the set $T_{\boldsymbol{z}^{\text{s}}}$ of NL tokens that are aligned to tokens in $\boldsymbol{z}^{\text{s}}$ (line 3). We present two variants of this approach, depending on the properties of the dataset (§8.4). In the first case (lines 5-6), similar to bilingual phrase extraction in machine translation [MT; 144], we create a single consecutive utterance span $\boldsymbol{u}_{m:n}$ via the outer bound of the aligned utterance tokens in $T_{\boldsymbol{z}^{\text{s}}}$ (*e.g.*, Block 1, Fig. 6.1a). In the second variant (lines 8-9), we find internally contiguous utterance spans in $T_{\boldsymbol{z}^{\text{s}}}$ and align them to $\boldsymbol{z}^{\text{s}}$. For instance, the sub-program (`?x1 art_directed M1`) in Block 2 of Fig. 6.1b aligns to two utterance spans: *M1 's* and *art director*. While this case does not have an exact analog in MT, it is reminiscent of the model of Chiang [40] which extracts translation rules with discontinuous phrase segments. Span-level alignments for a sub-program are then generated by pairing its program spans $\boldsymbol{z}_{p:q}$ (spans with consecutive program tokens) with all its aligned utterance spans (lines 11-12). Finally, we generate alignments for sketch spans in $\boldsymbol{z}$ by pairing them with any utterance tokens that have not yet been aligned to a sub-program (lines 13-14).

[1] Unlike D&L, we allow sub-programs to include non-consecutive (and possibly overlapping) spans of program tokens, *e.g.*, `{?x0 {edited_by {?x1}}` in Fig. 6.1b. We also permit non-disjoint sub-programs.

Algorithm 1: Span Alignment Extraction

input : Utterance $\boldsymbol{u}$, program $\boldsymbol{z}$, token-level alignment matrix $A_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$

output: Span-level alignment matrix $A^{\text{span}}_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$

1 Initialize set $A_S = \emptyset$ to store span-level alignments
2 **foreach** *sub-program* $\boldsymbol{z}^{\text{s}}$ **do**
3 $\quad T_{\boldsymbol{z}^{\text{s}}} = \{u_i | \exists z_j \in \boldsymbol{z}^{\text{s}}, a_{i,j} = 1\}, U_{\boldsymbol{z}^{\text{s}}} = \emptyset$
4 $\quad$ ▷ *Case 1 (Consecutive Alignment):*
5 $\quad m = \min_i\{u_i \in T_{\boldsymbol{z}^{\text{s}}}\}, n = \max_i\{u_i \in T_{\boldsymbol{z}^{\text{s}}}\}$
6 $\quad U_{\boldsymbol{z}^{\text{s}}} = \{\boldsymbol{u}_{m:n}\}$
7 $\quad$ ▷ *Case 2 (Nonconsecutive Alignment):*
8 $\quad$ **foreach** *consecutive span* $\boldsymbol{u}_{m:n} \subset T_{\boldsymbol{z}^{\text{s}}}$ **do**
9 $\quad\quad$ Add utterance span $\boldsymbol{u}_{m:n}$ to $U_{\boldsymbol{z}^{\text{s}}}$
10
11 $\quad$ **foreach** $\boldsymbol{z}_{p:q} \in \boldsymbol{z}^{\text{s}}, \boldsymbol{u}_{m:n} \in U_{\boldsymbol{z}^{\text{s}}}$ **do**
12 $\quad\quad$ Add span alignment $\boldsymbol{z}_{p:q} \leftrightarrow \boldsymbol{u}_{m:n}$ to A_S

▷ *Generate sketch-utterance span alignments:*

13 **foreach** *unaligned span* $\boldsymbol{z}_{p:q} \in \boldsymbol{z}$ *and* $\boldsymbol{u}_{m:n} \in \boldsymbol{u}$ **do**
14 $\quad$ Add span alignment $\boldsymbol{z}_{p:q} \leftrightarrow \boldsymbol{u}_{m:n}$ to A_S
15 Generate $A^{\text{span}}_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$, such that $a^{\text{span}}_{i,j} = 1$ *iff* $\exists \boldsymbol{z}_{p:q} \leftrightarrow \boldsymbol{u}_{m:n} \in A_S, i \in [m,n], j \in [p,q]$
16 **return** $A^{\text{span}}_{|\boldsymbol{u}|\times|\boldsymbol{z}|}$

6.3 Experiments

We evaluate span-level supervised attention on three benchmarks of compositional generalization.

6.3.1 Datasets and Models

SMCalFlow Compositional Skills (SMCal Flow-CS) is a new dataset created in this study based on the task-oriented dialogue corpus SMCalFlow [166]. Like the motivating story in §6.1, we create training data of single-turn utterances for skills $\mathbb{S}$ involving event creation (*e.g.*, *Schedule a meeting with Adam*) and organization structure (*e.g.*, *Who's on Adam's team?*), while evaluating on examples $\mathbb{C}$ featuring compositional skills (*e.g.*, *Add meeting with Adam and his team*). Utterances are annotated with Lisp-style programs (Fig. 6.1a). Zero-shot generalization in this setting is highly non-trivial due to novel language patterns (*e.g.*, *Adam and his team*) and pro-

$\mathbb{S}_{\text{EventCreation}}$ (24,403 Examples)	*Schedule dinner with Adam tomorrow.* *Please add dinner with Adam Wallen next Wednesday night at 6:00 PM.* *Put a reminder on my calendar half an hour before my dinner.*
$\mathbb{S}_{\text{OrgChart}}$ (733 Examples)	*Who's on Abby's team now?* *Who are the reports of Dan Schoffel?*
Compositional Skills ($\mathbb{C}$) (1,426 Examples)	*Add a meeting with my manager after lunch.* *Add Amanda and her boss to project meeting.* *Right after I'm done with breakfast, put a meeting with Sally's team.*

Table 6.1: Examples from SMCalFlow-CS

gram structures (*e.g.*, usage of `List(·)` to concatenate named entities) in the compositional evaluation set. We therefore consider a few-shot learning scenario, where we include a few compositional examples $\{16, 32, 64, 128\}$ into the training sets. The sizes of training (without compositional samples)/development/test splits are 23,838/1,298/1,298, respectively. Tab. 6.1 presents more examples in SMCalFlow-CS.

Compositional Freebase Questions (cfq) is a challenging compositional generalization dataset of $130K$ synthetic utterances with Sparql queries (Fig. 6.1b). Training and evaluation splits are constructed such that they have different distributions of compositional structures, while the distributions of atomic language (*e.g.*, *director*) and program (*e.g.*, `film.director`) constructs remain similar [87].

ATIS Text-to-SQL is a dataset of 3,809 SQL-annotated utterances about flight querying (*e.g.*, *Flights from Seattle to Austin.*). We follow Oren et al. [147] and use the program split [55], where training and evaluation programs do not overlap at template level.

Models We apply span-level supervised attention to strong neural models on each dataset. We evaluate two systems on SMCalFlow-CS: Bert2Seq, a sequence-to-sequence model with a Bert encoder and an LSTM decoder using copy mechanism, and Coarse2Fine [51], which uses (a Bert encoder and) a structured decoder that factorizes the generation of a program into sketch and value predictions. On cfq, we use T5-base [159], and apply attention supervision on all the cross-attention heads in the last decoder layer. For Atis, we take the best system from Oren et al. [147] that is tuned for better generalization on this dataset, which is a sequence-to-sequence model with an Elmo encoder and coverage-based attention mechanism [165].

We extract word alignments using IBM Model 4 in `GIZA++` [145], and canonicalized programs (*e.g.*, remove parentheses) to improve alignment quality. To extract span-level align-

$\lvert\mathbb{C}_{\text{train}}\rvert$	16		32		64		128	
Domain	$\mathbb{S}$	$\mathbb{C}$	$\mathbb{S}$	$\mathbb{C}$	$\mathbb{S}$	$\mathbb{C}$	$\mathbb{S}$	$\mathbb{C}$
BERT2SEQ	82.8 ±1.0	33.6 ±7.2	82.8 ±0.6	53.5 ±10.3	83.7 ±0.6	64.2 ±4.9	83.0 ±0.8	71.3 ±2.3
+TS (Token-level Sup.)	83.4 ±0.7	39.7 ±1.3	83.2 ±0.3	59.9 ±1.6	83.7 ±0.6	65.7 ±1.5	83.4 ±0.4	73.2 ±0.7
+SS (Span-level Sup.)	83.9 ±0.2	**46.8** ±1.2	83.5 ±0.7	**61.7** ±2.2	83.6 ±0.7	**66.9** ±1.0	83.5 ±0.9	**74.3** ±0.7
COARSE2FINE (DL18)	83.0 ±1.0	40.6 ±7.0	83.6 ±0.6	54.6 ±6.8	83.8 ±0.3	65.7 ±3.2	83.4 ±1.2	72.9 ±0.6
+TS (Token-level Sup.)	83.7 ±0.5	44.6 ±1.5	83.1 ±1.0	60.7 ±2.5	83.7 ±0.8	67.1 ±1.4	83.3 ±0.7	74.1 ±0.9
+SS (Span-level Sup.)	83.8 ±0.4	**47.4** ±2.1	83.7 ±1.0	**61.9** ±1.8	83.0 ±0.8	67.5 ±1.4	83.5 ±0.8	**75.0** ±1.2

Table 6.2: TEST. accuracies on the SMCALFLOW-CS Compositional Skills dataset w.r.t. the size of compositional examples included in the training set. We report both the results on the in-domain single-skill examples ($\mathbb{S}$) as well as the generalized multi-skill examples ($\mathbb{C}$). Results are averaged over five random random seeds. **Bold** results have p-values ≤ 0.05 when comparing to other systems in the same category under a permutation test.

ments, we use consecutive alignments (Case 1) in Algo. 1 for SMCALFLOW-CS and ATIS, as those datasets feature simple one-to-one mapping between sub-programs and utterance spans. For CFQ, we use nonconsecutive alignments (Case 2) to handle assertions aligned to disjoint NL spans (Fig. 6.1b). We apply Eq. (6.1) during model optimization using either the token and span level alignment matrix for token (+**TS**) and span (+**SS**) level supervised attention, respectively.

6.3.2 Results

Tab. 6.2 lists the evaluation results on SMCALFLOW-CS with varying numbers of compositional examples in the training set ($\mathbb{C}_{\text{train}}$).[2] We report accuracies on both the in-domain single-skill examples ($\mathbb{S}$) as well as on the generalized compositional-skill examples ($\mathbb{C}$). Both methods improve compositional generalization for BERT2SEQ and COARSE2FINE, while span-level supervised attention is more effective. Intuitively, span-level alignments could better capture the correspondence between sub-structures in utterances and programs, helping the parser to correctly predict such sub-programs in compositionally novel contexts by focusing on the corresponding utterance span. Besides more accurate predictions of sub-programs, models trained with span-level supervision also achieves better accuracies in predicting program sketches. For example, when using BERT2SEQ with $\lvert\mathbb{C}_{\text{train}}\rvert = 16$, the program sketch accuracy jumps from 41.9% without supervised attention to 68.3% using span-level supervision, compared to 63.3%

[2]We ran `GIZA++` and extracted span-level alignments for each training split separately.

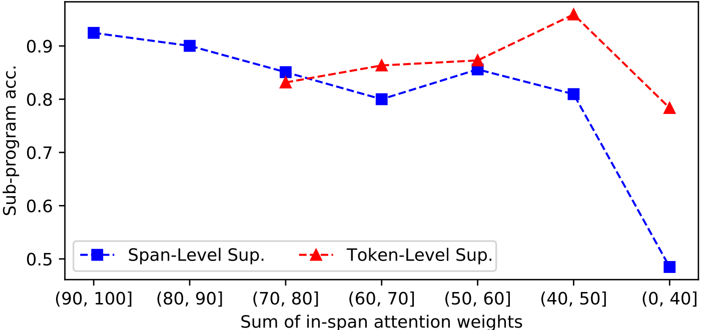


Figure 6.2: Sub-program prediction accuracy w.r.t. the sum of attention weights over oracle utterance spans. Models are trained on $|\mathbb{C}_{\text{train}}| = 32$. Results averaged over three runs.

for models trained with token-level supervision. Finally, we remark that in extremely low-resource learning scenarios with only a handful of compositional training samples, span-level supervised attention offers more gains in extreme low-resource settings ($|\mathbb{C}_{\text{train}}| = 16$), outperforming the base BERT2SEQ model by 13% absolute (33.6% v.s. 46.8% for BERT2SEQ).

Indeed, we found that more *alignment-like* attentions are associated with more accurate model predictions. For a BERT2SEQ model with span-level supervision trained on $|\mathbb{C}_{\text{train}}| = 64$, when predicting sub-programs for the `attendees` argument (*e.g.*, `attendees=FindManager(recipient=self)`) on compositional samples in $\mathbb{C}$, the model achieves 86% sub-program accuracy if it assigns a time-step average of at least 90% of its attention weights over the aligned utterance spans (*e.g.*, *with my manager*) identified by Algo. 1. Otherwise, the accuracy drops to 70%.

To further investigate the positive correlation between the "quality" of the attention distribution $p_{\text{att}}(u_i|z_j)$ (how concentrated $p_{\text{att}}(u_i|z_j)$ is) over an utterance span (*e.g.*, *with my manager*) and the prediction accuracy of its target sub-program (*e.g.*, `attendees=FindManager(·)`). Here we present more results. Specifically, we identify compositional examples in the Dev. set for which a model predicts sub-programs $\boldsymbol{z}^{\text{s}}$ for the `attendees`, `start`, and `location` arguments in a `CreateEvent` function call (refer to Fig. 6.1a for the first two arguments, `location` is used to specify event location). We compute the sum of the attention weights over the "oracle" utterance span identified by Algo. 1, and averaged over the decoder's time step when predicting $\boldsymbol{z}^{\text{s}}$. We measure the sub-program prediction accuracy w.r.t. the attention weights, as illustrated in Fig. 6.2. We observe that models trained with span-level supervised attention shows a stronger correlation between the sub-program accuracy and the degree the attention focuses on utterance tokens within the oracle span.

Split	MCD$_1$			MCD$_2$			MCD$_3$			Average
	C	R	All	C	R	All	C	R	All	
T5-BASE	**55.8** ±4.8	77.4 ±4.7	**62.4** ±4.5	34.8 ±2.9	29.4 ±2.5	33.0 ±2.4	**21.6** ±8.6	34.4 ±2.8	23.0 ±1.7	39.5
+ TS	44.9 ±4.7	**86.4** ±2.4	57.7 ±3.4	32.4 ±3.1	32.7 ±1.4	32.5 ±2.1	14.3 ±1.5	36.6 ±1.7	22.0 ±0.7	37.4
+ SS	48.2 ±4.4	80.5 ±2.2	58.2 ±2.8	34.8 ±2.3	**36.4** ±2.8	**35.4** ±1.6	14.6 ±2.1	**40.1** ±3.5	23.8 ±1.0	39.1

Table 6.3: Mean Test Accuracies on CFQ MCD splits with 95% confidence interval, for **C**onjunctive, **R**ecursive, and **All** the samples. The last column lists averaged accuracies for the three splits. **Bold** results have p-values ≤ 0.01 when comparing to other systems in the same category.

Moreover, supervised attention may be a sufficient *substitute* for structured model architectures in some cases. Despite the unstructured BERT2SEQ model's generally inferior performance without supervised attention, it matches the accuracies of COARSE2FINE when both models are trained with span-level supervision.[3] We also remark that span-based supervision maintains or improves performance on in-domain single-skill examples ($\mathbb{S}$). For instance, the accuracy for BERT2SEQ increases from 82.8% to 83.9% when $|\mathbb{C}_{\text{train}}| = 16$.

Next, on CFQ (Tab. 6.3), we report break-down results based on the syntactic types of questions: **R**ecursive questions with chained multi-hop relations (*e.g.*, $\boldsymbol{u}_r$:*Was M1 influenced by a German writer?*), and **C**onjunctive ones with only conjunctions of entities and relations and without chained relations (*e.g.*, $\boldsymbol{u}_c$:*Was M1 directed and edited by M2 and M3?*). While supervised attention is effective on recursive questions, it struggles on conjunctive ones. This may be because the model learns to attend to discontinuous utterance spans (*e.g.*, "*M1 directed*" and "*M2 and M3*" in $\boldsymbol{u}_c$) when predicting a relation (*e.g.*, `directed_by`) in a conjunction, which could be more sensitive to alignment errors. Additionally, utterance spans aligned to a sub-program in conjunctive questions are usually longer and more complex (*e.g.*, having multiple conjunctive entity mentions like *Did M1 write M2, M3, M4, and M5?*), which might require more fine-grained supervision than uniformly treating every aligned utterance tokens equally as in Eq. (6.1).

k

Finally, we present the results on the ATIS query splits in Tab. 6.4, where span-level supervision is comparable with token-level one, further improving upon an already-strong model that targets for compositional generalization (ELMO with coverage based attention). Interestingly, token-level supervised attention is slightly worse than the baseline model on the standard

[3] We found that the sketch and sub-program decoders in COARSE2FINE do not achieve their best DEV. accuracy at the same iteration during training, which could hurt performance in our few-short learning setting.

Model	*Query Split*		*i.i.d. Split*	
	DEV.	TEST.	DEV.	TEST.
Oren et al. [147]	28.9	34.4	78.4	74.5
+ Token-level Sup.	31.2 ±1.2	34.5 ±0.9	76.7 ±0.6	72.5 ±1.6
+ Span-level Sup.	31.1 ±0.6	35.0 ±2.0	78.4 ±0.8	74.0 ±0.5

Table 6.4: Accuracies and standard deviation on the ATIS text-to-SQL query (program template) and standard i.i.d. split splits. Results averaged over five random runs.

i.i.d. splits, while span-level supervision does not offer further improvements. Empirically we observe that the utterance-SQL alignments in ATIS are much noisier than other two datasets due to redundant structures in SQL queries (*e.g.*, `Join` statements with intermediary tables), whose aligned NL constituents are often not well defined.

6.4 Summary

In this chapter, we demonstrated the effectiveness of span-level supervised attention as a simple and flexible tool for improving neural sequence models in a diverse set of architectures and tests of generalization. Future work will include applications to other prediction tasks, as well as designing generative modeling approaches that capture span-level alignments as structured latent variable models (*c.f.*, Chapter 8), which jointly learn utterance-program alignments with parameters in the neural semantic parser.

Chapter 7

Ground Language to Schema without Labeled Data

Chapters 5 and 6 demonsrate how pre-training and regularized attention could capture the grounding of NL constituents in utterances (*e.g.*, *Who is my manager*) to structured logical predicates in domain schemas (*e.g.*, the program `FindManager(recipient=User)`). These approaches still require supervised learning using compositional utterances annotated with logical forms. 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. Specifically, we capture such mappings between NL phrases and MR predicates (*e.g.*, "*my manager*"↦`FindManager(recipient=User)`, "*Add meeting with* ?"↦`CreateEvent( attendees=?)`) in the domain's schema specification using a synchronous grammar, and automatically synthesize parallel examples of compositional utterances (*e.g.*, *Add meeting with my manager*) and programs from the grammar as training data. Intuitively, semantic parsers trained on the synthetic data with diverse compositionality patterns could implicitly capture the grounding of NL phrases to elements in the domain schema in a data driven fashion.

This work is currently under peer review:

- Pengcheng Yin, John Wieting, Avirup Sil, and Graham Neubig. On the ingredients of an effective zero-shot semantic parser. under reivew

7.1 Overview

Learning semantic parsers typically requires parallel data of NL utterances annotated with programs, and annotating such data requires significant expertise and cost. In §2.2, we reviewed a range of data efficient approaches, like the OverNight annotation framework that automatically synthesizes parallel training data using a synchronous grammar, where the synthetically generated canonical utterances are further manually paraphrased (§2.2.1). Later, Xu et al. [205] build upon OverNight and develop a *zero-shot* semantic parser replacing the manual paraphrasing process with an automatic paraphrase generator (§2.2.4, more in §7.2). While promising, there are still several issues with the current approach. First, such systems are not truly zero-shot — they still require labeled validation data (*e.g.*, to select the best checkpoint during training). Next, to ensure the quality and broad-coverage of synthesized canonical examples, existing models rely on heavily curated grammars (*e.g.*, with 800 production rules), which are cumbersome to maintain. More importantly, as suggested by Herzig and Berant [73] who study OverNight models using manual paraphrases, such systems trained on synthetic canonical samples suffer from fundamental mismatches between the distributions of the automatically synthesized examples and the *natural* ones issued by real users. Specifically, there are two types of gaps. First, there is a *logical gap* between the synthetic and real programs, as real utterances (*e.g.*, *Paper coauthored by Peter and Jane*) may exhibit logic patterns outside of the domain of those covered by the grammar (*e.g.*, *Paper by Jane*). The second is the *language gap* between the synthetic and real utterances, as paraphrased utterances (*e.g.*, $\boldsymbol{u}_1'$ in Fig. 7.1) still follow similar linguistic patterns as the canonical ones they are paraphrased from (*e.g.*, $\boldsymbol{u}_1$), while user-issued utterances tend to be more linguistically diverse (*e.g.*, $\boldsymbol{u}_2$).

In this chapter we analyze zero-shot parsers through the lenses of language and logical gaps, and propose methods to close those gaps (§7.3). Specifically, we attempt to bridge the language gap using stronger paraphrasers and more expressive grammars tailored to the domain-specific idiomatic language patterns. We replace the large grammars of previous work with a highly compact grammar with only 46 domain-general production rules, plus a small set of domain-specific productions to capture idiomatic language patterns (*e.g.*, $\boldsymbol{u}_2$ in Fig. 7.1, §7.3.1). We demonstrate that models equipped with such a smaller but more expressive grammar catered to the domain could generate utterances with more idiomatic and diverse language styles.

On the other hand, closing the logical gap is non-trivial. This is because canonical examples are generated by exhaustively enumerating all possible programs from the grammar up to a certain depth, and increasing the threshold to cover more complex real-world examples will lead

to exponentially more canonical samples, the usage of which is computationally intractable. To tackle the exponentially exploding sample space, we propose an efficient approach to sample canonical examples from the grammar by retaining canonical samples that most likely appear in real data (§7.3.1). Specifically, we approximate the likelihood of canonical examples using the probabilities of their utterances measured by pre-trained language models (LMs). This enables us to improve logical coverage of programs while maintaining a tractable number of highly-probable examples as training data.

In experiments, we show that with the proposed methods to bridge the language and logical gaps, our system achieves strong results on two semantic parsing datasets featuring realistic utterances (Scholar and Geo). Despite the fact that the proposed system uses *zero* annotated data for training and validation, it outperforms other supervised methods like OverNight and Granno [73] that require manual annotation. Our analysis also shows that current models are far from perfect, suggesting that logical gap still remains an issue, while stronger paraphrasing models are needed to further close the language gap.

7.2 Zero-shot Semantic Parsing via Data Synthesis

Problem Definition Semantic parsers translate a user-issued NL utterance $\boldsymbol{u}$ into a machine-executable program $\boldsymbol{z}$ (Fig. 7.1). We consider a zero-shot learning setting without access to parallel data in the target domain. Instead, the system is trained on a collection of machine-synthesized examples.

Overview Our system is inspired by the existing zero-shot parser by Xu et al. [205]. Fig. 7.1 illustrates our framework. Intuitively, we automatically create training examples with canonical utterances from a grammar, which are then paraphrased to increase diversity in language style. Specifically, there are two stages. First, a set of seed canonical examples (Fig. 7.1b) are generated from a **synchronous grammar**, which defines compositional rules of NL expressions to form utterances (Fig. 7.1a). Next, in the iterative training stage, a **paraphrase generation** model rewrites the canonical utterances to more natural and linguistically diverse alternatives (Fig. 7.1c). The paraphrased examples are then used to train a semantic parser. To mitigate noisy paraphrases, a filtering model, which is the parser trained on previous iterations, rejects paraphrases that are potentially incorrect. This step of paraphrasing and training could proceed for multiple iterations, with the parser trained on a dataset with growing diversity of language styles.

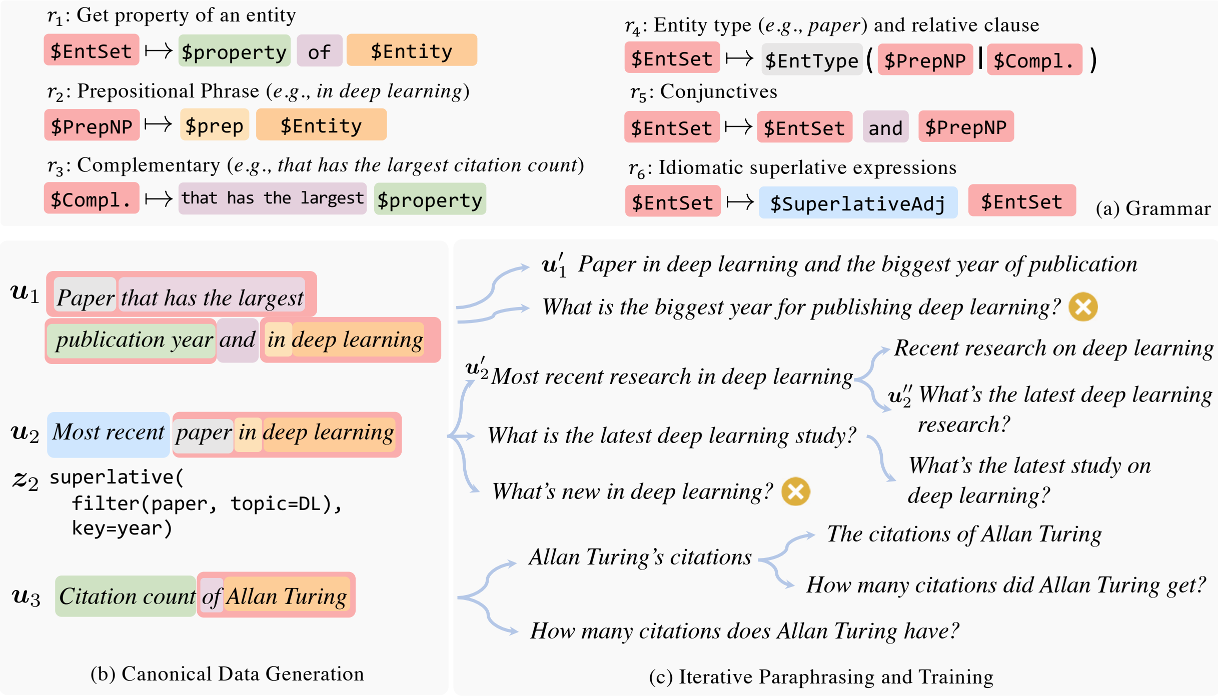


Figure 7.1: Illustration of the learning process of our zero-shot semantic parser with real model outputs. **(a)** Synchronous grammar with production rules. **(b)** Canonical examples of utterances with programs (only z_2 is shown) are generated from the grammar (colored spans show productions used). Unnatural utterances like u_1 can be discarded, as in §7.3.1 **(c)** At each iteration, canonical examples are paraphrased to increase diversity in language style, and a semantic parser is trained on the paraphrased examples. Potentially noisy or vague paraphrases are filtered (marked as ⊗) using the parser trained on previous iterations.

Synchronous Grammar Seed canonical examples are generated from a synchronous context free grammar (SCFG). Fig. 7.1a lists simplified production rules in the grammar. Intuitively, productions specify how utterances are composed from lower-level language constructs and domain lexicons. For instance, given a database entity `allan_turing` with a property `citations`, u_3 in Fig. 7.1 could be generated using r_1. Productions could be applied recursively to derive more compositional utterances (*e.g.*, u_2 using r_2, r_4 and r_6). Our SCFG is based on Herzig and Berant [73], consisting of domain-general rules of generic logical operations (*e.g.*, `superlative`, r_3) and domain-specific lexicons of entity types and relations. Different from Xu et al. [205] which uses a complex grammar with 800 rules, we use a compact grammar with only 46 generic rules plus a handful of idiomatic productions (§7.3.1) to capture domain-specific language patterns (*e.g.*, *"most recent"* in u_2, *c.f.*, u_1). Given the grammar, examples are enumerated exhaustively up to a threshold of number of rule applications, yielding a large set

of seed canonical examples $\mathbb{D}_{\text{can}}$ (Fig. 7.1b) for paraphrasing.

Paraphrase Generation and Filtering The paraphrase generation model rewrites a canonical utterance $\boldsymbol{u}$ to more natural and diverse alternatives $\boldsymbol{u}'$. $\boldsymbol{u}'$ is then paired with $\boldsymbol{u}$'s program to create a new example. Our paraphraser is a BART model specifically fine-tuned to produce lexically and syntactically diverse outputs. Some paraphrases are noisy or potentially vague (⊗ in Fig. 7.1c). We follow Xu et al. [205] and use the parser trained on previous iterations as the filtering model, and reject paraphrases for which the parser cannot predict their programs.

7.3 Bridging the Gaps between Canonical and Natural Data

Language and Logical Gaps The synthesis approach in §7.2 will generate a large set of paraphrased canonical data (denoted as $\mathbb{D}_{\text{par}}$). However, as noted by Herzig and Berant [73] (hereafter HB19), the synthetic examples cannot capture all the language and programmatic patterns of real-world natural examples from users (denoted as $\mathbb{D}_{\text{nat}}$). There are two mismatches between $\mathbb{D}_{\text{par}}$ and $\mathbb{D}_{\text{nat}}$. First, there is a **logical gap** between the programs in $\mathbb{D}_{\text{nat}}$ capturing real user intents, and the synthetic ones in $\mathbb{D}_{\text{par}}$. Notably, since programs are exhaustively enumerated from the grammar up to a certain compositional depth, $\mathbb{D}_{\text{par}}$ will not cover more complex programs in $\mathbb{D}_{\text{nat}}$ beyond the threshold. Ideally we could improve the coverage using a higher threshold. However, the space of possible programs will grow exponentially, and combinatorial explosion happens even with small thresholds.

Next, there is a **language gap** between paraphrased canonical utterances and real-world user-issued ones. Real utterances (*e.g.*, the $\boldsymbol{u}_2$ in Fig. 7.1, modeled later in §7.3.1) enjoy more lexical and syntactical diversity, while the auto-paraphrased ones (*e.g.*, $\boldsymbol{u}'_1$) are typically biased towards the monotonous and verbose language style of their canonical source (*e.g.*, $\boldsymbol{u}_1$). While we could increase diversity via iterative rounds of paraphrasing (*e.g.*, $\boldsymbol{u}_2 \mapsto \boldsymbol{u}'_2 \mapsto \boldsymbol{u}''_2$), the paraphraser could still fail on canonical utterances that are not natural English sentences at all, like $\boldsymbol{u}_1$.

7.3.1 Bridging Language and Logical Gaps

We introduce improvements to the system to close the language (§7.3.1) and logical (§7.3.1) gaps.

Idiomatic Productions

To close language gaps, we augment the grammar with productions capturing domain-specific idiomatic language styles. Such productions compress the clunky canonical expressions (*e.g.*, $\boldsymbol{u}_1$ in Fig. 7.1) to more succinct and natural alternatives (*e.g.*, $\boldsymbol{u}_2$). We focus on two language patterns:

Non-compositional expressions for multi-hop relations Compositional canonical utterances typically feature chained multi-hop relations that are joined together (*e.g.*, *Author that writes paper whose topic is NLP*), which can be compressed using more succinct phrases to denote the relation chain, where the intermediary pivoting entities (*e.g.*, `paper`) are omitted (*e.g.*, *Author that works on NLP*). The pattern is referred to as sub-lexical compositionality in Wang et al. [195] and used by annotators to compress verbose canonical utterances, while we model them using grammar rules. Refer to Appendix A.1 for more details.

Idiomatic Comparatives and Superlatives The general grammar in Fig. 7.1a uses canonical constructs for comparative (*e.g.*, *smaller than*) and superlative (*e.g.*, *largest*) utterances (*e.g.*, $\boldsymbol{u}_1$), which is not ideal for entity types with special units (*e.g.*, time, length). We therefore create productions specifying idiomatic comparative and superlative expressions (*e.g.*, *paper published before 2014*, and $\boldsymbol{u}_2$ in Fig. 7.1). Sometimes, answering a superlative utterance also requires reasoning with other pivoting entities. For instance, the relation in *"venue that X publish mostly in"* between authors and venues implicitly involves counting the papers that X publishes. For such cases, we create "macro" productions, with the NL phrase mapped to a program that captures the computation involving the pivoting entity (see Appendix A.1 for more information).

In line with Marzoev et al. [129], Su and Yan [179], we remark that such *functionality*-driven grammar engineering to cover patterns in real data is more efficient and cost-effective than example-driven annotation, as SCFG rules are easily comprehensible with basic knowledge of English syntax, and synthetic samples can be further paraphrased to significantly increase linguistic diversity.

Naturalness-driven Data Selection

To cover real programs in $\mathbb{D}_{\text{nat}}$ with complex structures while tackling the exponential sample space, we propose an efficient approach to sub-sample a small set of examples from this space as seed canonical data $\mathbb{D}_{\text{can}}$ (Fig. 7.1b) for paraphrasing. Our core idea is to only retain a set of examples $\langle \boldsymbol{u}, \boldsymbol{z} \rangle$ that most likely reflect the intents of real users. We use the probability

$p_{\text{LM}}(\boldsymbol{u})$ measured by a language model to approximate the "naturalness" of canonical examples.[1] Specifically, given all canonical examples allowed by the grammar, we form buckets based on their derivation depth d. For each bucket $\mathbb{D}_{\text{can}}^{(d)}$, we compute $p_{\text{LM}}(\boldsymbol{u})$ for its examples, and group the examples using program templates as the key (*e.g.*, $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ in Fig. 7.1 are grouped together). For each group, we find the example $\langle \boldsymbol{u}^*, \boldsymbol{z} \rangle$ with the highest $p_{\text{LM}}(\boldsymbol{u}^*)$, and discard other examples $\langle \boldsymbol{u}, \boldsymbol{z} \rangle$ if $p_{\text{LM}}(\boldsymbol{u}^*) - p_{\text{LM}}(\boldsymbol{u}) > \delta$ ($\delta = 5.0$), removing unlikely utterances from the group (*e.g.*, $\boldsymbol{u}_1$). Finally, we rank all groups in $\mathbb{D}_{\text{can}}^{(d)}$ based on $p_{\text{LM}}(\boldsymbol{u}^*)$, and retain examples in the top-K groups. This method offers trade-off between program coverage and efficiency and, more surprisingly, we show that using only $0.2\% \sim 1\%$ top-ranked examples also results in significantly better final accuracy (§7.4).

7.3.2 Generating Validation Data

Zero-shot learning is non-trivial without a high-quality validation set, as the model might overfit on the (paraphrased) canonical data, which is subject to language and logical mismatch. While existing methods [205] circumvent the issue using real validation data, in this work we create validation sets from paraphrased examples, making our method truly labeled data-free. Specifically, we consider a two-stage procedure. First, we run the iterative paraphrasing algorithm (§7.2) without validation, and then sample $\langle \boldsymbol{u}, \boldsymbol{z} \rangle$ from its output with a probability $p(\boldsymbol{u}, \boldsymbol{z}) \propto p_{\text{LM}}(\boldsymbol{u})^{\alpha}$ ($\alpha = 0.4$), ensuring the resulting sampled set $\mathbb{D}_{\text{par}}^{\text{val}}$ is representative. Second, we restart training using $\mathbb{D}_{\text{par}}^{\text{val}}$ for validation to find the best checkpoint. The paraphrase filtering model is also initialized with the parser trained in the first stage, which has higher precision and accepts more valid paraphrases. This is similar to iterative training of weakly-supervised semantic parsers [45], where the model searches for candidate programs for unlabeled utterances in multiple stages of learning.

7.4 Experiments

We evaluate our zero-shot parser on two datasets.

Scholar [81] is a collection of utterances querying an academic database (Fig. 7.1). Examples are collected from users interacting with a parser, which are later augmented with Turker paraphrases. We use the version from HB19 with programs represented in λ-calculus logical forms. The sizes of the train/test splits are 579/211.

[1] We use the GPT-2 XL model [158].

Geo [235] is a classical dataset with queries about U.S. geography (*e.g.*, *Which rivers run through states bordering California?*). Its database contains basic geographical entities like cities, states, and rivers. We also use the release from HB19, of size 537/237.

7.4.1 Setup

Models and Configuration Our semantic parser is a sequence-to-sequence model with a pre-trained $\text{BERT}_{\text{Base}}$ encoder [49] and an LSTM decoder augmented with a copy mechanism. The paraphraser is a $\text{BART}_{\text{Large}}$ model [108], fine-tuned on the paraphrase generation dataset released by Krishna et al. [93], which consists of lexically and syntactically diverse paraphrases. We use the same set of hyper-parameters for both datasets. Specifically, we synthesize canonical examples from the SCFG with a maximal program depth of 6, and collect the top-K ($K = 2,000$) GPT-scored sample groups for each depth as the seed canonical data $\mathbb{D}_{\text{can}}$ (§7.3.1). We perform the iterative paraphrasing and training procedure (§7.2) for two iterations. We create validation sets of size $2,000$ in the first stage of learning (§7.3.2), and perform validation using perplexity in the second stage. We select the above hyper-parameters using the *natural* Dev sets of Scholar. Note that our model does *not* use any natural examples in both datasets during model training and validation.[2]

Measuring Language and Logical Gaps We measure the language mismatch between utterances in the paraphrased canonical ($\mathbb{D}_{\text{par}}$) and natural ($\mathbb{D}_{\text{nat}}$) data using **perplexities** of natural utterances in $\mathbb{D}_{\text{nat}}$ given by a GPT-2 LM fine-tuned on $\mathbb{D}_{\text{par}}$. For logical gap, we follow HB19 and compute the **coverage** of natural programs $\boldsymbol{z} \in \mathbb{D}_{\text{nat}}$ in $\mathbb{D}_{\text{par}}$.

Metric We report **denotation accuracy** on the execution results of model-predicted programs. We ran all experiments with five random restarts.

7.4.2 Results

We first compare our model with existing approaches using labeled data. Next, we analyze how our proposed methods close the language and logical gaps. Tab. 7.1 reports accuracies of

[2]Ideally, we should not use any natural data when developing the model. However, it is hard to create a truly zero-shot scenario since we already have prior knowledge about the language style and query patterns of utterances in these datasets. We envision that a better evaluation setting is through shared tasks, where evaluation examples are held-out from participants, who are only presented with a handful of natural examples for model development.

System	Supervision	SCHOLAR	GEO
Supervised†	Labeled Examples	83.4	86.3
OVERNIGHT†	Manual Paraphrases	40.8	61.9
GRANNO†	Real Utterances, Manual Paraphrase Detection	69.2	72.0
Our System	–	**75.5**	**74.1**

Table 7.1: Denotation accuracies on TEST sets. †Results reported in Herzig and Berant [73].

various systems on the test sets, as well as their form of supervision. Specifically, the **supervised** parser uses a standard parallel corpus $\mathbb{D}_{\text{nat}}$ of real utterances annotated with programs. **OVERNIGHT** uses paraphrased synthetic examples $\mathbb{D}_{\text{par}}$ like our model, but with manually written paraphrases. **GRANNO** uses unlabeled real utterances $\boldsymbol{u}_{\text{nat}} \in \mathbb{D}_{\text{nat}}$, and manual paraphrase detection to pair $\boldsymbol{u}_{\text{nat}}$ with the canonical examples $\mathbb{D}_{\text{can}}$. Our model outperforms existing approaches on the two benchmarks without using any annotated data, while GRANNO, the currently most cost-effective approach, still spends $155 in manual annotation (besides collecting real utterances) to create training data for the two datasets (Herzig and Berant [73], HB19). Overall, the results demonstrate that zero-shot parser based on idiomatic synchronous grammars and automatic paraphrasing using pre-trained LMs is a data-efficient and cost-effective paradigm to train semantic parsers for emerging domains.

Still, our system falls behind fully supervised models trained on natural datasets $\mathbb{D}_{\text{nat}}$, due to language and logical gaps between $\mathbb{D}_{\text{par}}$ and $\mathbb{D}_{\text{nat}}$. In following experiments, we explore whether our proposed methods are effective at narrowing the gaps and improving accuracy. Since the validation splits of the two datasets are small (*e.g.*, only 99 samples for SCHOLAR), we use the full training/validation splits for evaluation to get more reliable results.

More expressive grammars narrow language and logical gaps We capture domain-specific language patterns using idiomatic productions to close language mismatch (§7.3.1). Tables 7.2 and 7.3 list the results when we gradually improve the expressiveness of the grammar by adding different types of idiomatic productions. We observe that more expressive grammars help close the language gap, as indicated by the decreasing perplexities. This is especially important for SCHOLAR, which features diverse idiomatic language styles that are hard to infer from plain canonical utterances. For instance, it could be non-trivial to paraphrase canonical utterances with multi-hop relations (*e.g.*, *Author that cites paper by X*) or superlative queries (*e.g.*, *Topic of the most number of ACL paper*) to more idiomatic alternatives (*e.g.*, "*Author that cites X*", and

Grammar	Acc.	PPL	Logical Coverage	
			$\mathbb{D}_{\text{can}}$	$\mathbb{D}_{\text{par}}$
Base	66.3	23.0	80.6	75.8
+Multihop Rel.	67.0	22.0	87.7	81.2
+Comparison	67.3	21.7	86.5	80.2
+Superlative	77.8	20.9	90.6	86.1
−Multihop Rel.	75.8	20.8	83.9	81.1

Table 7.2: Ablation of grammar categories on SCHOLAR.

Grammar	Acc.	PPL	Logical Coverage	
			$\mathbb{D}_{\text{can}}$	$\mathbb{D}_{\text{par}}$
Base	64.5	8.2	84.4	79.7
+Multihop Rel.	67.9	8.1	83.6	79.7
+Superlative	72.8	8.0	84.1	79.4
−Multihop Rel.	66.5	8.2	84.1	80.0

Table 7.3: Ablation study of grammar categories on GEO.

"*The most popular topic for ACL paper*"), while directly including such patterns in the grammar (+**Multihop Rel.** and +**Superlative**) is helpful.

Additionally, we observe that more expressive grammars also improve logical coverage. The last columns (**Logical Coverage**) of Tables 7.2 and 7.3 report the percentage of real programs that are covered by the seed canonical data before ($\mathbb{D}_{\text{can}}$) and after ($\mathbb{D}_{\text{par}}$) iterative paraphrasing. Intuitively, idiomatic grammar rules could capture compositional program patterns like multi-hop relations and complex superlative queries (*e.g.*, *Author that publish mostly in ACL*, §7.3.1) within a single production, enabling the grammar to generate more compositional programs under the same threshold on the derivation depth. Notably, when adding all the idiomatic productions on SCHOLAR, the number of exhaustively generated examples with a program depth of 6 is tripled ($530K \mapsto 1,700K$).

Moreover, recall that the seed canonical dataset $\mathbb{D}_{\text{can}}$ contains examples with highly-likely utterances under the LM (§7.3.1). Therefore, examples created by idiomatic productions are more likely to be included in $\mathbb{D}_{\text{can}}$, as their more natural and well-formed utterances often have higher LM scores. However, note that this could also be counter-productive, as examples created with idiomatic productions could dominate the LM-filtered $\mathbb{D}_{\text{can}}$, "crowding out" other useful examples with lower LM scores. This likely explains the slightly decreased logical coverage on GEO (Tab. 7.3), as more than 30% samples in the filtered $\mathbb{D}_{\text{can}}$ include idiomatic multi-hop relations directly connecting geographic entities with their countries (*e.g.*, "*City in US*", *c.f.* "*City in state in US*"), while such examples only account for $\sim 8\%$ of real data. While the over-representation issue might not negatively impact accuracy, we leave generating more balanced synthetic data as important future work.

Finally, we note that the logical coverage drops after paraphrasing ($\mathbb{D}_{\text{can}}$ *v.s.* $\mathbb{D}_{\text{par}}$ in Tables 7.2 and 7.3). This is because for some samples in $\mathbb{D}_{\text{can}}$, the paraphrase filtering model rejects all their

	K	Train Data Size		Acc	PPL	Logical Coverage		In Coverage		Out of Coverage	
		$\|\mathbb{D}_{\text{can}}\|$	$\|\mathbb{D}_{\text{par}}\|$			$\mathbb{D}_{\text{can}}$	$\mathbb{D}_{\text{par}}$	Acc	PPL	Acc	PPL
Scholar	500	1,554	45,269	74.0	22.0	79.4	76.0 (14.5)	82.3	23.4	47.6	18.2
	1000	3,129	80,481	75.9	21.4	88.0	82.0 (9.4)	81.4	21.3	50.6	21.7
	2000	5,486	129,955	77.8	20.9	90.6	86.1 (7.5)	82.2	20.7	50.2	22.7
	4000	9,239	202,429	78.4	20.7	91.9	87.4 (4.9)	83.2	20.5	45.3	22.0
	8000	17,077	330,548	75.5	21.5	92.0	88.2 (2.9)	79.8	21.4	43.4	22.4
Geo	500	1,351	29,835	61.6	8.4	70.3	64.4 (14.2)	79.2	7.6	29.8	9.9
	1000	2,586	55,117	68.5	8.2	80.5	74.9 (9.0)	81.4	7.4	28.8	11.3
	2000	5,413	112,530	72.8	8.0	84.1	79.4 (5.2)	82.0	7.4	37.6	10.8
	4000	11,085	182,469	67.5	8.2	84.9	78.3 (3.1)	75.5	7.6	38.8	11.2
	8000	16,312	243,343	67.9	8.2	85.4	78.0 (2.1)	75.5	7.5	41.3	11.2

Table 7.4: Results on Scholar and Geo with varying amount of canonical examples in the seed training data.

paraphrases. We provide further analysis later in a case study.

Do smaller logical gaps entail better performance? As in §7.3.1, the seed canonical data $\mathbb{D}_{\text{can}}$ consists of top-K highest-scoring examples under GPT-2 for each program depth. This data selection method makes it possible to train the model efficiently in the iterative paraphrasing stage using a small set of canonical samples that most likely appear in natural data out of the exponentially large sample space. However, using a smaller cutoff threshold K might sacrifice logical coverage, as fewer examples are in $\mathbb{D}_{\text{can}}$. To investigate this trade-off, we report results with varying K in Tab. 7.4. Notably, with $K = 1000$ and around $3K$ seed canonical data $\mathbb{D}_{\text{can}}$ (before iterative paraphrasing), $\mathbb{D}_{\text{can}}$ already covers 88% and 80% natural programs on Scholar and Geo, resp. This small portion of samples only account for 0.2% (1%) of the full set of $1.7M+$ ($0.27M$) canonical examples exhaustively generated from the grammar on Scholar (Geo). This demonstrates our data selection approach is effective in maintaining learning efficiency while closing the logical gap.

More interestingly, while larger K yields higher logical form coverage, the accuracy might not improve. This is possibly because while the recall of real programs improves, the percentage of such programs in paraphrased canonical data $\mathbb{D}_{\text{par}}$ (numbers in parentheses) actually drops. Out of the remaining $90\%+$ samples in $\mathbb{D}_{\text{par}}$ whose programs are not in $\mathbb{D}_{\text{nat}}$, many have unnatural intents that real users are unlikely to issue (*e.g.*, "*Number of titles of papers with the*

smallest citations", or "*Mountain whose elevation is the length of Colorado River*"). Such *unlikely* samples are potentially harmful to the model, causing worse language mismatch, as suggested by the increasing perplexity when $K = 8000$. Similar to HB19, empirically we observe around one-third of samples in $\mathbb{D}_{\text{can}}$ and $\mathbb{D}_{\text{par}}$ are unlikely. As later in the case study, such unlikely utterances have noisier paraphrases, which hurts the quality of $\mathbb{D}_{\text{par}}$.

Next, to investigate whether the model could generalize to utterances with out-of-distribution program patterns not seen in the training data $\mathbb{D}_{\text{par}}$, we report accuracies on the splits whose program templates are covered (**In Coverage**) and not covered (**Out of Coverage**) by $\mathbb{D}_{\text{par}}$. Not surprisingly, the model performs significantly better on the in-coverage sets with less language mismatch. An exception is K=500 on Scholar, where the perplexity on out-of-coverage samples is slightly lower. This is because utterances in Scholar tend to use compound nouns to specify compositional constraints (*e.g.*, *ACL 2021 parsing papers*), a language style common for in-coverage samples but not captured by the grammar. With smaller K and $\mathbb{D}_{\text{can}}$, it is less likely for the paraphrased data $\mathbb{D}_{\text{par}}$ to capture similar syntactic patterns. Anther factor that makes the out-of-coverage PPL smaller when $K = 500$ is that there are more (simpler) examples in the set compared to $K > 500$, and the relatively simple utterances will also bring down the PPL.

Our results are also in line with recent research in compositional generalization of semantic parsers [55, 101], which suggests that existing models generalize poorly to utterances with novel compositional patterns (*e.g.*, conjunctive objects like *Most cited paper by X and Y*) not seen during training. Still surprisingly, our model generalizes reasonably to compositionally novel (out-of-coverage) splits, registering $30\%\sim50\%$ accuracies, in contrast to HB19 reporting accuracies smaller than 10% on similar benchmarks for OverNight. We hypothesize that synthesizing compositional samples increases the number of unique program templates in training, which could be helpful for compositional generalization [2]. As an example, the number of unique program templates in $\mathbb{D}_{\text{par}}$ when $K = 2000$ on Scholar and Geo is $1.9K$ and $1.7K$, resp, compared to only 125 and 187 in $\mathbb{D}_{\text{nat}}$. This finding is reminiscent of data augmentation strategies for supervised parsers using synthetic samples induced from (annotated) parallel data [84, 191].

Impact of Paraphrasers Our system relies on strong paraphrasers to generate diverse utterances in order to close the language gap. Tab. 7.5 compares the performance of the system trained with our paraphraser and the one used in Xu et al. [205]. Both models are based on Bart, while our paraphraser is fine-tuned to encourage lexically and syntactically diverse outputs. We measure lexical diversity using token-level F_1 between the original and paraphrased

Paraphraser	Scholar			Geo		
	Tok. F_1↓	τ↓	Acc.↑	Tok. F_1↓	τ↓	Acc.↑
Ours	70.3	0.71	77.8	69.2	0.78	72.8
Xu et al. [205]	72.4	0.94	69.9	74.5	0.95	62.3

Table 7.5: Systems with different paraphrasers. We report end-to-end denotation accuracy, as well as F_1 and Kendall's τ rank coefficient between utterances and their paraphrases.

utterances $\langle \boldsymbol{u}, \boldsymbol{u}' \rangle$ [93, 161]. For syntactic divergence, we use Kendall's τ [104] to compute the ordinal correlation between $\boldsymbol{u}$ and $\boldsymbol{u}'$, which intuitively measures the number of times to swap tokens in $\boldsymbol{u}$ to get $\boldsymbol{u}'$ using bubble sort. Our paraphraser generates more diverse paraphrases (*e.g.*, *What is the biggest state in US?*) from the source (*e.g.*, *State in US and that has the largest area*), as indicated by lower token-level overlaps and ordinal coefficients, comparing to the existing paraphraser (*e.g.*, *The state in US with the largest surface area*). Nevertheless, our paraphraser is still not perfect, as discussed next.

Limitations Our parser still lags behind the fully supervised model (Tab. 7.1). To understand the remaining bottlenecks, we show representative examples in Tab. 7.6. First, the recall of the paraphrase filtering model is low. The filtering model uses the semantic parser trained on the paraphrased data generated in previous iterations. Since this model is less accurate, it can incorrectly reject valid paraphrases $\boldsymbol{u}'$ (⊗ in Tab. 7.6), especially when $\boldsymbol{u}'$ uses a different sentence type (*e.g.*, questions) than the source (*e.g.*, statements). Empirically, we found the recall of the filtering model at the first iteration of the second-stage training (§7.3.2) is only around 60%. This creates logical gaps, as paraphrases of examples in the seed canonical data $\mathbb{D}_{\text{can}}$ could be rejected by the conservative filtering model, leaving no samples with the same programs in $\mathbb{D}_{\text{par}}$.

Another issue is the imperfect paraphraser generating semantically incorrect predictions (*e.g.*, $\boldsymbol{u}'_{1,1}$), especially when the source canonical utterance contains uncommon concepts (*e.g.*, *venue* in $\boldsymbol{u}_1$), which tend to be ignored or interpreted as other entities (*e.g.*, *sites*). Additionally, while we have attempted to close the language gap using more idiomatic utterances, the paraphraser still fails to generate some language patterns in real utterances. This is especially non-trivial for relations not covered by our idiomatic productions (*e.g.*, the co-authorship multi-hop relation in $\boldsymbol{u}_2$). Besides the lack of coverage for idiomatic relations, utterances can also follow special compositionality patterns. For instance, $\boldsymbol{u}^*_{\text{nat}}$ in Example 3 uses compound nouns to denote the occurrence of a conference, which is difficult to automatically paraphrase from $\boldsymbol{u}_3$ (that uses

Example 1 (Uncommon Concept)		
$\boldsymbol{u}_1$	*Venue of paper by* `author`$_0$ *and published in* `year`$_0$	
$\boldsymbol{u}'_{1,1}$	`author`$_0$*'s paper, published in* `year`$_0$	❌
$\boldsymbol{u}'_{1,2}$	*Where the paper was published by* `author`$_0$ *in* `year`$_0$*?*	⊗
$\boldsymbol{u}'_{1,3}$	*Where the paper was published in* `year`$_0$ *by* `author`$_0$*?*	⊗
$\boldsymbol{u}^*_{\text{nat}}$	*Where did* `author`$_0$ *publish in* `year`$_0$*?*	(Wrong Answer)
Example 2 (Novel Relation)		
$\boldsymbol{u}_2$	*Author of paper by* `author`$_0$	
$\boldsymbol{u}'_{2,1}$	*Author of the paper written by* `author`$_0$	✅
$\boldsymbol{u}'_{2,2}$	*Author of* `author`$_0$*'s paper*	✅
$\boldsymbol{u}'_{2,3}$	*Who wrote the paper* `author`$_0$ *wrote?*	⊗
$\boldsymbol{u}^*_{\text{nat}}$	*Co-authors of* `author`$_0$	(Wrong Answer)
Example 3 (Novel Language Pattern)		
$\boldsymbol{u}_3$	*Author of paper published in* `venue`$_0$ *and in* `year`$_0$	
$\boldsymbol{u}'_{3,1}$	*Author of papers published in* `venue`$_0$ *in* `year`$_0$	✅
$\boldsymbol{u}'_{3,2}$	*Who wrote a paper for* `venue`$_0$ *in* `year`$_0$	⊗
$\boldsymbol{u}'_{3,3}$	*Who wrote the* `venue`$_0$ *paper in* `year`$_0$	⊗
$\boldsymbol{u}^*_{\text{nat}}$	`venue`$_0$ `year`$_0$ *authors*	(Correct)
Example 4 (Unlikely Example)		
$\boldsymbol{u}_4$	*Paper in* `year`$_0$ *and whose author is not the most cited author*	
$\boldsymbol{u}'_{4,1}$	*A paper published in* `year`$_0$ *that isn't the most cited author*	☑
$\boldsymbol{u}'_{4,2}$	*What's not the most cited author in* `year`$_0$	☑
$\boldsymbol{u}'_{4,3}$	*In* `year`$_0$*, he was not the most cited author*	☑

Table 7.6: Case Study on SCHOLAR. We show the seed canonical utterance $\boldsymbol{u}_i$, the paraphrases $\boldsymbol{u}'_{i,j}$, and the relevant natural examples $\boldsymbol{u}^*_{\text{nat}}$. ✅ and ❌ denote the correctness of paraphrases. ⊗ denotes false negatives of the filtering model (correct paraphrases that are filtered), ☑ denotes false positives (incorrect paraphrases that are accepted). Entities are canonicalized with indexed `slots`.

prepositional phrases) without any domain knowledge. While the model could still correctly answer $\boldsymbol{u}^*_{\text{nat}}$ in this case, $\boldsymbol{u}^*_{\text{nat}}$'s perplexity is high, suggesting language mismatch.

Finally, as discussed earlier, $\mathbb{D}_{\text{can}}$ contains around 30% unlikely canonical examples (*e.g.*, $\boldsymbol{u}_4$). Since these utterances have convoluted logic or do not resemble natural English sentences, their paraphrases are much noisier (*e.g.*, $\boldsymbol{u}'_{4,*}$). Empirically, we observe the paraphraser's accuracy is only around 30% for unlikely utterances, compared to 70% for the likely ones. The filtering model is also less effective on unlikely examples (false positives ✓). These noisy samples will eventually hurt performance of the parser. We leave modeling utterance naturalness as important future work.

7.5 Related Work

We present a systematic review of data efficient learning approaches for semantic parsers in §2.2. Our work is closely related to the family of models based on the OverNight framework [§2.2.1; 195], which synthesizes parallel corpora from synchronous context free grammars [37, 204] or neural sequence models [63]. The field has attempted to bridge the gaps between canonical and real utterances via paraphrase detection [73] and generation [171, 179], or representation learning [129].

7.6 Summary

In this chapter, we propose a cost-effective approach to collect parallel training data of utterances labeled with MRs using zero-shot data synthesis and iterative paraphrasing. We put forward methods to close the language and logical gaps between synthetic and real data. On Scholar and Geo, our model achieves competitive results compared to other approaches while using *zero* labeled data. This work demonstrates data synthesis based on grammar engineering as a viable paradigm for task adaptation of general-purpose pre-trained language models. On the one hand, pre-trained LMs, which capture open-domain knowledge of text, provide useful signals to filter unnatural synthetic examples. On the other hand, those paraphrased synthetic samples generated from idiomatic grammars could be used to fine-tune pre-trained LMs modeled as semantic parsers to quickly adapt to the target domain. This paradigm of task-specific data synthesis for fine-tuning LMs has shown promising results for natural language understanding tasks with structured data [232, 233].

In Part I we have discussed methods that model program syntax for more accurate parsing. The approach presented in this chapter can be viewed as a way to leverage natural language syntax to improve semantic parsing. Instead of building encoders that explicitly follow the syntax of utterances (*e.g.*, using tree LSTMs), we use SCFGs to model utterance syntax and train neural semantic parsers to implicitly capture such syntactic knowledge by learning to parse utterances following the pre-defined syntactic patterns, and also generalize to understand those with more diverse syntactic styles after iterative paraphrasing.

For future work, we plan to simplify the process of constructing SCFG rules and make this model a general-purpose and easy-to-use framework for developing semantic parsers for emerging new domains. More directions are discussed in Chapter 10.

Part III

Data Efficient Approaches

Chapter 8

Semi-supervised Learning

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 7 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

8.1 Overview

Recent advances in semantic parsing research are largely attributed to the success of neural network models [50, 81, 117, 202, 244]. However, as discussed in Chapter 1 and §2.2.1, 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 par-

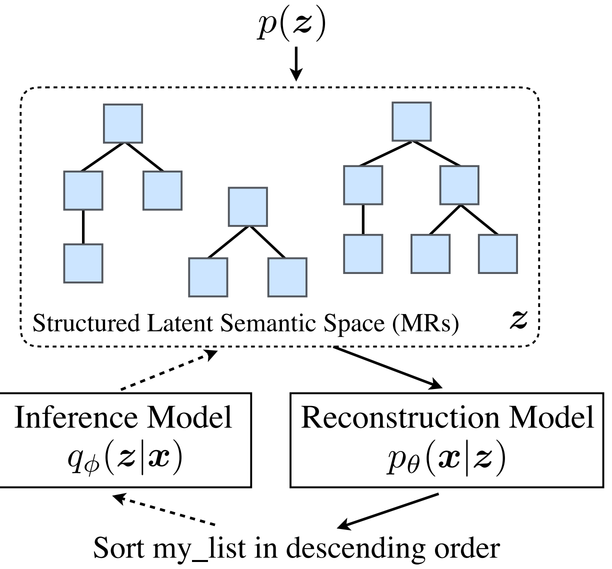


Figure 8.1: Graphical Representation of StructVAE

allel data has become the bottleneck of existing, purely supervised-based models. These data requirements can be alleviated with *weakly-supervised* learning (§2.2.2), where the denotations (*e.g.*, answers in question answering) of MRs (*e.g.*, logical form queries) are used as indirect supervision (Berant et al. [18], Clarke et al. [42], Liang et al. [113], *inter alia*), or *data-augmentation techniques* that automatically generate pseudo-parallel corpora using hand-crafted or induced grammars [84, 195] §2.2.1.

In this work, we focus on *semi-supervised* learning (§2.2.3), 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 [91, 134], and propose StructVAE — a principled deep generative approach for semi-supervised learning with tree-structured latent variables (Fig. 8.1). 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 (§8.3.2); (2) a *reconstruction model* decodes the latent MR into the original observed utterance (§8.3.1). 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 (§8.3.3).

In addition to these contributions to semi-supervised semantic parsing, StructVAE con-

tributes 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 [89], discrete symbols [134], or hybrids of the two [245].

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, §8.3.2) 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.

8.2 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.

8.2.1 Supervised and Semi-supervised Training

As noted in §2.1, 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 Fig. 4.1 in Chapter 4 for a running example Python AST), although application of a similar framework to graph-structured representations is also feasible.

In this chapter, we consider semi-supervised learning of semantic parsers (§2.2.3), which jointly maximizes the conditional likelihood $p_\phi(\boldsymbol{z}|\boldsymbol{u})$ using utterances $\boldsymbol{u}$ labled with MRs $\boldsymbol{z}$, $\mathbb{L} = \{\langle \boldsymbol{u}, \boldsymbol{z} \rangle\}$, and the marginal likelihood $p(\boldsymbol{u})$ over unlabled utterances $\mathbb{U}$ (Eq. (2.4)). StructVAE uses the variational auto-encoding framework to jointly optimize $p_\phi(\boldsymbol{z}|\boldsymbol{u})$ and $p(\boldsymbol{u})$ in Eq. (2.4), as outlined in §8.2.2 and detailed in §8.3.

8.2.2 VAEs for Semi-supervised Learning

From Eq. (2.4), our semi-supervised model must be able to calculate the probability $p(\boldsymbol{u})$ of unlabeled NL utterances. To model $p(\boldsymbol{u})$, we use VAEs, which provide a principled framework for

generative models using neural networks [89]. As shown in Fig. 8.1, VAEs define a *generative story* (bold arrows in Fig. 8.1, explained in §8.3.1) to model $p(\boldsymbol{u})$, where a latent MR $\boldsymbol{z}$ is sampled from a prior, and then passed to the *reconstruction* model to decode into the surface utterance $\boldsymbol{u}$. There is also an *inference* model $q_\phi(\boldsymbol{z}|\boldsymbol{u})$ that allows us to infer the most probable latent MR $\boldsymbol{z}$ given the input $\boldsymbol{u}$ (dashed arrows in Fig. 8.1, explained in §8.3.2). 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 $p(\boldsymbol{u})$ using the inference and reconstruction models, allowing us to effectively optimize the unsupervised and supervised objectives in Eq. (2.4) in a joint fashion (Kingma et al. [90], explained in §8.3.3).

8.3 StructVAE: VAEs with Tree-structured Latent Variables

8.3.1 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 $p_\theta(\boldsymbol{u}|\boldsymbol{z})$ then decodes the sampled MR $\boldsymbol{z}$ into the observed NL utterance $\boldsymbol{u}$.

Both the prior $p(\boldsymbol{z})$ and the reconstruction model $p(\boldsymbol{u}|\boldsymbol{z})$ takes tree-structured MRs as inputs. To model such inputs with rich internal structures, we follow Konstas et al. [92], 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 $p_\theta(\boldsymbol{u}|\boldsymbol{z}) \triangleq p_\theta(\boldsymbol{u}|\boldsymbol{z}^s)$[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 $p_\theta(\boldsymbol{u}|\boldsymbol{z})$. As we will explain in §8.4.3, 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 [123], augmented with a copying mechanism [62], allowing an out-of-vocabulary (OOV) entity in $\boldsymbol{z}^s$ to be copied to $\boldsymbol{u}$ (*e.g.*, the variable name `my_list` in Fig. 8.1).

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

8.3.2 Inference Model

StructVAE models the semantic parser $p_\phi(\boldsymbol{z}|\boldsymbol{u})$ as the inference model $q_\phi(\boldsymbol{z}|\boldsymbol{u})$ in VAE (§8.2.2), which maps NL utterances $\boldsymbol{u}$ into tree-structured meaning representations $\boldsymbol{z}$. $q_\phi(\boldsymbol{z}|\boldsymbol{u})$ can be any trainable semantic parser, with the corresponding MRs forming the structured latent semantic space. In this work, we primarily use the semantic parser proposed in Chapter 4 based on the Abstract Syntax Description Language (ASDL) framework [194] as the inference model. The parser encodes $\boldsymbol{u}$ into ASTs, which 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 §8.4.2 for details). Interested readers are referred to Chapter 4 for details of the semantic parser used as the inference model. 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 §8.4.3).

8.3.3 Semi-supervised Learning

In this section we explain how to optimize the semi-supervised learning objective Eq. (2.4) 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:

$$\mathcal{J}_s \triangleq \sum_{(\boldsymbol{u},\boldsymbol{z})\in\mathbb{L}} \big(\log q_\phi(\boldsymbol{z}|\boldsymbol{u}) + \log p_\theta(\boldsymbol{u}|\boldsymbol{z})\big) \tag{8.1}$$

Unsupervised Learning To optimize the unsupervised learning objective $\mathcal{J}_u$ in Eq. (2.4), we maximize the variational lower-bound of $\log p(\boldsymbol{u})$:

$$\log p(\boldsymbol{u}) \geq \mathbb{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|\boldsymbol{u})}\big(\log p_\theta(\boldsymbol{u}|\boldsymbol{z})\big) - \lambda \cdot \mathrm{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{u})||p(\boldsymbol{z})] = \mathcal{L} \tag{8.2}$$

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 [22, 134].

To optimize the parameters of our model in the face of non-differentiable discrete latent variables, we follow Miao and Blunsom [134], and approximate $\frac{\partial \mathcal{L}}{\partial \phi}$ using the score function

estimator (a.k.a. REINFORCE, Williams [198]):

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \phi} &= \frac{\partial}{\partial \phi}\mathbb{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|\boldsymbol{u})} \\ &\quad \underbrace{\Big(\log p_\theta(\boldsymbol{u}|\boldsymbol{z}) - \lambda\big(\log q_\phi(\boldsymbol{z}|\boldsymbol{u}) - \log p(\boldsymbol{z})\big)\Big)}_{\text{learning signal}} \\ &= \frac{\partial}{\partial \phi}\mathbb{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|\boldsymbol{u})} l'(\boldsymbol{u},\boldsymbol{z}) \\ &\approx \frac{1}{|\mathcal{S}(\boldsymbol{u})|}\sum_{\boldsymbol{z}_i\in\mathcal{S}(\boldsymbol{u})} l'(\boldsymbol{u},\boldsymbol{z}_i)\frac{\partial \log q_\phi(\boldsymbol{z}_i|\boldsymbol{u})}{\partial \phi}\end{aligned} \tag{8.3}$$

where we approximate the gradient using a set of samples $\mathcal{S}(\boldsymbol{u})$ drawn from $q_\phi(\cdot|\boldsymbol{u})$. To ensure the quality of sampled latent MRs, we follow Guu et al. [66] and use beam search. The term $l'(\boldsymbol{u},\boldsymbol{z})$ is defined as the *learning signal* [134]. 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 $b(\boldsymbol{u})$ to stabilize learning, and re-define the learning signal as

$$l(\boldsymbol{u},\boldsymbol{z}) \triangleq l'(\boldsymbol{u},\boldsymbol{z}) - b(\boldsymbol{u}). \tag{8.4}$$

Specifically, in StructVAE, we define

$$b(\boldsymbol{u}) = a\cdot\log p(\boldsymbol{u}) + c, \tag{8.5}$$

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

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

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

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 [173], in this chapter we try to empirically test some likely hypotheses. In Eq. (8.3), the gradient received by the inference model from each latent sample $\boldsymbol{z}$ is weighed by the learning signal $l(\boldsymbol{u},\boldsymbol{z})$. $l(\boldsymbol{u},\boldsymbol{z})$ can be viewed as the reward function in REINFORCE learning. It can also be viewed as weights associated with pseudo-training examples $\{\langle\boldsymbol{u},\boldsymbol{z}\rangle : \boldsymbol{z}\in\mathcal{S}(\boldsymbol{u})\}$ 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 $\log p_\theta(\boldsymbol{u}|\boldsymbol{z})$; and (2) have $\boldsymbol{z}$ be succinct and natural, yielding high prior probability. Let $\boldsymbol{z}^*$ denote the gold-standard MR of $\boldsymbol{u}$. Consider the ideal case where $\boldsymbol{z}^* \in \mathcal{S}(\boldsymbol{u})$ and $l(\boldsymbol{u}, \boldsymbol{z}^*)$ is positive, while $l(\boldsymbol{u}, \boldsymbol{z}')$ is negative for other imperfect samples $\boldsymbol{z}' \in \mathcal{S}(\boldsymbol{u})$, $\boldsymbol{z}' \neq \boldsymbol{z}^*$. In this ideal case, $\langle \boldsymbol{u}, \boldsymbol{z}^* \rangle$ would serve as a positive training example and other samples $\langle \boldsymbol{u}, \boldsymbol{z}' \rangle$ 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 [66]. We will present more empirical statistics and observations in §8.4.3.

8.4 Experiments

8.4.1 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 [50], 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. [157] to convert between logical forms and ASTs (see Chapter 4 for details).

Code Generation The Django dataset [146] 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 from Chapter 3 and use the `astor` package to convert ASDL ASTs into Python source code.

8.4.2 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. (8.2) requires sampling structured MRs from the inference model $q_\phi(\boldsymbol{z}|\boldsymbol{u})$. Due to the complexity of the semantic parsing problem, we cannot expect any valid samples from randomly initialized $q_\phi(\boldsymbol{z}|\boldsymbol{u})$. We therefore pre-train the inference and reconstruction models using the supervised objective Eq. (8.1) until convergence, and then optimize using the semi-supervised learning objective Eq. (2.4). Throughout all experiments we set α (Eq. (2.4)) and λ (Eq. (8.2)) to 0.1. The sample size $|\mathcal{S}(\boldsymbol{u})|$ 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 $q_\phi(\boldsymbol{z}|\boldsymbol{u})$. 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 $p(\boldsymbol{z})$ (§8.3.3) and the baseline function Eq. (8.5).

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

8.4.3 Main Results

Tab. 8.1 and Tab. 8.2 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. (8.1)). 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 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

[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.

$\vert\mathbb{L}\vert$	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 [237]		84.6	
WKZ14 [187]		**91.3**	
Seq2Tree [50][†]		84.6	
ASN [157][†]		85.3	
+ supervised attention		85.9	

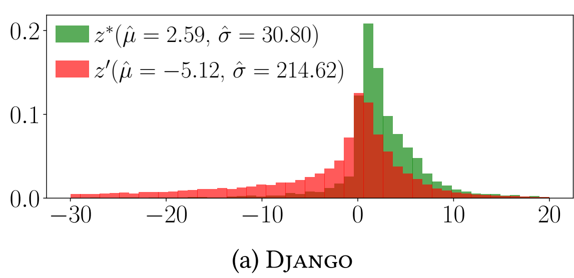
Table 8.1: Performance on Atis w.r.t. the size of labeled training data $\mathbb{L}$. [†]Existing neural network-based methods

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 system in Chapter 3, 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 Chapter 3.

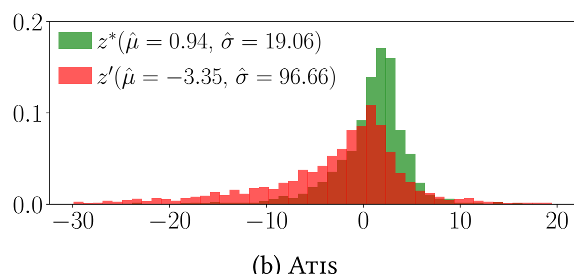
Semi-supervised Learning Next, we discuss our main comparison between StructVAE with the supervised version of the parser (recall that the supervised parser is used as the inference model in StructVAE, §8.3.2). First, comparing our proposed StructVAE with the supervised parser when there are extra unlabeled 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 (Chapter 3) using only *half* the training data (71.5 when $|\mathbb{L}| = 8000$ v.s. 71.6 for the model in Chapter 3). 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

$\lvert\mathbb{L}\rvert$	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.
Yin and Neubig [218] (Chapter 3)			71.6

Table 8.2: Performance on Django w.r.t. the size of labeled training data $\mathbb{L}$



(a) Django



(b) Atis

Figure 8.2: Histograms of learning signals on Django ($\lvert\mathbb{L}\rvert = 5000$) and Atis ($\lvert\mathbb{L}\rvert = 2000$). Difference in sample means is statistically significant ($p < 0.05$).

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 (§8.3.3), 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. (8.1) used by the supervised parser, StructVAE has the extra unsupervised objective Eq. (8.2), 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 §8.3.3, in semi-supervised learning, the gradient received by the inference model from each sampled latent MR is weighted by the learning

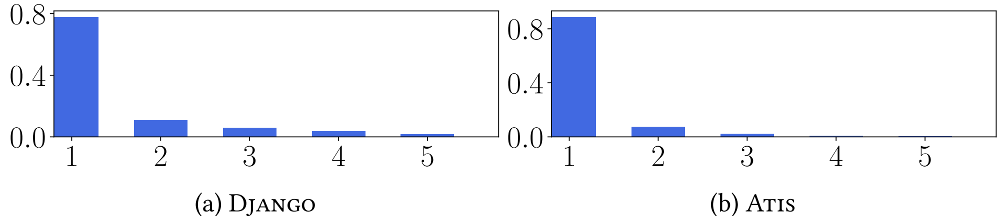


Figure 8.3: Distribution of the rank of $l(\boldsymbol{u}, \boldsymbol{z}^*)$ in sampled set

signal. Empirically, we would expect that on average, the learning signals of gold-standard samples $\boldsymbol{z}^*$, $l(\boldsymbol{u}, \boldsymbol{z}^*)$, are positive, larger than those of other (imperfect) samples $\boldsymbol{z}'$, $l(\boldsymbol{u}, \boldsymbol{z}')$. We therefore study the statistics of $l(\boldsymbol{u}, \boldsymbol{z}^*)$ and $l(\boldsymbol{u}, \boldsymbol{z}')$ for all utterances $\boldsymbol{u} \in \mathbb{U} - \mathbb{L}$, *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. FiguresFig. 8.2a andFig. 8.2b 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 $l(\boldsymbol{u}, \boldsymbol{z}^*)$ being positive and $l(\boldsymbol{u}, \boldsymbol{z})$ negative. Also note that the distribution of $l(\boldsymbol{u}, \boldsymbol{z}^*)$ 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 $l(\boldsymbol{u}, \boldsymbol{z}^*)$, among the learning signals of all samples of $\boldsymbol{u}$, $\{l(\boldsymbol{u}, \boldsymbol{z}_i) : \boldsymbol{z}_i \in \mathcal{S}(\boldsymbol{u})\}$. Results are shown in Fig. 8.3. 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 $\log p(\boldsymbol{u}|\boldsymbol{z})$ and the prior $\log p(\boldsymbol{z})$ to the learning signal, we present examples of inferred latent MRs during training (Tab. 8.3). 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 $\log p(\boldsymbol{u}|\boldsymbol{z})$, 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 $q(\boldsymbol{z}|\boldsymbol{u})$ are lower than those of $\boldsymbol{z}_2^s$. This result further demonstrates that learning signals provide informative gradient weights for

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

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

Table 8.3: Inferred latent MRs on Django ($|\mathbb{L}| = 5000$). For simplicity we show the surface representation of MRs ($\boldsymbol{z}^s$, source code) instead.

$\vert\mathbb{L}\vert$	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 8.4: Performance of the StructVAE-Seq on Atis w.r.t. the size of labeled training data $\mathbb{L}$

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 structure in semantic parsing [28, 178], 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 [123] 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. [36]. We use a classical tri-gram Kneser-Ney language model as the prior. Tab. 8.4 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 Tab. 8.1, we found that the gains are especially larger with few labeled examples — StructVAE-Seq achieves improvements of 8-10 points when $|\mathbb{L}| < 1000$. These results suggest that semi-supervision is especially useful in improving a mediocre parser in low resource settings.

Impact of Baseline Functions In §8.3.3 we discussed our design of the baseline function $b(\boldsymbol{u})$ incorporated in the learning signal (Eq. (8.3)) to stabilize learning, which is based on a language model (LM) over utterances (Eq. (8.5)). 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. Tab. 8.5 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

Atis				Django			
$\|\mathbb{L}\|$	Sup.	MLP	LM	$\|\mathbb{L}\|$	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 8.5: Comparison of StructVAE with different baseline functions $b(\boldsymbol{u})$, *italic*†: semi-supervised learning with the MLP baseline is worse than supervised results.

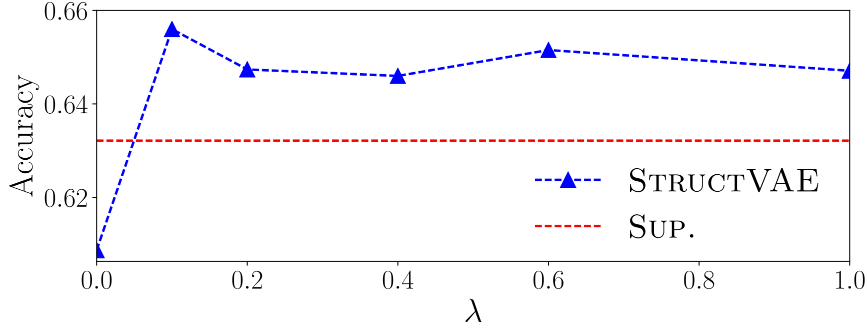


Figure 8.4: Performance on Django ($|\mathbb{L}| = 5000$) w.r.t. the KL weight λ

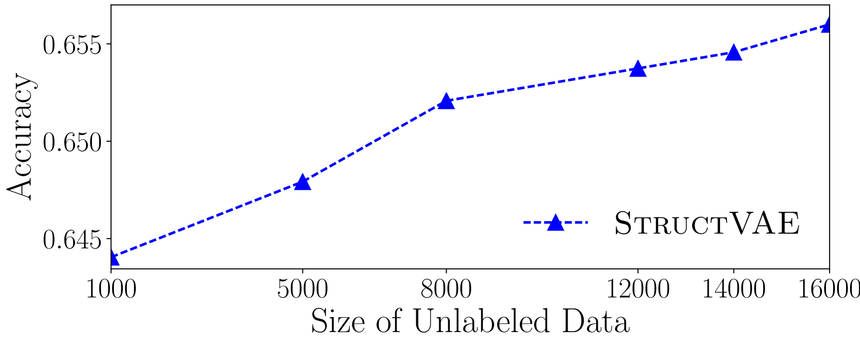


Figure 8.5: Performance on Django ($|\mathbb{L}| = 5000$) w.r.t. the size of unlabeled data $\mathbb{U}$

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 $p(\boldsymbol{z})$ Fig. 8.4 depicts the performance of StructVAE as a function of the KL term weight λ in Eq. (8.2). 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 Tab. 8.3 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 Fig. 8.5 illustrates the accuracies w.r.t. the size of unlabeled data. StructVAE yields consistent gains as the size of the unlabeled data increases.

8.5 Related Works

Semi-supervised Learning for NLP Semi-supervised learning comes with a long history [246], with applications in NLP from early work of self-training [214], and graph-based methods [44], to recent advances in auto-encoders [39, 174, 239] and deep generative methods [206]. Our work follows the line of neural variational inference for text processing [135], and resembles Miao and Blunsom [134], 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. k

Data Efficient Semantic Parsing We present a review of data efficient learning approaches in §2.2. Our work shares similar spirits with Kociský et al. [91], which 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.

8.6 Summary

In this chapter, we propose StructVAE, a deep generative model with tree-structured latent variables for semi-supervised semantic parsing. Under StructVAE, semantic parsers are modeled as inference networks to compute the posterior over latent meaning representations. We apply StructVAE to semantic parsing and code generation tasks, and show it outperforms a strong supervised parser using extra unlabeled data. For future work, an interesting direction is to generalize StructVAE to semi-supervised learning with graph-structured MRs as latent variables, such as AMR parsing [12]. Another important direction would be theoretical explanations of semi-supervised learning with VAEs, such as asymptotic analysis of their convergence rate and sample efficiency.

Chapter 9

Speeding Up Data Acquisition

Chapter 8 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

9.1 Overview

In order to be effective, statistical semantic parsers, like the code generation model presented in Chapter 3, 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 9.1. 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 [4], or one that picks all code blocks from answers that are highly ranked or *accepted* [80, 199].[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).

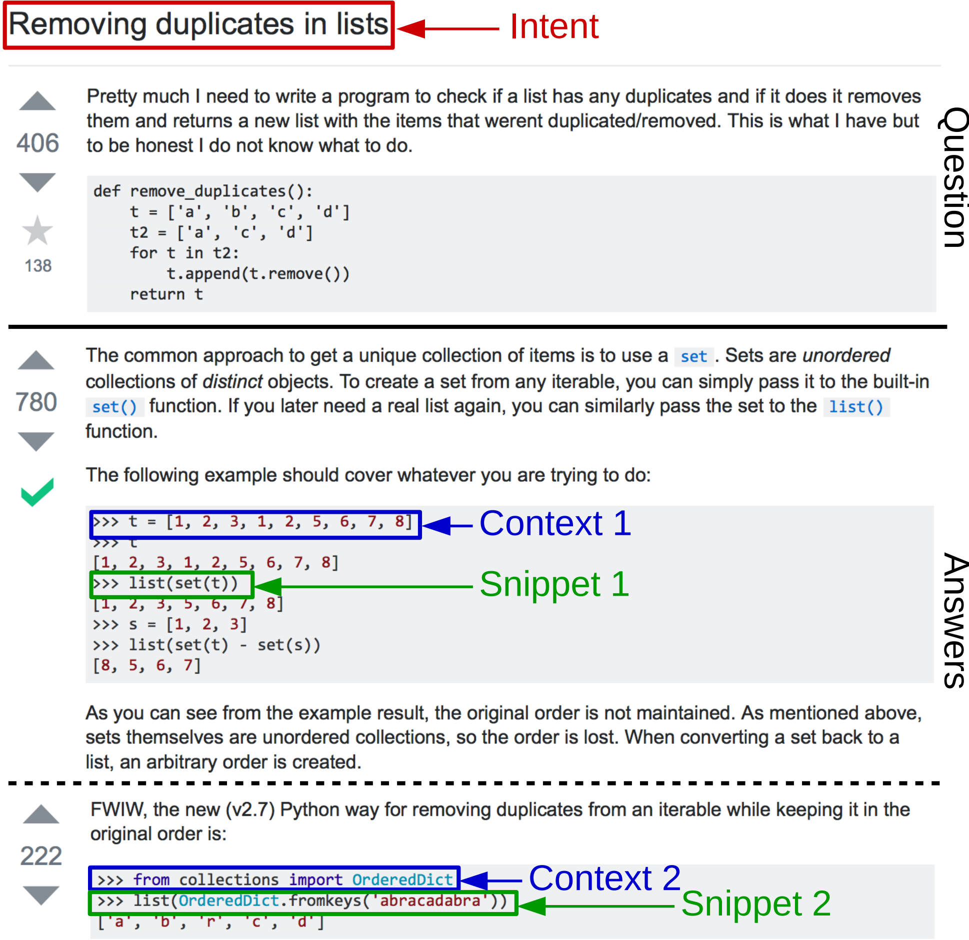


Figure 9.1: Excerpt from a SO post showing two answers, and the corresponding NL intent and code pairs.

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 9.1.

like one thrust of prior work [80, 199]); 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 [4], 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 9.2 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 [10], 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 [4, 80, 199], 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`.

9.2 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 9.1, 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 §9.3.2. 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 9.1. 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.

9.3 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.

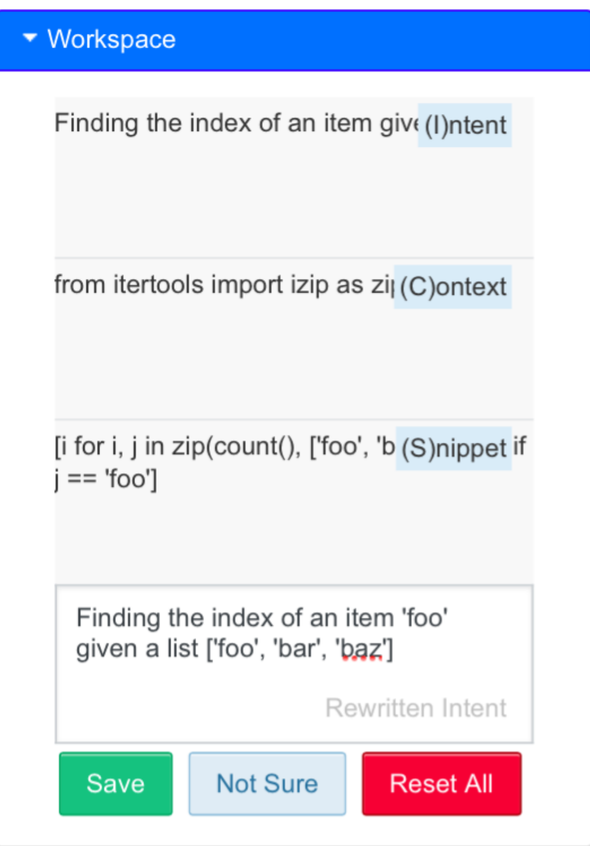
For each target programming language, we first obtained all questions from the official SO data dump[3] dated March 2017 by filtering questions tagged with that language. We then generated the set of questions to annotate by: (1) including all top-100 questions ranked by view count; and (2) sampling 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

9.3.1 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 9.1: 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 "`".

9.3.2 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 §9.2), 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 9.1 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 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

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

uses special HTML tags to highlight code blocks, recall the example in Figure 9.1) 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.

9.4 Mining Method

In this section, we describe our mining method (see Figure 9.2 for an overview). As mentioned in §9.2, 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 9.1, 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.*, $P(y = 1|I, S)$, based on a number of *features* (Sections §9.4.1 and §9.4.2). As is usual in supervised learning, our system first requires an offline *training* phase that learns the parameters (*i.e.*,

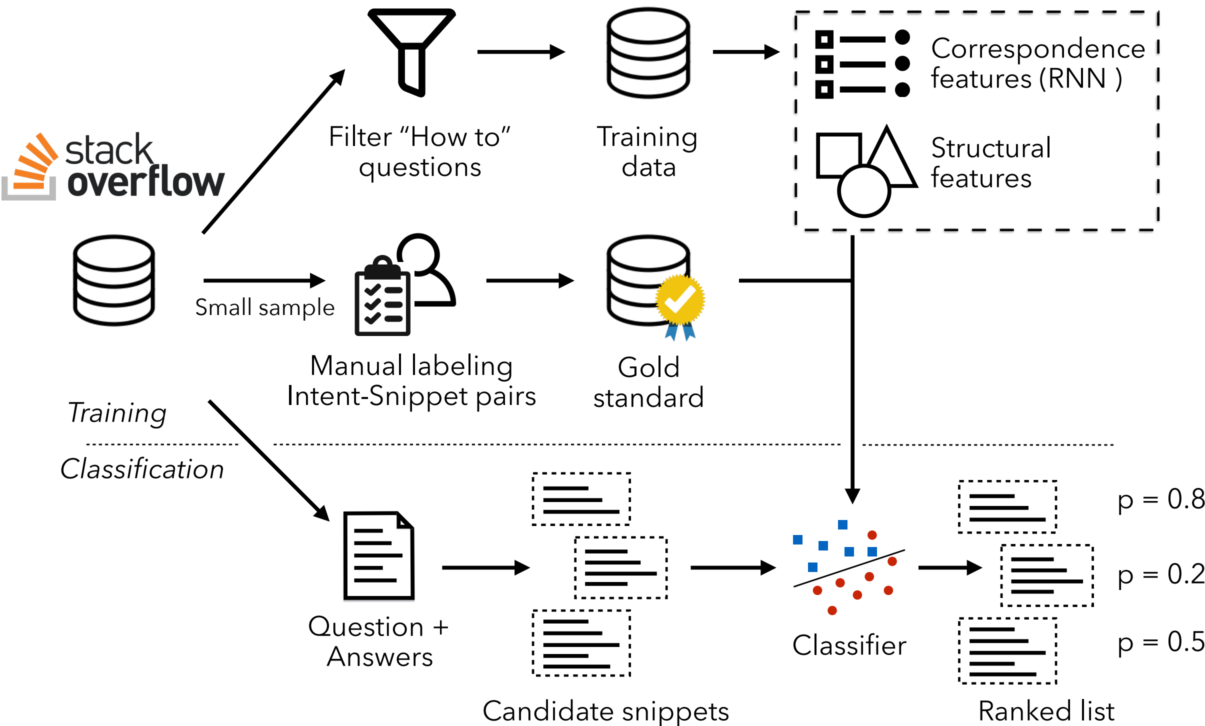


Figure 9.2: Overview of our approach.

feature weights) of the classifier, for which we use the annotated data described above (§9.3). 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 (§9.4.1) and machine learned *correspondence* features that predict whether intents and code snippets correspond to each-other semantically (§9.4.2). 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.

9.4.1 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 9.1, 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 [80, 199]. We use this feature and two other combination features: specifically ¬StartWithAssign + EndOfBlock and ¬StartWithAssign + NumLines1.

9.4.2 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 [10] 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 §9.4.1.

- **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.

9.5 Evaluation

In this section we evaluate our proposed mining approach. We first describe the experimental setting in §9.5.1 before addressing the following research questions:

1. How does our mining method compare with existing approaches across different programming languages? (§9.5.2)
2. How do the structural and correspondence features impact the system's performance? (§9.5.2)
3. Given that annotation of data for each language is laborious, is it possible to use a classifier learned on one programming language to perform mining on other languages? (§9.5.3)

[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).

Table 9.2: 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

4. What are the qualitative features of the NL-code pairs that our method succeeds or fails at extracting? (§9.5.4)

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.

9.5.1 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 [208].

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 [80] 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 [80]. We then extract intent/snippet pairs from all these questions as described in §9.4.2, 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 9.2.[6]

We implement our neural correspondence model using the DyNet neural network toolkit [141]. The dimensionality of word embedding and RNN hidden states is 256 and 512. We use dropout [176], a standard method to prevent overfitting, on the input of the last softmax layer over target words ($p = 0.5$), and recurrent dropout [57] on RNNs ($p = 0.2$). We train the network using the widely used optimization method Adam [88]. 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 struc-

[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.

tural and correspondence features described above, the latter computed by the previous neural network, we use our annotated intent/snippet data (§9.3.2)[7] during **5-fold cross validation**. 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 [80, 199]; 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.

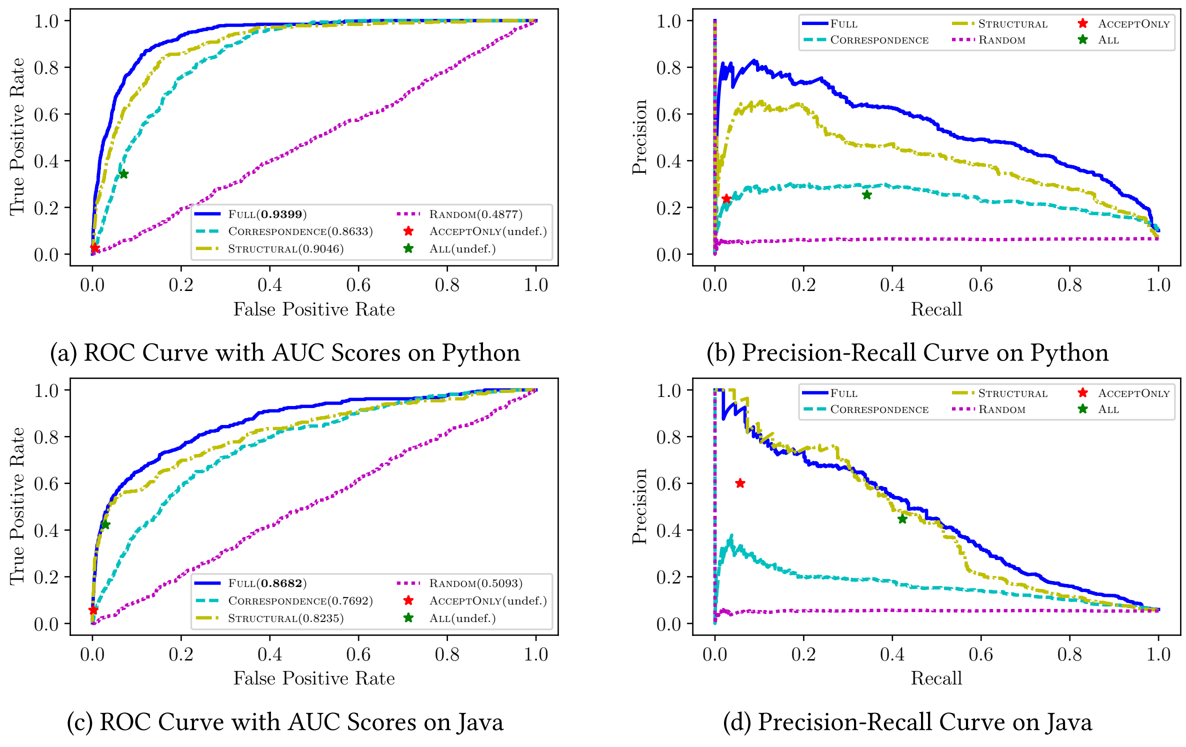
Similarly to our model, we enforce the constraint that all mined code snippets given by the baseline approaches should be parseable.

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.

9.5.2 Results and Discussion

Our main results are depicted in Figure 9.3. 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 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

[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 [80].



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

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 9.3b and 9.3d): 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; 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 re-

[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.

sults 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 §§9.4.2), 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. 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 9.1),

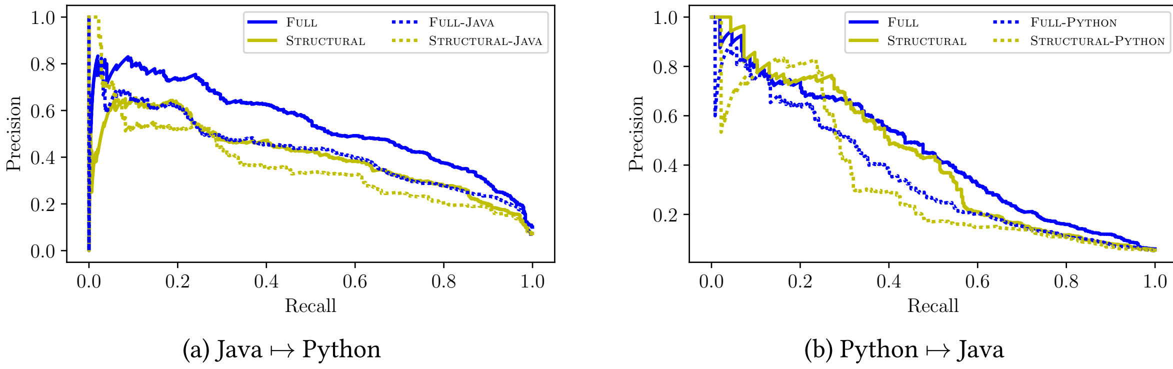


Figure 9.4: Precision-Recall Curves for Transfer Learning on Java ↦ Python (a) and Python ↦ Java (b)

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 9.3c, our full model achieves higher true positive rates compared with Structural, registering higher AUC scores.

9.5.3 Must We Annotate Each Language?

As discussed in §§9.3, 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 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 9.4a and 9.4b 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

[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.

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.

9.5.4 Successful and Failed Examples

As illustration, we showcase successful and failed examples of our proposed approach, for Python in Table 9.3 and for Java in Table 9.4. 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 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

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

Table 9.3: 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 9.4: 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 ✗ (annotation error)
```
public static int randInt(int min, int
    max) {
    Random rand = new Random();
    int randomNum = rand.nextInt((max -
        min) + 1) + min;
    return randomNum; }
```

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 §9.3, guaranteeing coverage in the annotation process is non-trivial, and we leave this as a challenge for future work.

9.6 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.*, [80, 199, 210, 234]), while there has been research on mining from other data sources such as API documentation [14, 29, 138], code comments [200], specialized sites [156], parameter/method/class names [3, 175], and developer mailing lists [148]. 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 [29, 199, 234]). 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 [143, 197]. 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, [210] 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 [115, 146], or vice-versa [195]. This allows for the generation of high-quality data, but is time consuming and does not scale beyond limited domains.

9.7 Threats to Validity

Besides threats related to the manual labeling (§9.3.2), 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 §9.3 and §9.5.4).

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.

9.8 Summary

In this chapter, we describe a novel method for extracting aligned code/natural language pairs from the Q&A website Stack Overflow. The method is based on learning from a small number of annotated examples, using highly informative features that capture structural aspects of the code snippet and the correspondence between it and the original natural language query. Experiments on Python and Java demonstrate that this approach allows for more accurate and more exhaustive extraction of NL-code pairs than prior work. We foresee the main impact of this chapter lying in the resources it would provide when applied to the full Stack Overflow data: the NL-code pairs extracted would likely be of higher quality and larger scale. Given that high-quality parallel NL-code data sets are currently a significant bottleneck in the development of new data-driven software engineering tools, we hope that such a resource will move the field forward. In addition, while our method is relatively effective compared to previous work, there is still significant work to be done on improving mining algorithms to deal with current failure cases, such as those described in §9.5.4. Our annotated data set and evaluation tools, publicly available, may provide an impetus towards further research in this area.

In addition, we note several intriguing future research directions emerging from our study. scientific questions in this general realm that came to light in the process of this study (and were mentioned briefly earlier in the paper). First, there is still a disconnect between the SO question title and a strict description of the functionality of the code, such as the modified intent described in §9.3.1. How to quantify and close this gap computationally is an interesting problem that could be a direction of future research. Second, given that multiple code snippets can be extracted for any particular intent, these snippets will each have their own trade-offs and compromises. Understanding these compromises, either by processing the textual descriptions on SO or through other means, is another challenging research problem; solving it holds great potential to software engineering applications such as code suggestion, automatic refactoring, and others. Third, Stack Overflow is not the only promising "Big Code" archive with potential for large-scale natural instances of aligned NL-code pairs. Open-source software forges host billions of lines of source code, often accompanied by natural language comments, documentation, or even discussion, for example around pull requests on GitHub. The relative generality of our approach and the relatively little labeled training data needed to make it work hold great promise for future expansion into these other data sources.

Chapter 10

Conclusions and Future Directions

This thesis put forward a series of approaches for neural semantic parsing. Our methods enables effective modeling of structures in domain knowledge schemas (Chapters 5 to 7) and meaning representations (Chapters 3 and 4), while presenting strategies to collect data (Chapter 9) and train semantic parsers in a data efficient manner (Chapters 7 and 8). Specifically, we propose 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 capable of scaling to complex open-domain programs (Chapters 3 and 4). Next, to capture structured domain knowledge like database schemas and API specifications, we explore pre-training over massive corpora of Web tables as a universal recipe for learning representations of tabular schemas and utterances (Chapter 5), followed by more explicit modeling of utterance-schema alignments to improve data efficiency and generalization ability (Chapter 6). To mitigate the paucity of limited training data, we present a data-efficient semi-supervised learning method that outperforms purely supervised systems with additional unlabeled utterances (Chapter 8), and an unsupervised model that learns alignments between utterances and database schemas without labeled data (Chapter 7). Finally, to speed-up data annotation, we also propose a machine-in-the-loop data acquisition pipeline for tasks with complex meaning representations (Chapter 9).

In this chapter, we first briefly summarize the contributions made in this thesis, while envisioning future avenues in this line of research.

10.1 Summary of Contributions

In this section we present a systematic summary of key contributions made in this thesis:

Structured Program Generation Models Natural language is highly flexible, while meaning representations derived from the NL utterances are conformed by their underlying syntactic structures. Given the gaps between the free-form natural language data and the structured symbolic semantic formalisms, the field is ripe with semantic parsing algorithms that capture structures in meaning representations [113, 236, 237]. On the other hand, the burgeoning development of neural sequence-to-sequence models has alleviated the need of explicit modeling of syntactic structures in MRs, as programs are treated as tokenized sequences, similar to natural language [50, 84, 117]. However, it is noted that such a simple solution is sub-optimal — a neural model without prior syntactic knowledge could yield grammatically incorrect outputs, and could also be data hungry, requiring more data to learn the underlying structures of MRs [84]. Therefore, in Part I of this thesis, we develop syntactic driven neural semantic parsing models that leverage the grammar of meaning representations as prior knowledge. Instead of directly generating program tokens as in vanilla neural sequence transduction models, this approach models the rich structures in meaning representations using abstract syntax trees, and employs a structured decoder to predict ASTs following the production rules in the grammar, which constrains the generation space to be MRs that are syntactically valid, hence effectively reduces the output space and improves data efficiency. In Chapter 3, we demonstrate this structured decoding model scales well to generating code in open-domain programming languages with complex grammars. Later in Chapter 4, we extend this model and develop a general-purpose syntax-driven semantic parsing framework with a configurable interface to encode grammars of MRs in various down-stream applications. From predicting λ-calculus to Prolog formulas, from generating SQL queries to Python code, we show this framework is highly extendable, achieving competitive results on benchmarks featuring a variety of grammar formalisms. This work also inspires a large body of literature along this line, and forms the foundation of many influential neural semantic parsers later developed.

Methods to Understand Domain Knowledge Schema Domain knowledge (*e.g.*, database schemas) is essential to interpret the semantics of utterances and infer their representations. Such task-specific knowledge schema also exhibit rich structures. For example, a database table holds structured factual information about the domain, which need to be first processed by a semantic parser in order to understand utterances and predict SQL queries related to the table. In Part II, we explore schema understanding models that could better encode such structured knowledge. To this end, in Chapter 5 we present a pre-trained encoder, TaBert, which jointly learns representations of NL utterances and (semi-)structured database tables. TaBert is the

first pre-trained language model for natural language understanding tasks with structured data, which is trained on a massive collection of 35 million Web tables and associated textual contexts. Through pre-training on large corpora, this model implicitly learns general-purpose, domain-agnostic representations of structured tabular data from its linearized form, while implicitly capturing the alignments between relevant NL phrases (*e.g.*, *flights from Pittsburgh*) and their grounded table cells and columns (*e.g.*, `Departure_City`). TaBert can be used as a drop-in replacement of any neural semantic parser's original encoder to produce representations of utterances and tables, improving their performance on down-stream tasks.

An issue with TaBert is that it only implicitly captures the alignments between NL phrases and related schema elements via free-form self-attention, which might not be data efficient. To more explicitly model NL-schema alignments, we propose a *supervised* attention mechanism (Chapter 6) that encourages the semantic parser to predict semantically coherent segments of MRs (*e.g.*, `FindManager(Jean)`) using the aligned localized spans in NL utterances (*e.g.*, *Schedule meeting with Jean's manager*). Here, we generalize the notion of domain knowledge schema from structured database tables to such alignment information between predefined functions in the domain and their natural language descriptions. Such schematic information can be viewed as a variant of API specification defining domain functions and their exemplar NL realizations in utterances, which is widely adopted by dialogue systems. We show supervised attention improves data efficiency, generalizing well to utterances with novel compositional contexts (*e.g.*, *Add meeting with Jean's manager and Peter*) with only a handful of labeled examples.

To further reduce the amount of labeled data required to understand domain schemas, we propose an unsupervised learning approach in Chapter 7, which trains a semantic parser only using utterances and MRs automatically synthesized from a specification of the domain schema defining canonical NL phrases and their MR implementations. We also present measures to improve the coverage of synthetic examples in terms of their language styles and logical patterns to better resemble the real-world user-issued utterances. Our data synthesis approach uses zero annotated data, and outperforms other efficient annotation approaches like OverNight (§2.2.1) and its variants [73].

Cost-effective Data Acquisition Paradigms Neural semantic parsers are data hungry, requiring relatively large amount of parallel data for learning. This thesis also explores cost-effective approaches to efficiently collect annotated data to train semantic parsers. Specifically, in Chapter 9, we present a semi-automatic mining pipeline to collect examples of utterances and Python implementations from community question answering websites (StackOverflow)

for code generation tasks. We propose a machine-in-the-loop method where human annotators modify proposal examples generated by a mining model, whose quality is iteratively improved using newly annotated examples from domain experts. Additionally, our zero-shot data synthesis model in Chapter 7 further lifts the tedious work of having annotators labeling individual examples — they just need to define a grammar specifying alignments of canonical NL phrases to MR implementations (*e.g.*, *Schedule a meeting*↔`CreateEvent(?)`), from which compositional examples are automatically synthesized and paraphrased. We demonstrate this cost-effective data acquisition method outperforms other more laborious data annotation frameworks [73, 195]

Data Efficient Learning Approaches Keeping the cost of acquiring parallel data in mind, we also made contributions in proposing more data efficient learning methods that require fewer amount of annotated data. In Chapter 8, we propose a semi-supervised learning framework StructVAE, which models the ASTs of MRs as tree-structured latent variables, and uses extra unlabeled utterances to train semantic parsers. Compared to purely supervised learning, StructVAE achieves similar performance using less labeled data. Meanwhile, our zero-shot data synthesis approach in Chapter 7 can also be viewed as a data efficient learning model, as it bootstraps a semantic parser through iterative rounds of synthetic data paraphrasing and self-training of parsers, without using *any* labeled data. Finally, our span-based supervised attention method in Chapter 6 also improves data efficiency, as it effectively captures the alignments between NL phrases and their logical representations in domain schemas, generalizing to compositionally novel samples with only a handful of training examples.

10.2 Open Problems and Future Directions

Although the proposed methods in this thesis have helped the field a bit, we are still faced by several open research questions that this thesis barely unveils, or has not touched so far. We hope the following summary of future avenues would help researchers get a grasp of challenging (yet promising) directions in the field.

Pre-training Methods for Semantic Parsing Nowadays, sequence-processing neural networks, especially those generative language models pre-trained on massive textual corpora, have become the new *de-facto* approach for a variety of natural language understanding tasks [26, 158, 159], and have also registered promising results on semantic parsing [Chapter 5; 75, 232, 233]. Still, there are several open challenges in this line:

- **Pre-training Data Collection** Pre-training for semantic parsing has been limited by the lack of high-quality, large-scale corpora of NL utterances and structured knowledge schemas (tables). Existing work uses tabular data with noisy contexts from the Web [Chapter 5; 75] or small-scale parallel data from other tabular understanding tasks [48]. Notably, recent works have demonstrated the potential of pre-training using large-scale Web tables paired with synthetically generated NL utterances [232, 233], where utterances are synthesized using predefined templates with sampled information from Web tables. It is promising to scale this approach to (1) generate large-scale pre-training data of utterances and complex DB schemas with multiple tables, and (2) leverage our data synthesis approach in Chapter 7 to generate more realistic pre-training corpora using carefully designed domain-specific grammars and paraphrasing.
- **Pre-training Encoders *v.s.* End-to-End Models** Most existing work on pre-training for semantic parsing has been only focused on improving the encoder of utterances and tables [75, 232, 233]. Recent advance of pre-training for NLP has considered jointly learning of encoder-decoder architectures [26, 108, 159]. Therefore, a promising direction is to developing semantic parsers with pre-trained encoder and decoder modules. However, as discussed in Part I and in Chapter 5, semantic parsing is a highly domain-specific task, and the architecture of decoding modules is heavily dependent on the application. It still remains an open research question to jointly pre-train the encoding and decoding networks for neural semantic parsing.
- **Leverage Knowledge in Pre-trained Generative LMs** Large-scale generative pre-trained LMs on text, like BART [108] and GPT-3 [26], have revolutionized NLP, as many NLP tasks can be formulated as text-to-text generation problems, and could be attacked by those generative LMs equipped with open-domain knowledge of natural language text. An important research question is to leverage the knowledge captured by the pre-trained generative LMs for semantic parsing. However, unlike many other NLP tasks where the prediction targets are NL sentences (*i.e.*, text generation), semantic parsing aims to generate task-specific MRs with rich structures, and also requires understanding of additional domain knowledge (Part II), which is not captured by those pre-trained LMs. Recent efforts attempt to "naturalize" MRs as canonical NL sentences (*c.f.*, Chapter 7) and cast semantic parsing as a text normalization problem, where NL utterances are normalized to their canonical versions, which can be trivially converted to MRs [171]. However, this approach still requires supervision with labeled data to teach LMs about domain knowl-

edge (*i.e.*, how to generate canonical utterances). Applying our zero-shot data synthesis model in Chapter 7 to this task would be an interesting future work.

Is Modeling Syntactic Structures still Necessary? In Part I, we have showed the importance of modeling syntactic information of MRs in semantic parsing. However, recent advance of pre-trained generative LMs on text and source code has demonstrated impressive abilities in generating complex, syntactically well-formed programs *without* any prior syntactic knowledge [Open AI Codex; 30]. Given this result, we have to ask the question: is it still necessary to model syntactic structures? First, modeling program syntax could still be useful in low data regime where pre-training is not possible. Additionally, we remark that the techniques we present in Part I could be generalized as a way to *control* neural auto-regressive decoding, where the prediction of a hypothesis (*e.g.*, ASTs) is controlled by a predefined grammar. Therefore, such methods provide a systematic approach for controllable generation of any targets other than ASTs that follow predefined syntax (*e.g.*, programs). In this sense, our grammar-constrained decoding approach could be used to control the generation process of existing pre-trained LMs that directly predict free-form source code (*e.g.*, constrain the model to generate code only using a specified library and its APIs), as long as the output space could be captured by a grammar of certain form. In the following paragraph, we consider generalizing this idea for more neural structured prediction tasks.

Generalized Constraint-based Neural Structured Prediction As discussed in the above paragraph, our syntax-driven program generation model could be potentially extended as a general-purpose approach for constraint-based neural structured prediction. Intuitively, most structured prediction tasks require that the outputs should follow certain pre-defined structure. For example, in dependency parsing, labels of an edge are dependent on the part-of-speech of its connected tokens. For dialogue systems, the dialogue state of a conversation could follow a pre-defined schema (*e.g.*, a state that keeps track of a user's flight booking request might maintain a key-value store with fixed slots for the requested date and destination). Our model presented in Part I could potentially be generalized to handle generation of structure in those applications if their domain-specific constraints can be specified in context-free grammars.

Interactive Semantic Parsing with Complex MRs Natural interactions between users and computational systems involve multiple turns of dialogues. Therefore the literature is ripe with methods for building such interactive dialogue agents. An important future work is to scale our syntax-based semantic parsing models (Part I) to such interactive scenarios. Existing work in interactive semantic parsing has focused on simpler domain-specific languages like

SQL [230, 231] or other task-oriented ones [166]. Generalizing interactive semantic parsing to applications with complex MRs, like Python code generation, would be a promising direction with significant impact. Such models need to understand the context of previous turns of utterances and generated MRs, predict new programs modify existing ones as response, and be able to leverage feedback from users to revise their predictions.

Unified Frameworks of Data Acquisition and Learning The paucity of annotated training data hurdles deploying semantic parsers to emerging new domains. This thesis explores a variety of efficient approaches for acquiring training data (Chapters 7 and 9) and learning semantic parsers (Chapter 8). An interesting future direction is to jointly model the process of data annotation and model training, using unified models to capture the life cycle of semantic parsing [34]. This is an important step towards developing general-purpose semantic parsing services that could be easily used by end users.

Appendix A

Appendix for Chapter 7

A.1 Synchronous Grammar

Our synchronous grammar is adapted from Herzig and Berant [73] and Wang et al. [195], which specifies alignments between NL expressions and logical form constituents in λ-calculus s-expressions.[1] The grammar consists of a set of domain-general production rules, plus domain-specific rules specifying lexicons and idiomatic productions. Specifically, domain-general productions define (1) generic logical operations like `count` and `superlative` (*e.g.*, r_3, Fig. 7.1), and (2) compositional rules to construct utterances following English syntax (*e.g.*, r_1, Fig. 7.1). Domain-specific rules, on the other hand, are typically used to define task-dependent lexicons like types (*e.g.*, `author`), entities (*e.g.*, `allen_turing`), and relations (*e.g.*, `citations`) in the database. This work also introduces idiomatic productions to specific common NL expression catered to a domain, as detailed later.

Tab. A.1 lists example domain-general productions in our SCFG. Fig. A.1 shows the derivation that applies those productions to generate an example utterance and program. Each production has a syntactic body, specifying how lower-level syntactic constructs are composed to form more compositional utterances, as well as a *semantic function*, which defines how programs of child nodes are composed to generate a new program. For instance, the production r_3 in Tab. A.1 generates a noun phrase from a unary noun phrase `UnaryNP` (*e.g.*, *paper*) and a complementary phrase `CP` (*e.g.*, *in deep learning*) by concatenating the child nodes `UnaryNP` and `CP` (*e.g.*, *paper in deep learning*). On the program side, the programs of two child nodes on Fig. A.1 are:

[1] We use the implementation in `Sempre`, `https://github.com/percyliang/sempre`

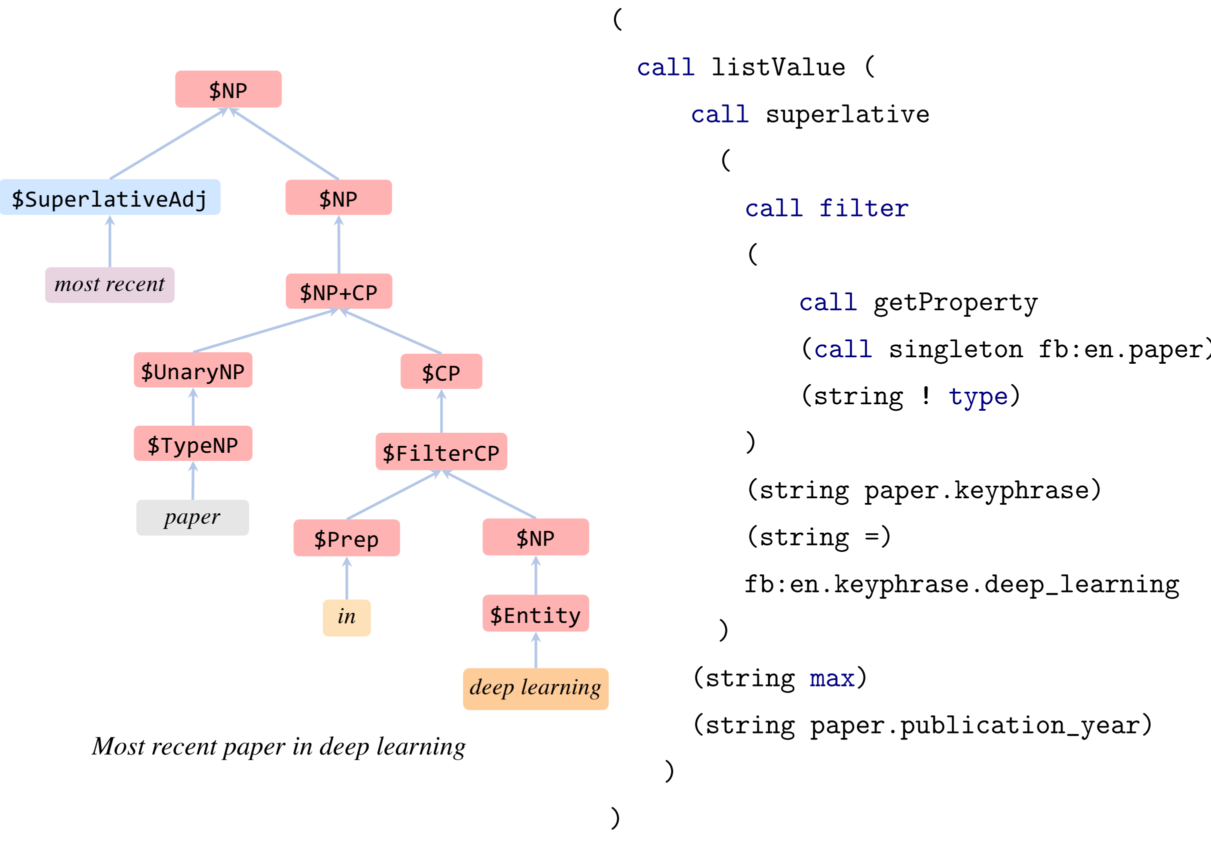


Figure A.1: (a) The derivation tree (production rule applications) to generate the example utterance and its program. (b) The program defined in s-expression.

```
# Get all entities whose type is paper
$UnaryNP: call getProperty (call singleton fb:en.paper) (string !type)
# A lambda function that returns entities in x whose relation paper.keyphrase
# is deep_learning
$CP: lambda x (call
          filter (x)
          (string paper.keyphrase)
          (string =)
          (fb:en.keyphrase.deep_learning)
     )
```

Id	Productions (Syntactic Body and Semantic Function)	Description
r_1	NP↦SuperlativeAdj NP	*e.g. most recent* ?
	`lambda rel, sub (` `call superlative (var sub) (string max) (var rel))`	lambda function to get the subject sub with the largest relation `rel`
r_2	NP↦NP+CP	A noun phrase head `NP` and a complementary phrase body `CP` (*e.g. paper in deep learning*)
	`IdentityFn`	An identity function returning child program
r_3	NP+CP↦UnaryNP CP	*e.g. paper in deep learning*
	`Lambda Beta Reduction: f(var x)`	Perform beta reduction, applying the function from `CP` (*e.g. in deep learning*) to the value of `UnaryNP` (*e.g. paper*)
r_4	UnaryNP↦TypeNP CP	Entity types, *e.g.*, *paper*
	`IdentityFn`	
r_5	CP↦FilterCP	–
	`IdentityFn`	
r_6	FilterCP↦Prep NP	*e.g. in deep learning*
	`lambda rel, obj, sub (` `call filter (var sub) (var rel) (string =) (var obj))`	Create a lambda function, which filters entities in a list `sub` such that its relation `rel` (*e.g. topic*) equals `obj` (*e.g. deep learning*)
r_5	NP↦Entity	Entity noun phrases *e.g. deep learning*
	`IdentityFn`	

Table A.1: Example domain-general productions rules in the SCFG

Where the program of UnaryNP is an entity set of papers, and the program of NP is a lambda function with a variable x, which filters the entity set. The semantic function of r_3 specifies how these two programs should be composed to form the program of their parent node NP+CP, which performs β reduction, assigning the entity set returned by UnaryNP to the variable x:

```
# Get all papers whose keyphrase is deep learning
$NP+CP: (call
           filter (
             call getProperty (call singleton fb:en.paper) (string !type)
           )
           (string paper.keyphrase)
           (string =)
           (fb:en.keyphrase.deep_learning)
        )
```

A.1.1 Idiomatic Productions

Multi-hop Relations We create idiomatic productions for non-compositional NL phrases of multi-hop relations (*e.g.*, *Author that writes paper in ACL*). We augment the database with entries for those multi-hop relations (*e.g.*, ⟨X, author.publish_in, acl⟩), and then create productions in the grammar aligning those relations with their NL phrases (*e.g.*, r_1 in Tab. A.2).

Comparatives and Superlatives We also create productions for idiomatic comparatives and superlative expressions. Those productions specify the NL expressions for the comparative/superlative form of some relations. For example, for the relation paper.publication_year with objects of date time, its superlative form would be *most recent* (r_2 in Tab. A.2) and *first* (r_3), while its comparative form could be prepositional phrases like *published before* (r_4) and *published after.* Those productions define the lexicons for comparative/superlative expressions, and could be used by the domain-general rules like r_1 in Tab. A.1 to compose utterances (*e.g.*, Fig. A.1).

Besides superlative expressions for relations whose objects are measurable, we also create idiomatic expressions for relations with countable subjects or objects. As an example, the utterance "*The most popular topic for papers in ACL*" involves grouping ACL papers by topic and return the most frequent one. Such computation is captured by the CountSuperlative operation in our SCFG based on Wang et al. [195], and we create productions aligning those relations with the idiomatic noun phrases describing their superlative form (*e.g.*, r_5 in Tab. A.2).

Perhaps the most interesting form of superlative relations are those involving reasoning

Id	Production Body (Child Nodes and Semantic Function)	Description
r_1	`RelVP`↦*publish in* `ConstantFn (string author.publish_in)`	Verb phrase for multi-hop relation *author that writes paper in ACL*
r_2	`SuperlativeAdj`↦*most recent* `ConstantFn (string paper.publication_year)`	Superlative adjectives to describe publication dates
r_3	`SuperlativeMinAdj`↦*first* `ConstantFn (string paper.publication_year)`	Superlative adjectives to describe the earliest publication dates
r_4	`SuperlativeAdj`↦*published before* `ConstantFn (string paper.publication_year)`	Comparative prepositions to describe publication dates
r_5	`CountSuperlativeNP`↦*the most popular topic for* `ConstantFn (string keyphrase.paper)`	Superlative form to refer to the most frequent keyphrase for papers

r_6	`MacroVP`↦*publish mostly in*
	`lambda author, venue (`
	`call countSuperlative`
	`(var venue)`
	`(string max)`
	`(string venue.paper)`
	`(call getProperty (var author) (string author.paper))`
	`)`
		Superlative form of verb relational phrases with complex computation. `countSuperlative` returns the entity `x` in `venue` for which the papers in `x` (via relation `venue.paper`) has the largest intersection with papers by `author` (via realtion `author.paper`)

Table A.2: Example idiomatic productions used in SCHOLAR

with additional entities. For instance, the relation in *"venue that X publish mostly in"* between the entity `author` and `venue` implicitly involves counting the papers that the author X publishes. For those relations, we create "macro" productions (*e.g.*, r_6 in Tab. A.2), which defines the lambda function that computes the answer (*e.g.*, return the publication venue where X publishes the most number of papers) given the arguments (*e.g.*, an author X).

Bibliography

[1] Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. Learning to generalize from sparse and underspecified rewards. In *ICML*, 2019. (pages 19 and 69)

[2] Ekin Akyürek, Afra Feyza Akyurek, and Jacob Andreas. Learning to recombine and resample data for compositional generalization. In *Proceedings of ICLR*, 2021. (page 96)

[3] 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. (page 138)

[4] Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of ICML*, 2015. (pages 42, 120, and 121)

[5] David Alvarez-Melis and Tommi S. Jaakkola. Tree-structured decoding with doubly recurrent neural networks. In *Proceedings of ICLR*, 2017. (page 40)

[6] Ion Androutsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *ArXiv*, cmp-lg/9503016, 1995. (pages 11 and 12)

[7] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transaction of ACL*, 2013. (pages 1 and 23)

[8] Yoav Artzi, Nicholas FitzGerald, and Luke Zettlemoyer. Semantic parsing with combinatory categorial grammars, 2013. (pages 1 and 8)

[9] 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`. (page 1)

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*, 2015. (pages 1, 9, 32, 33, 121, and 129)

[11] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Trans. Software Eng.*, 11(11), 1985. (page 14)

[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. (pages 1 and 117)

[13] Jun-Wei Bao, Nan Duan, Ming Zhou, and Tiejun Zhao. Knowledge-based question answering as machine translation. In *Proceedings of ACL*, 2014. (page 12)

[14] 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. (page 138)

[15] H. Bast, Björn Buchhold, and Elmar Haussmann. Semantic search on text and knowledge bases. *Found. Trends Inf. Retr.*, 10:119–271, 2016. (page 1)

[16] I Beltagy and Chris Quirk. Improved semantic parsers for if-then statements. In *Proceedings of ACL*, 2016. (pages 14, 36, 39, and 40)

[17] Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *Proceedings of ACL*, 2014. (pages 12 and 60)

[18] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of EMNLP*, 2013. (pages 1, 3, 12, 16, 17, 18, 60, and 104)

[19] Valts Blukis, Dipendra Misra, Ross A. Knepper, and Yoav Artzi. Mapping navigation instructions to continuous control actions with position visitation prediction. In *Proceedings of CoRL*, 2018. (page 1)

[20] Ben Bogin, Matt Gardner, and Jonathan Berant. Global reasoning over database structures for text-to-sql parsing. *ArXiv*, abs/1908.11214, 2019. (pages 70 and 73)

[21] Ben Bogin, Matthew Gardner, and Jonathan Berant. Representing schema structure with graph neural networks for text-to-sql parsing. In *Proceedings of ACL*, 2019. (pages 13, 22, and 60)

[22] 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. (page 107)

[23] 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. (page 14)

[24] 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. (page 14)

[25] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2), 1993. (page 77)

[26] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, J. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. Henighan, R. Child, A. Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020. (pages 144 and 145)

[27] Qingqing Cai and Alexander Yates. Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of ACL*, 2013. (pages 1 and 12)

[28] Bob Carpenter. *Type-logical Semantics*. 1998. ISBN 0-262-03248-1. (pages 7 and 115)

[29] 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. (page 138)

[30] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam

McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021. (pages 1, 22, and 146)

[31] 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. (pages 64, 72, and 74)

[32] Xinyun Chen, Chang Liu, and D. Song. Tree-to-tree neural networks for program translation. In *NeurIPS*, 2018. (pages 15 and 54)

[33] Xinyun Chen, Chen Liang, Adams Wei Yu, D. Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. *ArXiv*, abs/2008.06662, 2020. (page 76)

[34] Jianpeng Cheng. Lifecycle of neural semantic parsing. 2019. (page 147)

[35] Jianpeng Cheng and Mirella Lapata. Weakly-supervised neural semantic parsing with a generative ranker. In *CoNLL*, 2018. (page 19)

[36] Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. Learning structured natural language representations for semantic parsing. In *Proceedings of ACL*, 2017. (page 115)

[37] Jianpeng Cheng, Siva Reddy, V. Saraswat, and Mirella Lapata. Learning an executable neural semantic parser. *Computational Linguistics*, 45:59–94, 2019. (page 99)

[38] Jianpeng Cheng, Devang Agrawal, Héctor Martínez Alonso, Shruti Bhargava, J. Driesen, F. Flego, D. Kaplan, Dimitri Kartsaklis, Lin Li, Dhivya Piraviperumal, J. Williams, Hong Yu, Diarmuid Ó Séaghdha, and Anders Johannsen. Conversational semantic parsing for dialog state tracking. In *EMNLP*, 2020. (page 1)

[39] 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. (page 117)

[40] David Chiang. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of ACL*, 2005. (page 78)

[41] S. Clark and J. Curran. Log-linear models for wide-coverage ccg parsing. In *EMNLP*, 2003. (pages 8 and 9)

[42] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of CoNLL*, 2010. (pages 17 and 104)

[43] Marco Damonte, Rahul Goel, and Tagyoung Chung. Practical semantic parsing for spoken language understanding. In *NAACL-HLT*, 2019. (pages 1 and 23)

[44] Dipanjan Das and Noah A. Smith. Semi-supervised frame-semantic parsing for unknown predicates. In *Proceedings of HLT*, 2011. (page 117)

[45] 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. (pages 69 and 91)

[46] Michael Brown William Fisher Kate Hunicke-Smith David Pallett Christine Pao Alexander Rudnicky Deborah A. Dahl, Madeleine Bates and Elizabeth Shriber. Expanding the scope of the ATIS task: The ATIS-3 corpus. *Proceedings of the workshop on Human Language Technology*, pages 43–48, 1994. URL http://dl.acm.org/citation.cfm?id=1075823. (pages 16 and 22)

[47] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. Turl: Table understanding through representation learning. *Proc. VLDB Endow.*, 14:307–319, 2020. (pages 13 and 74)

[48] Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. Structure-grounded pretraining for text-to-sql. In *NAACL*, 2021. (pages 13, 74, and 145)

[49] 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. (pages 59, 65, and 92)

[50] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of ACL*, 2016. (pages 1, 9, 34, 37, 40, 51, 53, 76, 77, 103, 109, 111, and 142)

[51] Li Dong and Mirella Lapata. coarse-to-fine decoding for neural semantic parsing. In *Proceedings of ACL*, 2018. (pages 1, 14, 45, 73, 77, 78, and 80)

[52] Chris Dyer, Victor Chahuneau, and Noah A. Smith. A simple, fast, and effective reparameterization of IBM model 2. In *Proceedings of NAACL*, 2013. (page 77)

[53] Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. Nl-edit: Correcting semantic parse errors through natural language interaction. In *NAACL*, 2021. (page 23)

[54] Anthony Fader, Luke Zettlemoyer, and Oren Etzioni. Open question answering over curated and extracted knowledge bases. *Proceedings of the 20th ACM SIGKDD international*

conference on Knowledge discovery and data mining, 2014. (pages 12 and 13)

[55] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. In *Proceedings of ACL*, 2018. (pages 76, 80, and 96)

[56] Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Scharli. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *ArXiv*, abs/2007.08970, 2020. (page 76)

[57] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of NIPS*, 2016. (pages 37 and 131)

[58] Michael R. Glass, Mustafa Canim, A. Gliozzo, Saneem A. Chemmengath, Rishav Chakravarti, Avirup Sil, Feifei Pan, Samarth Bharadwaj, and Nicolas Rodolfo Fauceglia. Capturing row and column semantics in transformer based question answering over tables. In *NAACL*, 2021. (page 74)

[59] Yoav Goldberg. Assessing bert's syntactic abilities. *ArXiv*, abs/1901.05287, 2019. (page 59)

[60] Omer Goldman, Veronica Latcinnik, Udi Naveh, A. Globerson, and Jonathan Berant. Weakly-supervised semantic parsing with abstract examples. *ArXiv*, abs/1711.05240, 2018. (pages 19 and 20)

[61] Dan Goldwasser, Roi Reichart, J. Clarke, and D. Roth. Confidence driven unsupervised semantic parsing. In *Proceedings of ACL*, 2011. (page 21)

[62] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of ACL*, 2016. (pages 34 and 106)

[63] Daya Guo, Yibo Sun, Duyu Tang, Nan Duan, Jian Yin, Hong Chi, James Cao, Peng Chen, and M. Zhou. Question generation from sql queries improves neural semantic parsing. In *Proceedings of EMNLP*, 2018. (page 99)

[64] 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. (pages 1, 12, 13, 22, 60, 67, 70, and 73)

[65] Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of EMNLP*, 2018. (pages 1 and 23)

[66] 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. (pages 18, 20, 108, and 109)

[67] Tihomir Gvero. *Search Techniques for Code Generation.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2015. (page 14)

[68] Tatsunori B. Hashimoto, Kelvin Guu, Y. Oren, and Percy Liang. A retrieve-and-edit framework for predicting structured outputs. In *NeurIPS*, 2018. (page 15)

[69] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1111. URL `https://aclanthology.org/D18-1111`. (pages 15 and 54)

[70] Pengcheng He, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. X-sql: reinforce schema representation with context. *ArXiv*, abs/1908.08113, 2019. (page 73)

[71] V. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *ICLR*, 2020. (page 13)

[72] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *ArXiv*, abs/1606.08415, 2016. (page 66)

[73] Jonathan Herzig and Jonathan Berant. Don't paraphrase, detect! rapid and effective data collection for semantic parsing. In *Proceedings of EMNLP*, 2019. (pages 17, 86, 87, 88, 89, 93, 99, 143, 144, and 149)

[74] Jonathan Herzig and Jonathan Berant. Span-based semantic parsing for compositional generalization. In *Proceedings of EMNLP*, 2020. (pages 76 and 77)

[75] Jonathan Herzig, Pawel Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. Tapas: Weakly supervised table parsing via pre-training. In *ACL*, 2020. (pages 13, 22, 74, 144, and 145)

[76] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997. (page 32)

[77] 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. (page 54)

[78] Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. A comprehensive exploration on wikisql with table-aware word contextualization. *ArXiv*, abs/1902.01069, 2019. (pages 13, 60, 64, 72, and 73)

[79] Srini Iyer, Alvin Cheung, and Luke Zettlemoyer. Learning programmatic idioms for scalable semantic parsing. *ArXiv*, abs/1904.09086, 2019. (pages 15 and 54)

[80] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of ACL*, 2016. (pages 120, 121, 129, 131, 132, and 138)

[81] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of ACL*, 2017. (pages 1, 9, 13, 16, 22, 91, and 103)

[82] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Proceedings of EMNLP*, 2018. (pages 15 and 54)

[83] Mohit Iyyer, Wen tau Yih, and Ming-Wei Chang. Search-based neural structured learning for sequential question answering. In *ACL*, 2017. (page 22)

[84] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *Proceedings of ACL*, 2016. (pages 1, 2, 3, 9, 16, 34, 42, 77, 96, 104, and 142)

[85] 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. (pages 60 and 66)

[86] Rohit J. Kate and Raymond J. Mooney. Semi-supervised learning for semantic parsing using support vector machines. In *Proceedings of NAACL-HLT*, 2007. (page 20)

[87] Daniel Keysers, Nathanael Schärli, Nathan Scales, H. Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, D. Tsarkov, Xiao Wang, Marc van Zee, and O. Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *Proceedings of ICLR*, 2020. (pages 76 and 80)

[88] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. (page 131)

[89] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint*

arXiv:1312.6114, 2013. (pages 105 and 106)

[90] Diederik P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Proceedings of NIPS*, 2014. (page 106)

[91] 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. (pages 20, 42, 104, and 117)

[92] 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. (pages 20 and 106)

[93] Kalpesh Krishna, John Wieting, and Mohit Iyyer. Reformulating unsupervised style transfer as paraphrase generation. In *Proceedings of EMNLP*, 2020. (pages 92 and 97)

[94] Jayant Krishnamurthy. Learning to understand natural language with less human effort. 2015. (page 21)

[95] Jayant Krishnamurthy and Tom Mitchell. Weakly supervised training of semantic parsers. In *Proceedings of EMNLP*, 2012. (pages 17 and 21)

[96] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of EMNLP*, 2017. (pages 10, 19, and 60)

[97] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *Proceedings of NAACL*, 2013. (page 14)

[98] Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of EMNLP*, 2010. (page 9)

[99] T. Kwiatkowski, Luke Zettlemoyer, S. Goldwater, and Mark Steedman. Lexical generalization in ccg grammar induction for semantic parsing. In *EMNLP*, 2011. (page 9)

[100] 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. (page 12)

[101] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of ICML*, 2018. (pages 10, 76, and 96)

[102] Brenden M Lake. Compositional generalization through meta sequence-to-sequence learning. In *Proceedings of NeurIPS*, 2019. (page 76)

[103] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40: e253, 2017. doi: 10.1017/S0140525X16001837. (page 10)

[104] Mirella Lapata. Automatic evaluation of information ordering: Kendall's tau. *Computational Linguistics*, 32(4):471–484, 2006. doi: 10.1162/coli.2006.32.4.471. URL `https://www.aclweb.org/anthology/J06-4002`. (page 97)

[105] 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. (page 65)

[106] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. From natural language specifications to program input parsers. In *Proceedings of ACL*, 2013. (page 14)

[107] Wenqiang Lei, Weixin Wang, Zhixin Ma, T. Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. Re-examining the role of schema linking in text-to-sql. In *EMNLP*, 2020. (pages 3, 12, and 22)

[108] M. Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, A. Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of ACL*, 2020. (pages 76, 92, and 145)

[109] Yuanpeng Li, Liang Zhao, JianYu Wang, and Joel Hestness. Compositional generalization for primitive substitutions. In *Proceedings of EMNLP/IJCNLP*, 2019. (page 76)

[110] 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. (pages 18, 19, and 42)

[111] 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. (pages 13, 18, 20, 22, 67, 69, and 73)

[112] Percy Liang. Lambda dependency-based compositional semantics. *ArXiv*, abs/1309.4408, 2013. (page 10)

[113] Percy Liang, Michael I Jordan, and Dan Klein. Learning dependency-based compositional

semantics. In *Proceedings of ACL*, 2011. (pages 11, 104, and 142)

[114] Jindrich Libovický and Jindrich Helcl. Attention strategies for multi-source sequence-to-sequence learning. In *Proceedings of ACL*, 2017. (page 67)

[115] 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. (page 139)

[116] Xi Victoria Lin, R. Socher, and Caiming Xiong. Bridging textual and tabular data for cross-domain text-to-sql semantic parsing. In *FINDINGS*, 2020. (page 14)

[117] 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. (pages 14, 27, 29, 34, 35, 36, 37, 38, 40, 53, 103, and 142)

[118] Greg Little and Robert C. Miller. Keyword programming in java. *Autom. Softw. Eng.*, 16 (1), 2009. (page 14)

[119] Lemao Liu, Masao Utiyama, Andrew Finch, and Eiichiro Sumita. Neural machine translation with supervised attention. In *Proceedings of COLING*, 2016. (pages 77 and 78)

[120] Qian Liu, Shengnan An, Jianguang Lou, B. Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. Compositional generalization by learning analytical expressions. 2020. (page 76)

[121] 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. (page 73)

[122] 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. (page 59)

[123] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of EMNLP*, 2015. (pages 1, 50, 106, and 115)

[124] 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. (page 37)

[125] M. MacMahon, B. Stankiewicz, and B. Kuipers. Walk the talk: Connecting language,

knowledge, and action in route instructions. In *AAAI*, 2006. (page 23)

[126] Chris J Maddison and Daniel Tarlow. Structured generative models of natural source code. In *Proceedings of ICML*, 2014. (page 42)

[127] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating programming by example and natural language programming. In *Proceedings of AAAI*, 2013. (page 14)

[128] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rJgMlhRctm`. (page 23)

[129] Alana Marzoev, S. Madden, M. Kaashoek, Michael J. Cafarella, and Jacob Andreas. Unnatural language processing: Bridging the gap between synthetic and natural language data. *ArXiv*, abs/2004.13645, 2020. (pages 90 and 99)

[130] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. Learned in translation: Contextualized word vectors. In *Proceedings of NIPS*, 2017. (page 59)

[131] 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. (page 23)

[132] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional LSTM. In *Proceedings of CONLL*. (page 59)

[133] Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. Supervised attentions for neural machine translation. In *Proceedings of EMNLP*, 2016. (pages 10 and 77)

[134] Yishu Miao and Phil Blunsom. Language as a latent variable: Discrete generative models for sentence compression. In *Proceedings of EMNLP*, 2016. (pages 104, 105, 107, 108, and 117)

[135] Yishu Miao, Lei Yu, and Phil Blunsom. Neural variational inference for text processing. In *Proceedings of ICML*, 2016. (page 117)

[136] Dipendra Misra, Ming-Wei Chang ad Xiaodong He, and Wen tau Yih. Policy shaping and generalized update equations for semantic parsing from denotations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brusses, Belgium, October 2018. Association for Computational Linguistics. (pages 19 and 20)

[137] Dipendra K. Misra and Yoav Artzi. Neural shift-reduce CCG semantic parsing. In *Pro-*

ceedings of EMNLP, 2016. (page 1)

[138] 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. (page 138)

[139] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of ICLR*, 2016. (pages 69 and 73)

[140] Graham Neubig. lamtram: A toolkit for language and translation modeling using neural networks. http://www.github.com/neubig/lamtram, 2015. (pages 37 and 53)

[141] 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 dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017. (page 131)

[142] 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. (page 42)

[143] Haoran Niu, Iman Keivanloo, and Ying Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, pages 1–33, 2016. (page 138)

[144] Franz Josef Och. *Statistical machine translation: From single word models to alignment templates*. PhD thesis, RWTH Aachen University, Germany, 2002. (page 78)

[145] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1), 2003. (pages 77 and 80)

[146] 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. (pages 35, 51, 109, and 139)

[147] Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. Improving compositional generalization in semantic parsing. In *Proceedings of EMNLP-Findings*, 2020. (pages 77, 78, 80, and 84)

[148] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *In-*

ternational Conference on Program Comprehension (ICPC), pages 63–72. IEEE, 2012. (page 138)

[149] Ankur P. Parikh, Xuezhi Wang, Sebastian Gehrmann, Manaal Faruqui, Bhuwan Dhingra, Diyi Yang, and Dipanjan Das. Totto: A controlled table-to-text generation dataset. In *EMNLP*, 2020. (page 74)

[150] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Proceedings of ACL*, 2015. (pages 22, 61, 67, 69, and 73)

[151] Panupong Pasupat and Percy Liang. Inferring logical forms from denotations. *ArXiv*, abs/1606.06900, 2016. (page 19)

[152] 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. (page 59)

[153] 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. (page 73)

[154] Hoifung Poon. Grounded unsupervised semantic parsing. In *ACL*, 2013. (pages 21 and 22)

[155] Python Software Foundation. Python abstract grammar. https://docs.python.org/2/library/ast.html, 2016. (page 28)

[156] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of ACL*, 2015. (pages 14, 36, 40, and 138)

[157] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of ACL*, 2017. (pages 1, 3, 45, 50, 51, 53, 77, 78, 109, and 111)

[158] Alec Radford, Jeff Wu, R. Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. (pages 91 and 144)

[159] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140), 2020. (pages 76, 77, 80, 144, and 145)

[160] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: synthesizing what I mean:

code search and idiomatic snippet synthesis. In *Proceedings of ICSE*, 2016. (page 14)

[161] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *Proceedings of EMNLP*, 2016. (pages 59 and 97)

[162] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of IJCAI*, 2015. (page 14)

[163] Siva Reddy, Mirella Lapata, and Mark Steedman. Large-scale semantic parsing without question-answer pairs. *Transactions of ACL*, 2014. (page 11)

[164] Jake Russin, Jason Jo, R. O'Reilly, and Yoshua Bengio. Compositional generalization in a deep seq2seq model by separating syntax and semantics. volume abs/1904.09708, 2019. (page 76)

[165] Abigail See, Peter Liu, and Christopher Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of ACL*, 2017. (page 80)

[166] Semantic Machines, Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8, September 2020. (pages 1, 16, 23, 79, and 147)

[167] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT*, 2018. (page 13)

[168] Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, J. Wang, C. D. Santos, and Bing Xiang. Learning contextual representations for semantic parsing with generation-augmented pre-training. In *AAAI*, 2021. (pages 13 and 74)

[169] 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. (page 73)

[170] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. Program synthesis and semantic parsing with learned code idioms. In *NeurIPS*, 2019. (pages 15 and 54)

[171] Richard Shin, C. H. Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, D. Klein, J. Eisner, and Benjamin Van Durme. Constrained language models yield few-shot semantic parsers. *ArXiv*, abs/2104.08768, 2021. (pages 99 and 145)

[172] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. URL https://arxiv.org/abs/1912.01734. (page 23)

[173] Aarti Singh, Robert D. Nowak, and Xiaojin Zhu. Unlabeled data: Now it helps, now it doesn't. In *Proceedings of NIPS*, 2008. (page 108)

[174] 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. (page 117)

[175] 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. (page 138)

[176] 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. (page 131)

[177] Mark Steedman. Surface structure and interpretation. In *Linguistic inquiry*, 1997. (pages 1 and 8)

[178] Mark Steedman. *The Syntactic Process*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262194201. (pages 1, 8, and 115)

[179] Yu Su and Xifeng Yan. Cross-domain semantic parsing via paraphrasing. In *Proceedings of EMNLP*, 2017. (pages 18, 90, and 99)

[180] Alane Suhr, Ming-Wei Chang, Peter Shaw, and Kenton Lee. Exploring unexplored generalization challenges for cross-database semantic parsing. In *Proceedings of ACL*, 2020.

(page 76)

[181] 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. (page 73)

[182] 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. (pages 54 and 73)

[183] 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. (page 73)

[184] Lappoon R. Tang and Raymond J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of ECML*, 2001. (pages 1 and 22)

[185] 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. (pages 5, 65, and 77)

[186] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of NIPS*, 2015. (pages 34 and 51)

[187] Adrienne Wang, Tom Kwiatkowski, and Luke Zettlemoyer. Morpho-syntactic lexical generalization for CCG semantic parsing. In *Proceedings of EMNLP*, 2014. (pages 53 and 111)

[188] 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. (pages 1, 3, 13, 22, 54, 60, 70, and 73)

[189] Bailin Wang, Mirella Lapata, and Ivan Titov. Meta-learning for domain generalization in semantic parsing. *arXiv preprint arXiv:2010.11988*, 2020. (page 76)

[190] Bailin Wang, Mirella Lapata, and Ivan Titov. Learning from executions for semantic parsing. In *Proceedings of NAACL*, 2021. (page 22)

[191] Bailin Wang, Wenpeng Yin, Xi Victoria Lin, and Caiming Xiong. Learning to synthesize data for semantic parsing. In *Proceedings of NAACL*, 2021. (page 96)

[192] 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/. (page 54)

[193] 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. (pages 54 and 73)

[194] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of DSL*, 1997. (pages 46 and 107)

[195] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Proceedings of ACL*, 2015. (pages 17, 73, 90, 99, 104, 139, 144, 149, and 152)

[196] David H.D. Warren and Fernando C.N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8 (3-4):110–122, 1982. URL https://aclanthology.org/J82-3002. (page 11)

[197] 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/. (page 138)

[198] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992. (pages 18 and 108)

[199] 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. (pages 120, 121, 129, 132, and 138)

[200] 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. (page 138)

[201] William Woods, Ronald Kaplan, and Bonnie Webber. The lunar science natural language information system: Final report. 01 1972. (page 11)

[202] Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based structured prediction for semantic parsing. In *Proceedings of ACL*, 2016. (pages 1, 9, 28, 42, 45, and 103)

[203] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. *ArXiv*, abs/2004.09015, 2020. (pages 15 and 22)

[204] Silei Xu, Giovanni Campagna, Jian Li, and Monica S Lam. Schema2qa: High-quality and

low-cost q&a agents for the structured web. In *Proceedings of CIKM*, 2020. (page 99)

[205] Silei Xu, Sina J. Semnani, Giovanni Campagna, and M. Lam. Autoqa: From databases to qa semantic parsers with only synthetic training data. In *Proceedings of EMNLP*, 2020. (pages 21, 22, 86, 87, 88, 89, 91, 96, and 97)

[206] Weidi Xu, Haoze Sun, Chao Deng, and Ying Tan. Variational autoencoder for semi-supervised text classification. In *Proceedings of AAAI*, 2017. (page 117)

[207] Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating structured queries from natural language without reinforcement learning. *arXiv*, abs/1711.04436, 2017. (pages 1, 2, 14, 45, 54, and 73)

[208] 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. (page 131)

[209] 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. (page 59)

[210] 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. (page 138)

[211] Ziyu Yao, Xiujun Li, Jianfeng Gao, Brian Sadler, and Huan Sun. Interactive semantic parsing for if-then recipes via hierarchical reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 2547–2554, 2019. (page 23)

[212] Ziyu Yao, Yu Su, Huan Sun, and Wen tau Yih. Model-based interactive semantic parsing: A unified framework and a text-to-sql case study. In *EMNLP/IJCNLP*, 2019. (page 23)

[213] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. Learning structural edits via incremental tree transformations. *ArXiv*, abs/2101.12087, 2021. (pages 15 and 55)

[214] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of ACL*, 1995. (page 117)

[215] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Joshua B Tenenbaum. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. In *Advances in Neural Information Processing Systems (NIPS)*, 2018. (page

23)

[216] 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. (pages 11 and 60)

[217] Pengcheng Yin. On bridging the semantic gap in knowledge-based question answering, 2016. URL `http://hdl.handle.net/10722/240678`. (page 12)

[218] Pengcheng Yin and Graham Neubig. a syntactic neural model for general-purpose code generation. In *Proceedings of ACL*, 2017. (pages 53 and 112)

[219] 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. (page 70)

[220] Pengcheng Yin, John Wieting, Avirup Sil, and Graham Neubig. On the ingredients of an effective zero-shot semantic parser. under reivew. No citations.

[221] Pengcheng Yin, Nan Duan, B. Kao, Junwei Bao, and M. Zhou. Answering questions with complex semantic constraints on open knowledge bases. *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, 2015. (page 13)

[222] 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. No citations.

[223] 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. No citations.

[224] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. *ArXiv*, abs/1810.13337, 2019. (pages 15 and 55)

[225] 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. No citations.

[226] Pengcheng Yin, Hao Fang, Graham Neubig, Adam Pauls, Emmanouil Antonios Platanios, Yu Su, Sam Thomson, and Jacob Andreas. Compositional generalization for neural semantic parsing via span-level supervised attention. In *Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, June 2021. No citations.

[227] 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. (pages 3, 13, 54, and 73)

[228] 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. (pages 14, 54, and 73)

[229] 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. (pages 12, 16, 22, 61, and 66)

[230] Tao Yu, Rui Zhang, H. Er, Suyi Li, Eric Xue, B. Pang, Xi Victoria Lin, Y. Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander R. Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, V. Zhang, Caiming Xiong, R. Socher, Walter S. Lasecki, and Dragomir R. Radev. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *EMNLP*, 2019. (page 147)

[231] Tao Yu, Rui Zhang, Michihiro Yasunaga, Y. Tan, Xi Victoria Lin, Suyi Li, H. Er, Irene Z Li, B. Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, V. Zhang, Caiming Xiong, R. Socher, and Dragomir R. Radev. Sparc: Cross-domain semantic parsing in context. *ArXiv*, abs/1906.02285, 2019. (page 147)

[232] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Y. Tan, Xinyi Yang, Dragomir Radev, R. Socher, and Caiming Xiong. Grappa: Grammar-augmented pre-training for table semantic parsing. *ArXiv*, abs/2009.13845, 2021. (pages 13, 22, 74, 99, 144, and 145)

[233] Tao Yu, Rui Zhang, Alex Polozov, Christopher Meek, and Ahmed Hassan Awadallah. Score: Pre-training for context representation in conversational semantic parsing. In *ICLR*, 2021. (pages 13, 74, 99, 144, and 145)

[234] 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. (page 138)

[235] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of AAAI*, 1996. (pages 1, 16, 22, 60, and 92)

[236] Luke Zettlemoyer and Michael Collins. Learning to map sentences to logical form: struc-

tured classification with probabilistic categorial grammars. In *Proceedings of UAI*, 2005. (pages 8, 9, 16, and 142)

[237] Luke S. Zettlemoyer and Michael Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of EMNLP-CoNLL*, 2007. (pages 8, 9, 16, 53, 111, and 142)

[238] 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. (pages 60, 70, and 73)

[239] Xiao Zhang, Yong Jiang, Hao Peng, Kewei Tu, and Dan Goldwasser. Semi-supervised structured prediction with neural crf autoencoder. In *Proceedings of EMNLP*, 2017. (page 117)

[240] Yuchen Zhang, Panupong Pasupat, and Percy Liang. Macro grammars and holistic triggering for efficient semantic parsing. In *Proceedings of EMNLP*, 2017. (pages 19, 20, and 69)

[241] 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. (page 73)

[242] 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. (page 73)

[243] Kai Zhao and Liang Huang. Type-driven incremental semantic parsing with polymorphism. In *Proceedings of NAACL-HLT*, 2015. (pages 10 and 53)

[244] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv*, abs/1709.00103, 2017. (pages 1, 2, 13, 14, 16, 22, 45, 52, 54, 73, and 103)

[245] Chunting Zhou and Graham Neubig. Multi-space variational encoder-decoders for semi-supervised labeled sequence transduction. In *Proceedings of ACL*, 2017. (page 105)

[246] Xiaojin Zhu. Semi-supervised learning literature survey. Technical Report 1530, Computer Sciences, University of Wisconsin-Madison, 2005. (pages 20 and 117)

[247] Lei Zou, Ruizhe Huang, Haixun Wang, J. X. Yu, W. He, and Dongyan Zhao. Natural

language question answering over rdf: a graph data driven approach. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014. (page 11)