NATURAL LANGUAGE INTERFACES FOR SEMI-STRUCTURED WEB PAGES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Panupong Pasupat
August 2019

Abstract

Question answering (QA) systems take natural language questions and then compute answers based on a knowledge source. This dissertation focuses on improving QA systems along two axes. First, instead of operating on knowledge sources with a fixed schema such as a database, we propose to use web pages, which contain a large amount of up-to-date open-domain information (high BREADTH), as the knowledge source. Second, we want the QA system to understand more complex questions and perform different types of multi-step reasoning to compute the answer (high DEPTH). Unlike most previous works on retrieval-based QA (which operate on open-domain unstructured text but target only factoid questions) and knowledge-based QA (which can handle compositional questions but on knowledge sources with fixed schemata), we aim to address the two axes simultaneously.

One important aspect of web pages is that they are semi-structured: they contain structural constructs such as tables and template-generated product listings, but the schemata of such structures are not known in advance by the QA system. To explore the semi-structured nature of web pages, we first investigate the task of extracting a list of entities from the web page based on the natural language specification (e.g., from "*(What are) hiking trails near Baltimore*", extract the trail names from a table column). Then, to increase the complexity of the questions, we next study the task of answering complex questions on open-domain semi-structured web tables using question-answer pairs as supervision (e.g., answering "*Where did the last 1st place finish occur?*" in an athlete's statistics table). To handle compositional questions with different types of operations, we frame the task as learning a semantic parser, which maps questions into compositional logical forms that can be executed to get the answer. Our semantic parser can answer complex questions on unseen web tables and achieves an accuracy of 43.7% on the dataset. Overall, we show that while the unknown schema of the tables (increased BREADTH) and complexity in the questions (increased DEPTH) lead to an exploding search space of logical forms, our proposed methods control the search space to a manageable size, enabling us to train a QA system that can operate on open-domain web pages. The resulting QA system can potentially enable virtual assistants, search engines, and other similar products to handle a much wider range of user's utterances.

Acknowledgments

First of all, I am deeply indebted to my advisor, Percy Liang, who has been amazingly supportive throughout my whole Ph.D. career. Percy's academic excellence needs no introduction, and by being his advisee, I learned that Percy's work ethics, time management, lateral thinking, expertise in numerous research areas, and communication skills all contributed to his success. I was lucky to be among the first students in Percy's group, meaning that I had an opportunity to receive a lot of advice from him, both in academic and personal matters. Percy was also one of the most patient and understanding person I have ever met, and I do not think I can sanely survive the Ph.D. program without him. I am extremely grateful for the time and effort Percy dedicated to support me, and I would like to thank him from the bottom of my heart.

I would also like to thank my dissertation committee for their guidance and suggestions. First, I am grateful to Chris Manning for being a great runner of the Stanford NLP community and an advisor for my first-year rotation project. I also learned a lot from his information retrieval course while I was working as a course assistant. Next, I want to thank Dan Jurafsky for his support and advice throughout my career. His textbook also plays an important role for the content in this dissertation. I want to thank Chris Ré for asking critical questions during the oral defense and directing me to the work on information extraction. Finally, I want to thank Chris Potts for agreeing to be the oral committee chair and for his great suggestions during the defense.

During the past six years, I had great pleasure of working with many amazing colleagues and coauthors. For the work presented in this dissertation, I would like to especially thank Kelvin Guu and Evan Liu for being my friends and collaborators on various projects over the last three years; Yuchen Zhang for the work on macro rules; Allan Jiang and Tim Shi for the work on web interaction; and Reggy Long for for the work on sequential semantic parsing. I would like to also thank my collaborators on other projects at Stanford University: Jonathan Berant, Kartik Chandra, Robin Jia, Ramtin Keramati, Jason Koenig, Sumith Kulal, Mina Lee, Oded Padon, Sudarshan Seshadri, Megha Srivastava, and Sida Wang. I also want to extend my gratitude to Alex Aiken and Emma Brunskill for supervising me on my joint projects outside this dissertation.

The research presented in this dissertation would not exist without the funding by the following organizations: Amazon, Defense Advanced Research Projects Agency (DARPA), Google, Microsoft, National Science Foundation (NSF), and Tencent.

My research experience was also shaped by the research internships during my Ph.D. years. I would like to

thank Asli Celikyilmaz, Dilek Hakkani-Tür, and Gokhan Tür for my internship at Microsoft Research; Arne Mauser and Jakob Uszkoreit for my internship at Google; and Arash Einolghozati, Sonal Gupta, Mike Lewis, Karishma Mandyam, Mrinal Mohit, Rushin Shah, and Luke Zettlemoyer for my internship at Facebook.

I am also thankful for my marvelous friends: everyone at the Stanford Natural Language Processing Group, who were extremely friendly and supportive even during the tightest deadlines; and all my friends at the Stanford Thai Student Association, who taught me life skills and gave me a space to socialize. I want to especially thank Anak Yodpinyanee for sparking my interest in natural language processing; Pramook Khungurn for giving me advice on academic matters; and Warut Suksompong for hanging out with me and being a great friend during the past 10+ years.

Finally, I would like to express my gratitude to my family for raising me and giving me emotional supports throughout my life. Words cannot describe how much my father, my mother, and my sister mean to me and how much I appreciate them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Question answering (QA) has been both an academically critical and commercially viable task throughout the history of artificial intelligence. From the pedagogical viewpoint, the ability to comprehend and respond to natural language questions has been used to evaluate the "intelligence" of artificial intelligence systems since the early days (Turing, 1950; Winograd, 1972). And from the practical viewpoint, question answering has become the core functionality of crucial commercial products such as search engines, virtual assistants, and automated help systems. Needless to say, improving the capability of QA systems will have a large impact in both academic and business endeavors.

While question answering is a large and diverse field, almost all QA systems fundamentally share the same basic structure: given a natural language *question* x, a QA system references some *knowledge source* w to compute an *answer* y to the question. However, different QA systems target different types of questions and utilize different knowledge sources, as demonstrated in the examples below:

- An extractive reading comprehension system (Brill et al., 2002; Seo et al., 2016; Chen et al., 2016) can answer questions (e.g., $x =$ "*Where was Barack Obama born?*") by looking at unstructured text ($w =$ the Wikipedia article *Barack Obama*) and extracting a phrase to be used as the answer ($y =$ "*Honolulu, Hawaii*").

- A task-oriented virtual assistant (Pieraccini et al., 1991; Raymond and Riccardi, 2007; Mesnil et al., 2014) can handle domain-specific questions by issuing API queries to a database service. For instance, when the user asks the question $x =$ "*What's the weather in Seattle today?*", the system can issue a query of the form $\texttt{GetWeather}(\texttt{location} = \texttt{Seattle}, \texttt{date} = \texttt{GetToday}())$ to $w =$ a weather database to get the answer $y =$"*23°C sunny*".

- Some search engines such as Google and Microsoft Bing use *knowledge bases* as the knowledge source.

A knowledge base contains open-domain information that were either manually curated or automatically extracted from other knowledge sources. When such search engines detect a fact-based question (e.g., $x =$ "*Where was Barack Obama's wife born?*"), they can construct a knowledge base query that yields the answer ($y =$ "*Chicago, IL*").

This dissertation focuses on improving the capability of question answering systems along two axes: widening the scope of the knowledge source that the system can use (BREADTH) and increasing the complexity of the questions that the system can handle (DEPTH).

1.1.1 BREADTH: Scope of the knowledge source

Question answering systems are designed to work within different types of knowledge sources, each with different benefits and restrictions. On one end of the spectrum, natural language understanding systems in most virtual assistants (Pieraccini et al., 1991; Raymond and Riccardi, 2007; Mesnil et al., 2014) and natural language interfaces to databases (Hendrix et al., 1978; Androutsopoulos et al., 1995) are designed to work well in a knowledge source with a predefined domain, such as reserving flights or interacting with an internal financial database. The knowledge sources they use, such as databases and knowledge bases, have a predefined *schema* dictating what types of entities exist in the knowledge source and what relations they can have with one another. For instance, a database in the flight domain would have information about flights and airports, but might not contain other general knowledge such as tourist attractions. With a fixed data schema, the QA system can restrict the possible interpretations of the utterances x to the ones that fit the schema, making it easier to design or train the system. However, the fixed schema inherently limits the scope of utterances the system can handle. For instance, the flight database mentioned above would not be able to handle the question "*What are flights to Cornell University next weekend*" since the information about Cornell's location is not encoded in the database.

To handle a broader set of questions, a QA system needs to utilize *open-domain* knowledge sources with arbitrary data schema. For instance, instead of using a fixed domain-specific database, a natural language interface to databases that knows how to utilize any arbitrary databases (which might be selected on-the-fly based on the input question) would be able to answer a wider range of questions. However, the lack of a fixed data schema makes it difficult to interpret the questions. For instance, the question "*How many US presidents are from New York?*" might involve just a simple number lookup in one data schema, but could require complex filtering and counting in others. The QA system has to choose the correct method to compute the answer based on the data schema, which can be challenging if it has never encountered the schema before.

That said, using open-domain knowledge sources in a QA system is not an impossible feat. Unlike the closed-domain QA systems above, previous work on *retrieval-based QA* has been using open-domain knowledge found in unstructured data as the knowledge source. For instance, text-based QA systems (Voorhees and Harman, 1999; Brill et al., 2002; Seo et al., 2016; Chen et al., 2017) use reading comprehension models to extract answers from text paragraphs or documents, while visual QA systems (Antol et al., 2015; Tapaswi et al., 2016) can answer questions based on the given images or videos. Nevertheless, most previous retrieval-based

QA works do not target more complex questions that require multi-step reasoning; for instance, a retrieval-based QA system will have a hard time answering the question "*How many US presidents are from New York?*" if the number is not explicitly stated in the document. As we will discuss in the following subsection, the struggle to perform multi-step reasoning can prevent the system from answering a significant portion of interesting questions, even though the knowledge source is open-domain.

1.1.2 DEPTH: Complexity of the questions

Many QA systems handle questions that only require surface form matching or very few steps of reasoning to compute the answer. For instance, the lookup question "*Where was Obama born?*" could be handled by matching the surface form to the source documents, or by identifying the relevant fields {entity: *Barack Obama*, relation: *place of birth*} for querying a database. But occasionally, users will ask questions that require multiple non-trivial steps to answer. For instance, the question "*Where was the current US president born?*" requires identifying the current president before retrieving the place of birth, and "*How many presidents were born in New York before Trump?*" requires even more complex reasoning with filtering (*in New York*), comparison (*before Trump*), and aggregation (*how many*).

We consider two main aspects of question complexity. The first is *compositionality*: the number of steps of operations involved when computing the answer. Note that the question does not need to be long or highly nested to make the task compositional. For example, the short question "*Obama's aunt*" can be compositional if we only know how to look up the parents, siblings, and gender of each person in the database.

The second aspect is the *variety of operations*: the number of unique atomic operations that can be performed on the knowledge source to compute the answer. For instance, we want a QA system that can perform multiple types of reasoning such as comparison (e.g., to handle questions containing "*... more than 60 years*" or "*The oldest ...*"), navigation (e.g., "*... right after Lincoln*"), aggregation (e.g., "*How many ...*"), and calculation (e.g., "*The total ...*" or "*... difference between ...*") when computing the answer.

A class of QA systems known as *knowledge-based QA* systems were designed to tackle such complex questions by predicting the steps of reasoning required to compute the answers. Examples of knowledge-based QA systems include natural language interfaces to databases mentioned in the previous section, which use database queries to represent the steps of computation; and semantic parsers for questions answering (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2007; Liang et al., 2011), which convert questions into compositional and executable logical forms. However, these QA systems traditionally operate on structured knowledge sources with a limited domain and a fixed schema, such as a domain-specific database. And while more recent semantic parsing works consider using large knowledge bases with information extracted from open-domain sources, the fixed schema and incompleteness of the knowledge bases still limits the types of questions that can be answered.

1.2 Increasing BREADTH and DEPTH simultaneously

As mentioned earlier, the goal of this dissertation is to demonstrate QA systems that can handle broader, open-domain knowledge sources w as well as more complex questions x. We now give an overview of our work, which involves introducing new QA tasks with increased BREADTH and DEPTH, and then proposing novel algorithms to address the challenges introduced by the task settings.

1.2.1 New QA tasks with increased BREADTH and DEPTH

To aid our investigation, the first few chapters of this dissertation propose a series of QA tasks to address BREADTH and DEPTH *simultaneously*. In these tasks, we focus on the scope of questions that can be asked within the task settings, as well as the potential challenges that the QA systems will have to address.

BREADTH: Using web pages as the knowledge source. To increase the scope of the knowledge source, we focus on using *web pages*, which form a huge, open-domain, and up-to-date repository of information. One distinguishing factor of web pages is that they are *semi-structured*: a web page contains structural constructs (e.g., the internal tree representation of the whole page, tables, and template-generated product listings), but the schemata of such structures are arbitrary depending on the whim of the webmaster. This lack of a predefined schema allows web pages to represent a much wider variety of information than a database or other fixed-schema knowledge sources. At the same time, the structural nature of semi-structured data makes it appropriate for displaying certain types of information, such as a series of objects (e.g., list of national parks) and their statistical properties (e.g., the location and area of each national park), that would be awkward when written out as plain text.

In Chapter 3, we first explore the semi-structured nature of web pages. To aid our investigation, we consider the task of extracting a list of entities from the web page based on the input question (e.g., "*(What are) hiking trails near Baltimore*" $\rightarrow$ {"*Avalon Loop*", "*Hilton Area*", ...} extracted from the *Name* column of the table on the page). While most input queries we consider are not very complex, we end the chapter by highlighting the need for compositional reasoning in some queries (e.g., extracting "*tech companies in China*" requires filtering the cells from the table). This chapter is based on our works in Pasupat and Liang (2014).

DEPTH: Answering compositional questions on web tables. We turn to addressing question complexity by introducing the task of answering compositional questions. Among different types of structured information on web pages, we choose *HTML tables* as the source of information. Our rationale is that tables typically contain computable data suitable for answering complex questions (high DEPTH), while at the same time are open-domain and contain arbitrary data schemata (high BREADTH).

Chapter 4 formalizes our task setup and explains how we collect a dataset of compositional questions with diverse operations. The resulting WIKITABLEQUESTIONS dataset contains over 20,000 question-answer

pairs on HTML tables of various topics. We then compare the task and the dataset to related tasks in terms of BREADTH and DEPTH. This chapter is based on our work in Pasupat and Liang (2015).

1.2.2 Tackling the increased BREADTH and DEPTH

Using the task of answering complex questions on semi-structured tables as a benchmark, we next investigate techniques for handling diversified data schemata and compositional reasoning.

Semantic parsing. Chapter 5 presents our semantic parsing approach for answering complex questions on web tables. Given a question, our parser generates and scores executable *logical forms* that represent the semantics of the question (e.g., parsing "*How many presidents were born in Georgia?*" to the logical form `count(HasBirthState.Georgia)`, where `BirthState` is a table column and `Georgia` is a table cell). Logical forms are compositional, and thus are suitable for representing the steps needed to compute the answer. The parser is trained using the annotated answer for each question as a *distant supervision*. During training, the parser performs search over the space of logical forms, and then updates its parameters so that the logical forms executing to the correct answer are ranked higher than others.

On-the-fly information extraction. As mentioned in Section 1.1.2, semantic parsers traditionally operate on structured data with a known schema (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2007; Liang et al., 2011). One common technique to utilize open-domain data in semantic parsers is to apply *information extraction* (Hearst, 1992; Kushmerick, 1997; Banko et al., 2007; Mintz et al., 2009). Given a corpus of unstructured or semi-structured data, an information extraction system detects the information that satisfies the desired database schema, and then populate the database accordingly. While this allows open-domain knowledge to be aggregated and canonicalized for the downstream QA system, the assumed schema of the database still limits the type of knowledge that can be extracted.

To tackle this issue, we apply *on-the-fly information extraction*: from the web table of interest, we create a temporary structured data just for that table. In particular, we convert the table into a generic graph structure that can (1) encode arbitrary table fields as graph edges, and (2) encode uncertainty in how text in table cells should be interpreted. Logical forms can be executed on the resulting graph, and the semantic parser uses information from the graph to construct candidate logical forms.

We end Chapter 5 with empirical analyses of the dataset and the semantic parser. This chapter is based on our work in Pasupat and Liang (2015).

1.2.3 Tackling search problems

The WIKITABLEQUESTIONS dataset we introduced in Chapter 4 is *distantly supervised*: it was annotated with the answers to the questions, but not how the answers can be derived. While this makes the dataset method-agnostic and cheap to collect, the training procedure for our method now has to search for logical forms that execute to the correct answer. In the final two chapters, we outline the challenges caused by

this need to search over the large space of logical forms, and then provide two orthogonal solutions to the challenges.

Speeding up search. To enable the parser to answer a wider range of questions, we could augment the parser with more logical operators and constructs for generating logical forms. Unfortunately, this expands the space of possible logical forms and significantly slows down the search algorithm. Chapter 6 proposes a method to speed up search by using the *patterns* of logical forms found in previous training examples to guide the search for similar examples. For example, after parsing "*How many presidents were born in Georgia?*" into `count(HasBirthState.Georgia)`, we reuse the logical form pattern `count(Has{`*Col*#1`}.{`*Cell*#2`})` when parsing future questions of the form "*How many ... were ... in ... ?*". The resulting speedup allows the model to explore more compositional logical forms and more diverse logical operators in less amount of time. This chapter is based on our work in Zhang et al. (2017).

Precomputing semantically correct logical forms. While using patterns to guide search increases the overall efficiency, performing search over the logical forms during training still poses two other fundamental issues. First, if the search fails to find any logical form that executes to the correct answer, the parameters cannot be updated. Second, the search can generate spurious logical forms that execute to the correct answer for wrong reasons (e.g., for "*How many presidents were born in Georgia?*", it turns out `count(HasDeathState.Georgia)` also gives the same answer). These spurious logical forms become noises that could lead the model to learn incorrect associations.

Since distant supervision is the reason we need to perform search during training, we propose to add *direct supervision* to the dataset by augmenting each example with semantically correct logical forms. Chapter 7 presents algorithms that, given a training example, exhaustively enumerate logical forms that execute to the correct answer, and then filter out the spurious logical forms using a small amount of human supervision. The resulting logical forms can be used to train a semantic parser directly without having to perform search over logical forms. This chapter is based on our work in Pasupat and Liang (2016).

Chapter 2

Related Work

Before we discuss previous works on question answering (QA), we first look at the settings that QA systems are designed to operate on. First, Section 2.1 analyzes the types of *knowledge sources* that QA systems pull information from. We describe and contrast the nature of structured, unstructured, and semi-structured data, with a special emphasis on the sorts of information they tend to contain and the challenges they pose to the QA systems. Then in Section 2.2, we look at the *types of questions* different QA systems are designed to handle. We particularly focus on *fact-based*[1] questions and highlight the various types of complexity that can be present in the questions.

Afterward, we review the two main lines of work on question answering. First, Section 2.3 explores *retrieval-based QA* systems. As the name suggests, these QA systems first retrieve snippets from the knowledge source that are relevant to the question, and then use reading comprehension to extract the answer from the retrieved snippets. Second, in Section 2.4, we look at *knowledge-based QA* systems, which perform reasoning on the structure of the knowledge source to compute the answer. We particularly focus on the *semantic parsing* approach, which we use for the main task of this dissertation.

These two classes of QA systems traditionally operate on the opposite ends of the spectrum. As illustrated in Figure 2.1, retrieval-based QA works on open-domain unstructured data (high BREADTH) but traditionally aims to answer questions that do not require multi-step reasoning (low DEPTH). In contrast, the strength of knowledge-based QA is its ability to perform multi-step reasoning to answer complex questions (high DEPTH), but the knowledge sources are traditionally constrained to structured data with predefined schemata, such as databases or knowledge bases (low BREADTH). The goal of this dissertation (and several other recent works) is to build QA systems that can operate in high-BREADTH high-DEPTH settings.

Finally in Section 2.5, we discuss *information extraction* (IE) systems. An IE system transforms unstructured or semi-structured data into the structured format that knowledge-based QA systems can process, which allows them to answer questions that are both complex and open-domain. The goal of most IE systems in the

[1] While questions with factual answers are sometimes called "factoid" questions, the term "factoid" sometimes denotes simple questions that do not need complex reasoning. We use the term "fact-based" to avoid confusion.

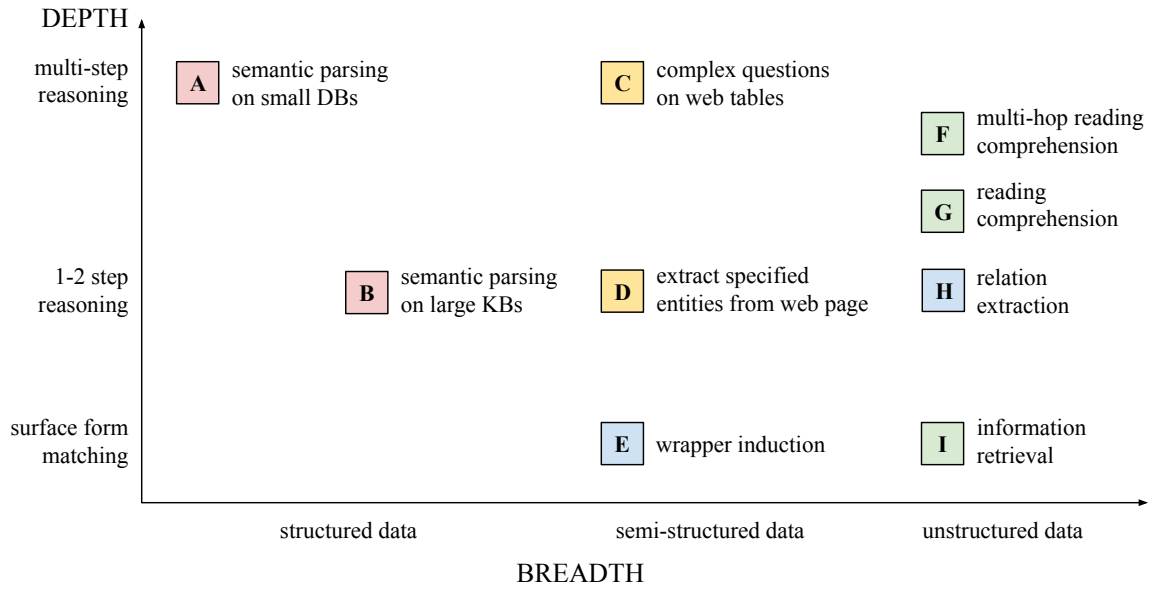


Figure 2.1: Comparison of retrieval-based QA (green), knowledge-based QA (red), and information extraction systems (blue) on the scope of the knowledge source (BREADTH) and the question complexity (DEPTH). This dissertation starts with exploring semi-structured data (Point D), and then progresses to a task that addresses both BREADTH and DEPTH simultaneously (Point C).

literature is to populate a central structured knowledge source with a predefined schema. While this allows information from multiple sources to be aggregated in a systematic manner, the predefined schema, along with precision and recall issues, inherently limits the amount of knowledge accessible by the downstream systems. As an alternative, the work in this dissertation employs *on-the-fly information extraction*: instead of populating a central fixed-schema database, we only convert parts of the source data that are relevant to the question into a temporary structured knowledge source. Without a predefined schema, we are able to capture more information from the source data; however, we lose some benefits of having a unified database (e.g., we can no longer aggregate data from different sources, and running extraction for every question can be costly).

2.1 Knowledge sources

We now classify the types of knowledge sources used in QA systems based on (1) the amount of structure specified in the knowledge sources and (2) whether the data schema is predefined. The BREADTH axis in Figure 2.1 lays out the different types of knowledge sources described below.

2.1.1 Structured data

A structured knowledge source has a well-defined *schema*, which dictates the types of objects that can exist in the knowledge source, as well as the types of relations among them. Database is a canonical example of structured knowledge sources: the database schema defines the value type in each field, the fields in each record, and the relationship between records across database tables. Other examples of structured knowledge sources include graph-structured knowledge bases (Google, 2013; Suchanek et al., 2007; Auer et al., 2007) and constrained interactive environments (e.g., a simulated world with blocks that can be moved around). Most knowledge-based QA systems, such as natural language interfaces to databases (Androutsopoulos et al., 1995) and semantic parsers (Zelle and Mooney, 1996; Berant et al., 2013; Chen and Mooney, 2011) operate on structured knowledge sources.

Using a structured knowledge source provides several benefits. A structured knowledge source can store a large amount of statistical data, and usually provides a convenient interface (e.g., database query languages) for performing computation on such statistics. Furthermore, structured data is a systematic way to consolidate knowledge. In contrast to unstructured text which may describe the same object, relation, or event using a wide variety of paraphrases, the schema of structured data forces the representation of information to be unified and canonicalized, making it easier to process.

On the flip side, structured data presents a trade off between knowledge coverage and maintainability. Closed-domain structured knowledge sources, such as a database for a specific application, only support questions that are asked within that domain. On the other hand, open-domain structured knowledge sources, such as large knowledge bases, support a wider range of questions but are difficult to populate and maintain.

Finally, the fixed schema in structured data is both a blessing and a curse. It is easier to design or learn a mapping between natural language phrases in the question and objects in the knowledge source when the schema is known in advance. However, the fixed schema limits the types of knowledge that can be included in the knowledge source.

2.1.2 Unstructured data

Unstructured knowledge sources are knowledge sources that do not contain a schema. While some QA works operate on *unstructured media* such as images or videos, most retrieval-based QA systems operate on *unstructured text*, which we will focus on for our comparison.

A large amount of open-domain world knowledge is encoded as unstructured text due to the flexibility and expressiveness of natural language. Apart from describing objects and their properties, unstructured text is also used to express more complex structures such as events and processes, or more vague concepts such as arguments and opinions. Using unstructured data as the knowledge source opens the door for a much wider range of questions.

As its downside, unstructured text is more difficult to process than structured data. Due to the flexibility of natural language, a QA system must be ready to handle different sentence structures and linguistic patterns

people use to express the same concept. Additionally, compared to structured data, plain text is a more clunky way to express a long series (e.g., list of all national parks in United States) or statistical data (e.g., the area of each national park). This means unstructured data is less suitable for questions that require computation; for instance, the data needed for computing the average area of national parks is less likely to be expressed in plain text (except when the average is already computed and explicitly stated in text).

2.1.3 Semi-structured data

Semi-structured data contains some semantic structures, but the schema of such structures is not predefined (Abiteboul, 1997; McHugh et al., 1997). Examples of semi-structured data include markup documents (e.g., schema-less XML files and HTML pages) and sections with uniform patterns within a document (e.g., tables, bullet lists, and template-generated product listings on web pages). While a spcific data source might use the same schema for all of its documents, that schema would not apply to the similar type of content from a different data source.

Semi-structured data inherits the advantages and disadvantages from both structured and unstructured data. Similar to structured data, semi-structured data usually contains computable data, but since the schema is not predefined, it can express a variety of open-domain knowledge depending on the author of the data. On the flip side, the lack of a predefined schema increases the difficulty of mapping phrases in the questions to objects in the knowledge source.

2.2 Types of questions

We now discuss the types of questions targeted by previous work in question answering. The family of questions our work focuses on is *fact-based* questions, which have objective and mostly unique answers with respect to the knowledge source (e.g., "*Where was Barack Obama born?*" or "*What is the average earning of the company in 1999?*"). Other than fact-based questions, previous work has also considered questions whose answers are long explanations or opinions (Burke et al., 1997; Soricut and Brill, 2006). Most systems for answering such open-ended questions treat the task as a information retrieval task, where the potential answer snippets are retrieved and then ranked.

2.2.1 Complexity in fact-based questions

Most retrieval-based and some knowledge-based QA systems target fact-based questions that only require simple reasoning (sometimes called "factoid" questions). As an example, consider the question "*Where was Barack Obama born?*" asked under different knowledge sources. When the knowledge source is a corpus of text documents, a retrieval-based QA system can fetch a snippet that is likely to contain the answer (e.g., "*Obama was born on August 4 . . . in* ***Honolulu, Hawaii***." from the Wikipedia article *Barack Obama*), and then extract the answer "*Honolulu, Hawaii*" directly from a sentence in that snippet. On the other hand, if

the knowledge source is a structured database, a knowledge-based QA system can identify the relevant entity (*Barack Hussein Obama*) and relation (*place of birth*) to find the answer. In either cases, given a portion of the knowledge source relevant to the question, the answer can be computed using few steps of reasoning.

As stated in Chapter 1, we want to build QA systems that can answer complex questions (i.e., high on the DEPTH axis in Figure 2.1). We consider the following sources of complexity that can arise in fact-based questions.

Multi-step reasoning. Performing multiple steps of reasoning has been the goal of many knowledge-based QA and interactive systems. As an example, consider an early conversational system SHRDLU (Winograd, 1972), which takes natural language commands or questions, manipulates blocks in a simulated three-dimensional environment, and then generates natural language responses. SHRDLU was designed to understand complex commands such as "*Find a block which is taller than the one you are holding and put it into the box.*" and "*Is at least one of them narrower than the one which I told you to pick up?*" (where *them* refers to the answer from the previous question). Another example is an early semantic parsing system by Zelle and Mooney (1996), which aims to answer highly compositional questions such as "*What states border states that border states that border states that border Texas?*" using a database as the knowledge source.

One common criticism of targeting multi-step reasoning is the argument that humans rarely ask questions with complex linguistic constructs. Indeed, since most commercial retrieval and question answering systems (e.g., search engines and early virtual assistants) cannot understand complex questions well, the users have learned to ask simple questions or even use a fragmented list of words for such products. Nevertheless, with the introduction of speech-enabled devices, users have started to use natural language more often, which increases the frequency of questions that require multiple steps of reasoning (though not as many steps as the examples given above).

Sublexical compositionality. Multi-step reasoning does not necessarily come from questions with high linguistic compositionality. Rather, depending on the type of information available in the knowledge source, even a simple word or phrase and invoke multi-step reasoning. An example of such *sublexical compositionality* (Wang et al., 2015b) is the question "*Who is Barack Obama's* ***aunt****?*". The question only requires one step of reasoning if the knowledge source explicitly mentions the *aunt* relationship. However, for a database where only a person's parents, siblings, and gender can be queried, multiple steps of reasoning will be needed to answer the question. Several other examples include "*weekend*" (Saturday or Sunday), "*same position as John*" (people with the same position as John, except John himself), and "*win*" (might require summing up the scores to be compared).

Diversity of operators. Another potential source of complexity in fact-based questions comes from the types of reasoning needed to compute the answer. For instance, a QA system dealing with statistics should be able to perform comparison ("*Which company has the highest revenue?*"), aggregation ("*What is the average*

revenue of tech companies?"), and computation ("*How much more does Walmart make than Apple?*") on the numerical data in the knowledge source.

As we will discuss in Section 2.4, many knowledge-based QA systems use *logical operators* to represent types of reasoning. Within this framework, the system has to choose the correct logical operators, their arguments, and the order they should be composed to compute the answer. This can be challenging as the number of possible combinations of operators scales with the size of the knowledge source and grows exponentially with the number of reasoning steps.

2.3 Retrieval-based question answering

We now discuss the first family of question answering systems, retrieval-based QA, which is often used to handle open-domain questions on unstructured knowledge sources. To answer a given fact-based question, most retrieval-based QA systems (1) retrieve snippets from the knowledge source that are relevant to the question, and then (2) use reading comprehension to extract the answer from the retrieved snippets.

Snippet retrieval. Given the question, the task of snippet retrieval is to fetch portions of the knowledge source that are relevant to the question, and then rank them based on relevancy.

For fetching the candidate snippets to be ranked, the most basic method is to find documents with large word overlap with the question. Weighting schemes such as tf-idf is often used to emphasize the matching of important content words (Jones, 1972; Robertson and Zaragoza, 2009). To increase recall, previous work has also used query expansion and query rewriting to generate similar queries for retrieving a larger number of relevant snippets (Carpineto and Romano, 2012).

Various factors can be considered when ranking the snippets. For instance, given user ratings as training data, previous work considered training a supervised model to assign scores to the retrieved snippets (Cao et al., 2006; Yue et al., 2007). Other factors such as prominence or credibility of the sources can also be taken into account (Haveliwala, 2002; Gyöngyi et al., 2004)

Reading comprehension. Given a snippet for the question, the next step is to extract the answer from the snippet. Early work in reading comprehension analyzes the question type and uses linguistics patterns to extract the answer (Brill et al., 2002; Paşca, 2003). With the availability of larger datasets and neural models, more recent works cast reading comprehension as predicting a token span inside the given snippet (Yang et al., 2015; Rajpurkar et al., 2016; Seo et al., 2016). This generally involves embedding the snippet and the question, computing interaction between the two, and then outputting the start and end positions of the span. For fact-based questions with few steps of reasoning, such neural models have matched the human performance on many benchmarks (Chen et al., 2016; Devlin et al., 2018).

Increasing question complexity. Traditionally, the questions that retrieval-based QA applies on are usually "factoid" questions that do not involve multi-step reasoning (Voorhees and Harman, 1999; Brill et al.,

2002; Paşca, 2003). However, more recent work has started to move toward questions that require more complex reasoning. For instance, datasets such as SQuAD 2.0 (Rajpurkar et al., 2018) and Natural Questions (Kwiatkowski et al., 2019) require the system to reason whether the question can be answered by the given snippet. The DROP dataset (Dua et al., 2019) contain questions that require various types of operations such as cross-referencing, comparison, and calculation to compute the answer. And the HotpotQA dataset (Yang et al., 2018) requires two-step reasoning where a second snippet has to be retrieved based on the information gained from the first snippet.

Nevertheless, the types of questions and their complexity are still partially limited by the nature of the knowledge source. As discussed in Section 2.1.2, questions that require aggregating the attributes of many entities (e.g., "*What is the average revenue of . . . ?*" or "*Who is the oldest . . . ?*") are usually covered by structured or semi-structured data rather than unstructured text.

Question answering on images. As a side note, a similar trend of increased question complexity has also been observed in the related task of visual question answering. After the success of object recognition (Krizhevsky et al., 2012; Szegedy et al., 2015; He et al., 2016) and image captioning (Farhadi et al., 2010; Lin et al., 2014; Fang et al., 2015; Mao et al., 2015), researchers turned to the task of answering questions about the images. Most earlier works consider either simpler questions on diverse images (Antol et al., 2015) or complex questions on synthetic images (Johnson et al., 2017a; Suhr et al., 2017). However, recent work has moved toward considering higher BREADTH and DEPTH simultaneously: understanding complex questions in the context of open-domain images (Suhr and Artzi, 2018; Hudson and Manning, 2019).

2.4 Knowledge-based question answering

Knowledge-based QA systems perform reasoning on the facts in the knowledge source to compute the answer. Among different ways to perform reasoning, we focus on the *semantic parsing* approach, which we use for our main task. In semantic parsing, the steps of computation are represented as a discrete representation called a *logical form*, and the task is thus to map the given question into a logical form that represents a correct way to compute the answer.

2.4.1 Executable semantic parsing

Broadly speaking, *semantic parsing* is the task of mapping natural language utterance to *some* meaning representation. As different subfields of natural language processing employ different schemes of meaning representations, the term "semantic parsing" has become a heavily overloaded term. The possible interpretations include shallow intent-slot tagging in dialog systems (Pieraccini et al., 1991; Raymond and Riccardi, 2007; Mesnil et al., 2014), identifying semantic frames (Gildea and Jurafsky, 2002; Hermann et al., 2014), and generating full semantic representations such as abstract meaning representation (AMR) (Banarescu et al., 2013; Flanigan et al., 2014; Wang et al., 2015a; Artzi and Zettlemoyer, 2015) among many others.

For the purpose of question answering, we will focus on *executable semantic parsing*, which is based on model-theoretic formal semantics (Montague, 1973). In this framework, the semantic representations are interpreted under the knowledge source (traditionally called a "model"). The result of this interpretation is called *denotation*, which denotes some part of the knowledge source. For instance, in a database of countries in the world, the denotation of $\texttt{CapitalOf}(\texttt{France})$ is a single entity `Paris`, while the denotation of `CapitalOf` is the mapping between countries and their capital cities.

In practical terms, executable semantic parsing is the task of mapping the input utterance x into a semantic representation z (usually called *logical forms*) that can be executed like a computer program on the knowledge source w to give some desired denotation $y = [\![z]\!]_w$. For instance, the utterance $x =$ "*Tell me what 2 + 2 is.*" can be mapped to $z = \texttt{add}(2, 2)$, which can be executed to give the denotation $y = 4$. The logical form is not required to capture all semantic details of the utterance as long as its denotation under the knowledge source is correct (e.g., in the example above, the *Tell me* part is not represented in the logical form).

Semantic parsing has been applied in many tasks that require compositional reasoning. Some examples include: parsing questions into database queries for retrieving the answers (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2007; Berant et al., 2013; Dong and Lapata, 2016; Zhong et al., 2017), parsing the user's queries into API calls (Quirk et al., 2015), parsing commands for navigating an agent (Chen and Mooney, 2011; Tellex et al., 2011; Artzi and Zettlemoyer, 2013; Andreas and Klein, 2015), parsing commands for manipulating objects (Guu et al., 2017; Fried et al., 2018), and parsing specifications into code in a programming language (Kushman and Barzilay, 2013; Ling et al., 2016; Yin and Neubig, 2017; Rabinovich et al., 2017; Iyer et al., 2018).

In the following subsections, we will focus on works in semantic parsing for question answering that have motivated our task settings.

2.4.2 Highly compositional semantic parsers

Early semantic parsing systems focus on understanding very complex utterances in a predefined domain. For example, the SHRDLU system (Winograd, 1972) mentioned earlier can understand very compositional commands such as "*Find a block which is taller than the one you are holding and put it into the box.*", but with hand-crafted rules, it is difficult to generalize to linguistic variations or larger knowledge sources. Another example is the work by Zelle and Mooney (1996), which learns to parse questions about geography into database queries. Again, the system targets highly compositional questions, but the database used is small and fixed.

Following Zelle and Mooney (1996), the main focus of semantic parsing work in the next decade was on incorporating statistical models into the parser. By training the parser to predict the annotated logical forms in the training data (Kate and Mooney, 2007; Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2011), the parser becomes more robust to lexical variations. However, the knowledge sources these work consider are still limited to small, fixed, and closed-domain databases. In fact, one capability these models are expected to learn is the association between the finite number of objects in the knowledge source and input natural

language phrases.

2.4.3 Learning from denotations

Training a statistical model for semantic parsing requires some form of supervision. The most straightforward supervision is to annotate each question x with a logical form z, and then train the model to prefer the annotated z over other logical forms. However, while logical form is a very strong signal, it often requires expertise and time to annotate, making it difficult to scale up to larger datasets or larger knowledge sources.

Instead, previous work has considered using the denotation y (e.g., the correct answer to the question x) as the form of supervision (Clarke et al., 2010; Liang et al., 2011). The model is now trained to prefer any logical form z that executes to the annotated denotation y. Denotations are easier to annotate by non-experts, making it cheaper to scale up to larger-scale data. Moreover, denotations are not tied to any formalism of semantic representations, making the dataset easier to use in different training paradigms with different semantic representations. The tasks in this dissertation will use denotations as supervision.

2.4.4 Scaling up to large knowledge bases

To expand the scope of the knowledge source, later semantic parsing works consider using large knowledge bases as the source of information (Cai and Yates, 2013; Berant et al., 2013). These knowledge bases such as Freebase (Google, 2013) contain millions of entities and relations across multiple domains. In order to handle the increased BREADTH of the knowledge source, previous work employs various techniques such as paraphrase models (Berant and Liang, 2014) and distributional representations (Bordes et al., 2015) to link natural language to the corresponding objects in the knowledge base. Unfortunately, as the trend shifts toward handling BREADTH, the questions these work consider are often much less compositional. For instance, the commonly studied WEBQUESTIONS dataset (Berant et al., 2013) and the subsequent SIMPLEQUESTIONS dataset (Bordes et al., 2015) of factoid questions on Freebase, most questions can be answered by identifying the correct entity (e.g., *Barack Obama*) and relation (e.g., *place of birth*) (Yao et al., 2014).

Another important point to note is that while knowledge bases contain information from many domains, the included information is inherently limited by the rate in which the data gets populated, and by the data schema set up by the knowledge base maintainer. This makes knowledge bases not fully open-domain. Research on knowledge base population (Ji and Grishman, 2011; Ellis et al., 2015; Chaganty et al., 2017) and attribute discovery (Cafarella et al., 2008; Yakout et al., 2012) tries to address these issues, but it would be more preferable if a semantic parsing system can process open-domain data from the source directly instead of waiting for the knowledge base to be populated.

2.4.5 Semantic parsing on open-domain tables

Our work on complex question answering on web tables aims to increase the scope of the domain by using web tables as the knowledge source, while at the same time bring back the compositionality in the input

questions. Even though the table is much smaller than a knowledge base, different questions are asked on different tables with possibly unseen schemata, making it more open-domain than a fixed knowledge base. As for task complexity, while the questions in our dataset are not as compositional as the extreme examples from earlier works, they still require non-trivial numbers of steps and a diverse type of operations (e.g., aggregation, comparison, calculation) to derive the answers.

After our work, there have been a few other tasks and datasets related to question answering on open-domain tables. Notable ones include the WIKISQL dataset (Zhong et al., 2017), which focuses on identifying table columns and values in the `SELECT` and `WHERE` clauses of SQL queries; and the SPIDER dataset (Yu et al., 2018a,b), which considers very complex SQL queries add incorporates multiple tables with `JOIN` clauses. We will compare our dataset with these related datasets in Chapter 4.

2.4.6 Reasoning without discrete logical forms

Discrete representation, such as logical forms, is not the only way for computing the answer to a complex question. With the recent development of using neural network as a universal computation device (Graves et al., 2014; Weston et al., 2015; Reed and de Freitas, 2016), previous work has considered representing the intermediate result of multi-step computation as a continuous state vector (Neelakantan et al., 2016; Yin et al., 2016; Mou et al., 2017; Iyyer et al., 2017). Under this paradigm, a neural model takes in the question and the knowledge source, and then updates the state vector for several time steps. The final state vector is then used to produce the final answer. Chapter 8 describes the details of previous works that utilize continuous representations to tackle the main dataset of this dissertation.

One benefit of using continuous representations is the ability to train the model in an end-to-end fashion. However, compared to the semantic parsing approaches that use discrete logical forms, the continuous representation is less interpretable, making it difficult to diagnose mistakes. Moreover, learning to perform complicated operations (e.g., subtracting the start date from end date for each event) requires a large amount of training data and a model expressive enough to encode the operations.

2.5 Information extraction

The main goal of this dissertation is to create a system for understanding complex questions (high DEPTH) that works directly on open-domain data (high BREADTH). Instead of this challenging paradigm, one alternative solution to creating a system in a high-BREADTH high-DEPTH regime is a pipeline approach: apply an *information extraction* (IE) system to convert the source unstructured or semi-structured data into a structured knowledge source (e.g., a database), and then have a knowledge-based QA system use the populated structured data to answer complex questions. In this section, we give an overview of previous work on information extraction, and then explain the limitations of this pipeline approach.

2.5.1 Information extraction on unstructured text

As most databases and knowledge bases represent relations between objects, one common information extraction task is *relation extraction*: extract binary relations between two entities mentioned in the text (e.g., from the snippet "*Obama was born on August 4 . . . in Honolulu, Hawaii*", extract the relation *place of birth* between *Barack Obama* and *Honolulu, Hawaii*). Early relation extraction works find snippets that express the desired relations with explicit linguistic patterns, which can be manually written or bootstrapped (Hearst, 1992, 1998; Agichtein and Gravano, 2000). Later works learn classifiers to detect relations between entity mentions. To train the classifiers, previous works consider using texts annotated with relation as a direct supervision (Zeng et al., 2014; Miwa and Bansal, 2016) or known relations between entities as a distant supervision (Mintz et al., 2009; Riedel et al., 2010).

When the exact relations to extract is not given, *open information extraction* can be used instead (Banko et al., 2007; Fader et al., 2011; Etzioni et al., 2011; Mausam et al., 2012; Mitchell et al., 2015). These systems take unstructured text and extract binary relations between entities, where the relation is derived from the text string instead of a predefined list of relations.

Beyond binary relations between two entities, previous work has also considered extracting more complex structures such as events and temporal series of events (Riedel and McCallum, 2011; Li et al., 2013; Chen et al., 2015).

2.5.2 Information extraction on semi-structured data

The main line of works on information extraction from semi-structured data is *wrapper induction* (Kushmerick, 1997; Crescenzi et al., 2001; Dalvi et al., 2011). We give a brief overview of wrapper induction and defer the details to Chapter 3 when we discuss our proposed task of entity list extraction. A *wrapper* is any machinery (e.g., a pattern or a program) that extracts the information specified by the user. For instance, on a web page, selectors such as regular expressions and XPaths have been used as patterns for extracting the desired information. The process of inducing a wrapper is called wrapper induction, which is usually done based on a few given examples of the extraction targets. From the examples, a wrapper generator generates candidate wrappers that correctly extract the examples, and then a ranker ranks the wrapper to find the one that is likely to generalize to new source documents.

2.5.3 Advantages and disadvantages of the pipeline approach

We now examine the pipeline approach of extracting information from open-domain data and then applying knowledge-based QA on the populated database. With the clear separation of the two pipeline steps, it is easier to develop the information extractor and semantic parser independently. The resulting database is highly interpretable and can be directly used outside the context of semantic parsing. Moreover, information from multiple data sources can be *aggregated* to form a unified consensus of the data. This increases the factual correctness when answering questions (e.g., for "*Where was Barack Obama born?*", some documents

give a wrong answer), and increases the coverage when the answer is an open-ended list (e.g., "*What are some important buildings in New York?*").

However, the pipeline approach also has several limitations. First, information extraction can produce errors that are propagated to the database ("precision" problem). This can sometimes be mitigated by leveraging redundancy and verifying the extracted facts across multiple sources.

Second, data extraction is a lossy process and could discard a large amount of information from the source documents ("recall" problem). For example, information that has to be inferred from multiple sentences are generally difficult to detect. This limitation can be quite severe even when extracting information in non-specialized domains. In knowledge base population challenges (Ji and Grishman, 2011; Ellis et al., 2015), in which specified properties about the specified entities should be extracted from text documents, the recall of even the best system is usually much lower than the human's performance.

Finally, the types of extracted data is limited by the schema of the database. Handling questions outside the database schema requires solutions such as knowledge inference based on other existing information (Nickel et al., 2011; Riedel et al., 2013; Neelakantan et al., 2015) or rerunning the information extractor to extract the required information.

2.5.4 On-the-fly information extraction

Semi-structured data, such as web pages and web tables, contains an arbitrary schema depending on the whim of its authors. As such, in order to retain as much knowledge as possible from a semi-structured knowledge source, we should not force the extracted data to conform to a predefined schema. Instead of populating a central database with a predefined schema, an alternative approach we take is *on-the-fly information extraction*: upon retrieving the relevant snippets from the knowledge source (e.g., a web table), we apply IE to create a temporary structured data just for that snippet. The schema of the produced structured data depends on the source data, which allows it to represent any types of relation present in the source data. However, this increased flexibility comes with a trade off: since the schema is not predefined, it is more difficult to map the phrase in the question to the objects and relations in the produced structured data.

On-the-fly information extraction has been explored by some previous QA systems. For instance, the START system by Katz et al. (2002) first translates the user query into a logical form (e.g., "*Who directed gone with the wind?*" is mapped to `(get "imdb-movie" "Gone with the Wind (1939)" "DIRECTOR")`). The execution of the logical form involves fetching the web page specified by the logical form (e.g., IMDB page for the movie *Gone with the Wind*), and then using a wrapper to extract the answer (e.g., use an HTML wrapper to read off the director's name from the information table). Another example is the Octopus system (Cafarella et al., 2009), which facilitate data integration tasks. From the given query (e.g., "*VLDB program committee*"), Octopus first retrieves and ranks a list of web tables matching the query. The user can then use data integration operators to aggregate the retrieve information and construct a unified structured table.

Chapter 3

Semi-Structured Data on Web Pages

In this chapter, we investigate the nature of *semi-structured data* on web pages, as well as the techniques for handling them in a question answering system. As explained in Section 2.1.3, a semi-structured knowledge source contains some form of structures, but the schemata of such structures are not defined in advance. For instance, Figure 3.1 shows a web page where hiking trails and their descriptions are listed using a uniform structure. However, there is no universal way for presenting a list of hiking trails. The webmaster gets to choose the pieces of information to be presented in each entry and how they should be laid out on the web page. As such, a different website might use a different data schema, presenting different pieces of information in a different format.

To aid our investigation, we consider the task of extracting a specified list of entities from the semi-structured data on the web page. Concretely, given a natural language query describing an entity category and a relevant web page, the task is to extract entities that belong to the category from the page. For example, from the web page in Figure 3.1 the question "*(What are) hiking trails near Baltimore*" should be mapped to the the list {"*Avalon Super Loop*", "*Patapsco Valley State Park*", ...}. We particularly focus on the semi-structured part of the web page: the entity strings are expected to be extracted from the HTML elements with a coherent structure (e.g., cells from the same column of a table, or the header of each page section).

We first formalize the task and give a few examples from the OPENWEB dataset we collected for the task. Afterward, we introduce *extraction path*, an XPath-like latent selector that selects HTML elements that share the same structure. The selected elements are then scores using features that capture the homogeneity of structural and linguistic information in the elements. Finally, we present the experimental results of models that were trained to generate good extraction paths.

Reference. The results described in this section have been published as Pasupat and Liang (2014).

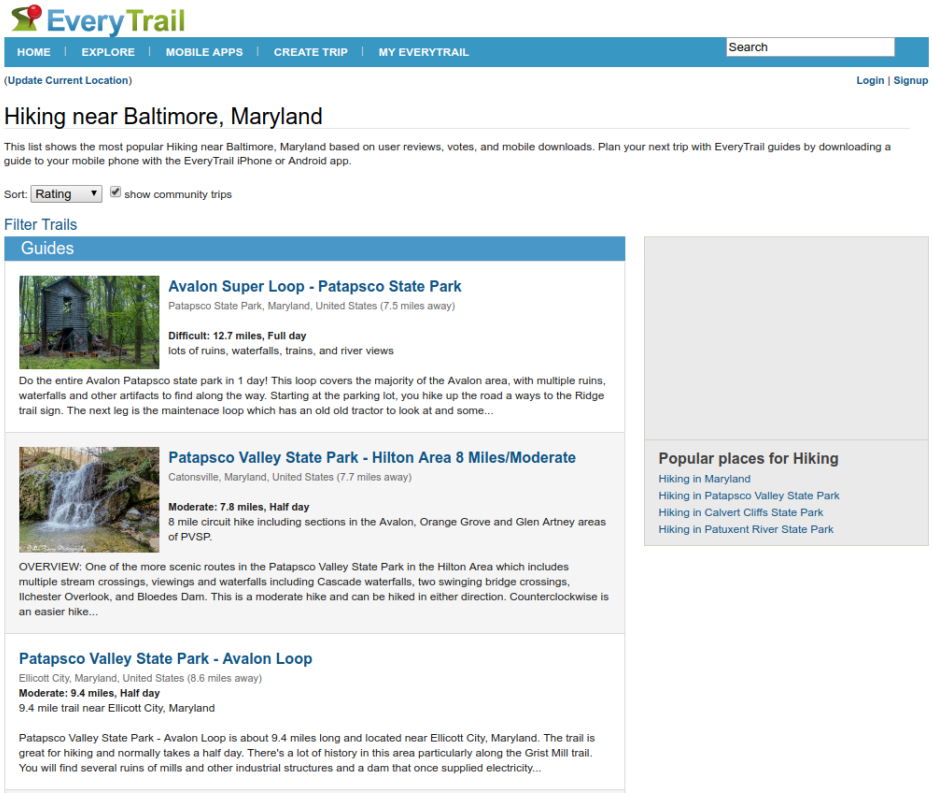


Figure 3.1: The list of hiking trails are presented in a semi-structured format on a web page. We consider the task of extracting the specified list of entities (e.g., "*hiking trails near Baltimore*") from the web page.

3.1 The entity list extraction task

Task. Given a web page w (e.g., Figure 3.1) and a natural language query x (e.g., "*hiking trails near Baltimore*"), the task is to extracts a list y of entities that satisfy the query x from the web page (e.g., $y =$ {"*Avalon Super Loop*", "*Patapsco Valley State Park*", ...}).

Dataset. We collected a new dataset, OPENWEB, with a diverse set of queries and web pages. The data collection process is as follows:

1. We used the method from Berant et al. (2013) to generate a diverse set of search queries. Specifically, we used the Google Suggest API, which takes a partial query (e.g., "*list of* ____ *movies*") and outputs complete queries (e.g., "*list of horror movies*").

   The queries were generated using breadth-first search over the query space. We started with the seed partial queries "*list of* •____" where • is one or two initial letters (e.g., "*list of pr*____"). In each iteration, we called the Google Suggest API to complete the queries, and then applied the transformation rules in Table 3.1 to generate more partial queries. We ran the procedure until we obtained 100K

Full query	**New partial queries**
list of X IN Y	list of X ____
where IN is a preposition	list of ____ X
(*list of* $[hotels]_X$ *in* $[Guam]_Y$)	list of X IN ____
	list of ____ IN Y
list of X CC Y	list of X ____
where CC is a conjunction	list of ____ X
(*list of* $[food]_X$ *and* $[drink]_Y$)	list of Y ____
	list of ____ Y
list of X w	list of w ____
(*list of* $[good\ 2012]_X$ $[movies]_w$)	list of ____ w
	list of X ____

Table 3.1: Rules for generating new partial queries from complete queries. (X and Y are sequences of words; w is a single word.)

queries.

2. For each query, we downloaded the top 2–3 Google search results, sanitized the web pages, and randomly submitted 8000 web pages along with their associated queries to Amazon Mechanical Turk. Each worker must either mark the web page as irrelevant or identify the first, second, and last entities (as strings) according to the query. The dataset only contains examples where at least two workers agreed on the three entities.

The first, second, and last entities of the list are used as a surrogate for the full list, which is tedious to annotate. We say that a list y is *consistent* with the annotation if its first, second, and last entries match the annotation.

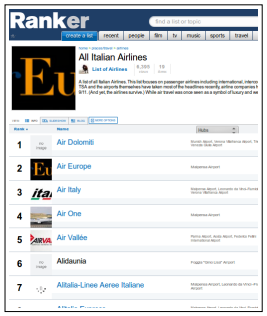
Statistics. The OPENWEB dataset contains 2,773 examples from 2,269 distinct queries. The queries span a variety of topics from the more common ones (e.g., movies, companies, characters) to more obscure ones (e.g., enzymes, proverbs, headgears). The web pages are also diverse: they come from 1,438 different web domains, of which 83% appears only once in the dataset.

Figure 3.2 shows several queries and web pages from the dataset. Besides the wide range of queries, another main challenge of the dataset comes from the diverse data representation formats such as tables, grids, lists, headings, and paragraphs.

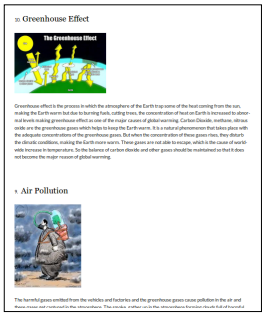
Queries

airlines of italy
natural causes of global warming
lsu football coaches
bf3 submachine guns
badminton tournaments
foods high in dha
technical colleges in south carolina
songs on glee season 5
singers who use auto tune
san francisco radio stations
actors from boston

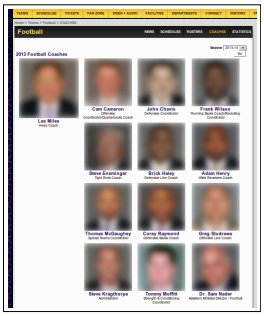
Examples (web page, query)



airlines of italy



natural causes of global warming



lsu football coaches

Figure 3.2: Some examples illustrating the diversity of queries and web pages from the OPENWEB dataset.

3.2 Representation of the web page

The set of web pages across web domains form a semi-structured knowledge source: different web pages encode similar information with different data schemata. This schema mismatch sometimes manifests even within the same web domain. For example, while Wikipedia has guidelines on how certain categories of articles should be formatted, the data schemata still diverge drastically between articles.

To handle arbitrary data schemata, we propose to use a generic domain-independent structure to represent web pages. In particular, we use the Document Object Model (DOM) to represent the given web page w as a tree of objects. Unlike the full HTML DOM standard which includes multiple types of nodes (e.g., text nodes and comment nodes), we use a simplified representation where all tree node are element nodes, as shown in Figure 3.3. For each DOM tree node, we consider its HTML properties (tag name and HTML attributes) as well as its text content. (The text content of non-leaf nodes is the concatenation of all text content within the subtree.)

3.3 Extraction paths

We use latent *extraction paths* z to model the process of extracting entities from the semi-structured web page w. An extraction path is a simplified XML path (XPath) such as

$$z = \texttt{/html/body/table[2]/tr/td[1]} \tag{3.1}$$

Formally, an extraction path is a sequence of *path entries*, where each path entry is either a tag name (e.g., `tr`) or a tag name with an index (e.g., `td[1]`). An extraction path z can be executed on the web page w to get $[\![z]\!]_w$, a list of matching HTML elements, as follows:

- The base extraction path $z = \texttt{/html}$ executes to a list containing only the root `<html>` element of w.

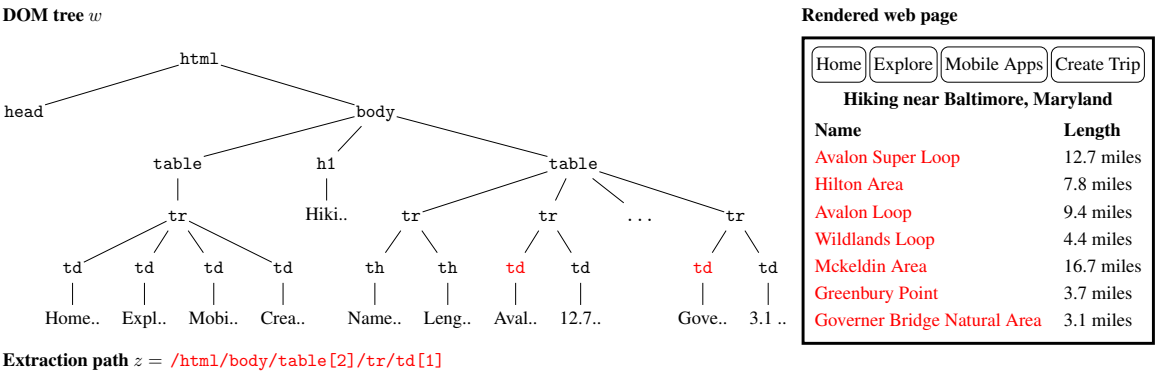


Figure 3.3: A simplified example of a DOM tree of the web page w and an extraction path z, which selects a list of entity strings $y = [\![z]\!]_w$ from the web page (highlighted in red).

- To execute z/t (where z is an extraction path and t is a tag): for each element $e \in [\![z]\!]_w$, select all children with tag t of e. Compile the selected elements into a list.
- To execute $z/t[i]$ (where z is an extraction path, t is a tag, and i is an index): for each element $e \in [\![z]\!]_w$, consider all children with tag t of e, and then select the ith one (1-indexed) if there are at least i such children. Compile the selected elements into a list.

For example, as illustrated in Figure 3.3, the extraction path $z =$ `/html/body/table[2]/tr/td[1]` executes to the first cell in each row of the second table on the page. Note that all lists of elements are sorted by the order they appear in the pre-order traversal of the DOM tree.

From the list $[\![z]\!]_w$ of elements, we can extract a list y of entity strings by compiling the text contents of all elements in $[\![z]\!]_w$ that are shorter than 140 characters. This filtering is employed to control the search space. This means an extraction path might yield an empty list of entities (e.g., `/html/body/div[1]/p` gives no entities if all the selected `<p>` elements contain more than 140 characters).

Generating extraction paths. From a given web page w, we can generate a set $\mathcal{Z}(w)$ of candidate extraction paths as follows. For each element in the DOM tree, we find an extraction path that selects only that element (using only the indexed path entries), and then generalizes the path by removing any subset of indices of the last k path entries. For instance, when $k = 2$, an extraction path ending in `.../tr[5]/td[2]` will be generalized to `.../tr[5]/td[2]`, `.../tr/td[2]`, `.../tr[5]/td`, and `.../tr/td`. This generalization step allows us to select multiple elements of the same structure (e.g., table cells from the same column, or list items from the same section). We keep any generalized extraction path where the extracted list of entities contain at least two entities. Note that several extraction paths might produce the same list of entities.

We use $k = 8$ in our experiments, which gives at most 2^8 generalized extraction paths for each element. Among the development examples of the OPENWEB dataset, we generate on average approximately 8500 extraction paths, which evaluate to approximately 1200 unique entity lists.

3.4 Approach

Given a query x and a web page w, we generate a set $\mathcal{Z}(w)$ of candidate extraction paths z as described in Section 3.3. We then choose $z \in \mathcal{Z}$ with the highest model probability $p_\theta(z \mid x, w)$ and extract the list of entities from z.

Model. From the query x and the web page w, we define a log-linear distribution over the extraction paths in $\mathcal{Z}(w)$ as

$$p_\theta(z \mid x, w) \propto \exp\left[\theta^\top \phi(x, w, z)\right], \tag{3.2}$$

where θ is the parameter vector and $\phi(x, w, z)$ is the feature vector.

We train the model by maximizing the log-likelihood of the extraction paths that produce entity lists consistent with the annotation (first, second, and last entities). From training examples $(x^{(i)}, w^{(i)}, c^{(i)})$, where the indicator function $c^{(i)}(z)$ is 1 if z is consistent with the annotation and 0 otherwise, we seek θ that maximizes

$$J(\theta) := \sum_i \log p_\theta(c^{(i)} = 1 \mid x^{(i)}, w^{(i)}) - \Omega(\theta) \tag{3.3}$$

where

$$p_\theta(c = 1 \mid x, w) = \sum_{z \in \mathcal{Z}(w)} p_\theta(z \mid x, w)\, c(z) \tag{3.4}$$

and $\Omega(z) = \frac{\lambda}{2} \|\theta\|_2^2$ (with $\lambda = 0.01$) is a regularization term. We optimize θ with AdaGrad (Duchi et al., 2010), running for 5 epochs over the training data.

3.5 Feature extraction

The selected HTML elements from the web page w contains both structured information (e.g., their locations in the DOM tree) and unstructured information (e.g., their text contents). As such, we define the feature vector $\phi(x, w, z)$ as a concatenation of two types of features: *structural features* $\phi_{\mathrm{s}}(w, z)$ which capture structured information of the HTML elements, and *denotation features* $\phi_{\mathrm{d}}(x, y)$ which capture the linguistic information of the extracted entity list y.

Recipe for defining features on lists. Many of our features are defined on a list of objects (e.g., a list of HTML elements or a list of entity strings). When defining such features, we want to incorporate the information about individual list members, as well as how the list looks like as a whole (e.g., whether the members are diverse in some property). As illustrated in Figure 3.4, we propose the following two-step recipe for generating features from a list:

1. **Abstraction.** We map each list member to an abstract value by extracting one of its properties. For instance, we can map each each HTML element to its number of children, or map each entity string to

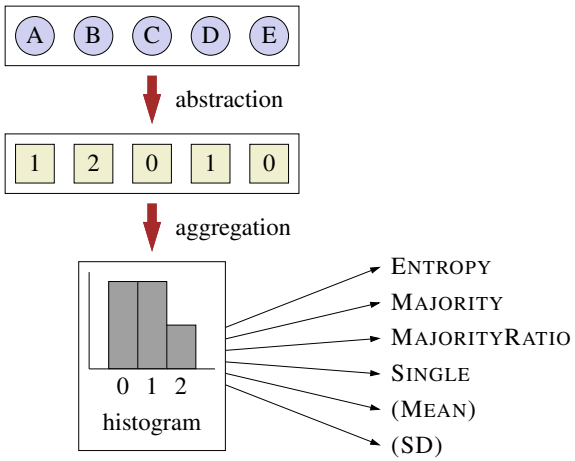


Figure 3.4: The recipe for defining features on a list of objects: (i) the *abstraction* step converts list elements into abstract values; (ii) the *aggregation* step defines features using the histogram of the abstract values.

its part-of-speech tag sequence.

2. **Aggregation.** We create a histogram of the abstract values, and then define features based on the following statistics of the histogram:

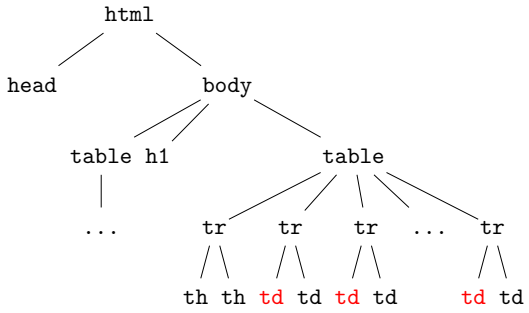
   - ENTROPY: entropy normalized to the maximum value of 1. The ENTROPY is 0 if all abstract values are identical, and is 1 if all abstract values are distinct.
   - MAJORITY: the most frequent abstract value.
   - MAJORITYRATIO: percentage of abstract values sharing the MAJORITY value.
   - SINGLE: whether all abstract values are identical.

   For abstract values with finitely many possible values (e.g., part-of-speech tags), we also use the normalized count of each value as a feature. And for numeric abstract values, we also use the mean (MEAN) and standard deviation (SD). In our implementation, real-valued features are converted to indicator features by binning.

We use the recipe above to define structural features and denotation features. Figure 3.5 demonstrates some features we can extract from the example in Figure 3.3.

Structural features. Although different web pages represent data in different formats, they still share some common hierarchical structures in the DOM tree. We use structural features $\phi_{\text{s}}(w, z)$ to capture the structural information from the extraction path z and the selected elements $[\![z]\!]_w$:

- **Features on the list of selected elements.** We apply our recipe on the list of elements in $[\![z]\!]_w$ using the following abstract values of each element:



Selected entities
Avalon Super Loop
Hilton Area
Avalon Loop
Wildlands Loop
Mckeldin Area
Greenbury Point
Governer Bridge Natural Area

Structural feature	Value
Features on selected nodes:	
TAG-MAJORITY = td	1
INDEX-ENTROPY	0.0
Features on parent nodes:	
CHILDRENCOUNT-MAJORITY = 2	1
PARENT-SINGLE	1
INDEX-ENTROPY	1.0
HEADHOLE (The first node is skipped)	1
Features on grandparent nodes:	
PAGECOVERAGE	0.6
...	...

Denotation feature	Value
WORDSCOUNT-MEAN	2.42
PHRASESHAPE-MAJORITY = Aa Aa	1
PHRASESHAPE-MAJORITYRATIO	0.71
WORDSHAPE-MAJORITY = Aa	1
PHRASEPOS-MAJORITY = NNP NN	1
LASTWORD-ENTROPY	0.74
WORDPOS = NN (normalized count)	0.53
...	...

Figure 3.5: A small subset of features from the example *hiking trails near Baltimore* in Figure 3.3.

  - The HTML TAG and attributes (ID, CLASS, HREF, etc.)
  - CHILDRENCOUNT: the number of children of the element in the DOM tree.
  - SIBLINGCOUNT: the number of siblings in the DOM tree.
  - INDEX: the position among its siblings.
  - PARENT: the parent element (e.g., the resulting PARENT-SINGLE feature indicates that all elements share the same parent).

- **Hole features.** If the extraction path does not select all elements in the same DOM tree level, it is likely that the selection is incorrect. For instance, an extraction path ending in `.../ul[1]/li/a` selects links inside an unlabeled list. However, if some list entries do not have links for some reason, then the selected list will be incomplete. In this case, it is preferred to use `.../ul[1]/li` to select the list entries themselves instead.

  To capture this intuition, we define a NOHOLE feature if all parents of the selected elements has at least one of the selected elements as a child. For the example above, if some list entries `.../ul[1]/li`

do not have links, then the extraction path .../ul[1]/li/a will not generate a NOHOLE feature. To account for the common case of selecting table cells (.../tr/td[i]), where the first row <tr> usually contains <th> instead of <td>, we define a HEADHOLE feature if only the first parent is violating our criterion.

- **Coverage feature.** We define a real-valued feature PAGECOVERAGE as the number of HTML elements that are descendents of the selected elements, divided by the total number of elements on the web page.
- **Features on ancestor elements.** We also define the same feature set on the list of ancestors of the selected elements. In our experiments, we traverse up to 5 levels of ancestors and define separate structural features on each level.

Denotation features. Structural features are not powerful enough to distinguish entity lists with the same structure (e.g., different columns of the table, or different lists on the same web page). To resolve this issue, we define denotation features $\phi_{\mathrm{d}}(x, y)$ on the entity list y extracted from the selected elements:

- **Linguistic features.** We observe that the correct entities often share some linguistic features. For example, people and place names usually contain 2–3 word tokens with part-of-speech tag NNP (proper noun). In contrast, incorrect lists of elements (e.g., various elements across the page) tend to have texts of diverse lengths and linguistic tags. With this reasoning, we apply our recipe on the list of entity strings using the following abstract values of each string:
  - WORDCOUNT: number of tokens.
  - PHRASESHAPE: abstract shape of the string (e.g., "*Stanford University*" becomes "Aa Aa").
  - WORDSHAPE: abstract shape of each word. The features are defined on the list of words instead of the list of entity strings.
  - FIRSTWORD and LASTWORD
  - PHRASEPOS: the part-of-speech sequence of the whole string.
  - WORDPOS: part-of-speech of each word. Like WORDSHAPE, the features are defined on the list of words instead.
- **Query-denotation features.** Different query types denote entities with different linguistic properties. For instance, queries "*mayors of Chicago*" and "*universities in Chicago*" will produce entities of different lengths, path-of-speech sequences, and word distributions. This suggests incorporating features that depend on the query x.

  To this end, we define features of the form $(f(y), g(x))$, where $f(y)$ is a linguistic feature as described above, and $g(x)$ is the category of the query x. For our experiments, we manually classify the queries into 7 categories: person, media title, location/organization, abstract entity, word/phrase, object name, and miscellaneous. In an actual system, the queries should be classified automatically instead, but we leave this as future work.

Path suffix pattern	Count
{a, table, tbody, td[*], tr}	1792
{a, tbody, td[*], text, tr}	1591
{a, table[*], tbody, td[*], tr}	1325
{div, table, tbody, td[*], tr}	1259
{b, div, div, div, div[*]}	1156
{div[*], table, tbody, td[*], tr}	1059
{div, table[*], tbody, td[*], tr}	844
{table, tbody, td[*], text, tr}	828
{div[*], table[*], tbody, td[*], tr}	793
{a, table, tbody, td, tr}	743

Table 3.2: Top 10 path suffix patterns found by the baseline. We blank out the index numbers and disregard the order of path entries, making them a multiset instead of a sequence.

As the query-denotation features require additional manual annotation, we exclude them from the main experiments. Instead, we will investigate the effect of query-denotation features in our model analysis (Section 3.7).

3.6 Experiments

We evaluate our model on the OPENWEB dataset. The main evaluation metric is *accuracy*: the fraction of examples where the model predicts an entity list consistent with the annotation. We also report *accuracy at 5*, the fraction of examples where one of the five highest-scoring extraction path yields a consistent entity list. Finally, we report the *oracle* score, the fraction of examples where the model can find at least one consistent entity list among all extraction paths.

To measure how the consistency-based accuracy tracks the actual correctness, we sampled 100 web pages which have at least one valid extraction path and manually annotate them with the full entity lists. We found that in 85% of the examples, the longest consistent list y is the correct list, and many lists in the remaining 15% miss the correct list by only a few entities.

Baseline. As a baseline, we list all 5-entry suffixes of the correct extraction paths in the training data, and then sort the suffixes by frequency. Table 3.2 shows the most frequent suffixes. To increase generalization, we blank out the index numbers and discard the order of path entries. At test time, we choose an extraction path with the most frequent suffix pattern. The baseline should work reasonably well if the web pages are relatively homogeneous.

Main results. We split the dataset into 70% training and 30% test data. In addition to testing on test data, we also report the results on 10 random 80-20 splits of the training data.

	10 random splits		Test data	
	Acc	**Acc@5**	**Acc**	**Acc@5**
Baseline	10.8 ± 1.3	25.6 ± 2.0	10.3	20.9
Our approach	41.1 ± 3.4	58.4 ± 2.7	40.5	55.8
Oracle	68.7 ± 2.4	68.7 ± 2.4	66.6	66.6

Table 3.3: Main results on the OPENWEB dataset using the default set of features. (Acc = accuracy, Acc@5 = accuracy at 5)

As reported in Table 3.3, our approach gets a test accuracy of 40.5%. The oracle accuracy of 66.6% reveals that there are two categories of errors: (i) *coverage errors* (33.4%), which are when the system cannot find any consistent entity list, and (ii) *ranking errors* (26.1%), which are when a consistent list of entities exists but does not have the highest probability.

3.7 Analysis

Tables 3.4 and 3.5 show the breakdown of the coverage and ranking errors on one of the development splits.

Analysis of coverage errors. From Table 3.4, about 36% of coverage errors happen when the extraction path for the correct entities also captures irrelevant parts of the web page (Reason C1). For example, many Wikipedia articles contain the *See Also* section that lists related articles in an unordered list (`.../ul/li/a`). This is problematic when the entities to be selected are also represented in the same way.

Another main source of errors is the inconsistency in HTML tag usage (Reason C2). For instance, some web pages use `<b>` and `<strong>` interchangeably for bold texts, or switch between `<b><a>...</a></b>` and `<a><b>...</b></a>` across entities. Normalizing the web page, using alternative web page representations, or using a more expressive representation than extraction paths could help reduce this type of errors.

Analysis of ranking errors. From Table 3.5, a large number of errors are attributed to selecting non-content elements such as navigation links or content headings (Reason R1). It turns out that structural and linguistic statistics of these elements can sometimes be more coherent than those of the correct entities. Since our features try to capture the homogeneity of the selected elements, the model can end up selecting these irrelevant elements. A possible solution is to detect the saliency of the element and favor the elements that are in the content part of the page.

Feature ablation. Table 3.6 shows the ablation results on 10 development splits. We observe while the structural features are responsible for most of the accuracy, denotation features is also helpful. By examining the actual predictions, we found that if there are multiple parallel lists on the web page (e.g., different columns

	Reason	Example	Count
C1	Answers and non-answer elements are selected by the same extraction path.	Select entries in the "*See Also*" section in addition to the content because they are all `<li>`.	48
C2	HTML tag usage is inconsistent.	The page uses both `<b>` and `<strong>` for headers interchangeably.	16
C3	The query applies to only some sections of the matching entities.	Need to select only companies in China from the table of all Asian companies.	20
C4	Answers are embedded in running text.	Answers are in a comma-separated list.	13
C5	Text normalization issues.	Selected "*Silent Night Lyrics*" instead of "*Silent Night*".	19
C6	Other issues.	Incorrect annotation / Entities are permuted when the web page is rendered	18
	Total		134

Table 3.4: Breakdown of coverage errors from the development data.

	Reason	Example	Count
R1	Select non-content strings.	Select navigation links, headers, footers, or sidebars.	25
R2	Select entities from a wrong field.	Select book authors instead of book names.	22
R3	Select entities from wrong section(s).	For the query "*schools in Texas*", select all schools on the page, or select the schools in Alabama instead.	19
R4	Also select headers or footers.	Select the table header in addition to the answers.	7
R5	Select only entities with a particular formatting.	From a list of answers, select only entities that are links (`<a>`).	4
R6	Select headings instead of the contents or vice versa.	Select the categories of rums in `<h2>` instead of the rum names in the tables.	2
R7	Other issues.	Incorrect annotation / Multiple sets of answers appear on the same page / etc.	9
	Total		88

Table 3.5: Breakdown of ranking errors from the development data.

Setting	Acc	Acc@5
All features	41.1 ± 3.4	58.4 ± 2.7
Oracle	68.7 ± 2.4	68.7 ± 2.4
Structural features only	36.2 ± 1.9	54.5 ± 2.5
Denotation features only	19.8 ± 2.5	41.7 ± 2.7
Query-denotation features only	25.0 ± 2.3	48.0 ± 2.7
Structural + query-denotation	41.7 ± 2.5	58.1 ± 2.4
Concatenate a random web page + structural + denotation	19.3 ± 2.6	41.2 ± 2.3
Concatenate a random web page + structural + query-denotation	29.2 ± 1.7	49.2 ± 2.2

Table 3.6: System accuracy with different feature and input settings on the development data. (Acc = accuracy, Acc@5 = accuracy at 5)

of a tables), denotation features help the system select the correct list. In contrast, structural features act as a prior that prevents the system from selecting random entities outside the main part of the web page.

Query-denotation features. The model so far does not include query-denotation features, meaning that the input query x is ignored entirely. Despite this, since the web page was obtained from a search engine, the most prominent entities on the web pages are likely to satisfy the query already, and thus we were able to get some reasonable accuracy.

Table 3.6 shows that using only query-denotation features (no structural features) increases the accuracy over using normal denotation features by 5.2%. When combined with structural features, the query-denotation features help the model select entities that are more appropriate for the query, as shown in Table 3.7. However, it does not increase the overall accuracy. This is perhaps because the web page is already related to the query as argued earlier.

To test query-denotation features in scenarios where the most prominent elements might not give correct entities, we conduct experiments on a modified dataset where in each example, the original web page and an unrelated web page are concatenated (in a random order). We find that query-denotation features (with structural features) significantly improve the accuracy over using the normal denotation features from 19.3% to 29.2%.

3.8 Related work and discussion

Web page understanding. To understand the overall structure of web pages, work on web page segmentation aims to segment the page into semantically meaningful blocks (Bing et al., 2014; Kreuzer et al., 2015), identify salient parts of the web page (Gamon et al., 2013; Shen and Zhao, 2014), align the structures with

Query	Default features	+ query-denotation
euclid's elements book titles (category = *media title*)	"*Prematter*" "*Book 1*" "*Book 2*" ...	"*Book I. The fundamentals ...*" "*Book II. Geometric algebra.*" "*Book III. Theory of circles.*" ...
soft drugs (category = *object name*)	"*Hard drugs*" "*Soft drugs*" "*Some drugs cannot be classi...*" ...	"*methamphetamine*" "*psilocybin*" "*caffeine*" ...
professional athletes with concussions (category = *person*)	"*Pistons-Knicks Game Becomes Site of Incredible Dance Battle*" "*Toronto Mayor Rob Ford ...*" ...	"*Mike Richter*" "*Stu Grimson*" "*Geoff Courtnall*" ...

Table 3.7: The predictions after changing denotation features into query-denotation features.

the same semantics between web pages (Kumar et al., 2011), or detect the semantics of web page regions (Spengler and Gallinari, 2010; Kumar et al., 2013).

Extracting information from web pages. Our presented approach is closely related to the work on wrapper induction (Kushmerick, 1997). A wrapper is a function that takes the web page as input and return the extracted information. Previous work on wrapper induction mostly focuses on a small set of web domains, and assume that the web pages follow a fixed template (Muslea et al., 2001; Crescenzi et al., 2001; Cohen et al., 2002; Arasu and Garcia-Molina, 2003; Dalvi et al., 2011). Later work tries to generalize across web pages, but relies on restricted data format (Wong et al., 2009) or prior knowledge of the web page structure for the desired data to extract (Zhang et al., 2013).

Given a web page generated from a template (e.g., a product listing page on an online shop), previous work has also considered using the regularity of web page elements to extract structured information. For instance, Grigalis and Cenys (2014) extracts data records by looking at contiguous elements that are visually similar, and Omari et al. (2016) factors the raw data from the web template.

Our setting differs from previous work on wrapper induction in two major ways. First, most previous works on wrapper induction operate on a fixed website. The induced wrapper only match the data schema of that website, and is not likely to work on a different websites or even a different version of the same website. Similar to some previous work that generalize across pages mentioned above (Wong et al., 2009; Zhang et al., 2013), our approach does not make an assumption about the schema of the web page. Second, in most wrapper induction systems, the user specifies a few examples of entities to be extracted, and the system tries to find wrappers that can extract such entities. Our setting has natural language as the specification of what entities to extract, which requires less effort from the user. Note that while the query-dependent features has

minor impact on the overall extraction accuracy, the features do help in scenarios where the most prominent HTML elements on the page do not produce the specified entities.

Collating information from web pages. Instead of extracting only the specified type of information, one might want to mine and process all possible information from a collection of web pages. This can reveal corpus-level statistics that cannot be captured by processing individual web pages. For instance, WebTables (Cafarella et al., 2008) compiles a large corpus of web tables, resulting in a corpus that can be used for many downstream tasks such as deriving the correlation between data schemata (column headers). Other works consider methods for processing the mined data, such as recovering the missing column headers (Venetis et al., 2011) or converting raw data to canonical formats (Sarawagi and Chakrabarti, 2014; Embley et al., 2016).

One way to view these families of information extraction methods is to compare them with relation extraction from unstructured text (Section 2.5). Wrapper induction and other query-based extraction methods is parallel to relation extraction with a predefined set of relations (Hearst, 1992; Mintz et al., 2009; Miwa and Bansal, 2016). In contrast, WebTables and other works that collate information from the structured parts of web pages is parallel to open information extraction (Banko et al., 2007; Fader et al., 2011; Etzioni et al., 2011; Mausam et al., 2012).

Interacting with web pages. Previous work has considered a variety of interfaces for humans to interact with web pages. Some examples include keystrokes (Spalteholz et al., 2008), speech (Ashok et al., 2014), haptics (Yu et al., 2005), and eye gaze (Kumar et al., 2007).

In contrast to the methods above, automated web scripts can be used for web page interaction with minimal human intervention. Most script frameworks refer to elements with logical selectors such as CSS matchers and XPaths. Due to the brittle nature of these selectors, the script can break when the web page changes its layout, or when the attributes of HTML elements are automatically generated (either due to the web framework used or for obfuscation purposes) (Hammoudi et al., 2016).

Previous work has proposed alternative methods for referencing web elements in automated web scripts. A few examples include Sikuli (Yeh et al., 2009), which uses images of the elements, and TagUI (Soh, 2017), which uses natural language. These tools were designed for productivity, but also have robustness as a side benefit. For instance, even when an element changes its attributes or location, a visual-based selector will still be able to select it.

Instead of manually authoring the scripts, previous work has also considered learning an agent to perform actions on web page based on the given goal. The agent can be learned from demonstrations (Allen et al., 2007), reinforcement learning (Branavan et al., 2008, 2009) or both (Shi et al., 2017; Liu et al., 2018).

The need for compositionality. While doing error analysis, we observe multiple examples that require compositional understanding of the input query x to obtain the correct answers. A few of such examples are elaborated below:

Rank	Firm	Headquarters
1	Bridgewater Associates	Westport, CT
2	Man Group	London
3	J.P. Morgan Asset Management	New York
4	Brevan Howard Asset Management	London
5	Och-Ziff Capital Management Group	New York
6	Paulson & Co.	New York
7	BlackRock Advisors	New York

Figure 3.6: The query "*list of hedge funds in New York*" requires filtering the list of entities by the value in the Headquarters column.

Actresses > Madhuri Dixit > Movies (Filmography)

FULL FILMS LIST | RECENT | UPCOMING | IN PRODUCTION

Year	Film Name	Cast
2014	Gulaab Gang	Madhuri Dixit, Juhi Chawla, Priyanka Bose, Vinitha Menon, Divya Jagdale
2014	Dedh Ishqiya	Arshad Warsi, Naseeruddin Shah, Madhuri Dixit, Huma Qureshi, Vijay Raaz
2013	Yeh Jawaani Hai Deewani	Ranbir Kapoor, Deepika Padukone, Aditya Roy Kapur, Kunaal Roy Kapur, Kalki Koechlin, Madhuri Dixit
2007	Aaja Nachle	Madhuri Dixit, Konkona Sen Sharma, Akshaye Khanna, Irfan Khan, Kunal Kapoor
2002	Devdas	Shahrukh Khan , Aishwarya Rai, Madhuri Dixit, Jackie Shroff
2002	Hum Tumhare Hain Sanam	Madhuri Dixit, Shahrukh Khan, Salman Khan, Atul Agnihotri, Suman Ranganathan
2001	Lajja	Manisha Koirala, Madhuri Dixit, Mahima Choudhary, Rekha, Anil Kapoor
2001	Yeh Raaste Hain Pyaar Ke	Ajay Devgan, Madhuri Dixit, Preity Zinta, Shakti Kapoor, Raju Mawani
2000	Gaja Gamini	Madhuri Dixit, Inder Kumar, Naseeruddin Shah, Shahrukh Khan,Shilpa Shirodkar

Figure 3.7: The query "*list of Salman Khan and Madhuri Dixit movies*" requires filtering the list of entities by the value in the Cast column.

- Rhizome
  - *Curcuma longa* (turmeric)
  - *Panax ginseng* (ginseng)
  - *Arthropodium* spp. (rengarenga, vanilla lily, and others)
  - *Canna* spp. (canna)
  - *Cordyline fruticosa* (ti)
  - *Maranta arundinacea* (arrowroot)
  - *Nelumbo nucifera* (lotus root)
  - *Typha* spp. (cattail or bulrush)
  - *Zingiber officinale* (ginger, galangal)
- Tuber
  - *Apios americana* (hog potato or groundnut)
  - *Cyperus esculentus* (tigernut or chufa)
  - *Dioscorea* spp. (yams, ube)
  - *Dioscorea polystachya* (Chinese yam, white name or white ñame)
  - *Helianthus tuberosus* (Jerusalem artichoke or sunchoke)
  - *Hemerocallis* spp. (daylily)
  - *Lathyrus tuberosus* (earthnut pea)
  - *Oxalis tuberosa* (oca or New Zealand yam)
  - *Plectranthus edulis* and *P. esculentus* (kembili, dazo, and others)
  - *Solanum tuberosum* (potato)
  - *Stachys affinis* (Chinese artichoke or crosne)
  - *Tropaeolum tuberosum* (mashua or añu)
  - *Ullucus tuberosus* (ulluku)

Root-like stem [edit]

- *Zamia pumila* (Florida arrowroot)

Figure 3.8: The query "*list of plants that are tubers*" requires locating the correct portion of the list based on the list indentation.

- In Figure 3.6, the query "*list of hedge funds in New York*" requires filtering the list of entities from the Firm column by the values in the Headquarters column. No extraction paths we defined can perform such filtering.
- In Figure 3.7, the query "*list of Salman Khan and Madhuri Dixit movies*" requires finding movies that have *both* people in the Cast column.
- In Figure 3.8, the query "*list of plants that are tubers*" requires identifying the correct subsection to extract the entities from. This requires understanding that list indentation signifies membership to the category *Tuber*.

We also observed that data *tables* on web pages provide a rich structure suitable for compositional reasoning, as illustrated in some of the examples above. This observation serves as the inspiration for the task of answering complex questions on web tables, which we will cover in the subsequent chapters.

3.9 Conclusion

In this chapter, we have explored the nature of semi-structured data on web pages. Instead of assuming a fixed data schema, we use the DOM tree as a generic representation of the structures on web pages. The process of extracting HTML elements in the tree is represented as an extraction path, which is discrete and domain-independent. To rank the extraction paths, we use structural and denotation features to access the schematic information of the selected elements. For instance, TAG structural features help distinguish entities from non-entities (e.g., `<a>` and `<td>` elements are more likely to represent entities), while PHRASESHAPE denotation features capture the types of the selected entities (e.g., a phrase with the shape "`Aa Aa`" is likely to be a named entity).

Moving forward, we are going to apply similar techniques when we consider more complex questions in the subsequent chapters. The semi-structured knowledge source will be represented using a generic graph representation that can encode artibrary data schemata, similar to the DOM tree representation used in this chapter. The process of computing the answer will be represented as *logical forms*, which are discrete and domain-independent constructs like extraction paths, but can also encode multi-step reasoning. Features similar to structural and denotation features presented here will be used to rank the produced logical forms. However, unlike the finite number of extraction paths, it is rarely possible to enumerate all logical forms, and thus we will extensively study the methods to search over the space of logical forms.

Chapter 4

Compositional Question Answering on Web Tables

The previous chapter considers a task on open-domain semi-structured web pages (high BREADTH) that only requires a shallow understanding of the input natural language query (low DEPTH). As the examples at the end of the previous chapter show, compositional understanding of the query and multi-step reasoning are important for handling even seemingly simple questions.

To simultaneously increase the scope of the knowledge source (high BREADTH) and the complexity of the questions (high DEPTH), we consider a new task of answering complex questions based on a given semi-structured web table. As an illustration, Figure 4.1 shows several question-answer pairs and the associated table from the *Summer Olympic Games* article on Wikipedia.[1] The questions have a wide range of complexity and require various types of reasoning such as comparison (x_2), superlatives (x_3), aggregation (x_4), and arithmetic (x_5). Meanwhile, the table contains an arbitrary schema depending on the whim of the article's editors, so a system for solving the task needs to generalize to potentially unseen data schema.

This chapter discusses the nature of the task and compares it with related tasks and datasets. Section 4.1 formalizes the task, while Section 4.2 explains how we collect a dataset, WIKITABLEQUESTIONS, with open-domain web tables and complex questions. Finally, in Section 4.3, we analyze the BREADTH and DEPTH of the dataset with respect to other related datasets.

4.1 Task description

In our task, the system is given a context table w ("world") and a question x about the table. The system should output a list y of values that answers the question according to the table. The task does not assume that the table follows any particular format, and the output values can be arbitrary strings. Figure 4.1 shows

[1] The table was retrieved in 2015 and was heavily simplified for illustrative purposes.

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...	...	...	...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

x_1: *"Greece held its last Summer Olympics in which year?"*
y_1: {2004}
x_2: *"In which city's the first time with at least 20 nations?"*
y_2: {Paris}
x_3: *"Which years have the most participating countries?"*
y_3: {2008, 2012}
x_4: *"How many events were in Athens, Greece?"*
y_4: {2}
x_5: *"How many more participants were there in 1900 than in the first year?"*
y_5: {10}

Figure 4.1: Our task is to answer a highly compositional question from an HTML table. Each example in the training data contains a question x, a table w, and an answer y.

an example of a table w, a few question x_i, and their answers y_i.

At training time, the system has access to training data $\{(x^{(i)}, w^{(i)}, y^{(i)})\}_{i=1}^{N}$ of questions, tables, and answers. In particular, the training data does not contain *how* the answers were derived from the questions. To test that the system generalizes to unseen table schemata, we ensure that the tables in training and test sets do not overlap.

4.2 The WIKITABLEQUESTIONS dataset

As a benchmark for the task, we created a new dataset, WIKITABLEQUESTIONS, of question-answer pairs on tables from Wikipedia.

The data collection process is as follows. From the Wikipedia dump, we randomly selected data tables with at least 8 rows and 5 columns. We gave preference to tables with a high fraction of numerical data, as more interesting questions with comparison and calculation can be asked about the table.

After selecting the tables, we created two tasks on Amazon Mechanical Turk, a crowdsourcing service. In the first task, we asked the workers to write four trivia questions about the displayed table. For each question, we randomly gave the worker one of the 36 generic prompts in Table 4.1 to encourage more complex

Category	Prompts
Counting	How many ... ... number of ... must require counting
Superlatives	... the most ... ... the least ... ... top ...
	... ____est (e.g., largest, tallest) ...
Comparison	... at least ... ... at most ... ... more/less than ...
	... above/below ... ... same ... as ... must require comparison
	... ____er (e.g., larger, taller) ...
Arithmetic	How long ... ... difference ... ... average ...
	... total ... must require calculation
Quantification	... no ... ... not ... ... each ... ... only ...
Conjunction	... and ... ... but ... ... or ... ... other ...
Anaphora	... his/her ... ... its/their ...
Ordering	... before ... ... after ... ... first ... ... last ...
	... previous ... ... next ... ... consecutive ...

Table 4.1: Prompts for soliciting questions with a diverse set of operations.

questions. The worker can elect to change the prompt if it is impossible or impractical to follow.

In the second task, we asked workers to answer the questions from the first task on the given table. To aid the workers, we provided shortcuts for copying answers from the table, counting the number of selected cells, and computing statistics of the numbers in selected cells. We only kept the answers that were agreed upon by at least two workers.

The final dataset contains 22,033 examples on 2,108 tables of various topics. We set aside 20% of the tables and their associated questions as test set and only develop on the remaining examples. We performed simple preprocessing on the tables: we remove all non-textual contents (e.g., images and hyperlink targets), and if there is a merged cell spanning many rows or columns, we unmerge it and copy the content into each unmerged cell. In the later releases of the dataset, we also cleaned up encoding issues and enforced formatting consistencies among the answers.

4.3 Dataset analysis

With respect to contemporary datasets, the WIKITABLEQUESTIONS dataset was designed to cover a larger scope of the knowledge source (BREADTH) and more complex questions (DEPTH). We will analyze the dataset along the following four aspects:

- *Diversity of data schema* (a BREADTH aspect). The schema of an knowledge source defines the types of entities and relations that can exist between entities. For instance, in a single database table, we can treat the field names (e.g., $\{Id, Name, \dots\}$) and field types (e.g., `VARCHAR`) as the schema for that table. To increase BREADTH, we want the dataset to contain knowledge sources with a large variety or

an open set of data schemata.

- *Knowledge coverage* (a BREADTH aspect). Large knowledge bases such as Freebase (Google, 2013) contain much more knowledge than a domain-specific database. However, the amount of knowledge is bounded by the fixed data schema and the rate in which the knowledge base is populated. An open-domain knowledge source, such as the web, covers a larger amount of up-to-date knowledge.
- *Compositionality* (a DEPTH aspect). One measure of task complexity is the number of compositions or reasoning steps needed to complete the task. Note that due to sublexical compositionality (Wang et al., 2015b), the task can be compositional even when the question itself is not. For instance, finding someone's *aunt* is compositional if the knowledge source only allows querying the parent, sibling, and gender of a person.
- *Types of operations* (a DEPTH aspect). Finally, we consider the diversity of operations that are needed to compute the answers. Other than the number of unique operations, we also look at the properties of those operations (e.g., what the operations take as inputs).

4.3.1 Analysis of the WIKITABLEQUESTIONS dataset

We now quantitatively analyze the WIKITABLEQUESTIONS dataset along the different aspects of BREADTH and DEPTH, and compare the dataset with related datasets when applicable.

Diversity of data schema (BREADTH). We consider the number of different fields (indicated by the column header strings) as a rough measure of data schema diversity. The WIKITABLEQUESTIONS dataset contains 2,108 tables. Among them, there are 13,396 columns and 3,929 unique column headers. The diversity of column headers is much higher than traditional semantic parsing datasets, such as GEOQUERY (Zelle and Mooney, 1996) and ATIS (Dahl et al., 1994), which use small and fixed databases.

It is more tricky to compare our dataset with the semantic parsing datasets on knowledge bases such as FREE917 (Cai and Yates, 2013) and WEBQUESTIONS (Berant et al., 2013). While large knowledge bases can cover a large number of relations among entities, they usually have a predefined schema. In contrast, tables from the web can have any arbitrary schema depending on the authors of the tables.

Knowledge coverage (BREADTH). We first compare the scope of knowledge in our dataset with the information in knowledge bases. To do so, we sample 50 examples from the WIKITABLEQUESTIONS dataset and tried to answer them manually by querying Freebase. Even though Freebase contains some information extracted from Wikipedia, we can answer only 20% of the questions, indicating that the dataset covers a broader range of knowledge than Freebase.

On the other hand, it is also true that a large amount of knowledge is not encoded as tables. Nevertheless, we argue that parallel structures on the web page, such as the product listing or the news feed, can also be interpreted as a table where each parallel structure becomes a table column. For instance, in a product listing,

the product names can be detected based on the shared HTML properties, and the identified elements form a "column" in our virtual table. The same idea can be applied across different web pages that were generated from the same template (e.g., Amazon product pages).

Nevertheless, regarding the distribution of questions, one downside of our data collection mechanism is that the crowd workers write the questions *after* seeing the tables. This is the reverse of how an actual QA system would operate: the user first asks the question, and then relevant tables are retrieved based on the question. As such, unlike the OPENWEB dataset in Chapter 3, the distribution of questions is skewed away from the natural distribution of questions. Moreover, questions that cannot be answered by the given table are not present (except for annotation errors), even though detecting unanswerable questions is crucial for a real QA system. We found these sacrifices to be necessary for our task setting: most naturally occurring questions are either not factual ("*How can I lose weight?*"), not complex ("*Where is Toronto?*"), or not readily suitable to be answered by a table, and thus a considerable amount of data filtering effort would be needed to construct a dataset with a natural distribution of questions.

Compositionality (DEPTH). One selling point of traditional semantic parsing datasets is the complexity of the questions. One extreme example from the GEOQUERY dataset is the question "*What states border states that border states that border states that border Texas?*", which requires four levels of reasoning.

While an average example in the WIKITABLEQUESTIONS dataset is not as compositional, a non-trivial fraction of examples do involve at least three or four levels of reasoning. We sample 200 examples from the training data and note several questions that require compositional reasoning in Table 4.2.

In Section 5.6.4, we will return to quantitatively analyze compositionality by looking at the complexity of logical forms.

Types of operations (DEPTH). The questions in the WIKITABLEQUESTIONS dataset require a diverse set of operations to answer the questions. We manually classified the 200 examples above based on the types of operations for answering the questions. The statistics in Table 4.3 shows that while a few questions only require a simple operation such as table look-up or counting, the majority of the questions demands more complex operations.

One important aspect of our data collection procedure is that we *softly* specify the types of questions via question prompts. As Table 4.3 shows, this results in a sizable number of questions requiring the types of reasoning that we did not anticipate. Some examples of such "other phenomena" include identifying the second-highest value ("*what is the next highest hard drive available after the 30gb model?*"), summing up time periods ("*how many people stayed at least 3 years in office?*"), or using world knowledge ("*how far did they make it in the fa cup after 2009?*" requires knowing that "*Quarter-final*" is better than "*Round of 16*"). This is in contrast to data collection methods that restrict the types of questions more strongly. As an extreme example, the "overnight" data collection method (Wang et al., 2015b) generates a set of queries (e.g., logical forms or database queries) that the dataset wants to target, translates them into natural language questions using templates, and then asks humans to write natural paraphrases to the questions. By design, the

Link	Question	Comments
203-705	how many people stayed at least 3 years in office?	Requires computing the duration between *Took Office* and *Leave Office* dates.
203-116	which players played the same position as ardo kreek?	Requires finding people with that *Position*, but excluding Ardo Kreek himself.
203-104	which athlete was from south korea after the year 2010?	Requires filtering on two criteria, one of which is a comparison.
204-475	in how many games did the winning team score more than 4 points?	Requires finding the winning team in each row by comparing the two scores.
204-920	how many consecutive friendly competitions did chalupny score in?	Requires interpreting "consecutive", which could be complex depending on the available operations in the system
204-725	who earned more medals–vietnam or indonesia?	Requires comparing the *Total* column but only for the two countries in question.
204-255	which district has the greatest total number of electorates?	Requires summing up the *Number of electorates* in each district before comparing.
204-157	how many games were only won by 20 points or less?	Requires finding only the games with *W* (win) status, and with the difference between the scores not exceeding 20, before counting them.
203-109	what year did monaco ratify more international human rights treaties than they did in 1979?	Requires counting the number of treaties ratified in each year, and then find the year with more ratifications than 1979.
203-146	jones won best actress in a play in 2005. which other award did she win that year?	Requires finding the award name (in the *Category* column) in 2005 but excluding the "Best Actress in a Play" award.

Table 4.2: Examples of compositional questions in the WIKITABLEQUESTIONS dataset. The code in the *Link* column is the table code in the dataset.

resulting questions are guaranteed to have corresponding queries, but the diversity of questions is inherently limited by the templates used to generate the queries. Admittedly, we also observe a similar effect in the WIKITABLEQUESTIONS dataset, where the prompt words "*next*" and "*previous*" cause the operation "look at the next or previous rows" to be slightly over-represented.

4.3.2 Comparison with other semantic parsing for question answering datasets

we provide an overview of semantic parsing datasets in the literature for comparison. For each dataset, we focus on its motivation (i.e., what challenges it is trying to address) as well as the scope of the knowledge source (BREADTH) and the complexity of the questions (DEPTH).

Semantic parsing on databases. As explained in Section 2.4, early semantic parsing datasets focus on answering highly compositional questions (high DEPTH) on a small well-defined domain (low BREADTH).

Operation	Amount
look up cells in the specified column	13.5%
+ look at the next or previous rows	+ 5.5%
+ aggregation (e.g., counting, summing)	+ 15.0%
+ comparison (e.g., largest, highest)	+ 24.5%
+ arithmetic, union, intersection	+ 20.5%
+ other phenomena	+ 21.0%

Table 4.3: The operations required to answer the questions in 200 random examples from the WIKITABLEQUESTIONS dataset.

Some of the representative datasets are described below:

- GEOQUERY (Zelle and Mooney, 1996) contains 880 geography questions that can be answered with a database of 8 tables. Each question is annotated with a Prolog program. While the questions can be very compositional, the database contains few relations (35 fields total) and entities (737 unique non-numeric values). A similar dataset, JOBS (Tang and Mooney, 2001), contains 640 questions about job postings.

- ATIS (Dahl et al., 1994) is a popular dataset for natural language interfaces to database (NLIDB). Previous work such as He and Young (2006) and Zettlemoyer and Collins (2007) uses 4,978 training and 448 test examples, all of which were annotated with SQL queries. The dataset also contains a database of 25 tables with a total of 131 fields.

- OVERNIGHT (Wang et al., 2015b) datasets was created using the "overnight" data collection framework: generate logical forms, translate them into questions using templates, and then crowdsource natural paraphrases of those questions to be used as the actual questions. This results in 8 datasets from 8 different domains that were collected within a day. The datasets have recently been used as a cross-domain semantic parsing dataset (Su and Yan, 2017; Herzig and Berant, 2018).

As mentioned previously, while the WIKITABLEQUESTIONS dataset does not include extremely compositional (and arguably less natural) questions like in GEOQUERY or related datasets, it does contain examples with complex reasoning in the action space as demonstrated in Table 4.2.

Semantic parsing on knowledge bases. To extend the scope of the knowledge source, previous work has considered replacing small databases with large knowledge bases, which contain entities and relations from a variety of domains. The main focus of these dataset are mostly to scale up to the new larger knowledge source (higher BREADTH), and thus the question complexity is toned down (lower DEPTH).

- FREE917 (Cai and Yates, 2013) and WEBQUESTIONS (Berant et al., 2013) contain factoid questions

on Freebase, a large knowledge base with more than 2 billion facts, each describing a relation between two entities (e.g., (`BarackObama`, `PlaceOfBirth`, `Honolulu`)). Some questions involve multi-step reasoning (e.g., "*what is the name of justin bieber brother?*" requires filtering the answer by gender) and operations such as counting (e.g., "*how many australian states and territories?*"). However, most questions are not compositional and usually involve finding the correct anchor entity (e.g., `BarackObama`) and a short sequence of relations to reach the answer (e.g., `Spouse–PlaceOfBirth`) (Yao and Van-Durme, 2014). The two datasets contain 917 and 5,810 questions, respectively.

While FREE917 is annotated with logical forms, WEBQUESTIONS was designed to have only a distant supervision of question-answer pairs. Yih et al. (2016) later cleans up and annotates the questions with SPARQL queries, resulting in the WEBQUESTIONSSP dataset of 4,737 answerable questions.

- SIMPLEQUESTIONS (Bordes et al., 2015) is a large-scale dataset of more than 108,442 questions on a subset of Freebase. The dataset was created to study question answering at scale. As such, it exclusively contains questions that only involve identifying the correct anchor entity and a single relation.

- SPADES (Bisk et al., 2016a) contains 93,319 clozed-style questions derived by blanking out an entity from each declarative sentence. The sentences themselves come from a subset of CLUEWEB09 that were annotated with Freebase entities (Gabrilovich et al., 2013). While some sentences contain multiple entities, the dataset as a whole is not very compositional.

To address the reduced DEPTH of the datasets above, more recent datasets on knowledge bases aim to increase the complexity of the questions:

- QALD (Lopez et al., 2013) is a series of shared tasks on question answering over knowledge bases. Some of the most frequent challenges include multilingual question answering, where the question can be from any of the given list of languages; and hybrid question answering, where both knowledge bases and *unstructured text* are needed to answer the questions. While the scope of the knowledge source is large and many questions are compositional, especially in the hybrid task, the training dataset is small (less than 500 examples), and most participating systems resort to non-learning-based approaches.

- GRAPHQUESTIONS (Su et al., 2016) contains 5,166 questions that can be answered by graph queries on Freebase. In contrast to the other datasets above, GRAPHQUESTIONS contains more compositional questions with multiple relations, counting (e.g., "*how many* . . . "), comparison (e.g., ". . . *older than* . . . "), and superlatives (e.g., ". . . *most recent* . . . ").

- COMPLEXWEBQUESTIONS (Talmor and Berant, 2018) creates complex questions by modifying the SPARQL queries from the WEBQUESTIONSSP dataset. The resulting dataset contains 34,689 questions with complex operations such as conjunctions, superlatives, comparatives, and compositions.

Semantic parsing on tables. After the release of the WIKITABLEQUESTIONS dataset, several datasets of question answering on tables have been released, each of which targets a different aspect of answering complex questions on tables. Similar to the setup of WIKITABLEQUESTIONS, the datasets below ensure that the tables in the training and test sets do not overlap.

- WIKISQL (Zhong et al., 2017) has a similar setting to WIKITABLEQUESTIONS but focuses on generating SQL queries. The dataset is large: it contains 80,654 SQL queries with human-annotated descriptions on 24,241 Wikipedia tables. All queries follow a similar pattern of selecting a column (with an optional aggregation such as `COUNT`) and up to 4 `WHERE` clauses. While this means the query pattern is not diverse, multi-step reasoning is present in the form of multiple `WHERE` clauses. Moreover, the dataset contains a large number of tables, making it a suitable dataset for learning to identify the correct table elements based on the question in an open-domain setting.

- SPIDER (Yu et al., 2018b) is a recent text-to-SQL dataset with a focus on high compositionality. It contains 10,181 question-query pairs on 200 databases, each with multiple tables. In addition to the sheer number of SQL operations and nested queries, many examples also require using `JOIN` to connect the information across tables. Therefore, in addition to identifying the right SQL operators, the system also needs to choose the table fields from a large set of inter-related fields in the database.

Sequential semantic parsing. To extend beyond answering a single independent question, previous work has proposed tasks of answering a series of follow-up questions or performing actions sequentially. The task complexity mainly manifests in the sequential nature of the task. To perform well in these tasks, one needs to interpret the utterance under the context of previous utterances as well as the intermediate computation state.

- The original ATIS dataset (Dahl et al., 1994) contains long session data (7 turns on average) where the user asks a sequence of follow-up questions to the agent (e.g., asking "*On American Airlines*" as a follow-up question to "*Show me flights from Seattle to Boston next Monday*"). To perform well, the system needs to keep track of the dialog context and interprets the questions with respect to the query results from the previous input utterances. However, the majority of context-dependent utterances only involve ellipsis: the utterance adds, changes, or queries some pieces of information from the utterances preceding it (e.g., the "*On American Airlines*" example above adds the airline to the query in the previous utterance).

- SQA (Iyyer et al., 2017) contains sequential questions on Wikipedia tables. The questions were written based on the compositional questions in our WIKITABLEQUESTIONS dataset. Concretely, each step of computation is converted into a question annotated with the intermediate result (e.g., a set of selected cells). Like most sequential semantic parsing datasets, the questions feature context-dependent structures such as anaphora (e.g., "*Which of **them** won a Tony after 1960?*") and ellipsis (e.g., "*Who had the biggest gap between their two award wins?*" where the list of people considered is the filtered list from the previous steps).

- There are several datasets that involve manipulating objects in a space based on a sequence of utterances. These datasets feature context-dependent spatial reasoning as well as anaphora and ellipsis (e.g., "*Then move to the left of it*"). Examples include SAIL (MacMahon et al., 2006; Chen and Mooney, 2011), where an agent navigates around a 2D grid map; SCONE (Long et al., 2016), where objects in a 1D space move, change properties, or interact with other objects; and BLOCKS (Bisk et al., 2016b), where blocks in 2D or 3D spaces are moved based on either low-level or high-level commands.

It is arguable that using multiple context-dependent utterances to describe a task is more natural than forming a single complex question (Iyyer et al., 2017). However, this requires factoring the task into steps, which might be difficult when the space of possible actions and intermediate computation states are not known by the user. For instance, during the creation of the SQA dataset above, some questions in WIKITABLEQUESTIONS does not have clear intermediate computation results, and the question writers were asked to write up a new question in such cases. Furthermore, some types of compositional computations can be easily phrased as a single question (e.g., "*which players played **the same** position as ardo kreek?*").

Visual question answering. Images provide a nice playground for complex reasoning: each object has various properties (e.g., object type, color, size, amount) and relations with other objects (e.g., position, relative properties), while the whole scene could describe actions involving the objects. Following the success of object recognition (Krizhevsky et al., 2012; Szegedy et al., 2015; He et al., 2016) and image captioning (Farhadi et al., 2010; Lin et al., 2014; Fang et al., 2015; Mao et al., 2015), researchers turn to the task of answering questions on images, which requires more complex reasoning.

- Early visual QA datasets focus on factoid questions. For instance, VQA (Antol et al., 2015) contains free-form and open-ended questions on images. Most images come from the MS COCO image captioning dataset (Lin et al., 2014), which contains a diverse set of photographs. Answering the questions demands various skills such as recognizing objects, recognizing activities, and factual reasoning. However, most questions require only one or two steps of reasoning, and many could be guessed without looking at the image at all (Agrawal et al., 2016; Goyal et al., 2017). A later dataset, GQA (Hudson and Manning, 2019), has a similar setting to VQA but with more compositional questions.

- SHAPES (Andreas et al., 2016) and CLEVR (Johnson et al., 2017a) address task complexity (DEPTH) with highly compositional questions on synthetic scenes. The tasks are reminiscent of the questions in the SHRDLU system (Winograd, 1972) and feature a diverse types of reasoning such as spatial reasoning, object property identification (shape, size, color, and texture), aggregation (e.g., "*How many* ..."), and comparison (e.g., "... *same size as* ..."). A follow-up dataset CLRVR-HUMANS (Johnson et al., 2017b) provides human-authored sentences for the scenes in the CLEVR dataset.

- In NLVR (Suhr et al., 2017) and NLVR2 (Suhr et al., 2018), the task is to identify whether the statement describes the image or not. Similar to CLEVR, the questions are compositional, and require

both spatial and logical reasoning. NLVR contains synthetic images, while NLVR2 contains real photographs.

Parsing into domain-specific languages. When the knowledge source does not expose its internal content, a QA system needs to interact with it using an interface specific to that knowledge source. For instance, a web service might expose a set of APIs that can be called, and a QA system needs to predict the correct API function and arguments. We now give some examples of datasets whose outputs are in specific programming languages for interacting with such knowledge sources. Note that some of these datasets are not QA datasets, and their evaluation is based on whether the predicted program matches the annotated program, either at the surface level or at the behavioral level.

- REGEXP824 (Kushman and Barzilay, 2013) and NL-RX (Locascio et al., 2016) contain regular expressions and their natural language descriptions. The system's output is evaluated on its equivalence with the annotated regular expression.

  One challenge highlighted by Kushman and Barzilay (2013) is that while multiple regular expressions are equivalent to each other, some of them align better with the natural language description. A similar sentiment is shared by our dataset: there are multiple ways to derive the answer. However, while regular expression has a deterministic mechanism for checking equivalence, it is more difficult to distinguish correct ways of deriving the answer from the ones that get the right answer for wrong reasons. We will study this phenomenon in detail in Chapter 7.

- IFTTT (Quirk et al., 2015) is a dataset of if-this-then-that recipes annotated with short descriptions (e.g., "*Archive your missed calls from Android to Google Drive*"). Each recipe contains a trigger and an action, both of which are API functions (e.g., `Google_Drive.Add_row_to_spreadsheet`) with several parameters. While the number of distinct functions is limited, the main challenge comes from the noisy natural language descriptions, which often under-specify the output.

- NL2BASH (Lin et al., 2018) is a recent dataset of Bash one-liners. While the number of Bash utilities is limited to 135, each utility has a distinct parameter convention (i.e., distinct schema) and should be considered as a separate domain. Furthermore, the commands contain a fair amount of compositionality with logical connectives (`&&`, `||`) and nested commands (achieved via `|`, `$(...)`, and `<(...)`).

- DJANGO (Oda et al., 2015) was originally a code-to-pseudocode dataset, but has been reversed into a semantic parsing dataset (Ling et al., 2016). The output is a single line of Python from the Django project, a real code base with a wide variety of constructs.

- In the MTG and HEARTHSTONE datasets (Ling et al., 2016), the system is given the text and metadata of a card in a collectible card game, and should output a class implementing the behavior of the card. While the task is in a closed domain, the main challenge is to infer complex card behaviors from a short high-level description.

- In the CONCODE dataset (Iyer et al., 2018), the system is given a description of a Java method along with the signatures of fields and other methods in the surrounding class. The system should then generate the method body. Unlike MTG and HEARTHSTONE, the CONCODE dataset operates in an open-domain setting: the code and descriptions come from Github repositories, and the output code has to be generated based on what other methods are available in the class.

- In the datasets above, the output is evaluated on surface form metrics (e.g., exact match and BLEU score) instead of functional correctness. Recent datasets such as ALGOLISP (Polosukhin and Skidanov, 2018) and NAPS (Zavershynskyi et al., 2018) address this by providing test cases to verify the correctness of the generated programs. Similar to test-driven program synthesis tasks (Gulwani, 2011; Devlin et al., 2017; Shi et al., 2019), a few test cases are also available to the system, allowing it to search over the possible programs to find one that pass the test cases. The generated programs are then tested against hidden test cases.

In contrast to these datasets, the WIKITABLEQUESTIONS dataset does not include logical forms or programs for computing the answers. Apart from saving the annotation cost, the lack of logical forms creates an opportunity for different types of approaches to solving the task, including ones that do not use discrete symbols to represent steps of reasoning. And even among semantic parsing approaches, the system is free to choose a logical form language that suits the task or the model. Nevertheless, as we will explore in the subsequent chapters, having only the distant supervision from the annotated answers makes it more difficult to learn a semantic parsing model.

4.4 Conclusion

We have presented a new question answering task that features semi-structured tables from the web as the knowledge source (high BREADTH) and more complex questions that require various types of reasoning and compositional understanding (high DEPTH). The WIKITABLEQUESTIONS dataset we collected contains question-answer pairs and not the steps for computing the answers, making it a flexible dataset for different types of approaches.

In the next chapter, we explore a semantic parsing approach for tackling the task. To handle the unseen data schema, the table will be converted to a generic graph representation. The task is then to parse the question into a domain-generic logical form that can be executed on the graph to get an answer. Using this framework, we will highlight the challenges caused by the unseen schema and question complexity, and then address the challenges in subsequent chapters.

Chapter 5

Semantic Parsing with Flexible Composition

In the next three chapters, we cast the task of answering a question x on a web table w as a *semantic parsing* task. The core idea is to model the steps for computing the answer y as a compositional *logical form* z generated based on the input question x.

As a running example, consider the question $x =$ "*Where did the last 1st place finish occur?*" on the table w in Figure 5.1. To derive the answer "*Thailand*", one could identify the rows where the column *Position* contains the text *1st* (2nd and 4th rows), select the last one (4th row), and then look for the answer in the *Venue* column. This process for computing the answer can be expressed as the logical form $z = \texttt{VenueOf.argmax}(\texttt{HasPosition.1st}, \lambda r[\texttt{IndexOf}.r])$. (We will explain the semantics of such logical forms in Section 5.3.) The result of executing this logical form z on the table is a *denotation* $y = \{\texttt{Thailand}\}$, which we use as the answer. Using the training data, we train our system to generate and rank logical forms z so that the highest-scoring logical form executes to the correct denotation.

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

$x =$ "*Where did the last 1st place finish occur?*"
$z = \texttt{VenueOf.argmax}(\texttt{HasPosition.1st}, \lambda r[\texttt{IndexOf}.r])$
$y = \{\texttt{Thailand}\}$

Figure 5.1: Our running example based on a training example from the WIKITABLEQUESTIONS dataset.

Using logical forms as the intermediate representation provides several benefits. As logical forms are compositional and executable, they are suitable for representing the steps needed for answering complex questions. Logical forms are also highly interpretable: they explicitly show how the answer is computed. Finally, the syntax and semantics of logical forms, as well as how larger logical forms are constructed from parts, can be customized to suit the task at hand. In fact, we will be adapting *lambda dependency-based compositional semantics* (lambda DCS), a formal language designed for querying knowledge graphs (Liang, 2013; Berant et al., 2013), to fit our task of answering questions on tables.

In this chapter, we describe our framework for learning a semantic parser with *distant supervision*: the training dataset contains only the correct answers and not how the answers can be derived. We first formalize our semantic parsing framework. We then describe the syntax and semantics of logical forms, detailing how they can be executed on a given table. Afterward, we explain our model that learns to parse questions into logical forms via flexible bottom-up generation. Finally, we report the results of our experiments and provide analysis of our model.

Reference. The results described in this chapter have been published as (Pasupat and Liang, 2015). Reproducible experiments are hosted on the CodaLab platform at

`https://worksheets.codalab.org/worksheets/0xf26cd79d4d734287868923ad1067cf4c`.

5.1 Framework overview

Task. Given a table w and a question x about the table, the system should produce a list y of values that answers the question. For training, the system is given a training dataset $\{(x^{(i)}, w^{(i)}, y^{(i)})\}_{i=1}^{N}$, where each example contains a question $x^{(i)}$, a table $w^{(i)}$, and the correct answer $y^{(i)}$.

Modeling. Given the question x and a table w, we use a logical form z to represent the process for computing the answer y. Formally, we define a generative model that generates y from inputs x and w via a latent variable z. Given a space $\mathcal{Z}$ of *all* possible logical forms, we define a probability distribution $p(z \mid x, w)$ over all $z \in \mathcal{Z}$. The probability of producing a denotation y can be computed by maginalizing over z:

$$p(y \mid x, w) = \sum_{z \in \mathcal{Z}} \mathbb{I}(\llbracket z \rrbracket_w = y)\, p(z \mid x, w) \tag{5.1}$$

where $\llbracket z \rrbracket_w$ is the denotation from executing z on w.

Prediction. Given a question x on a table w, we predict a logical form z and an answer y using an approximation of the generative model above. Our prediction framework is illustrated in Figure 5.2. Since it is impossible to enumerate the set $\mathcal{Z}$ of all possible logical forms, we first generate a finite set of candidate logical forms $\mathcal{Z}_x$ by parsing the question x using the information from the table w (Section 5.4). Each generated

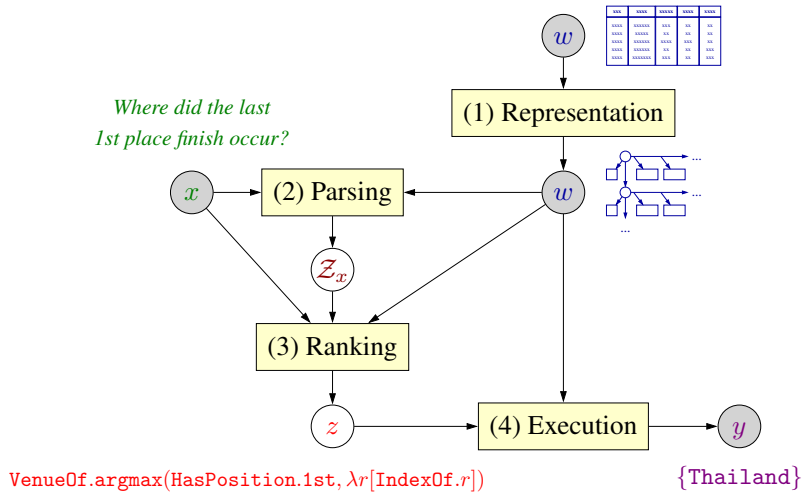


Figure 5.2: The semantic parsing framework for answering questions x on web tables w. At test time: (1) the table w is represented as a knowledge graph as shown in Figure 5.3; (2) with information from w, the question x is parsed into candidate logical forms in $\mathcal{Z}_x$; (3) the highest-scoring candidate $z \in \mathcal{Z}_x$ is chosen; and (4) z is executed on w, yielding the denotation y. The model is trained to maximize the probability of predicting the correct denotation.

logical form $z \in \mathcal{Z}_x$ is associated with its denotation $y = [\![x]\!]_w$ and a score $s_\theta(z, x, w)$, where s_θ is a scoring function (Section 5.5) with trainable parameters θ. We define a distribution over the candidate logical forms as

$$p_\theta(z \mid x, w) \propto \exp\left[s_\theta(z, x, w)\right]. \tag{5.2}$$

To predict y, we skip the marginalization over z as written in Equation 5.1, and instead just return the denotation of z with the highest model probability $p_\theta(z \mid x, w)$ (or equivalently, the highest score $s_\theta(z, x, w)$).

Training. Given training data $\{(x^{(i)}, w^{(i)}, y^{(i)})\}_{i=1}^N$, we use a gradient ascent method to optimize θ to maximize one of the following objective functions:

1. *Log-likelihood of the correct denotations.* To compute the log-likelihood of a denotation y, we marginalize over the logical forms $z \in Z_x$ that executes to y:

$$p_\theta(y \mid x, w) = \sum_{z \in \mathcal{Z}_x} \mathbb{I}([\![z]\!]_w = y)\, p_\theta(z \mid x, w). \tag{5.3}$$

   The objective function is

$$J_{\text{LL}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y^{(i)} \mid x^{(i)}, w^{(i)}) - \Omega(\theta) \tag{5.4}$$

   where Ω is a regularization function. Intuitively, a gradient update will push up the scores of *all*

consistent logical forms in $\mathcal{Z}_x$ (i.e., the ones with the correct denotation), and push down the scores of *all* inconsistent logical forms in $\mathcal{Z}_x$.

A few previous studies have used the log-likelihood object to train a semantic parser (Kwiatkowski et al., 2011; Liang et al., 2011; Berant et al., 2013). While this objective function correctly maximizes the marginalized probability of $y^{(i)}$, it does not match the prediction process, which does not marginalize over z.

2. *Contrastive loss.* From $\mathcal{Z}_x$, we pick a logical form z_+ with the highest model probability among consistent logical forms (i.e., logical forms giving the correct denotation), and another logical form z_- with the highest probability among inconsistent logical forms. The objective function is defined as

$$J_{\text{cnt}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \log \frac{p_\theta(z_+^{(i)} \mid x^{(i)}, w^{(i)})}{p_\theta(z_-^{(i)} \mid x^{(i)}, w^{(i)})} - \Omega(\theta) \tag{5.5}$$

Intuitively, a gradient update will push up the scores of *one* consistent logical forms and push down the scores of *one* inconsistent logical forms. Unlike the usual contrastive loss used in previous semantic parsing work (Zettlemoyer and Collins, 2007, 2009), we always update the parameter even when the probability of z_+ already exceeds that of z_-.

The next four sections give more details about each component of our framework. Section 5.2 describes how we represent the table as a *knowledge graph*, which aids logical form execution and helps us maintain uncertainty over possible interpretations of the table. Section 5.3 describes the syntax and semantics of *lambda DCS*, our logical form language. Afterward in Section 5.4, we look at how the set $\mathcal{Z}_x$ of candidate logical forms is generated from the input. Finally, Section 5.5 defines the scoring function $s_\theta(z, x, w)$.

5.2 Graph representation of the table

Inspired by the graph representation of knowledge bases, we represent the table as a *knowledge graph* as illustrated in Figure 5.3. The extensible graph structure allows us to easily encode any additional information by adding custom edges and nodes. In particular, we will augment the graph with structures for answering different types of questions and maintaining uncertainty over different interpretations of the cell contents.

Basic construction. To construct a knowledge graph, we first convert each row into a *row node* (e.g., the first row becomes r_1), and convert cells into *cell nodes* (e.g., the cell with text "*Hungary*" becomes a node `Hungary`). Then, we convert each column into directed *column edges* from the row nodes to the corresponding entity nodes of that column, and label the edges with the column header (e.g., we construct an edge with label `Venue` from r_1 to `Hungary`). Note that columns are automatically renamed to have unique texts if necessary.

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

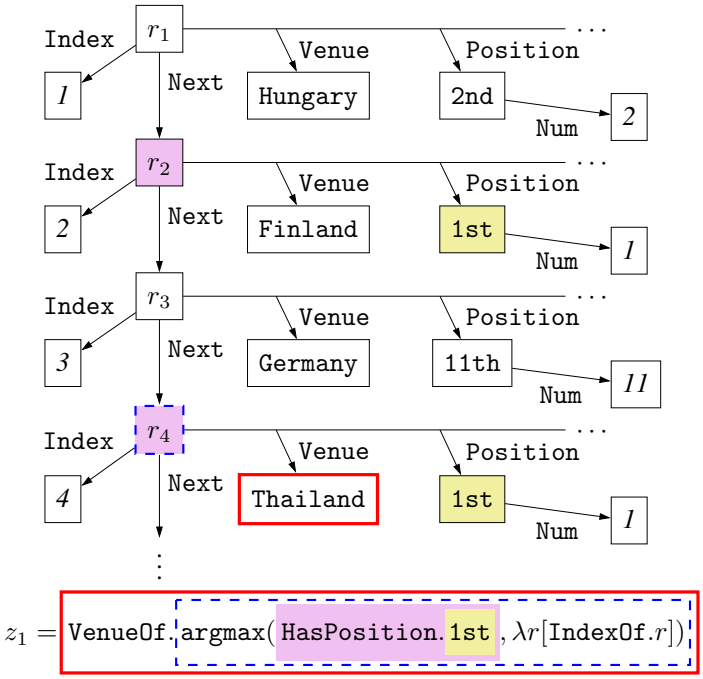


Figure 5.3: The table w is represented as a *knowledge graph*. The recursive execution of logical form z is shown via the different colors and styles.

Graph augmentation. One benefit of the graph representation is that we can freely augment the graph with additional information that helps us answer the questions.

The first type of augmentation we employ is *normalization nodes*. Some cell strings (e.g., "*2001*") can be interpreted as a number, a date, or a proper name depending on the context, while some other strings (e.g., "*200 km*" and "*21-14*") have multiple parts. Instead of committing to one normalization scheme, we introduce normalization nodes for the possible ways to interpret the cell strings, and use special edges to link cell nodes to normalization nodes. In this chapter, we consider two types of normalization: `Num` (value of the first number in the cell) and `Date` (year, month, and/or date that can be parsed from the cell). For instance, a cell node `2001` will have a `Num` edge pointing to the atomic value *2001.0*, and a `Date` edge pointing to the atomic value *2001-XX-XX*.

The second type of augmentation are row-specific edges. Questions about tables usually involve reasoning about the order of the rows. For instance, "*What is the next . . . ?*" or "*Who came before . . . ?*" require looking at adjacent rows, while "*Who is the last . . . ?*" requires looking at the last row among a group of rows. To help answer these types of questions, we augment each row node with an edge labeled `Next` pointing to the next row node, and an edge labeled `Index` pointing to the row index number (starting from 1).

5.3 Logical forms syntax and semantics

We use *lambda dependency-based compositional semantics* (Liang, 2013), or *lambda DCS*, as the language of our logical forms. The language was originally designed for semantic parsing on large knowledge graphs (Berant et al., 2013). As the context table can be converted into a knowledge graph (Section 5.2), the lambda DCS formalism naturally transfers to our setting.

We now describe the syntax and semantics of lambda DCS constructs. The knowledge graph in Figure 5.3 will be used as a running example.

Types of logical forms. A lambda DCS logical form is either a *unary* or a *binary*. A unary represents a set of objects (e.g., $\{\texttt{Finland}, \texttt{Thailand}\}$). A binary represents a mapping from objects to objects, which we will write as a set of pairs of objects (e.g., $\{(r_2, \texttt{Finland}), (r_4, \texttt{Thailand})\}$). We use $[\![z]\!]_w$ to denote the denotation of z with respect to the knowledge graph w (i.e., the result of executing z on w).

Primitives. Primitives are the smallest building blocks of lambda DCS logical forms. Under the context of a knowledge graph w from Section 5.2, the primitive unaries are cell nodes (e.g., $z = \texttt{Finland}$ with denotation $[\![z]\!]_w = \{\texttt{Finland}\}$), atomic values such as numbers and dates (e.g., $z =$ *2001-XX-XX*; $[\![z]\!]_w = \{$*2001-XX-XX*$\}$), and the special `allRows` symbol that executes to the set of all row nodes (e.g., in our running example, $[\![\texttt{allRows}]\!]_w = \{r_1, r_2, r_3, r_4, r_5\}$).

The primitive binaries are the graph edges (e.g., $z = \texttt{Venue}$; $[\![z]\!]_w = \{(r_1, \texttt{Hungary}), (r_2, \texttt{Finland}), \ldots\}$) and special binaries `=`, `!=`, `<`, `<=`, `>`, and `>=` (e.g., $z = <$; $[\![z]\!]_w = \{(p, q) \mid p < q\}$).

Name	Logical form z	Denotation $[\![z]\!]_w$
Reverse	$\mathbf{R}[b]$	$\{(p,q) \mid (q,p) \in [\![b]\!]_w\}$
Example:	$\mathbf{R}[\texttt{Venue}]$	$\{(p,q) \mid q \xrightarrow{\texttt{Venue}} p\}$
Join	$b.u$	$\{p \mid \exists q, q \in [\![u]\!]_w \wedge (p,q) \in [\![b]\!]_w\}$

Note: To avoid confusion regarding the direction of Join, we use the following notations when the binary b is a graph edge: $b.u$ is written as $\texttt{Has}b.u$, and $\mathbf{R}[b].u$ is written as $b\texttt{Of}.u$.

Example:	HasYear.2001	$\{p \mid p \xrightarrow{\texttt{Year}} \texttt{2001}\} = \{r_1\}$
	DateOf.2001	$\{p \mid \texttt{2001} \xrightarrow{\texttt{Date}} p\} = \{\textit{2001-XX-XX}\}$
	IndexOf.HasPosition.1st	$\{p \mid \exists q, q \xrightarrow{\texttt{Index}} p \wedge q \xrightarrow{\texttt{Position}} \texttt{1st}\} = \{2,4\}$
	>=*.4*	$\{p \mid p \geq 4\}$
Union	$u_1 \sqcup u_2$	$\{p \mid p \in [\![u_1]\!]_w \vee p \in [\![u_2]\!]_w\}$
Example:	$\texttt{Finland} \sqcup \texttt{Hungary}$	$\{\texttt{Finland}, \texttt{Hungary}\}$
Intersection	$u_1 \sqcap u_2$	$\{p \mid p \in [\![u_1]\!]_w \wedge p \in [\![u_2]\!]_w\}$
Example:	>=*.1980* $\sqcap$ <*.1990*	$\{p \mid 1980 \leq p < 1990\}$
Aggregation	$A(u)$	$\{p \mid p = A([\![u]\!]_w)\}$

Choices of the aggregation function A include:
- count: number of elements in the set
- min and max: minimum and maximum value (can only be used on a set of numbers or dates)
- sum and avg: sum and average value (can only be used on a set of numbers)

Example:	$\texttt{count}(\texttt{HasEvent.400m})$	$\{p \mid p = \texttt{count}(\{q \mid q \xrightarrow{\texttt{Event}} \texttt{400m}\})\} = \{3\}$
Arithmetic	$\texttt{sub}(u_1, u_2)$	$\{p \mid \exists q \exists q', q \in [\![u_1]\!]_w \wedge q' \in [\![u_2]\!]_w \wedge p = q - q'\}$

For simplicity, we only consider subtraction since other arithmetic operations are rare in our dataset.

Example:	$\texttt{sub}(\texttt{IndexOf.HasVenue.China}, \texttt{IndexOf.HasVenue.Hungary})$	$\{5-1\} = \{4\}$
Superlative	$\texttt{argmax}(u, b)$ (argmin is defined similarly)	$\{p \mid p \in [\![u]\!]_w$ such that for any other $p' \in [\![u]\!]_w$, $[\exists q \forall q', (q,p) \in [\![b]\!]_w \wedge (q',p') \in [\![b]\!]_w \Rightarrow q \geq q']\}$

Intuitively, the binary b can be thought of as a function mapping $p \in [\![u]\!]_w$ to a set of values. The argmax operator chooses the values p that give the maximum mapped value.

Example:	allRows	$\{r_1, r_2, r_3, r_4, r_5\}$
	$\lambda r[\texttt{IndexOf}.r]$	$\{(1,r_1), (2,r_2), (3,r_3), (4,r_4), (5,r_5)\}$
	$\texttt{argmin}(\texttt{allRows}, \lambda r[\texttt{IndexOf}.r])$	$\{r_1\}$ (giving the minimum value 1)
	HasPosition.1st	$\{r_2, r_4\}$
	$\texttt{argmax}(\texttt{HasPosition.1st}, \lambda r[\texttt{IndexOf}.r])$	$\{r_4\}$

Table 5.1: The syntax and semantics of lambda DCS logical operators. (Variables u and b denote a unary and a binary, respectively. Variables p and q denote arbitrary values.)

Operators. Larger logical forms can be constructed from smaller ones using *logical operators*. Table 5.1 lists the operators we use in our work. Note that the direction of the binary in our superlative operator (`argmax` and `argmin`) is the reverse of that in Liang (2013). This is to make the syntax more consistent with the key-based comparison functions in many programming languages (e.g., the logical form $\texttt{argmax}(u, \lambda p[f(p)])$ is equivalent to "`max(u, key=lambda p: f(p))`" in Python and "`u.max_by {|p| f(p)}`" in Ruby).

Lambda abstraction. As seen in the examples of superlatives in Table 5.1, one way to construct more complex logical forms is to use *lambda abstraction* to construct a binary. For a logical form $f(p)$ containing a free variable p, the construct $\lambda p[f(p)]$ is a binary with denotation $\{(q, p) \mid q \in [\![f(p)]\!]_w\}$. For example, the binary $\lambda p[\texttt{count}(\texttt{HasEvent}.p)]$ counts the number of rows for each value in the *Event* column, and thus executes to $\{(3, \texttt{400m}), (2, \texttt{Relay})\}$.[1]

Terminologies. Each token in a logical form is called a *predicate*. A logical form z is *consistent* with a value y if its denotation $[\![z]\!]_w$ matches y; otherwise, it is *inconsistent*. Note that the matching can be done loosely to get around text normalization issues in the annotated answers (e.g., a logical form whose denotation is $\{32\}$ is still treated as consistent with the correct answer "*32 km*").

5.4 Parsing the utterance into logical forms

Given a knowledge graph w,[2] we want to parse the input question x and generate a set $\mathcal{Z}_x$ of candidate logical forms. This section describes a bottom-up semantic parser that produce the logical forms and their scores.

5.4.1 Deduction rules

The space of possible logical forms given the knowledge graph w and the question x is defined recursively by a set of *deduction rules*, which dictate how logical forms can be constructed either from the inputs or from smaller logical forms (Zettlemoyer and Collins, 2007; Berant et al., 2013). Informally, our parser maintains a set of logical forms, and then repeatedly applies deduction rules to construct larger logical forms from smaller ones in the set. Logical forms that are "well-formed" are finally compiled into the set $\mathcal{Z}_x$ of candidate logical forms. The parsing algorithm will be described more formally in Section 5.4.2.

Deduction rules are divided into two types: terminal rules and compositional rules.

[1] Technically, the denotation also contains $(0, p)$ for any other values p.

[2] We overload the variable w for both the table and its knowledge graph representation. Clarification will be made when necessary.

Terminal rules. A terminal rule creates logical forms based on the input question x and knowledge graph w. Each terminal rule follows one of the following templates:

$$\textit{TokenSpan}[s] \rightarrow c[f(s)] \tag{5.6}$$

$$\varnothing \rightarrow c[f()] \tag{5.7}$$

A rule of Template 5.6 takes a token span from the question x and applies the *semantic function* f, which generates a set of logical forms. The resulting logical forms will be associated with a *category* c, which is used to enforce type consistency. For instance, let match be a function that takes a string s and returns cell nodes whose cell content strings match s. Then the deduction rule

$$\textit{TokenSpan}[s] \rightarrow \textit{Entity}[\mathrm{match}(s)] \tag{5.8}$$

can construct logical forms of category *Entity* by applying the function match on some token span s of x.

A rule of Template 5.7 works similarly, but does not require any string from the question x. Apart from generating input-independent logical forms (e.g., `>=` and `allRows`), this template is also useful when the logical form is difficult to infer deterministically from the question. For example, in most questions, the relevant column names are indirectly mentioned or implicitly implied (e.g., the question "*Where did the last 1st place finish occur?*" in Figure 5.1 indirectly mentions the columns `Venue` and `Position`). We can use the following deduction rule to generate such columns from the table:

$$\varnothing \rightarrow \textit{Relation}[\mathrm{columns}()], \tag{5.9}$$

where the function $\mathrm{columns}$ returns all unique column edges from the knowledge graph w.

Compositional rules. A compositional rule constructs larger logical forms from smaller ones. Each compositional rule follows one of the following templates:

$$c_1[z_1] \rightarrow c[g(z_1)] \tag{5.10}$$

$$c_1[z_1] + c_2[z_2] \rightarrow c[g(z_1, z_2)] \tag{5.11}$$

A rule of Template 5.10 takes a child logical form z_1 of category c, and then constructs a new logical form $g(z_1)$ of category c. For instance, if $g(z_1) = \texttt{count}(z_1)$, we can construct $\texttt{count}(\texttt{HasPosition.1st})$ from an existing logical form `HasPosition.1st`. A rule of Template 5.11 operates similarly but takes two children logical forms.

List of deduction rules. Tables 5.2 and 5.3 detail the deduction rules used in our parser. The deduction rules are designed to be closely mimic the compositional syntax of lambda DCS. However, as some lambda DCS constructs are very generic and can generate many nonsensical logical forms (e.g., the subtraction operator

Rule	Semantics	Example
TokenSpan $\rightarrow$ *Entity*	$\text{match}(s)$	`Finland`
(cell node with string s)		from *"Finland"*
TokenSpan $\rightarrow$ *Atomic*	$\text{value}(s)$	*2012-07-XX*
(interpretation of s at an atomic value)		from *"July 2012"*
$\varnothing \rightarrow$ *Relation*	$\text{columns}()$	`Venue`
(all column edges)		
$\varnothing \rightarrow$ *Relation*	$\text{normalizedColumns}()$	$\lambda x[\texttt{HasYear.HasDate}.x]$
(binaries formed by joining a column edge with a normalization edge)		
$\varnothing \rightarrow$ *Records*	`allRows`	
$\varnothing \rightarrow$ *RecordFn*	$\lambda r.[\texttt{IndexOf}.r]$	

Table 5.2: Terminal deduction rules. Entities and atomic values (numbers and dates) are constructed from token spans while other predicates are not.

can subtract any two arbitrary numbers), some operators are restricted to be applied only in certain contexts (e.g., only allow subtracting numbers from two cells from the same column). This trade-off prevents us from answering some small number of sophisticated questions, but we found it necessary for making the space of generated logical forms manageable.

From the table, we can see that many deduction rules construct logical forms without referencing the question. These include the terminal rules of the form $\varnothing \rightarrow c[f()]$ and various compositional rules that generate operators (e.g., `argmax` can be generated even when the question does not have any word that expresses superlative). This is intentional for two reasons:

- Many logical form predicates do not explicitly align to any token from the question.
- Even when the alignment exist, we want to learn such an alignment from the data. This will be achieved by the features in the scoring module that relate logical form predicates to the tokens in the question.

5.4.2 Floating parser

To parse the question x based on the deduction rules, we propose a new parser named *floating parser* that can construct logical forms in a flexible order.

Chart parser. To understand the motivation behind the floating parser, let us first consider a more common bottom-up parsing algorithm: the CKY algorithm for chart parsing.

Given an input sentence x with tokens $x_1, \ldots, x_n$, the algorithm constructs and stores partial parses with category c of the token span $x_{i:j} := (x_i, \ldots, x_{j-1})$ in a *cell* labeled (c, i, j). Being a dynamic programming algorithm, the CKY algorithm populates the cells in the increasing order of their span lengths $j - i$. To construct a parse for the cell (c, i, j), we have the following choices:

Rule	Semantics	Example
	Join + Aggregate	
Entity or *Atomic* $\rightarrow$ *Values*	z_1	`Finland`
Atomic $\rightarrow$ *Values*	$c.z_1$	`>=`*.30*
($c \in$ {`<`, `>`, `<=`, `>=`})		
Relation + *Values* $\rightarrow$ *Records*	$z_1.z_2$	`HasVenue.Finland`
Relation + *Records* $\rightarrow$ *Values*	$\mathbf{R}[z_1].z_2$	`YearOf.(HasVenue.Finland)`
Records $\rightarrow$ *Records*	`HasNext`.z_1	`HasNext.(HasVenue.Finland)`
Records $\rightarrow$ *Records*	`NextOf`.z_1	`NextOf.(HasVenue.Finland)`
Values $\rightarrow$ *Atomic*	$A(z_1)$	`count(HasVenue.Finland)`
($A \in$ {`count`, `max`, `min`, `sum`, `avg`})		
Values $\rightarrow$ *ROOT*	z_1	
	Union + Intersection	
Entity + *Entity* $\rightarrow$ *Values*	$z_1 \sqcup z_2$	`Finland` $\sqcup$ `Germany`
Records + *Records* $\rightarrow$ *Records*	$z_1 \sqcap z_2$	`HasPosition.1st` $\sqcap$ `HasEvent.Relay`
	Superlative over rows	
(A *RecordFn* $\lambda r[f(r)]$ is a function that maps row nodes r into comparable values)		
Relation $\rightarrow$ *RecordFn*	$\lambda r[\mathbf{R}[z_1].r]$	λr[`NumOf.TimeOf`.r]
Records + *RecordFn* $\rightarrow$ *Records*	$S(z_1, z_2)$	`argmax(allRows,` λr[`NumOf.TimeOf`.r]`)`
($S \in$ {`argmax`, `argmin`})		`argmin(HasPosition.1st,` λr[`IndexOf`.r]`)`
	Arithmetic	
(A *ValueFn* $\lambda v[f(v)]$ is a function that maps values v (cells or atomic values) into comparable values)		
Relation $\rightarrow$ *ValueFn*	$\lambda v[A(z_1.v)]$	λv[`count(HasEvent.`v`)`]
Relation + *Relation* $\rightarrow$ *ValueFn*	$\lambda v[\mathbf{R}[z_1].z_2.v]$	λv[`NumOf.TimeOf.HasEvent.`v]
ValueFn + *Values* + *Values* $\rightarrow$ *Values*	`sub`$(z_1.z_2, z_1.z_3)$	`sub(count(HasEvent.400m), count(HasEvent.Relay))`

Table 5.3: Compositional deduction rules. Each rule $c_1, \ldots, c_k \rightarrow c$ takes logical forms $z_1, \ldots, z_k$ constructed over categories $c_1, \ldots, c_k$, respectively, and produces a logical form based on the semantics.

- Apply a terminal rule of the form

  $$\textit{TokenSpan}[s] \rightarrow c[f(s)]$$

  on the token span $s = x_{i:j}$ to get logical forms $z \in f(s)$.

- Apply a compositional rule of the form

  $$c_1[z_1] + c_2[z_2] \rightarrow c[g(z_1, z_2)]$$

  on z_1 from cell (c_1, i, k) and z_2 from cell (c_2, k, j) (for some $k \in \{i, \ldots, j-1\}$) to get a logical form $z = g(z_1, z_2)$

- Apply a compositional rule of the form

$$c_1[z_1] \rightarrow c[g(z_1)]$$

on another logical form z_1 from the same cell (c, i, j) to get $z = g(z_1)$. To avoid an infinite loop, one can either design the deduction rules to not have loops, or use heuristics to detect and stop loops.

In any case, the parse is an abstract object containing the constructed logical form z, plus any other metadata necessary to score the logical form (e.g., when applying Rule 5.4.2, we can store the child parses where z_1 and z_2 come from). To restrict the search space to a reasonable size, *beam search* is usually employed: the populated cells are pruned down to some fixed number of parses that have the highest scores. The parses in cell $(\textit{ROOT}, 0, n)$ form the set $\mathcal{Z}_x$ of final logical forms.

Challenges. Chart parsing found success in syntactic parsing, where all words end up participating in the parse, and the parse forms a proper tree with no reordering. In contrast, for our semantic parsing task, the chart parser is restrictive for several reasons:

- While each parse in chart parsing belongs to some token span $x_{i:j}$, some semantic predicates do not naturally align to a token. Consider the following question on a table about Olympic games:

"*Greece held its last Summer Olympics in which year?*"
$\texttt{DateOf.YearOf.argmax}(\texttt{HasCountry.Greece}, \lambda r[\texttt{IndexOf}.r])$

While the cell node `Greece` can be generated from the word "*Greece*", some logical form predicates such as `Country` cannot be aligned to a token span. We could potentially learn to generate the whole `HasCountry.Greece` from "*Greece*", but this requires writing more custom deduction rules.

- Conversely, some token does not align with a semantic predicate. In the example above, "*Summer*" and "*Olympics*" do not manifest in the logical form.

- Finally, the order of question tokens and the logical form predicates may not align well. In the example above, the word "*year*" comes last, but the predicate `Year` comes first.

Previous semantic parsing work addresses the challenges above by using a *lexicon* that maps utterance phrases to corresponding logical form fragments (Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2010, 2011; Berant et al., 2013). The lexicon can either be manually specified or jointly learned with the parser. However, with an open-domain knowledge source with unknown data schema, such as a random web table, the lexicon is unlikely to have coverage over the unseen relations.

We instead opt for a more general solution: we allow some predicates to be constructed from other sources than the utterance. The relation between the utterance and the predicate is instead softly captured by the scoring function. As a result, our *floating parser* can achieve a high coverage over logical forms even for tables with unseen data schema.

Floating parser. As motivated above, the floating parser does not require logical form predicate to be constructed from utterance tokens. We replace the cells (c, i, j), with *floating cells* (c, s), which will store logical forms of category c that have size s. The size is measured as the number of compositional rules applied during the construction of the logical form, and not the number of logical form predicates.

The deduction rules now operate as follows:

- A terminal rule of the form
$$\textit{TokenSpan}[s] \rightarrow c[f(s)]$$
constructs $z \in f(s)$ in cell $(c, 0)$.
- A terminal rule of the form
$$\varnothing \rightarrow c[f()]$$
constructs $z \in f()$ in cell $(c, 0)$. This allows logical form predicates to be generated out of thin air.
- A compositional rule of the form
$$c_1[z_1] + c_2[z_2] \rightarrow c[g(z_1, z_2)]$$
takes z_1 from cell (c_1, s_1) and z_2 from cell (c_2, s_2) to construct $z = g(z_1, z_2)$ in cell $(c, s_1 + s_2 + 1)$.
- A compositional rule of the form
$$c_1[z_1] \rightarrow c[g(z_1)]$$
takes z_1 from cell (c_1, s_1) to construct $z = g(z_1)$ in cell $(c, s_1 + 1)$.

Figure 5.4 shows an example parse generated by our floating parser. After populating the cells up to some maximum size $s = s_{\max}$, the parses in cell $(\textit{ROOT}, s)$ for all sizes s are compiled into the set $\mathcal{Z}_x$ of final logical forms.

Pruning. The floating parser is very flexible: it can skip tokens, generate tokens out of thin air, and combine logical forms in any order. This flexibility might seem too unconstrained, but we can use several techniques to prevent bad logical forms from being constructed:

- **Denotation-based pruning.** Constructed logical forms can be executed on the knowledge graph w to get denotations. We prune logical forms that execute to an empty set (e.g., `HasYear.HasNum.`*1*). While it is tempting to prune logical forms that do not execute, care must be taken since some constructed logical forms are partial and are meant to be used as an argument of a larger logical form (e.g., logical forms of category *ValueFn* are meant to be used in superlative logical forms).
- **Heuristic pruning.** Some logical forms can be properly executed, but are unlikely to be relevant for answering the questions. For instance, a union of objects from different columns (e.g., `5th␣Germany`)

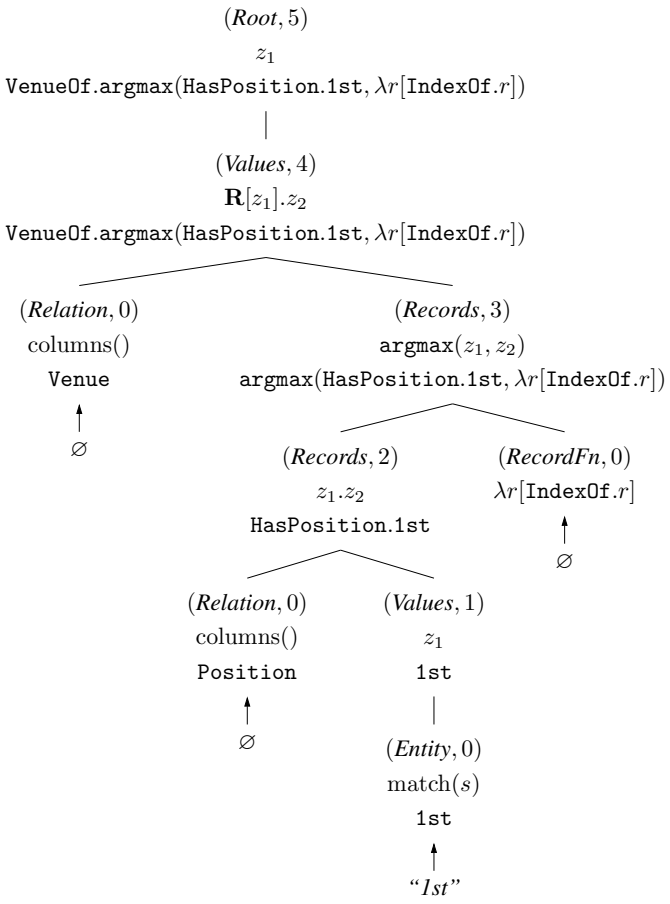


Figure 5.4: A derivation tree for the utterance "*Where did the last 1st place finish occur?*". Each tree node shows the cell (c, s), the semantic function used, and the resulting logical form. Among the terminal rule applications (indicated by arrows), only `1st` is generated from a phrase "*1st*" using the match function; other predicates are not generated based on the utterance.

are less likely to be reflecting what the question asks. We use several heuristics listed in Table 5.4 to prune logical forms. Note that some heuristics could potentially prevent us from answering some questions (e.g., preventing joining two `Next` predicates could prevent us from answering "*What comes before the person before . . . ?*"). However, these questions are rare enough, and pruning these patterns would prune a large portion of the search space, which increases speed and prevents overfitting to incorrect logical form patterns.

- **Score-based pruning (beam search).** Even with the pruning strategies above, the set of possible logical forms in each set might still be large. To control the search space, we employ beam search by scoring the logical forms and only keep the $B = 50$ highest-scoring logical forms in each cell.

Criterion for pruning a logical form z	**Example**
z cannot be executed or executes to an empty set	`HasVenue.1st`
z contains a relation joined with its inverse	`PositionOf.HasPosition.1st`
z contains a join of two `Next` predicates	`NextOf.NextOf.HasPosition.1st`
z is a union or intersection of identical logical forms	`1st` $\sqcap$ `1st`
z is a union or intersection of values with different types (Cells from different columns are considered different types.)	`5th` $\sqcup$ `Germany`
z is a superlative on a set of size 1	`argmax`(`HasIndex.`*1*, ...)
z contains multiple superlatives	`argmax`(...) $\sqcap$ `argmin`(...)
z is a *ROOT* logical form with only a single predicate	`1st` (at category *ROOT*)
z is a *ROOT* logical form whose denotation contains > 10 values	

Table 5.4: Heuristics for controlling the search space of the floating parser.

5.5 Scoring logical forms

Each logical form z is associated with a score $s_\theta(z, x, w)$. We adopt a linear model with features that capture the relationship between the question x and the logical form. Concretely,

$$s_\theta(z, x, w) := \theta^\top \phi(z, x, w) \tag{5.12}$$

where $\phi(z, x, w)$ is a feature vector. Table 5.5 shows example features from each feature type described below:

- **phrase-predicate**: For each n-gram p_x from the question x (up to 3-grams) and a predicate p_z from z, we define a lexicalized indicator feature "phrase $= p_\text{x}$, predicate $= p_\text{z}$". These features capture the correspondence between phrases and either close-classed predicates (e.g., "*last*" correlates with `argmax`) or frequent context-dependent predicates (e.g., "*when*" correlates with `Year`).

  Additionally, to handle open-domain predicates from the table, we define an unlexicalized feature "phrase = predicate" when the phrase p_x matches the string form of p_z. The unlexicalized features can also have more specific details based on substring matching, part-of-speech tag of the phrase, and the predicate type (e.g., "phrase = suffix of predicate, POS = W N, predicate type = entity").

- **missing-predicate**: We define unlexicalized indicator features "missing entity" and "missing relation" when there are entities or relations mentioned in x that are not present in z.

- **denotation**: From the denotation $y = [\![z]\!]_w$, we use its size (i.e., number of elements in the set y) and its type to define unlexicalized features. The type can be a primitive type (e.g., NUM, DATE) or the column containing the values in y in case those values are cells.

- **phrase-denotation**: For each n-gram p_x from the question x, we define a lexicalized feature "phrase =

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

$x =$ "*The last 1st place finish is in which year?*"
$z =$ `NumOf.YearOf.argmax`$($`allRows`$, \lambda r[$`IndexOf`$.r])$
$y = \{$*2008*$\}$ (type: NUM, column: `Year`)

Feature Name	Note
phrase-predicate	
phrase $=$ "*last*", predicate $=$ `argmax`	lexicalized
phrase $=$ predicate, type $=$ relation	unlexicalized ($\cdot,\cdot$ "*year*" $=$ `Year`)
missing-predicate	
missing entity	unlexicalized ($\cdot,\cdot$ missing "*1st*")
denotation	
denotation type $=$ NUM	
denotation column $=$ YEAR	
phrase-denotation	
phrase $=$ "*which year*", denotation type $=$ NUM	lexicalized
phrase $=$ denotation column	unlexicalized ($\cdot,\cdot$ *"year"* $=$ `Year`)
headword-denotation	
$Q =$ "*which*", denotation type $=$ NUM	lexicalized
$H =$ "*year*", denotation type $=$ NUM	lexicalized
$H =$ denotation column	unlexicalized ($\cdot,\cdot$ "*year*" $=$ `Year`)

Table 5.5: Example features defined for the (incorrect) logical form z. All features are binary features.

p_{x}, denotation type $= p_{\text{y}}$" where p_{y} is the type of y. Like the phrase-predicate features, we also define an unlexicalized feature "phrase $=$ denotation column" when the type p_{y} is a column whose string matches p_{x}. Like phrase-predicate features, the unlexicalized feature can be refined by substring match and part-of-speech tags.

- **headword-denotation**: From the part-of-speech tags of the question x, we deterministically identify the question word q_{x} (e.g., *who*, *when*, *what*) and the head word h_{x} (the first noun after the question word). We then define lexicalized features joining "denotation type $= p_{\text{y}}$" with "$Q = q_{\text{x}}$", "$H = h_{\text{x}}$", or both. We also define unlexicalized features if q_{x} or h_{x} matches the column string of the denotation.

	dev		test	
	acc	**ora**	**acc**	**ora**
IR baseline	13.4	69.1	12.7	70.6
WQ baseline	23.6	34.4	24.3	35.6
Floating parser	37.0	76.7	37.1	76.6

Table 5.6: Accuracy (acc) and oracle scores (ora) on the development sets (3 random splits of the training data) and the test data.

5.6 Experiments

We develop our model on three 80:20 splits of the training portion of the WIKITABLEQUESTIONS dataset (14,152 examples), where we report the average of three evaluation scores. For the final evaluation, we train on the training portion and test on the "unseen" test portion (4,345 examples).

5.6.1 Main evaluation

The main evaluation metric is *accuracy*: the fraction of test examples where the parser predicts the correct denotation (in other words, the highest-ranking logical form is consistent with the correct answer). We also report the *oracle* score: the fraction of test examples where the at least one of the logical forms in the final candidate list $\mathcal{Z}_x$ is consistent with the correct answer.

Model details. We train the model using AdaGrad (Duchi et al., 2010) with an initial learning of 1.0. For the experiments in this chapter, we use the log-likelihood objective (Equation 5.4) and lazy L1 regularization with coefficient 0.001. We take 3 passes over the training data. The beam size is set to $B = 200$.

Baselines. We compare our systems to two baselines:

- **Information retrieval baseline (IR)**: The IR baseline selects a cell y among the table cells by applying a log-linear model over the cells. The features are conjunctions of the phrases of x and the properties of y, which covers all features of our parser that do not depend on the logical form.
- **WEBQUESTIONS baseline (WQ)**: We restrict the logical form operators to the ones present in the previous semantic parsing work on the WEBQUESTIONS dataset (Berant et al., 2013). This includes the join and `count` operators.

Main results. Table 5.6 shows the accuracy and oracle scores of the floating parser and the baseline systems. Our parser outperforms the baselines by a significant margin. In the following sections, we use one of the development splits of the training data to analyze various aspects of the WIKITABLEQUESTIONS dataset and the floating parser.

5.6.2 Error analysis

The error on the development data can be divided into the following categories:

Unhandled question types (21%). Due to our choices of knowledge graph representation and logical form syntax, some questions in the dataset cannot be answered with logical forms. The majority of them are:

- Questions with incorrect annotations.
- Yes-no questions (e.g., "*is the are of saint helena more than that of nightingale island?*"). Our logical formalism does not support boolean values.
- Questions with the word "*same*" or something similar (e.g., "*which players played the same position as ardo kreek?*"). The answer needs to exclude the name mentioned in the question, which is not doable with our current set of deduction rules.
- Questions with the word "*consecutive*" or something similar (e.g., "*how many consecutive friendly competitions did chalupny score in?*"). The concept of *consecutive* records cannot be computed without augmenting the knowledge graph (e.g., with `consecCompetition` values tallying the number of consecutive repeated values so far in the *Competition* column).

Failure to match cells (25%). The cell predicates and atomic values in the logical forms are created from two terminal rules:

$$\textit{TokenSpan}[s] \rightarrow \textit{Entity}[\mathrm{match}(s)]$$
$$\textit{TokenSpan}[s] \rightarrow \textit{Atomic}[\mathrm{value}(s)]$$

While $\mathrm{match}(s)$ uses string matching to identify the cell from the token s, sometimes the question does not use the exact string from the cell (e.g., "*Italian*" referring to `Italy`, or "*no zip code*" referring to empty cells). While it is possible to generate cell predicates with a rule of the form $\varnothing \rightarrow \textit{Entity}[f(s)]$ like how the column predicates are generated, such a rule would explode the search space as most table has a large number of cells. On the other hand, $\mathrm{value}(s)$ interprets the utterance token span as a single cell value, and thus cannot represent a set of multiple dates (e.g., "*1980s*" should match all years from 1980 to 1989).

Complex cell content (29%). While we use normalization edges to handle different interpretation of the cell string, we observe several types of strings we cannot handle. Some of these include times (e.g., "*1:50.81*") and multi-part strings (e.g., "*Belo Horizonte, Brazil*", "*Brazil v Germany*", or the scores "*7-1*").

Ranking errors (25%). Finally, we have ranking errors where the consistent logical form is scored lower than the top logical form. The most common cause is rare column names that are not mentioned directly in

	acc	ora
Our system	37.0	76.7
(a) **Feature Ablation**		
all − features involving predicate	11.8	74.5
all − phrase-predicate	16.9	74.5
all − lex phrase-predicate	17.6	75.9
all − unlex phrase-predicate	34.3	76.7
all − missing-predicate	35.9	76.7
all − features involving denotation	33.5	76.8
all − denotation	34.3	76.6
all − phrase-denotation	35.7	76.8
all − headword-denotation	36.0	76.7
(b) **Anchor operations to trigger words**	37.1	59.4
(c) **Rule Ablation**		
join only	10.6	15.7
join + count (= WQ baseline)	23.6	34.4
join + count + superlative	30.7	68.6
all − $\{\sqcap, \sqcup\}$	34.8	75.1

Table 5.7: Average accuracy and oracle scores on development data in various system settings.

the question (e.g., "*airplane*" not matching the column header `Model`). A model that incorporates continuous word representations could potentially reduce this type of errors.

5.6.3 Ablation analysis

Effect of features. Table 5.7(a) shows the development accuracy when a subset of features are ablated. The most important features are the lexicalized phrase-predicate features, which learn the direct association between words from the questions and logical form predicates (e.g., associating "*last*" to `argmax`, or associating "*who*" with the column `Name`).

Table 5.8 shows the features whose parameter weights have high magnitude. We observe that the model indeed learns to associate question phrases with either built-in logical predicates or common column names. It also learns some biases in the dataset; for instance, a numerical answer is less likely to be zero or negative.

Associating operators with tokens. In our floating parser, built-in edges (e.g., `Index`, `Next`, `Num`) and logical operators (e.g., `count`, `argmax`) are not generated based on the question tokens, but the model ends up learning the association from these predicates to the tokens. As an experiment, we consider an alternative approach where these predicates are explicitly generated based on some set of "trigger" phrases. Based on the training data, we manually specify the trigger phrases for some logical predicates as listed in Table 5.9. We then change the deduction rules so that the predicates can be constructed only when one of the phrases is

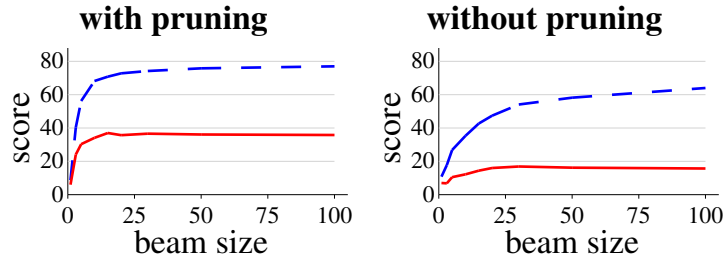


Figure 5.5: Accuracy (solid red) and oracle (dashed blue) scores with different beam sizes.

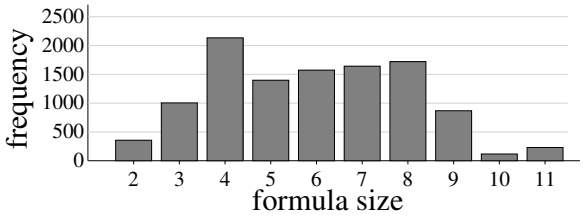


Figure 5.6: Sizes of the highest-scoring correct candidate logical forms in development examples.

present in the question.

As an explicit prior, the trigger words help decrease the number of generated logical forms (before pruning to beam size). However, the result in Table 5.7(b) shows that trigger words do not significantly affect the accuracy, suggesting that our floating parser can successfully learn the association between phrases and logical operators without requiring a lexicon.

Effect of beam size. Figure 5.5 shows the accuracy and oracle scores as the beam size changes. A lower beam size increases efficiency and prevents bad logical forms from clogging up the beam, but when the beam size gets too low, the accuracy and oracle scores decrease.

5.6.4 Additional dataset analysis

Using the trained parser, we continue our investigation from Section 4.3 and empirically analyze several aspects of the WIKITABLEQUESTIONS dataset.

Logical form coverage. The WIKITABLEQUESTIONS dataset contains various types of reasoning to answer the questions. Table 5.7(c) shows the drop in accuracy and oracle scores when only a subset of deduction rules are allowed. The *join only* subset corresponds to table lookup, while the *join* + *count* subset covers the same scope of logical forms as the previous work on the WEBQUESTIONS dataset (Berant et al., 2013). Finally, the *join* + *count* + *superlative* subset roughly corresponds to the coverage of the GEOQUERY dataset (Zelle and Mooney, 1996).

Compositionality. The histogram in Figure 5.6 tracks the size of the logical forms (as the number of compositional deduction rules) that are consistent with the correct answer. If there are consistent multiple logical forms in the candidate set $\mathcal{Z}_x$, we choose the shortest one. The histogram shows that a significant number of logical forms have non-trivial sizes.

5.6.5 True oracle score

From Table 5.6, our parser successfully generates a logical form consistent with the correct denotations (but may or may not be the highest-scoring logical form) in 76.7% of the development examples. Based on the large gap between the accuracy (37.0%) and this number, one might believe that the accuracy can be improved mainly with a better scoring model. Unfortunately, the oracle score is misleading due to the existence of *spurious logical forms* that give the right answer for wrong reasons. For instance, in our running example (Figure 5.1), consider the spurious logical form

$$\texttt{VenueOf.argmax}(\texttt{HasPosition.1st}, \lambda r[\texttt{NumOf.TimeOf}.r]). \tag{5.13}$$

From $[\![\texttt{HasPosition.1st}]\!]_w = \{r_2, r_4\}$, the logical form picks the row with the highest *Time* instead of the highest index, but eventually arrives at the correct denotation $\{\texttt{Thailand}\}$. We found that in many examples, the parser only found spurious logical forms and not a semantically correct one.

To measure the true oracle score, we sample 300 examples and manually annotate them with semantically correct logical forms, and see if the trained system can generate the annotated logical form as a candidate.[3] Out of 300 examples, we find that 84% can be manually annotated, while the rest are unhandled questions as explained in Section 5.6.2.

The system successfully generates the annotated logical forms in only 53.5% of the examples. This indicates that the main method to improve the parser is to increase the *coverage* over answerable questions, such as by augmenting the knowledge graph and expanding deduction rules. We will explore this direction of increasing coverage, along with the efficiency problem it entails, in the next chapter.

5.7 Related work and discussion

Logical form generation. Apart from the bottom-up generation used in our floating parser, previous work has modelled the process of generating logical forms with various paradigms.

Like the logical form structure, the syntactic parse of a sentence is also hierarchical. As such, previous work has considered using syntactic structures to guide the generation of semantic parses. For instance, some take the syntactic parse from an external parser and convert it into a semantic parse (Poon, 2013; Reddy et al., 2016). Others learn a joint model for syntactic and semantic parses. One popular example is the use of

[3] This method ignores generated logical forms that are semantically equivalent to the annotated logical forms. Chapter 7 will present a more precise method that takes logical form equivalency into account.

Combinatory Categorial Grammar (Steedman, 1996, 2000), or CCG, which can capture syntax and semantics jointly (Zettlemoyer and Collins, 2005, 2007; Kwiatkowski et al., 2010, 2011).

Without using syntactic structures, one can also learn to decode the semantic parse directly. We follow the line of work that uses bottom-up generation (Berant et al., 2013; Berant and Liang, 2014, 2015). One benefit of bottom-up generation is that the intermediate parses are complete logical forms that can be executed, and the resulting denotations can be used to prune unpromising logical forms (Wang et al., 2018) or as additional information to the scoring model (Guu et al., 2017).

Early *neural* semantic parsing approaches treat logical form generation as a sequence prediction problem and just generate the predicates from left to right (Jia and Liang, 2016). This works well for generating canned idiomatic expressions, but the resulting sequences are not guaranteed to be valid logical forms. A popular alternative is top-down generation, where at each recursion step, the parser first selects an operator and then recursively builds the arguments as child subtrees (Dong and Lapata, 2016; Krishnamurthy et al., 2017; Rabinovich et al., 2017; Cheng et al., 2017). One benefit of the top-down approach is that the information from the parent node is properly propagated to the corresponding children subtrees, even when the subtree is generated many steps after the parent operator is generated.

Hard mapping versus soft mapping. Early semantic parsing work assumes a strict mapping between words in the utterance and predicates in the logical forms. This mapping is often called a *lexicon*, which could be hand-specified (Zelle and Mooney, 1996; Unger and Cimiano, 2011; Unger et al., 2012), learned separately from the data (Cai and Yates, 2013; Berant et al., 2013), or learned jointly while training the parser (Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2010, 2011). While this works well for a specific domain, learning such a lexicon for open-domain knowledge sources is difficult due to the open set of phrases and predicates. Previous work addressed this using factorized lexicon (Kwiatkowski et al., 2011) or by translating the lexicon output to match the target domain on the fly (Kwiatkowski et al., 2013).

Our floating parser takes a similar approach to the free-form generation of logical forms in Berant and Liang (2014). Instead of a strict mapping, we try out different logical form predicates and uses the scorer to rank them. This idea of soft mapping has become the norm in neural-based parsers (Jia and Liang, 2016; Dong and Lapata, 2016; Krishnamurthy et al., 2017), where the generation of each predicate is guided by the embedding of the input, while the attention mechanism (Bahdanau et al., 2015; Luong et al., 2015) is used as a soft alignment between certain utterance tokens and the generated logical predicate. Open-domain predicates (e.g., column and cell nodes in our case) are generated using explicit copy mechanism (Gu et al., 2016; Jia and Liang, 2016), which plays a similar role to our terminal rules.

Table normalization. One challenge with working on web tables is that the table format is not standardized. This is surprisingly severe on Wikipedia: while there are some general guidelines, Wikipedia editors end up using the most suitable table format for each article.

One way to handle the different table formats is to convert them into canonical formats. For instance, different table formatting can be canonicalized to database-like formats (Embley et al., 2016). Entities in the

table can be linked to canonical entities in some structured knowledge source (Limaye et al., 2010; Bhagavatula et al., 2015). Numerical cells can be parsed into the correct values and units (Madaan et al., 2016). And missing data can be inferred from an aggregation of multiple tables (Venetis et al., 2011). The canonicalization makes it easier to perform computation on the data, but it requires designing a good canonical schema. Moreover, the conversion process is noisy and might introduce errors for ambiguous cell contents. For instance, the string "*1990*" is usually tagged as a date by a named entity recognition (NER) tagger, but could be a plain number or a proper noun in some contexts.

Another way to handle tables is to extract just the necessary information from the table (Govindaraju et al., 2013; Zhang, 2015). The extracted information, usually a relation between two entities, can then be used to populate a database with a known schema (Ellis et al., 2015). Apart from having clean data suitable for computation, this type of extraction also allows the information from both structured data and unstructured text to be unified and reasoned on jointly. However, it precludes the use of other data available in the table that were not extracted.

Our approach is to maintain uncertainty in table interpretation via normalization edges, and then let the scorer choose the correct form of normalization. This process prevents the loss of information due to extraction. Moreover, additional ways to interpret the cells can be added dynamically, as we will explore in Chapters 6 and 7. However, this also increase the search space of logical forms, which is sometimes wasteful when the column is clearly of a particular type. For instance, the cell with text *1990* is very likely to be a year if all other cells also look like a year, and keeping around the possible `Num` interpretation wastefully expands the space of possible logical forms.

5.8 Conclusion

In this chapter, we presented a semantic parsing approach for the task of answering complex question on web tables. To handle open-domain tables with unknown schemata, we (1) use knowledge graph as a domain-generic representation of the tables, and (2) use a floating parser that can flexibly generate logical form predicates based on the table schema. The scoring function is then responsible for relating the question to the generated logical forms.

To train the model with the correct answers as distant supervision, we need to generate a set of candidate logical forms and up-weight the ones that are consistent with the annotated answers. Unfortunately, with more complex questions, the space of possible logical forms rapidly expands with the number of reasoning steps and logical operators. This makes it expensive to generate a set of candidates that contain a consistent logical form. In the next two chapters, we will look at techniques to control the search space of logical forms.

Feature type	Feature	Weight
headword-denotation	Q = "*what*", H = denotation column	5.11
missing-predicate	missing relation	–4.21
headword-denotation	Q = "*which*", H = denotation column	4.09
phrase-predicate	phrase = "*before*", predicate = `<`	4.07
phrase-predicate	phrase = "*over*", predicate = `>`	4.02
phrase-predicate	phrase = "*before*", predicate = `HasNext`	4.02
headword-denotation	Q = "*what*", H = denotation column, denotation type = DATE	3.74
phrase-predicate	phrase = "*below*", predicate = `<`	3.72
denotation	denotation is the number 0	–3.72
phrase-denotation	phrase = "*how*", denotation type = NUM	3.71
phrase-predicate	phrase = "*less*", predicate = `<`	3.71
headword-denotation	Q = "*that*", H = denotation columns	–3.70
phrase-predicate	phrase = "*each*", predicate = `Index`	–3.66
custom-denotation	denotation is a negative number	–3.55
phrase-predicate	phrase = "*last*", predicate = `argmax`	3.46
phrase-predicate	phrase = suffix of predicate, POS = W N, predicate type = entity	–3.44
phrase-predicate	phrase = "*less than*", predicate = `<`	3.38
headword-denotation	Q = "*what*", H = "*number*", denotation type = NUM	3.36
phrase-denotation	phrase = "*many*", denotation type = NUM	3.34
headword-denotation	Q = "*how many*", denotation type = NUM	3.32
phrase-predicate	phrase = "*when*", predicate = `DateOf`	3.25
phrase-predicate	phrase = "*team do*", predicate = `Index`	3.18
phrase-denotation	phrase = "*how many*", denotation type = NUM	3.17
phrase-predicate	phrase = "*list before*", predicate = `HasNext`	3.17
phrase-predicate	phrase = "*under*", predicate = `<`	3.12
phrase-predicate	phrase = "*after*", predicate = `NextOf`	3.08
phrase-predicate	phrase = "*previous*", predicate = `HasNext`	3.07
phrase-predicate	phrase = "*after*", predicate = `>`	3.05
phrase-predicate	phrase = "*from*", predicate = `DateOf`	–3.03
phrase-predicate	phrase = "*the top*", predicate = `<=`	3.00

Table 5.8: The top 30 features when sorted by the magnitude of the parameter weights. (Q = question word; H = headword)

Predicate	Triggers
⊔	and, or
`Next`	next, previous, after, before, above, below
`>`, `>=`	than, more, least, above, after
`<`, `<=`	than, less, most, below, before
`count`	how, many, total, number
`sum`	all, combine, total
`avg`	average
`sub`	difference, between, and, much
`argmax`, `argmin`	top, first, bottom, last, *any word with part-of-speech tag JJR, JJS, RBR, or RBS*

Table 5.9: To test whether the model learns to associate logical operators with tokens, we perform an experiment where some logical forms predicates must be explicitly triggered by some predefined phrases.

Chapter 6

Guiding Search with Logical Form Patterns

As mentioned at the end of the previous chapter (Section 5.6.5), we would like to increase the coverage over answerable questions by expanding the space of logical forms that can be generated. One way to achieve this is by expanding the set of deduction rules. In particular, we add new rules listed in Table 6.1, which allows cells nodes to be generated from approximate string matching (e.g., the phrase "*Chinese*" can now map to `China`), and introduces superlatives on sets of cell nodes as demonstrated in Table 6.2.

Adding just these two rules already increases the true oracle number from 53.5% by almost 10% absolute. This is because both rules directly address the major coverage errors as analyzed in Section 5.6.2. However, since the added rules can potentially generate many more candidate logical forms (e.g., there can be multiple cells approximately matching each utterance token span), the increased coverage comes with a cost of slower running time. On average per example, the training time increases from 619 ms to 1,117 ms, and the prediction time increases from 645 ms to 1,150 ms. This doubling of running time could mean additional days to train and tune a model. Even worse, the running time will increase even more when the space of logical form is expanded further to gain more question coverage.

In this chapter, we propose to speed up the parser by making the search algorithm prioritize logical forms that "look good" according to its past experience. Our approach is based on two key ideas:

- **Good logical forms share common patterns.** For example, consider the question "*Which location comes after Germany?*" for the table in Figure 6.1. One possible semantically correct logical form is

  `VenueOf.NextOf.HasVenue.Germany`, (6.1)

  which identifies the cell below *Germany* in the same *Venue* column. From this logical form, we can abstract out question-specific and table-specific predicates (i.e., columns, cells, and primitive values)

Rule	Semantics	Example
TokenSpan $\rightarrow$ *Entity*	fuzzymatch(s)	`China` (from "*Chinese*")
(fuzzymatch(s) generates cell nodes with string approximately matching s)		
Values + *ValueFn* $\rightarrow$ *Values*	$S(z_1, z_2)$	`argmax(EventOf.allRows,` λv[`count(HasEvent.`v`)`]`)`
(A *ValueFn*, as defined in Table 5.3, maps cells or atomic values into comparable values)		

Table 6.1: Additional deduction rules to increase the coverage on answerable questions.

"*Which driver appears the most?*"	`argmax(DriverOf.allRows,` λv[`count(HasDriver.`v`)`]`)`
"*Is English for French spoken more?*"	`argmax(English` $\sqcup$ `French,` λv[`count(HasLanguage.`v`)`]`)`
"*Who is taller, Rose or Time?*"	`argmax(Rose` $\sqcup$ `Tim,` λv[`NumOf.HeightOf.HasName.`v]`)`

Table 6.2: Examples of superlative logical forms applied on sets of cell nodes.

to get the following *macro*:

$$\{Col\#1\}\texttt{Of.NextOf.Has}\{Col\#1\}.\{Cell\#2\}, \tag{6.2}$$

which identifies the cell below $\{Cell\#2\}$ in the same $\{Col\#1\}$ column. Such a macro captures a domain-independent computational pattern that generalizes across different table contexts. The main idea of our training procedure is to extract macros from logical forms found from search, and then use them to efficiently construct logical forms in new contexts.

- **Similar utterances are likely to share a logical form pattern.** Though the space defined by macros is smaller than the original search space, the set of extracted macros will eventually grow with the number of processed examples. To prioritize macros that are more likely to match for the input question x, we propose *holistic triggering*: find the K examples with questions most similar to x, and then only test macros extracted from the consistent logical forms found in those examples.

Based on the two ideas above, we propose an online learning algorithm that jointly learns a semantic parser and a set of macros encoded as *macro deduction rules*. For each training example, the algorithm first tries to find consistent logical forms by using holistic triggering to invoke a subset of promising macro rules to apply. If it succeeds, the logical forms are used to update the parameters of the semantic parser. Otherwise, it falls back to the base deduction rules to perform a more exhaustive search, and then uses the discovered consistent logical forms to derive more macro deduction rules. At test time, we only use the learned macro deduction rules.

The next two sections describe the details of macro deduction rules and the training algorithm. Afterward, we evaluate our approach on the WIKITABLEQUESTIONS dataset in terms of accuracy and speed. Training with macro rules yields a 11x speedup compared to training with the base deduction rules, and at test time, we achieve a slightly better accuracy of 43.7% with a 16x speedup compared to the baseline.

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

$x =$ "*Which location comes after Germany?*"
$z =$ `VenueOf.NextOf.HasVenue.Germany`
$y =$ {`Thailand`}

Figure 6.1: Our running example from before but with a different question.

Reference. The results described in this chapter have been published as Zhang et al. (2017). Reproducible experiments are hosted on the CodaLab platform at

https://worksheets.codalab.org/worksheets/0x4d6dbfc5ec7f44a6a4da4ca2a9334d6e.

6.1 Macros

6.1.1 Deriving macros from logical forms

A *macro* M characterizes an abstract logical form structure. For any given logical form z, we can compute its macro by transforming its derivation tree as illustrated in Figure 6.2:

1. For each terminal deduction rule (i.e., leaf node), substitute it with a placeholder, and label it with the category of that derivation (i.e., the category on the right hand side of the deduction rule).

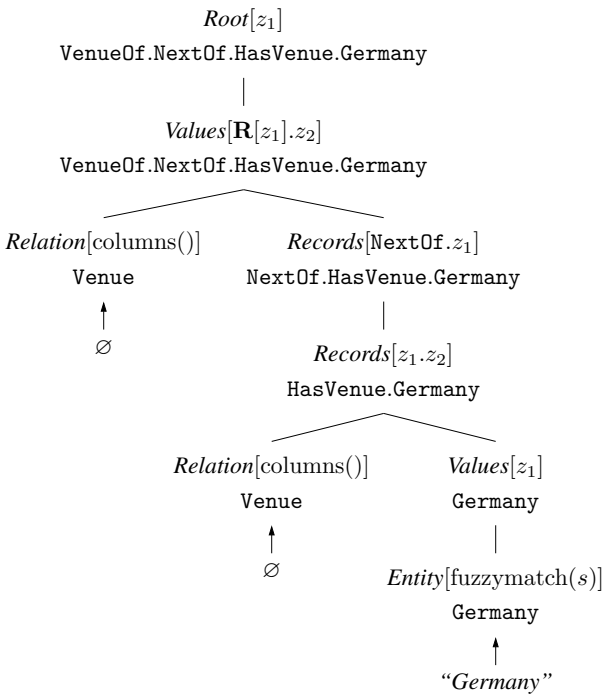
2. Merge leaf nodes that represent the same partial logical forms. For instance, the two mentions of `Nation` in Figure 6.2b are merged to signify that the two column names have to be identical.

The resulting macro M can be serialized as a flat string by assigning an index to each placeholder as illustrated in the figure. While the resulting macro is not a tree, we will use the terms *root* and *leaf* to refer to nodes that were roots or leaves in the original derivation tree.

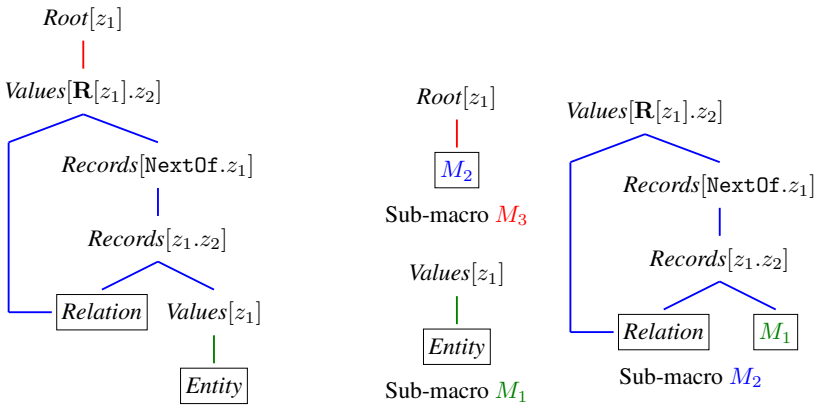
A logical form z' satisfies the macro if it can be obtained by filling each placeholder in M with a value of the matching category. For instance, the logical form $z' =$ `YearOf.NextOf.HasYear.1st` satisfies the macro M in the figure (even though it executes to an empty set).

6.1.2 Macro deduction rules

For any macro M, we can construct a set of *macro deduction rules* that, when combined with terminal rule from the original set of deduction rules (called "base deduction rules" henceforth), generates exactly the



(a) Derivation tree (z_i represents the ith child)



(b) Macro (c) Atomic sub-macros

Figure 6.2: From the derivation tree (a), we extract a macro (b), which can be further decomposed into atomic sub-macros (c). Each sub-macro is converted into a macro rule.

logical forms that satisfy the macro M. The most basic approach is to construct a single rule for the whole macro: if the placeholders of M are labeled $c_1, \ldots, c_k$, we construct a deduction rule

$$c_1[z_1] + \cdots + c_k[z_k] \rightarrow \textit{Root}[f(z_1, \ldots, z_k)] \tag{6.3}$$

where $f(z_1, \ldots, z_k)$ is a semantic function that substitute $z_1, \ldots, z_k$ to the corresponding placeholder nodes of M. For example, the macro in Figure 6.2b yields a deduction rule

$$\textit{Relation}[z_1] + \textit{Entity}[z_2] \rightarrow \textit{Root}[\mathbf{R}[z_1].\texttt{NextOf}.z_1.z_2] \tag{6.4}$$

Decomposed macro deduction rules. To speed up the model, we want to avoid processing the same logical form fragment more than once. For instance, when we use macro deduction rules to generate

$$\texttt{max}(\texttt{NumOf.YearOf.HasPosition.HasNum}.\textit{1}) \tag{6.5}$$

and

$$\texttt{VenueOf.argmin}(\texttt{HasPosition.HasNum}.\textit{1}, \lambda r.\texttt{IndexOf}.r), \tag{6.6}$$

we want to avoid constructing and featurizing the same fragment $\texttt{HasPosition.HasNum}.\textit{1}$ multiple times. To achieve this goal, we decompose macros M into *sub-macros* and define deduction rules from them. A subgraph M' of M is a sub-macro of M if

1. M' contains at least one non-leaf node
2. M' connects to the rest of the graph ($M \setminus M'$) only through the root of M'.

A macro M is *atomic* if its only sub-macro is itself. The process of decomposing a non-atomic macro M, as illustrated in Figure 6.2c, is as follows:

1. Detach an atomic sub-macro M' from M and define a macro rule

$$c'_1[z_1] + \cdots + c'_k[z_k] \rightarrow c'_{\text{out}}[f(z_1, \ldots, z_k)] \tag{6.7}$$

   where $c'_1, \ldots, c'_k$ are the categories of the leaf nodes of M', and $f(z_1, \ldots, z_k)$ is a semantic function that substitutes $z_1, \ldots, z_k$ into the macro M'. The name of the category c'_{out} is computed by serializing M' as a string; this way, if the sub-macro M' appears in a different macro, the category name will be shared, and any logical form satisfying M' will only be constructed once.

2. Substitute the subgraph M' in M by a placeholder node with name c'_{out}.

3. Repeat Steps 1 and 2 on the new graph until we are left with an atomic macro, and define a final macro deduction rule for it.

Algorithm 1 Process a training example with macro rules

Require: training example (x, w, y), macro rules, base rules with terminal rules $\mathcal{T}$
1: Select a set $\mathcal{R}$ of macro rules based on x with holistic triggering
2: Generate a set $\mathcal{Z}_x$ of candidate logical forms from rules $\mathcal{R} \cup \mathcal{T}$
3: **if** $\mathcal{Z}_x$ contains consistent logical forms **then**
4: Update model parameters
5: **else**
6: Apply the base rules to search for a consistent logical forms
7: Augment the macro rules
8: Associate utterance x with the highest-scoring consistent logical form found

6.2 Training algorithm

We now describe an online algorithm that jointly learns the semantic parser and macro deduction rules. Algorithm 1 describes how our algorithm proceeds for each training example (x, w, y). It first tries to use a subset of macro deduction rules to search for logical forms. If the search succeeds, then the semantic parser parameters are updated as usual. Otherwise, it falls back to the base deduction rules, and then add new macro rules based on the consistent logical forms found.

6.2.1 Holistic triggering

The first step of our algorithm is to select a subset $\mathcal{R}$ of macro rules that are likely to generate consistent logical forms for the question x. The selection is done as follows. Throughout training, we maintain a mapping $\mathcal{S}$ from each previously processed question to a consistent logical form found while processing that question. (Questions for which a consistent logical form has not been found are not included.) Given a new training question x, we identify K questions in $\mathcal{S}$ that are the most "similar" to x (i.e., K-nearest neighbors of x under some similarity metric), and then let $\mathcal{R}$ be the set of macro deduction rules that were extracted from their associated logical forms.

Question similarity metric. We use token-level Levenshtein distance as the distance metric for computing the nearest neighbors. To compute the distance between two sentences x and x', we first preprocess them by lemmatizing the tokens, and then removing all determiners and infrequent nouns that appear in less than 2% of the training questions. The distance is then defined as the Levenshtein distance between the two sequences of remaining tokens. For instance, "***Who*** *ranked* ***right after*** *Germany*" and "***Who*** *took office* ***right after*** *Uriah Forrest*" have distance 4. Despite its simplicity, our distance is good at capturing structural similarity between questions.

To speed up training, we pre-compute a sorted list of $K_{\max} = 100$ nearest neighbors for every question in the training data. When processing a question x during training, we calculate $\mathcal{R}$ as the intersection of the pre-computed list for x and the set of questions in $\mathcal{S}$.

6.2.2 Updating the macro rules

The computed subset $\mathcal{R}$ of macro deduction rules is combined with the set $\mathcal{T}$ of base terminal rules (for building basic units such as *Cell* and *Col*). The floating parser parses the training question x using deduction rules $\mathcal{R} \cup \mathcal{T}$, resulting in a set $\mathcal{Z}_x$ of logical forms (Line 2 of Algorithm 1).

If $\mathcal{Z}_x$ contains a consistent logical form, we update the model parameters as usual (Line 4); otherwise, we fall back to performing beam search using the base rules (Line 6). For efficiency, we stop the search either when a consistent logical form is found, or when the total number of generated logical forms exceeds some threshold T. These early stopping criteria prevent the model from spending too much time on difficult examples (e.g., when the context table is large). While we might miss consistent logical forms and thus their macros on such examples, we could potentially induce the same macro from more straightforward examples.

When the algorithm succeeds at finding a consistent logical form z with the base rules, we extract its macro M and construct the corresponding decomposed macro rules (Line 7). Parameters of the parser are not updated when the beam search with base rules is invoked.

Finally, if a consistent logical form z is found, the mapping $\mathcal{S}$ is updated to map x to z (Line 8).

6.2.3 Prediction

At test time, we follow steps 1–2 of Algorithm 1 to generate a set $\mathcal{Z}_x$ of candidate logical forms from the triggered macro rules and base terminal rules. We then output the highest-scoring logical form $z \in \mathcal{Z}_x$. We do not fall back to beam search at test time.

6.3 Experiments

We evaluate our approach on the WIKITABLEQUESTIONS dataset. In addition to the test accuracy, we will also compare the running times during both training and test time.

Model details. We use the same features and logical form pruning strategies from Chapter 5. We use a beam size of $B = 100$ for beam search, and use $K = 40$ for the K-nearest neighbor in holistic triggering. We take 3 passes over the training data during training. When falling back to beam search, we stop the search when the number of generated logical forms reaches $T = 5000$ in the first pass. To further increase speed, we disallow falling back to beam search during the subsequent passes (which means that no more macro deduction rules are constructed after the first pass).

6.3.1 Main results

Test accuracy. Table 6.3 shows the test accuracy of our approach and other parses on the WIKITABLEQUESTIONS dataset. Compared to the original floating parser, learning with macros slightly improves the

	Dev	Test
Original deduction rules (Chapter 5)	37.0%	37.1%
New deduction rules	**40.6%**	42.7%
New deduction rules + macros	40.4%	**43.7%**

Table 6.3: Results of learning with macros on the WIKITABLEQUESTIONS dataset.

		Time (ms/ex)	
	Acc.	Train	Pred
Original deduction rules (Chapter 5)	37.0%	619	645
New deduction rules	**40.6%**	1,117	1,150
New deduction rules + macros	40.4%	**99**	**70**
no holistic triggering	40.1%	361	369
no macro decomposition	40.3%	177	159

Table 6.4: Comparison and ablation study. The columns report averaged prediction accuracy, training time, and prediction time (milliseconds per example) on the three development splits.

test accuracy from 42.7% to 43.7%, while the averaged development accuracy on three development splits are similar for the two approaches (40.6% vs 40.4%).

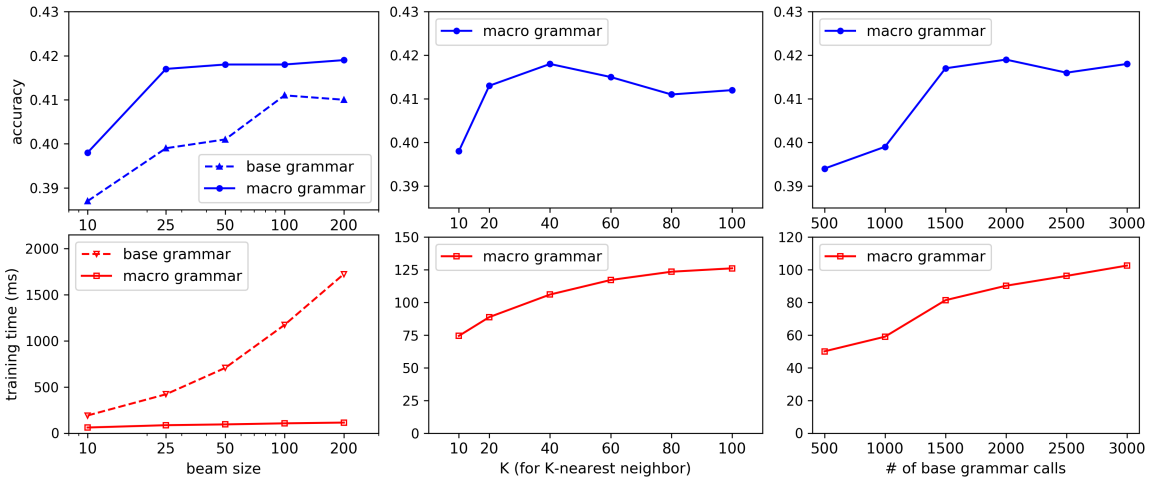
Running time. Table 6.4 lists the running time needed to process an example in the original floating parser and our new approach. We trained all parsers on a machine with Xeon 2.6GHz CPU and 128GB memory without parallelization. Learning with macros is substantially more efficient, with 11x speedup during training and 16x speedup at test time.

Ablation analysis. We run two ablations of our algorithm: removing holistic triggering (i.e., consider all macro rules when parsing a question x) and removing macro decomposition (i.e., construct a single deduction rule for each macro). Table 6.4 shows that the ablated models are slower to process examples. This is because holistic triggering greatly narrows down the set of macro rules for parsing questions, while decomposed macro rules save us from featurizing the same logical form fragments multiple times.

6.3.2 Coverage of macros

Using the base deduction rules, the floating parser generates an average of 13,700 partial logical forms for each training example, and discovers a consistent logical form in 81.0% of the examples. When learning with macros, the numbers become 1,300 and 75.6%.

At a first glance, learning with macros seems to reduce the coverage of logical forms, but this is untrue once we factor out spurious logical forms (i.e., logical forms that execute to the right denotation for wrong



(a) Varying beam size (b) Varying neighbor size (c) Varying base rules usage count

Figure 6.3: Accuracy and training time (per example) with various hyperparameter choices, reported on the first train-dev split.

reasons). To measure the coverage over semantically correct logical forms, we turn to the 300 examples that we annotated with correct logical forms in Chapter 7. We find that for 48.7% of these examples, the top consistent logical form produced by the base deduction rules is semantically correct (i.e., exactly matches or is equivalent to the annotated logical form). When learning with macros, the number stays at 48.7%, meaning that the effective coverage of the learned macro deduction rules is as good as the base ones.

6.3.3 Influence of hyperparameters

We compare the accuracy and training time per example on the first development split of the dataset when hyperparameters varies.

Beam size. Figure 6.3a shows that for all beam sizes B, training with macros is more efficient than with the base deduction rules, and the speedup rate grows with beam size. The accuracy is robust to the varying beam size as long as $B \geq 25$.

Number of neighbors in holistic triggering. We now consider K, the number of neighboring questions when computing the subset $\mathcal{R}$ of macro rules in Section 6.2.1. Figure 6.3b shows that retrieving fewer similar questions speeds up training but decreases accuracy. On the other hand, retrieving too many similar questions can overload the list $\mathcal{R}$ with potentially spurious macros, which could explain why the accuracy slightly drops when K is too high.

Number of fallback searches. We perform ablation experiments where we limit the number of times that the algorithm can fall back to beam search on base deduction rules to at most m times. This effectively limits the number of macros that the algorithm can discover: the set of macro rules will be augmented at most m times. With smaller m, the accuracy grows with m, which suggests that discovering a richer set of macros improves the accuracy.

However, the accuracy changes much more slowly once m hits 1,500. This suggests that we could speed up the model even further by stopping the model from finding macros after it has processed some fraction of the training data. Indeed, we try training a model with $B = 50$, $K = 40$, and $m = 2000$. The resulting model reduces the development accuracy very slightly (down from 40.4% to 40.2%), but it achieves more speed up during both training time (up from 11x the base floating parser to 15x) and test time (up from 16x to 20x).

6.4 Related work and discussion

Modeling recurring structures. A few generative models have been proposed to model recurring structures. Examples of these models include adaptor grammars (Johnson et al., 2006) and fragment grammars (O'Donnell, 2011), which were designed to assign higher probabilities to the structures that use the reoccurring parts. One example application of such models in semantic parsing is the idiom-based program synthesis work by Iyer et al. (2019), where the common program idioms, such as nested for-loops, can be extracted from a large library of unlabeled data. The idioms are then added to the set of possible production rules in a similar way to how we add macro rules (without decomposition). And like how our macro rules are used, the idiom can be invoked during decoding to produce a whole section for the desired program.

The main difference between our macro grammar and these generative models is that the generative models aim to improve modeling, using the intuition that learning to predict the repeating structure as a single action is easier than learning to generate all parts. In contrast, our training algorithm caches the patterns only to speed up search, while the score assignment is done by a discriminative scorer. Nevertheless, we also notice a slight modeling improvement on the test data.

Explicit mapping of phrases to logical form patterns. Previous semantic parsing work has proposed a few methods to infer explicit mappings from utterance phrases to logical form patterns. For instance, the higher-order unification method (Kwiatkowski et al., 2010, 2011) learns such a mapping in an online fashion during training. The result is a lexicon where each entry contains a logical form template and a set of possible phrases for triggering the template.

Our macros are triggered not by explicit mappings, but instead use holistic triggering to over-suggest the plausible macros to use. This is similar to how the floating parser generates some predicates independently from the utterance with a soft guidance from the scorer. This allows us to generate logical forms for utterances containing unseen lexical paraphrases. Moreover, our holistic triggering also works when the triggers is not a local contiguous part of the utterance. For instance, the question "***Who is*** *older, Jack* ***or*** *John?*" can trigger

the macro learned from "***Who is*** *taller, Rose* ***or*** *Tim?*": the logical form pattern of applying `argmax` on the union of two entities is dependent on the whole structure of the sentence.

Sketch-based and retrieval-based program synthesis. The process of producing a logical form pattern and filling in the details is closely related to sketch-based program synthesis (Solar-Lezama et al., 2005). To generate a program, the program sketch containing placeholders is first chosen, and then a synthesis algorithm (search-based or solver-based) is used to find the values for the placeholders that make the program satisfy the program specification. This technique has been recently applied in the context of neural semantic parsing (Dong and Lapata, 2018), where both the sketch and the actual logical forms are generated using neural decoders: the sketch is first generated based on the input, then a second network encodes the sketch and decodes the actual logical form. When the target logical form is available, the sketch can be directly inferred and used to train the sketch generator. In our case, we use the logical forms found by beam search to create macros (cf. sketches), and the macros in turn helps make search faster.

Another related approach is retrieval-based synthesis (Gu et al., 2017; Hayati et al., 2018; Guu et al., 2018). To generate an output for the current input x, the output y' of a similar input x' in the corpus is retrieved, and then the model uses y' to generate an output y for our input x. Unlike our work which derives and caches macros, these retrieval-based approaches can use the raw corpus without having to preprocess it.

6.5 Conclusion

We have presented an algorithm to speed up both the training and prediction time of our floating parser. Under the assumption that logical forms for similar utterances tend to share the same pattern, we use macro deduction rules to encode logical form patterns, and use holistic triggering to choose the macro rules that are the most relevant for the question. Our approach can trade off between exploitation (applying macro deduction rules) and exploration (falling back to beam search) to achieve a large speed up without sacrificing the accuracy. This allows us to efficiently expand the deduction rules and knowledge graph representation so that more types of questions can be answered.

One downside of our approach is that, as a caching mechanism, the macro rules do not kick in during the beginning of the training process. For the first few training examples, not many macro rules have been accumulated, which curtails the level of speed up seen in the final experiments. Even worse, during early training iterations, the parameters of the scoring function are still not well-calibrated. This leads to beam search pruning away essential logical form parts, and thus we are less likely to find a consistent logical form when falling back to beam search.

Chapter 7

Precomputing Semantically Correct Logical Forms

So far, we have been using the correct denotation (annotated answers) as a distant supervision for training the semantic parser. As argued in Chapter 4, using distant supervision has several benefits. Since the annotators do not have to learn the syntax of logical forms, more annotations can be gathered more cheaply. Furthermore, the annotation is independent of logical formalism, meaning that we are free to choose any logical formalism to attack the task. Finally, more diverse types of questions can be gathered since we do not limit the types of logical operations that can be performed.

Despite these benefits, training with denotation as supervision requires running a search algorithm to find consistent logical forms for parameter updates. While doing so is fine in closed-domain settings, with the increased domain size (BREADTH) and question complexity (DEPTH), it is increasingly difficult to search over the space of logical forms for two reasons:

1. **Exploding search space.** the number of possible logical forms grows exponentially with the size of logical forms. In the previous chapters, we use various techniques such as beam search, deduction rule design, pruning strategies, and macro deduction rules (Chapter 6) to control the search space. However, such techniques can prune away consistent logical forms, which could slow down training.

2. **Spurious logical forms.** Spurious logical forms are logical forms that execute to the right answer for a wrong reason. For instance, the logical forms z_1, z_2, and z_3 in Figure 7.1 are *semantically correct* as they follow what the question x asks; however, z_4 and z_5 are *spurious*: they execute to the correct answer $\{\texttt{Thailand}\}$ but do not reflect what the question x asks. While increasing the search space helps with coverage and generalization, many spurious logical forms get generated. Spurious logical forms provide deceptive signals during training. For example, questions with the phrasing "*X or Y*" tend to have a lot of spurious logical forms, and the model may not learn the correct construct $X \sqcup Y$ if it keeps updating toward spurious logical forms.

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

x: "*Where did the last 1st place finish occur?*"
y: Thailand

Consistent and semantically correct

z_1: `VenueOf.argmax(HasPosition.1st, λr[IndexOf.r])`
(Among the rows with *Position* = 1st, pick the one with maximum index and return its *Venue*.)

z_2: `VenueOf.HasIndex.max(IndexOf.HasPosition.1st)`
(Find the maximum index of rows with *Position* = 1st. Return the *Venue* of the row with that index.)

z_3: `VenueOf.argmax(HasPosition.HasNum.`*1*`, λr[DateOf.YearOf.r])`
(Among the rows with *Position* number 1, pick the one with the largest *Year*. Return its *Venue*.)

Consistent but spurious

z_4: `VenueOf.argmax(HasPosition.HasNum.`*1*`, λr[NumOf.TimeOf.r])`
(Among the rows with *Position* number 1, pick the one with the largest *Time*. Return its *Venue*.)

z_5: `VenueOf.HasYear.HasNum.sub(NumOf.YearOf.argmax(allRows, λr[IndexOf.r]),` *1*`)`
(Subtract 1 from the *Year* in the last row, then return the *Venue* of the row with that *Year*.)

Inconsistent

$\tilde{z}$: `VenueOf.argmin(HasPosition.1st, λr[IndexOf.r])` $\rightarrow [\![\tilde{z}]\!]_w = \{\texttt{Finland}\}$
(Among the rows with *Position* = 1st, pick the one with minimum index and return its *Venue*.)

Figure 7.1: Six logical forms generated from the question x. The first five are *consistent*: they execute to the correct answer y. Of those, *semantically correct* logical forms z_1, z_2, and z_3 are different ways to represent the semantics of x, while *spurious* logical forms z_4 and z_5 get the right answer y for the wrong reasons.

As distant supervision is the root cause of the challenges above, we propose to convert distant supervision into *direct supervision*. In other words, we propose to preprocess each example (x, w, y) in the training data by augmenting it with a set $\mathcal{Z}_x^{\mathrm{S}}$ of semantically correct logical forms. We can then use $\mathcal{Z}_x^{\mathrm{S}}$ to train a semantic parser that uses logical forms for supervision. By factoring out the search process into a preprocessing step, we can avoid running the expensive and potentially spurious search process at training time, thus sidestepping the two challenges outlined above.

Our approach for computing $\mathcal{Z}_x^{\mathrm{S}}$ from the given example (x, w, y) consists of two steps:

1. **Enumerate consistent logical forms.** Given a question x, a table w, and a target denotation y, compute a set $\mathcal{Z}_x^{\mathrm{C}}$ of logical forms consistent with the denotation y. To ensure the coverage of consistent logical forms over an exponentially large search space of logical forms, we propose *dynamic programming on denotations* (DPD) which uses the denotations of logical forms to compress the search space to a manageable size.

2. **Filter out spurious logical forms.** From x, w, y, and $\mathcal{Z}_x^{\mathrm{C}}$, compute a subset $\mathcal{Z}_x^{\mathrm{S}} \subseteq \mathcal{Z}_x^{\mathrm{C}}$ of semantically correct logical forms. As the denotation y alone does not provide enough information to detect spurious logical forms, we instead turns to external signals from humans. We propose *fictitious tables* as a framework to crowdsource signals for filtering spurious logical forms.

We will explain the two steps in the following sections, and then wrap up with methods that can use the resulting direct supervision to train the model.

Reference. The results described in this chapter have been published as (Pasupat and Liang, 2016). Reproducible experiments are hosted on the CodaLab platform at

https://worksheets.codalab.org/worksheets/0x47cc64d9c8ba4a878807c7c35bb22a42.

7.1 Enumerating consistent logical forms

For a given example (x, w, y), our first step toward computing semantically correct logical forms is to generate the set $\mathcal{Z}_x^{\mathrm{C}} = \{z \in \mathcal{Z}_x \mid [\![z]\!]_w = y\}$ of consistent logical forms. We first reason why performing regular beam search on the space of logical forms is intractable. After that, we propose *dynamic programming on denotations* (DPD), which performs search on the space of denotations instead.

7.1.1 Generic deduction rules

In order to gain more coverage over possible logical forms, we will use a collection of deduction rules that is extremely more general than the one used in our floating parser. Table 7.1 shows the new deduction rules. The differences from the original rules are as follows:

	Rule	Semantics
		Terminal Rules
T1	*TokenSpan* $\rightarrow$ *Set*	fuzzymatch(s)
	(cell or cell part fuzzily matching the text: "*chinese*" $\rightarrow$ `China`)	
T2	*TokenSpan* $\rightarrow$ *Set*	value(s)
	(interpreted value: "*march 2015*" $\rightarrow$ *2015-03-XX*)	
T3	$\varnothing \rightarrow$ *Set*	`allRows`
T4	$\varnothing \rightarrow$ *Set*	closedClass()
	(entities from a column with few unique entities)	
	(e.g., `400m` or `relay` from the *Event* column)	
T5	$\varnothing \rightarrow$ *Rel*	graphEdges()
	(*any* graph edge: `Venue`, `Index`, `Next`, `Num2`, ...)	
T6	$\varnothing \rightarrow$ *Rel*	`!=` | `<` | `<=` | `>` | `>=`
		Compositional Rules
C1	*Set* + *Rel* $\rightarrow$ *Set*	$z_2.z_1 \mid \mathbf{R}[z_2].z_1$
C2	*Set* $\rightarrow$ *Set*	$A(z_1)$
	($A \in \{$`count`, `max`, `min`, `sum`, `avg`$\}$)	
C3	*Set* + *Set* $\rightarrow$ *Set*	$z_1 \sqcap z_2 \mid z_1 \sqcup z_2 \mid$ `sub`(z_1, z_2)
	(subtraction is only allowed on numbers)	
		Compositional Rules with Maps
		Initialization
M1	*Set* $\rightarrow$ *Map*	(z_1, x)
	(identity map)	
		Operations on Map
M2	*Map* + *Rel* $\rightarrow$ *Map*	$(u_1, z_2.b_1) \mid (u_1, \mathbf{R}[z_2].b_1)$
	((u_1, b_1) is the logical form of the *Map* argument; z_2 is the *Rel* argument)	
M3	*Map* $\rightarrow$ *Map*	$(u_1, A(b_1))$
	($A \in \{$`count`, `max`, `min`, `sum`, `avg`$\}$)	
M4	*Map* + *Set* $\rightarrow$ *Map*	$(u_1, b_1 \sqcap z_2) \mid (u_1, b_1 \sqcup z_2) \mid (u_1,$ `sub`$(b_1, z_2))$
M5	*Map* + *Map* $\rightarrow$ *Map*	$(u_1, b_1 \sqcap b_2) \mid (u_1, b_1 \sqcup b_2) \mid (u_1,$ `sub`$(b_1, b_2))$
	(Rule M5 is allowed only when $u_1 = u_2$)	
		Finalization
M6	*Map* $\rightarrow$ *Set*	`argmin`$(u_1, \lambda x[b_1]) \mid$ `argmax`$(u_1, \lambda x[b_1])$

Table 7.1: Our new set of generic deduction rules. The logical form of the i-th argument is denoted by z_i (or (u_i, b_i) if the argument is a *Map*). The set of final logical forms contains any logical form with category *Set*.

- The categories are more general. Instead of tying the categories to table constructs such as cells and columns, we use table-independent categories *Set*, *Rel*, and *Map*. We also increase the generality of several compositional rules. For example, subtraction can now be applied on any two sets.

- To increase the recall of cell predicates, we add the rule $\varnothing \rightarrow \textit{Set}[\mathrm{closedClass}()]$, which can generate any cell predicate from a column with few unique cell contents. For instance, if a column with header "*alive*" only contain "*yes*" or "*no*", the question might say "*... is alive ...*" or "*... is dead ...*" to refer to these cells. Instead of relying on the $\mathrm{fuzzymatch}$ function, which will fail in this case, we generate `Yes` and `No` from these two cells. Our assumption here is that a cell from a column with a large number of unique cell contents is usually referred to directly, while a cell from a column with only a few possible values could be mentioned in a more indirect way.

- The special *Map* category represents a partially construct lambda that can be executed. In the original set of deduction rules, the lambda functions for `argmin` and `argmax` are constructed independently from the set they operates on, making them not executable in some cases. For instance, the lambda $\lambda x.\texttt{count}(x)$ cannot be executed without the knowledge without the knowledge of x. *Map* addresses this problem by including the domain that the lambda operates on.

  A *Map* object is a tuple (u, b) of a *finite* set u and a binary relation b. Its denotation $[\![(u, b)]\!]_w$ is $([\![u]\!]_w, [\![b]\!]'_w)$ where the binary $[\![b]\!]'_w$ is $[\![b]\!]_w$ with the domain restricted to the set $[\![u]\!]_w$. For instance, the logical form $\texttt{argmax}(\texttt{HasPosition.1st}, \lambda r[\texttt{IndexOf}.r])$ can be constructed following the derivation tree in Figure 7.2.

On top of the additional deduction rules, we also introduce two new normalization edges to the knowledge graph. First, we use `Num2` for the second number in the cell, which usually denotes a score of the opposing team or a measurement in an alternative unit (e.g., the cell node with text "*3–4*" has a `Num2` edge to atomic value *4*). Second, we use `Part` for items in a list under a set of predefined list delimiters (e.g., the cell with text "*PC, Mac, Linux*" has three `Part` edges to `PC`, `Mac`, and `Linux`). The $\mathrm{fuzzymatch}$ method in Rule T1 can generate these nodes representing cell parts based on the utterance token spans.

7.1.2 Dynamic programming on denotations

From the deduction rules, we could use our floating parser beam search to generate a set of logical forms. However, due to the generality of our deduction rules, the set of of possible logical forms grows quickly as the size of the logical forms increase. As such, partial logical forms that are essential for constructing the desired logical forms might fall off the beam early on, resulting in an incomplete coverage.

Space of denotations. In order to reduce the search space, our key observation is that while the number of logical forms explodes quickly, the number of *distinct denotations* of those logical forms grows at a much slower pace, as multiple logical forms can share the same denotation. For instance, the five consistent logical

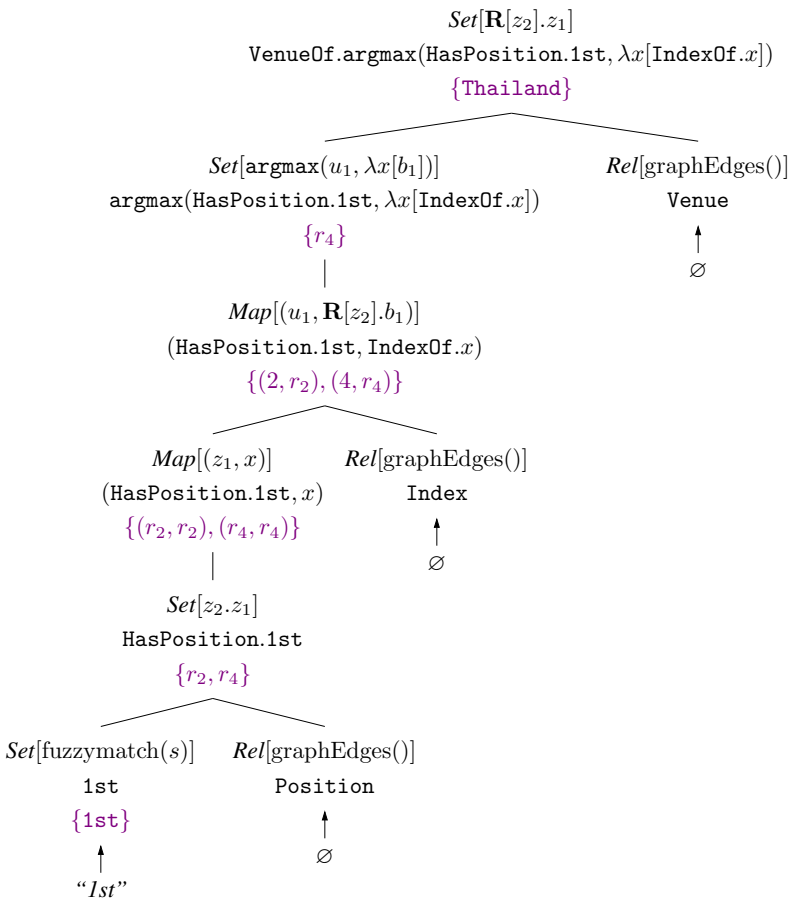


Figure 7.2: A derivation tree for the utterance "*Where did the last 1st place finish occur?*" under our new set of deduction rules. The denotations are shown in purple.

forms in Figure 7.1 execute to the same denotation {Thailand}, and variations in the arguments of logical operators would create even more logical forms with that denotation.

Therefore, instead of directly enumerating logical forms, we proposed *dynamic programming on denotations* (DPD). Inspired by similar methods in programming induction (Lau et al., 2003b; Liang et al., 2010; Gulwani, 2011; Devlin et al., 2017), the main idea is to group logical forms with the same denotations together. Instead of using cells (c, s) of the floating parser (where c is the category and s the the logical form size), we perform dynamic programming on cells (c, s, d) where d is a denotation. For instance, the logical form HasPosition.1st belongs to the cell $(\textit{Set}, 1, \{r_0, r_2\})$.

One requirement for DPD to work is that the deduction rules must be *denotationally invariant*, meaning that the denotation of the resulting logical form must only depend on the denotations of its child logical forms.

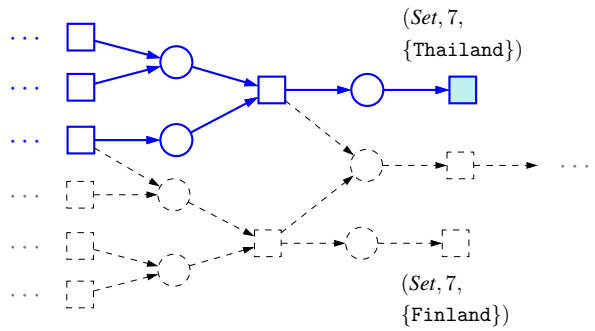


Figure 7.3: The first phase of DPD constructs cells (c, s, d) (square nodes) using denotationally invariant semantic functions (circle nodes). The second phase enumerates all logical forms along paths that lead to the correct denotation y (solid lines).

For example, a compositional rule with semantic function $g(z_1, z_2)$ is denotationally invariant if

$$[\![z_1]\!]_w = [\![z_1']\!]_w \ \wedge \ [\![z_2]\!]_w = [\![z_2']\!]_w \implies [\![g(z_1, z_2)]\!]_w = [\![g(z_1', z_2')]\!]_w \tag{7.1}$$

As a counterexample, if the execution of $g(z_1, z_2)$ involves comparing the numbers of logical operators in z_1 and z_2, the rule is not denotationally invariant.

All deduction rules in Table 7.1 are denotationally invariant. As such, if any of our compositional rule with semantic function $g(z_1, z_2)$ is applied on z_1 from cell (c_1, s_1, d_1) and z_2 from cell (c_2, s_2, d_2), the resulting parse will belong to the same cell $(c, s_1 + s_2 + 1, d)$ for some denotation d regardless of which z_1 and z_2 are chosen. The same conclusion applies for compositional rules with only one child. We will use this fact that logical forms in the same cell are interchangeable to compress the search space in the DPD algorithm.

Algorithm. As illustrated in Figure 7.3, the DPD algorithm consists of two phases. The first phase finds the possible combinations of cells (c, s, d) that lead to the correct denotation y, while the second phase enumerates the actual logical forms belonging to the cells found in the first phase.

1. In the first phase, we are only concerned about finding relevant cell combinations and not the actual logical forms. Therefore, any logical form that belong to a cell could be used as an argument of a deduction rule to construct further logical forms. Thus, we "collapse" the logical forms by keeping only at most one logical form per cell. Subsequent logical forms generated for the same cell are discarded, but the combinations of children cells are recorded for later backtracking.

   After populating the cells up to some maximum size $s_{\max}$, we list all cells $(\textit{Set}, s, y)$ with the correct denotation y, and then note all possible cell-rule combinations $(\text{cell}_1, \text{rule})$ or $(\text{cell}_1, \text{cell}_2, \text{rule})$ leading to those final cells, including the combinations that yield discarded logical forms.

2. The second phase retrieves the actual logical forms by populating the cells (c, s, d) with actual logical

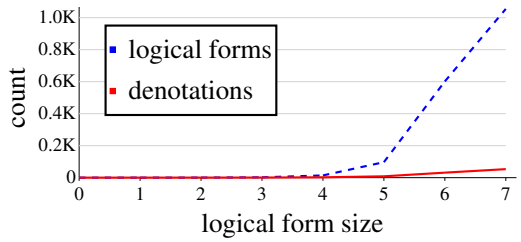


Figure 7.4: The median of the number of logical forms (dashed) and denotations (solid) as the formula size increases. The space of logical forms grows much more quickly than the space of denotations.

forms using only the relevant cell-rule combinations from the first phase. Even though we are searching over the space of logical forms, the elimination of irrelevant cell-rule combinations effectively reduces the search space. As our experiment in Section 7.1.3 will show, the number of cells needed to consider is reduced by 98.7%.

Pruning. While denotations help us narrow down the search space, there are a few cases where the search is still out of control. We decide to sacrifice some generality by introducing the following pruning procedures:

- We prune logical forms that are clearly redundant. For instance, we prune any rule application that does not change the denotation (e.g., applying `max` on a set of size 1).
- We restrict the union operation to between two cells (e.g., `Germany` ⊔ `Thailand`) since union rarely appears in other contexts.
- We forbid the subtraction operation when building a *Map*.
- We do not allow `count` on a set of size 1. This is the most restrictive heuristic since a small number of questions require counting a set of size 1. However, without this restriction, there are too many logical forms that execute to $\{1\}$, which can then be applied in other mostly irrelevant contexts (e.g., selecting the first row with `HasIndex.count`(z) where $[\![\texttt{count}(z)]\!]_w = \{1\}$ will almost always be spurious.)

7.1.3 Experiments

Oracle. We manually annotate 300 training examples from the WIKITABLEQUESTIONS dataset with semantically logical forms if possible. Among them, we successfully annotated 84% of the examples, which is a pretty good coverage since the crowd workers who wrote the questions could have used the tables in any way they wanted. The remaining 16% contains the types of reasoning outside our setup. Some of these include special table layouts, answers that appear inside running texts or images (instead of appearing as the whole cell or a clearly demarcated part), and questions that require common sense knowledge (e.g., comparing "*Quarter-final*" with "*Round of 16*" cannot be done by our `argmax`).

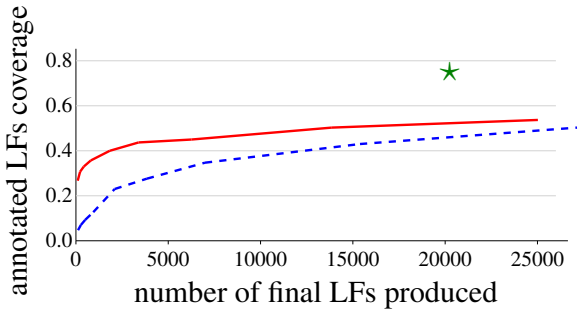


Figure 7.5: The number of annotated logical forms that can be generated by beam search, both uninitialized (dashed) and initialized (solid), increases with the number of candidates generated (controlled by the beam size), but still lacks behind DPD (star).

Search space. To demonstrate the savings gained by collapsing logical forms with the same denotation together, we plot the number of distinct logical forms and their denotations as the logical form size increases. Figure 7.4 shows that the space of logical forms explode much more quickly than the space of denotations. Since the first phase of DPD operates on the space of denotations, the search is much more controlled than using beam search over the space of logical forms.

Moreover, DPD allows us to ignore a large number of irrelevant partial logical forms before doing a fine-grained search in the space of logical forms. On average over all 14,152 training examples, DPD generates approximately 25,000 consistent logical forms. The first phase of DPD generates approximately 153,000 cells (c, s, d). After filtering for the cell-rule combinations that lead to the correct denotations, only about 2,000 cells from 8,000 cell-rule combinations remain, resulting in a 98.7% reduction of the number of cells we need to consider.

Coverage. To measure how well DPD enumerates consistent logical forms, we consider the following experiment. For each of the 300 training examples from earlier, we run the DPD algorithm to generate a set of consistent logical forms, and then check if the annotated logical form is in the generated set. We compare DPD against two baselines: beam search with random parameters, and beam search with parameters trained on the training data of WIKITABLEQUESTIONS using the method from Chapter 5. The evaluation metric is coverage: the fraction of examples where the annotated logical form is among the generated logical forms.

Figure 7.5 plots the coverage against the number of logical forms produced by the algorithms. DPD successfully generates more annotated logical forms (76%) than beam search (53.7%), even when beam search is guided by the learned parameters.

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

↓

Year	Venue	Position	Event	Time
2001	Finland	7th	relay	46.62
2003	Germany	1st	400m	180.32
2005	China	1st	relay	47.12
2007	Hungary	7th	relay	182.05

Figure 7.6: An example fictitious table generated from the table in our running example.

7.2 Filtering spurious logical forms

7.2.1 Fictitious tables

After computing the set $\mathcal{Z}_x^{\mathrm{C}}$ of consistent logical forms, the next step is to filter out spurious logical forms. Unfortunately, spurious logical forms and semantically correct ones cannot be distinguished solely by considering their denotations with respect to the table, since the denotations will be identical by definition. Instead, we use the following key observation: if the same answer is asked on a different but similar tables (e.g., having the same table schema but different cell contents), semantically correct logical forms should give a correct answer to the question in the new contexts, while spurious logical forms will inevitably fail in some contexts. This is similar to how test cases are used to test programs: a semantically correct program should correctly handle the test inputs that follow the program's input specification, while buggy programs may fail some test cases.

Generating fictitious tables. Based on the observation above, we generate *fictitious tables* $w_1, w_2, \ldots$ where each table w_i is a slight alteration of the original table w. As illustrated in Figure 7.6, the fictitious tables maintain the original table schema (i.e., columns names), but the cells in each column are re-sampled from the pool of cell contents collected from the original column. The sampling process follows several guidelines:

- If the cell contents of that column are originally all distinct, sampling is done without replacement (i.e., the cells are permuted). Otherwise, sampling is done with replacement.
- We want the cell predicates in the logical forms $z \in \mathcal{Z}_x^{\mathrm{C}}$ to be present in the fictitious table w_i. To do so, any cell predicates that can be generated by the terminal rules in Table 7.1 need to appear in w_i.

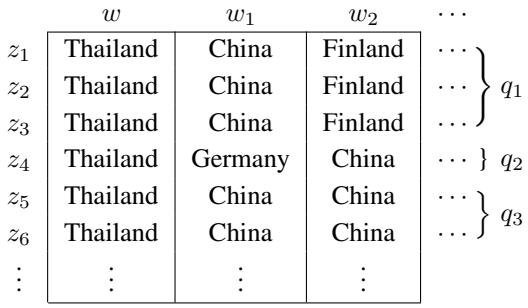


Figure 7.7: Consistent logical forms $z_i \in \mathcal{Z}_x^{\mathrm{C}}$ are executed on fictitious tables to get denotation tuples $[\![z_i]\!]_W$. Logical forms with the same denotation tuple are grouped into the same equivalence class q_j.

- Columns where the cells are sorted usually have special semantics. For instance, in Figure 7.6, the latest event can be identified by performing `argmax` on either the dates (literal semantics) or the row indices (implied semantics: the rows are ordered chronologically). While one could argue that performing `argmax` on row indices is spurious under other contexts, we will make a judgment call and decide that both approaches are both semantically correct. As such, when constructing a fictitious table w_i, we choose to keep sorted columns sorted to preserves the implied semantics.

Equivalence classes. Fictitious tables can help us identify logical forms that are semantically equivalent (under our assumption of implied column semantics above). Let $W = (w_1, \ldots, w_k)$ be a tuple of all possible fictitious tables. For a logical form $\mathcal{Z}_x^{\mathrm{C}}$, we define its *denotation tuple* as $[\![z]\!]_W = ([\![z]\!]_{w_1}, \ldots, [\![z]\!]_{w_k})$. We can now create equivalence classes q of logical forms with the same denotation tuple, which we will denote by $[\![q]\!]_W$. For instance, in Figure 7.7, the logical forms z_1, z_2, and z_3 have the same denotation across all fictitious tables, and thus we group them into an equivalence class q_1.

When the question is unambiguous, we expect at most one equivalence class to contain semantically correct logical forms.

7.2.2 Annotation

To filter equivalence classes with spurious logical forms, we acquire the correct answer to the question x with respect to each fictitious table w_i in W from humans (e.g., via crowdsourcing). We can filter out equivalence classes whose denotation tuple does not match the annotated answers, and then compile the rest into the collection $\mathcal{Z}_x^{\mathrm{S}}$ of semantically correct logical forms.

However, it is impractical and usually redundant to collect the answer for all exponentially many fictitious tables. So instead, we will approximate the process by selecting some subset $W' = (w'_1, \ldots, w'_\ell)$ of ℓ fictitious tables to be annotated with answers.

Choosing tables to annotate. We want to choose a subset W' that can potentially give us the most information about the correct equivalence class as possible. Let $\mathcal{Q}$ be the collection of all equivalence classes. Using the denotation tuples with respect to $W' = (w'_1, \ldots, w'_\ell)$, we can divide $\mathcal{Q}$ into partitions

$$F_t = \{q \in \mathcal{Q} : [\![q]\!]_{W'} = t\}, \tag{7.2}$$

where t is a denotation tuple of length ℓ. For instance, in Figure 7.7, if W' contains only w_2, then q_2 and q_3 will belong to the same partition $F_{(\texttt{China})}$.

Let the human-annotated answers form a tuple $t^* = (y'_1, y'_2, \ldots, y'_k)$. Assuming that there is at least one semantically correct logical form, and that the human annotations are correct, the answer tuple t^* will identify exactly one partition F_{t^*}. This partition will include the semantically correct equivalence class, but can also include spurious equivalence classes that are indistinguishable from the correct one based on the fictitious tables in W'. Our goal is to make the partitions as *small and numerous* as possible, so that no matter what answer t^* we get from humans, the matched partition F_{t^*} is likely to be small (i.e., a large number of spurious logical forms are pruned away).

To make our objective more formal, we choose W' that maximizes the *expected information gained* about the correct equivalence class given the annotation. Define a random variable $Q \in \mathcal{Q}$ representing the correct equivalence class, and another random variable $T^*(W')$ for the annotation tuple (which depends on our choice of W'). Our objective is to find

$$W' = \arg\max_{W'} \left(\mathbb{H}[Q] - \mathbb{H}[Q \mid T^*(W')]\right) = \arg\min_{W'} \mathbb{H}[Q \mid T^*(W')], \tag{7.3}$$

where the entropy terms $\mathbb{H}[\cdot]$ measure the expected information, and we want the entropy to reduce as much as possible after observing the annotation.

We assume a uniform prior on the correct equivalence class Q:

$$p(Q = q) = \frac{1}{|\mathcal{Q}|}, \tag{7.4}$$

and assume that the annotation is accurate with respect to Q:

$$p(T^*(W') = t \mid Q = q) = \begin{cases} 1 & \text{if } [\![q]\!]_{W'} = t \\ 0 & \text{otherwise.} \end{cases} \tag{7.5}$$

We can write

$$p(T^*(W') = t, Q = q) = \begin{cases} 1/|\mathcal{Q}| & \text{if } [\![q]\!]_{W'} = t \\ 0 & \text{otherwise} \end{cases} \tag{7.6}$$

$$p(T^*(W') = t) = \frac{|F_t|}{|\mathcal{Q}|} \tag{7.7}$$

(there are $|F_t|$ equivalence classes with $[\![q]\!]_{W'} = t$ due to the definition of F_t), and therefore

$$\mathbb{H}[Q \mid T^*(W')] = -\sum_{q,t} p(q,t) \log \frac{p(q,t)}{p(t)} \tag{7.8}$$

$$= -\sum_{t} \sum_{q;[\![q]\!]_{W'}=t} \frac{1}{|Q|} \log \frac{1/|Q|}{|F_t|\,/\,|\mathcal{Q}|} \tag{7.9}$$

$$= \frac{1}{|\mathcal{Q}|} \sum_{t} |F_t| \log |F_t| \,. \tag{7.10}$$

We can search for W' that minimizes the term (7.10) above. This matches our original intuition as the term is small when the partitions F_t are small and numerous.

In practice, we make a few more approximations:

- The full tuple W of fictitious tables is approximated as $k = 30$ randomly generated tables. We then choose W' by exhaustively searching over all possible combinations of $\ell = 5$ fictitious tables.
- We cannot guarantee that the annotations are all correct. In our experiments, we propose several methods to trade off accuracy with the amount of pruned spurious logical forms.

7.2.3 Experiments

We continue our investigation on the training examples of the WIKITABLEQUESTIONS dataset. The set $\mathcal{Z}_x^{\mathrm{C}}$ of consistent logical forms are computed by the DPD algorithm.

Equivalence classes. Using 30 randomly generated fictitious tables, we get an average of 1,237 equivalence classes per example. To measure how well the true equivalence classes (based on all possible fictitious tables) are approximated by using on 30 fictitious tables, we verify the equivalence classes against the ones computed on 300 fictitious tables. We found that only 5% of the logical forms are separated from the original equivalence classes.

Oracle annotation. For each of the 252 examples that were annotated with semantically correct logical forms z^* in Section 7.1.3, we use the denotation tuple $t^* = [\![z^*]\!]_{W'}$ to simulate perfect human annotations.

By choosing a subset W' of 5 fictitious tables that minimizes our proposed objective function (7.10), we are able to prune out 98.7% of the spurious equivalence classes, which equate to 98.3% of spurious logical

forms. Additionally, we were able to filter down to just one equivalence class in 32.7% of the examples, and down to three classes in 51.3% of the examples. When more than one equivalence classes remain, most of the time only one class will contain many equivalent logical forms, while other classes are small and contain unusual logical forms such as the spurious z_5 in Figure 7.1.

If the 5 fictitious tables in W' are chosen at random instead of using our proposed method, the number of examples that can be filtered down to one and three classes reduce from 32.7% to 22.6% and from 51.3% to 36.5%, respectively.

The average size of the equivalence class containing the annotated logical form is approximately 3,000 with a standard deviation of approximately 8,000. As our logical formulation is very expressive, there are many different logical forms that equivalently represent the same semantics.

Human annotation. We now consider the answer tuple t^* collected from crowdsourcing on Amazon Mechanical Turk. We use 177 examples where at least two crowd workers agree on the answer of the original table w.

The crowdsourced data is more susceptible to errors. When using the answers on all 5 fictitious tables to filter equivalence classes, the entire set $\mathcal{Z}_x^{\mathrm{C}}$ of consistent logical forms are pruned away in 11.3% of the examples, and the semantically correct equivalence class is pruned in 9% of the examples. These errors are caused by annotation errors, inconsistent data in the tables (e.g., the *Total* column for sport medals does not match the sum of the *Gold*, *Silver*, and *Bronze* columns), and different interpretation of the questions on the fictitious tables. For the remaining example, we are able to filter out 92.1% of spurious logical forms (which equates to 92.6% of spurious equivalence classes).

To prevent semantically correct logical forms from being pruned, we relax our assumptions and keep any equivalence class that disagree with the annotation t^* in at most one fictitious tables. The number of times the entire $\mathcal{Z}_x^{\mathrm{C}}$ is pruned out is reduced from 11.3% to to 3%, but the number of spurious logical forms pruned also decreases from 92.1% to to 78%.

7.3 Using the generated logical forms

The generated st $\mathcal{Z}_x^{\mathrm{S}}$ of semantically correct logical forms can be used to train semantic parsers that take logical forms as supervision. For instance, (Krishnamurthy et al., 2017) trains a neural semantic parser by maximizing the log-probability of generating 100 shortest logical forms from $\mathcal{Z}_x^{\mathrm{S}}$. By using the list $\mathcal{Z}_x^{\mathrm{S}}$, the model does not need to perform search during training. Their single model achieves a test accuracy of 43.3%, while the ensemble model achieves an accuracy of 45.9% .

Filtering spurious logical forms with fictitious tables also turns out to be crucial. When the model in Krishnamurthy et al. (2017) is trained on the set $\mathcal{Z}_x^{\mathrm{C}}$ of consistent logical forms generated by DPD instead of the filtered $\mathcal{Z}_x^{\mathrm{S}}$, the accuracy of the single model drops from 43.3% to 36.3% (Mudrakarta et al., 2018).

7.4 Related work and discussion

Types of supervision for semantic parsing. As stated in the related work chapter (Section 2.4), machine learning models for semantic parsers were mainly trained with logical forms as supervision (Tang and Mooney, 2001; Wong and Mooney, 2007; Zettlemoyer and Collins, 2007; Jia and Liang, 2016; Dong and Lapata, 2016; Zhong et al., 2017). The logical forms provide a clear and semantically correct signal, but they are expensive to annotate, and the logical formalism would prematurely restricts the types of questions present in the dataset. Furthermore, as Kushman and Barzilay (2013) argues, the annotated logical forms might not be the one that best align with the utterance among the equivalent logical forms.

Learning a semantic parser from denotations (Clarke et al., 2010; Liang, 2011; Berant et al., 2013) was mainly motivated by the easiness of acquiring annotated data. However, for a broader domains and complex questions, searching for semantically correct logical forms become more difficult. Some work tries to sidestep search, such as by representing the process for computing answers as continuous vectors instead of discrete symbols (Yin et al., 2016; Neelakantan et al., 2016). Our work provides a general solution by converting the distant supervision into direct supervision, which opens the door for the large class of models that use logical forms as supervision.

Previous works also considered other types of supervision. Without annotated logical forms or denotations, GUSP (Poon, 2013) learns to translate dependency parses into semantic parse by encouraging the result to be consistent with the database schema. Krishnamurthy and Mitchell (2012) and Reddy et al. (2014) encourage agreement between the semantic parse and some other structure (syntactic parse or semantic graph) derived from the same declarative sentence. Several previous works use interactive data, such as dialog sessions or user feedback, as a weak signal for interpreting the previous utterance (Artzi and Zettlemoyer, 2011; Iyer et al., 2017; Labutov et al., 2018).

Test-driven program synthesis. The process of finding logical forms consistent with the denotation is closely related to test-driven program synthesis in the programming language community. The goal of program synthesis is to generate a program that satisfies some specification, such an input-output pairs. Previous work has devised a similar idea of dynamic programming on denotations on more restricted spaces of programs (Lau et al., 2003a; Liang et al., 2010; Feser et al., 2015; Yaghmazadeh et al., 2016), When the program predicates are invertible, such as in string manipulation (Gulwani, 2011; Parisotto et al., 2017; Devlin et al., 2017), the search can be made even more efficient. More recently, Wang et al. (2017) applies a similar technique to DPD on program synthesis, but goes a step further to collapse denotations of the same classes together as another layer of abstraction. However, doing so requires a bit more machinery during unpacking to search over logical forms.

Limitations of DPD. Dynamic programming on denotations depends on two assumptions about the search space. First, the number of unique denotations must be low to prevent the number of cells in Phase 1 from exploding. This means some logical form patterns, such as arbitrary unions or subtraction between *Set*s

or *Map*s, cannot be handled effectively. Luckily, such patterns do not occur often in the distribution of natural questions. Second, the number of unique logical forms that execute to each denotation must be low; otherwise, Phase 2 will have to search over too many logical form combinations. One particular consequence is that `count` logical forms with denotation $\{1\}$ will make the search space explode, since many generated logical forms execute to sets of size 1. This unfortunately reduces the coverage over correct logical forms by approximately 2%.

Test case generation. Our fictitious tables functions similarly to test cases. Previous work has considered generating new test cases (Miller et al., 1990), but the goal there is to converge on a single program rather than identifying a class of programs.

Avoiding spurious logical forms. When training with distant supervision, the model can incorrectly updates toward spurious logical forms found during search. One way to avoid this without using additional annotations is to "hedge" the updates: in early training iterations, perform updates toward consistent logical forms more equally instead of proportionally to the probabilities assigned by the model. This intuition, coupled with decoding strategies that encourage more exploration, can prevent the model from making bad updates toward spurious logical forms (Guu et al., 2017; Misra et al., 2018).

7.5 Conclusion

In this chapter, we presented techniques for converting distant supervision (denotations) into direct supervision (logical forms). Given the correct answer for a question we efficiently enumerate logical forms consistent with the answer using dynamic programming on denotations, and then filter out spurious logical forms with fictitious tables. The resulting set of logical forms can be used to train a directly supervised parser, effectively sidestepping search problems during training.

The value of having logical forms as direct supervision has been noted by Yih et al. (2016), who used a labeling interface to annotate questions in the WEBQUESTIONS dataset (Berant et al., 2013) with SPARQL queries. Training with annotated queries as supervision yields an increased accuracy, and compared to the distantly supervised model, the number of incorrect relations learned through spurious logical forms is decreased. Our method of pre-computing logical forms provides a more automatic method for annotating examples with logical forms than the annotation interface. While it still requires some human effort, we do not require experts even when annotating complex questions.

Up to this chapter, we have considered a new QA task that operates on a high-BREADTH high-DEPTH regime, as well as algorithms based on semantic parsing to tackle the task. The next chapter concludes our findings, lists alternative QA methods in the literature for solving our proposed task, and provides several potential paths for future work.

Chapter 8

Conclusion

8.1 Summary

New QA tasks with increased BREADTH and DEPTH. In this dissertation, we focus on improving the capability of question answering systems along two axes. First, to increase the scope of the knowledge sources that the systems operate on (BREADTH), we consider using web pages, which contain open-domain semi-structured data, as the knowledge source (Chapter 3). As tabular data on web pages contain computable knowledge suitable for complex reasoning, we propose the task of answering compositional questions on web tables, which aims to increase task complexity (DEPTH) and the scope of the knowledge source simultaneously (Chapter 4). For the task, we have collected a new WIKITABLEQUESTIONS dataset of complex questions on tables and their answers. The dataset contains comparable or higher task complexity, as well as a broader open-domain knowledge source, than contemporary semantic parsing datasets.

Training a semantic parser from denotations. We cast the proposed QA task as learning a semantic parser using the annotated answers as distant supervision (Chapter 5). We address the increased question complexity (DEPTH) by using compositional logical forms to express the multiple steps needed for computing answers. To construct candidate logical forms, the parser uses a set of deduction rules to build larger logical forms from smaller parts.

Meanwhile, to address the potentially unseen table schema (BREADTH), we represent tables with schema-independent knowledge graphs, and then use floating deduction rules to flexibly generate open-domain logical form predicates based on the table. In contrast to offline information extraction, which assumes a fixed and canonicalized database schema, our on-the-fly information extraction approach can adapt to arbitrary data schemata and maintain uncertainty over the possible interpretations of the table. This ultimately allows us to answer a wider range of open-domain questions.

	Test accuracy	
System	**Single model**	**Ensemble**
Original deduction rules (Chapter 5)	37.1	-
Expanded deduction rules (Chapter 6)	42.7	-
Expanded deduction rules + macros (Chapter 6)	43.7	-
Neural Programmer (Neelakantan et al., 2017)	34.2	37.7
Neural Multi-Step Reasoning (Haug et al., 2018)	34.8	38.7
Rule-based + Abductive Matching (Dhamdhere et al., 2017b)	40.4	-
Neural Decoding with Type Constraints (Krishnamurthy et al., 2017)	43.3	45.9
Memory Augmented Policy Optimization (MAPO) (Liang et al., 2018)	43.9	46.3
MAPO + Meta Reward Learning (Agarwal et al., 2019)	44.1	46.9
MAPO + Neural Program Planner (Biloki et al., 2019)	43.9	47.2

Table 8.1: Test accuracy of the previous work on the WIKITABLEQUESTIONS dataset.

Speeding up the parser. The coverage over answerable questions can be increased by expanding the set of deduction rules and augmenting the knowledge graph representation. To compensate for the increased space of logical forms from this expansion, we proposed a way to improve search by reusing useful logical form patterns (Chapter 6). When processing an input question, similar questions in the training dataset are retrieved, and the logical form macros from those questions are used to construct logical forms. This allows the parser to skip the generation of logical forms that are less likely to be a good parse.

Converting distant supervision to direct supervision. Training a parser with distant supervision requires performing search during training, which introduces several challenges. With the expanding space of logical forms, it is more difficult to find logical forms consistent with the correct answer during training, and the discovered logical forms might be spurious (right for wrong reasons), leading to bad parameter updates. We propose to factor out the search and precompute the semantically correct logical forms for each example, which effectively converts the dataset into a directly supervised one (Chapter 7). The resulting logical forms can then be used to train a parser that takes logical forms as supervision without having to perform search during training.

8.2 State-of-the-art on the WIKITABLEQUESTIONS dataset

Table 8.1 summarizes previous work on the WIKITABLEQUESTIONS dataset. Descriptions and insights of the methods and their related works are described below. The annotated answers are used as distant supervision unless stated otherwise.

8.2.1 Using continuous representations for computation

Logical form operators are interpretable and efficient to execute. However, a system generating such symbolic operators as discrete actions cannot be trained end-to-end with backpropagation. As an alternative, the following works consider using differentiable operators to represent the steps of computing the answer.

- Neural Programmer (Neelakantan et al., 2016, 2017) proposes to model the operators as predefined continuous functions such as matrix multiplication and pooling. At each time step, the model applies each operator on the current state vectors, and then compute a weighted average of the results (where the weight of each operator is predicted by the model) to be used as new state vectors. The final state vectors are used to compute the final answer.

  Since all operators are differentiable, the model can be trained end-to-end with backpropagation from distant supervision. The model will have to learn to compute the weights of the operators based on the input utterance and table.

- Similar to Neural Programmer, Neural Enquirer (Yin et al., 2016) also uses continuous functions to model the steps of computation. However, the hand-engineered networks for the operators in Neural Programmer are replaced with a single general network for computing the state vectors. The network can be trained using a loss function defined on the final denotation. The system is evaluated on synthetic questions that are compositional but closed-domain.

- The work by Mou et al. (2017) models the steps of computations in two ways: continuous functions, which are trained end-to-end; and symbolic operators, whose decoder is trained with REINFORCE (Williams, 1992). The key observation is that parts of the state vectors from the continuous functions are interpretable (e.g., as attention over table rows). Such information can serve as a guidance when learning to decode symbolic operators. This helps mitigate the cold start problem of REINFORCE by boosting the chance of getting reward when training the symbolic decoder. The system is evaluated on the dataset by Yin et al. (2016) above.

Continuous operators allows the model to be trained in an end-to-end fashion from denotations without having to perform search. However, it can be difficult to design or learn new operators to perform more complex reasoning; for instance, adding an operator to subtract one column from another would require either a careful design for a predefined operator, or a large amount of training data for a trainable operator. Trainable operators like in Yin et al. (2016) and Mou et al. (2017) are also less interpretable than the discrete logical forms.

8.2.2 Different generation and scoring mechanisms

The following works use logical forms to represent the steps of computation, but employ different methods for generating and scoring the logical forms.

- Neural Multi-Step Reasoning (Haug et al., 2018) uses the floating parser (Chapter 5) to generate logical form candidates, but then applies a neural scorer to score logical forms. In particular, the scorer uses pre-trained word embeddings and information from the table to embed the utterance and a natural language paraphrase of the logical form into the same space. The similarity score between the two embeddings becomes the score of the logical form.

- The work by Dhamdhere et al. (2017a,b) uses a rule-based system to process the table, generate semantic parses from the input, and score them. To fill in the operands that the rule-based system cannot handle well (e.g., matching the phrase "*movie*" to the column *Title*), a machine learning system is trained to perform abductive reasoning and infer the missing values. The rule-based system is easy to debug and generates a controllable number of parses (8.7 on average as opposed to exponentially many parses from the floating parser), while the machine learning part helps with the difficult and open-domain predicates.

- Neural Decoding with Type Constraints (Krishnamurthy et al., 2017) trains a top-down symbolic decoder using the logical forms generated by DPD and fictitious tables (Chapter 7) as supervision. Type constraints mined from logical forms are used to ensure the validity of the output, as well as to control the space of possible actions at each time step.

- Memory Augmented Policy Optimization (MAPO) (Liang et al., 2018) applies a policy gradient algorithm to train an agent that generates logical forms in Lisp-like syntax. To address the cold start problem when training from a random policy, the work introduces (1) systematic exploration that caches the explored trajectories so that consistent logical forms are found more quickly; (2) a memory buffer for storing logical forms with high reward; and (3) an unbiased low-variance policy gradient update that incorporates both logical forms in the memory buffer and on-policy trajectories. Later works improves the reinforcement learning part with meta reward learning (Agarwal et al., 2019) and a model-based program planner (Biloki et al., 2019).

8.3 Future directions

Knowing when to abstain. Our main evaluation metric, accuracy, encourages the model to always predict an answer even when (1) the question cannot be answered from the given context, and (2) the model is not confident about the answer. This is problematic is real-world settings where precision is more important than recall; for instance, a virtual assistant should not perform actions with misleading or disastrous consequences (Dhamdhere et al., 2017b). One possible future direction is to make a model that can abstain from answering a question, and can make a trade-off between precision and recall (Yao et al., 2014; Rajpurkar et al., 2018; Dong et al., 2018).

Avoiding the long tail of phenomena. WIKITABLEQUESTIONS is a challenging dataset containing both compositional utterances with little constraints, and tables with arbitrary data and formatting. As such, while the model errors can be categorized at a high-level (Section 5.6.2), addressing an individual error type with some heavy machinery would push up the score by only a small amount. Moreover, interesting phenomena that occur less frequently in the dataset will likely to be ignored by the model.

One alternative solution is to come up with a family of datasets, each focusing on some particular challenge (e.g., rare logical operators or table understanding) while keeping the rest fixed. This way, methods for handling the phenomena could be developed more quickly with clear results. Another potential solution is to gather new examples for the rare phenomena in an interactive fashion.

Combining information from multiple sources. Finally, the tasks introduced in this dissertation are limited to using one source of data for all computation steps. In reality, the information needed to perform a task is often scattered around in multiple knowledge sources. For instance, to handle the query "*Direction to John's house*", the system would need to identify who John is (e.g., by finding a close friend on Facebook with name John), find his address (e.g., by looking up on John's personal web page), and then find the direction (e.g., using GPS and a map application).

Several question answering and semantic parsing datasets such as QALD (Lopez et al., 2013), Spider (Yu et al., 2018b), HotpotQA (Yang et al., 2018), and DROP (Dua et al., 2019) require the fusion of knowledge from multiple sources. We believe that the ability to plan complex actions on multiple, open-domain environments is the right path toward an ideal natural language interface.

Bibliography

S. Abiteboul. 1997. Querying semi-structured data. In *International Conference on Database Theory*.

R. Agarwal, C. Liang, D. Schuurmans, and M. Norouzi. 2019. Learning to generalize from sparse and underspecified rewards. *arXiv preprint arXiv:1902.07198*.

E. Agichtein and L. Gravano. 2000. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the fifth ACM conference on Digital Libraries*.

A. Agrawal, D. Batra, and D. Parikh. 2016. Analyzing the behavior of visual question answering models. In *Empirical Methods in Natural Language Processing (EMNLP)*.

J. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, M. Swift, and W. Taysom. 2007. PLOW: A collaborative task learning agent. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1514–1519.

J. Andreas and D. Klein. 2015. Alignment-based compositional semantics for instruction following. In *Empirical Methods in Natural Language Processing (EMNLP)*.

J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. 2016. Neural module networks. In *Computer Vision and Pattern Recognition (CVPR)*.

I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. 1995. Natural language interfaces to databases – an introduction. *Journal of Natural Language Engineering*, 1:29–81.

S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, and D. Parikh. 2015. Vqa: Visual question answering. In *International Conference on Computer Vision (ICCV)*, pages 2425–2433.

A. Arasu and H. Garcia-Molina. 2003. Extracting structured data from web pages. In *ACM SIGMOD international conference on Management of data*, pages 337–348.

Y. Artzi and K. L. L. Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Y. Artzi and L. Zettlemoyer. 2011. Bootstrapping semantic parsers from conversations. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 421–432.

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

V. Ashok, Y. Borodin, S. Stoyanchev, Y. Puzis, and I. V. Ramakrishnan. 2014. Wizard-of-Oz evaluation of speech-driven web browsing interface for people with vision impairments. In *Web for All Conference*.

S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. 2007. DBpedia: A nucleus for a web of open data. In *International semantic web conference and Asian semantic web conference (ISWC/ASWC)*, pages 722–735.

D. Bahdanau, K. Cho, and Y. Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*.

L. Banarescu, C. B. S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. 2013. Abstract meaning representation for sembanking. In *7th Linguistic Annotation Workshop and Interoperability with Discourse*.

M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. 2007. Open information extraction from the web. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2670–2676.

J. Berant, A. Chou, R. Frostig, and P. Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*.

J. Berant and P. Liang. 2014. Semantic parsing via paraphrasing. In *Association for Computational Linguistics (ACL)*.

J. Berant and P. Liang. 2015. Imitation learning of agenda-based semantic parsers. *Transactions of the Association for Computational Linguistics (TACL)*, 3:545–558.

C. S. Bhagavatula, T. Noraset, and D. Downey. 2015. TabEL: entity linking in web tables. In *International Semantic Web Conference (ISWC)*.

J. Biloki, C. Liang, and N. Lao. 2019. Neural program planner for structured predictions. In *Deep Reinforcement Learning Meets Structured Prediction Workshop at ICLR 2019*.

L. Bing, R. Guo, W. Lam, Z. Niu, and H. Wang. 2014. Web page segmentation with structured prediction and its application in web page classification. In *ACM Special Interest Group on Information Retreival (SIGIR)*.

Y. Bisk, S. Reddy, J. Blitzer, J. Hockenmaier, and M. Steedman. 2016a. Evaluating induced CCG parsers on grounded semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Y. Bisk, D. Yuret, and D. Marcu. 2016b. Natural language communication with robots. In *North American Association for Computational Linguistics (NAACL)*.

A. Bordes, N. Usunier, S. Chopra, and J. Weston. 2015. Large-scale simple question answering with memory networks. *arXiv preprint arXiv:1506.02075*.

S. Branavan, H. Chen, J. Eisenstein, and R. Barzilay. 2008. Learning document-level semantic properties from free-text annotations. In *Association for Computational Linguistics (ACL)*.

S. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay. 2009. Reinforcement learning for mapping instructions to actions. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pages 82–90.

E. Brill, S. Dumais, and M. Banko. 2002. An analysis of the AskMSR question-answering system. In *Association for Computational Linguistics (ACL)*, pages 257–264.

R. D. Burke, K. J. Hammond, V. Kulyukin, S. L. Lytinen, N. Tomuro, and S. Schoenberg. 1997. Question answering from frequently asked question files: Experiences with the FAQ finder system. *AI magazine*, 18.

M. J. Cafarella, A. Halevy, and N. Khoussainova. 2009. Data integration for the relational web. In *Very Large Data Bases (VLDB)*.

M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. 2008. WebTables: exploring the power of tables on the web. In *Very Large Data Bases (VLDB)*, pages 538–549.

Q. Cai and A. Yates. 2013. Large-scale semantic parsing via schema matching and lexicon extension. In *Association for Computational Linguistics (ACL)*.

Y. Cao, J. Xu, T. Liu, H. Li, Y. Huang, and H. Hon. 2006. Adapting ranking SVM to document retrieval. In *ACM Special Interest Group on Information Retreival (SIGIR)*.

C. Carpineto and G. Romano. 2012. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys (CSUR)*, 44.

A. T. Chaganty, A. Paranjape, J. Bolton, M. Lamm, J. Lei, A. See, K. Clark, Y. Zhang, P. Qi, and C. D. Manning. 2017. Stanford at TAC KBP 2017: Building a trilingual relational knowledge graph. In *Text Analytics Conference*.

D. Chen, J. Bolton, and C. D. Manning. 2016. A thorough examination of the CNN / Daily Mail reading comprehension task. In *Association for Computational Linguistics (ACL)*.

D. Chen, A. Fisch, J. Weston, and A. Bordes. 2017. Reading Wikipedia to answer open-domain questions. In *Association for Computational Linguistics (ACL)*.

D. L. Chen and R. J. Mooney. 2011. Learning to interpret natural language navigation instructions from observations. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 859–865.

Y. Chen, L. Xu, K. Liu, D. Zeng, and J. Zhao. 2015. Event extraction via dynamic multi-pooling convolutional neural networks. In *Association for Computational Linguistics (ACL)*.

J. Cheng, S. Reddy, V. Saraswat, and M. Lapata. 2017. Learning structured natural language representations for semantic parsing. In *Association for Computational Linguistics (ACL)*.

J. Clarke, D. Goldwasser, M. Chang, and D. Roth. 2010. Driving semantic parsing from the world's response. In *Computational Natural Language Learning (CoNLL)*, pages 18–27.

W. W. Cohen, M. Hurst, and L. S. Jensen. 2002. A flexible learning system for wrapping tables and lists in HTML documents. In *World Wide Web (WWW)*, pages 232–241.

V. Crescenzi, G. Mecca, P. Merialdo, et al. 2001. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, volume 1, pages 109–118.

D. A. Dahl, M. Bates, M. Brown, W. Fisher, K. Hunicke-Smith, D. Pallett, C. Pao, A. Rudnicky, and E. Shriberg. 1994. Expanding the scope of the ATIS task: The ATIS-3 corpus. In *Workshop on Human Language Technology*, pages 43–48.

N. Dalvi, R. Kumar, and M. Soliman. 2011. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment*, 4(4):219–230.

J. Devlin, M. Chang, K. Lee, and K. Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning (ICML)*.

K. Dhamdhere, K. S. McCurley, R. Nahmias, M. Sundararajan, and Q. Yan. 2017a. Analyza: Exploring data with conversation. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces*.

K. Dhamdhere, K. S. McCurley, M. Sundararajan, and A. Taly. 2017b. Abductive matching in question answering. *arXiv preprint arXiv:1709.03036*.

L. Dong and M. Lapata. 2016. Language to logical form with neural attention. In *Association for Computational Linguistics (ACL)*.

L. Dong and M. Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Association for Computational Linguistics (ACL)*.

L. Dong, C. Quirk, and M. Lapata. 2018. Confidence modeling for neural semantic parsing. In *Association for Computational Linguistics (ACL)*.

D. Dua, Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner. 2019. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *North American Association for Computational Linguistics (NAACL)*.

J. Duchi, E. Hazan, and Y. Singer. 2010. Adaptive subgradient methods for online learning and stochastic optimization. In *Conference on Learning Theory (COLT)*.

J. Ellis, J. Getman, H. Simpson, K. Griffitt, H. T. Dang, R. Grishman, H. Ji, C. DePrince, T. Riese, and N. Kuster. 2015. TAC KBP 2015 slot descriptions. *Linguistic Data Consortium*.

D. W. Embley, M. S. Krishnamoorthy, G. Nagy, and S. C. Seth. 2016. Converting heterogeneous statistical tables on the web to searchable databases. *International Journal on Document Analysis and Recognition (IJDAR)*, 19:119–138.

O. Etzioni, A. Fader, J. Christensen, S. Soderland, and Mausam. 2011. Open information extraction: the second generation. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

A. Fader, S. Soderland, and O. Etzioni. 2011. Identifying relations for open information extraction. In *Empirical Methods in Natural Language Processing (EMNLP)*.

H. Fang, S. Gupta, F. Iandola, R. K. Srivastava, L. Deng, P. Dollár, J. Gao, X. He, M. Mitchell, and J. C. Platt. 2015. From captions to visual concepts and back. In *Computer Vision and Pattern Recognition (CVPR)*.

A. Farhadi, M. Hejrati, M. A. Sadeghi, P. Young, C. Rashtchian, J. Hockenmaier, and D. Forsyth. 2010. Every picture tells a story: Generating sentences from images. In *European Conference on Computer Vision (ECCV)*, pages 15–29.

J. K. Feser, S. Chaudhuri, and I. Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Programming Language Design and Implementation (PLDI)*.

J. Flanigan, S. Thomson, J. G. Carbonell, C. Dyer, and N. A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Association for Computational Linguistics (ACL)*.

D. Fried, J. Andreas, and D. Klein. 2018. Unified pragmatic models for generating and following instructions. In *North American Association for Computational Linguistics (NAACL)*.

E. Gabrilovich, M. Ringgaard, and A. Subramanya. 2013. Facc1: Freebase annotation of clueweb corpora. http://lemurproject.org/clueweb09/.

M. Gamon, T. Yano, X. Song, J. Apacible, and P. Pantel. 2013. Identifying salient entities in web pages. In *Conference on Information and Knowledge Management (CIKM)*.

D. Gildea and D. Jurafsky. 2002. Automatic labeling of semantic roles. *Computational Linguistics*, 28:245–288.

Google. 2013. Freebase data dumps (2013-06-09). https://developers.google.com/freebase/data.

V. Govindaraju, C. Zhang, and C. Ré. 2013. Understanding tables in context using standard NLP toolkits. In *Association for Computational Linguistics (ACL)*.

Y. Goyal, T. Khot, D. Summers-Stay, D. Batra, and D. Parikh. 2017. Making the V in VQA matter: Elevating the role of image understanding in visual question answering. In *Computer Vision and Pattern Recognition (CVPR)*.

A. Graves, G. Wayne, and I. Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

T. Grigalis and A. Cenys. 2014. Unsupervised structured data extraction from template-generated web pages. *Journal of Universal Computer Science*, 20:169–192.

J. Gu, Z. Lu, H. Li, and V. O. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Association for Computational Linguistics (ACL)*.

J. Gu, Y. Wang, K. Cho, and V. O. Li. 2017. Search engine guided non-parametric neural machine translation. *arXiv preprint arXiv:1705.07267*.

S. Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 46(1):317–330.

K. Guu, T. B. Hashimoto, Y. Oren, and P. Liang. 2018. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics (TACL)*, 0.

K. Guu, P. Pasupat, E. Z. Liu, and P. Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Association for Computational Linguistics (ACL)*.

Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen. 2004. Combating web spam with trustrank. In *Very Large Data Bases (VLDB)*.

M. Hammoudi, G. Rothermel, and P. Tonella. 2016. Why do record/replay tests of web applications break? *IEEE International Conference on Software Testing, Verification and Validation*.

T. Haug, O. Ganea, and P. Grnarova. 2018. Neural multi-step reasoning for question answering on semi-structured tables. In *European Conference on Information Retrieval*.

T. H. Haveliwala. 2002. Topic-sensitive pagerank. In *World Wide Web (WWW)*.

S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig. 2018. Retrieval-based neural code generation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition (CVPR)*.

Y. He and S. Young. 2006. Spoken language understanding using the hidden vector state model. *Speech Communication*, 48:262–275.

M. A. Hearst. 1992. Automatic acquisition of hyponyms from large text corpora. In *Interational Conference on Computational linguistics*, pages 539–545.

M. A. Hearst. 1998. Automated discovery of wordnet relations. *WordNet: an electronic lexical database*.

G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum. 1978. Developing a natural language interface to complex data. *ACM Transactions on Database Systems (TODS)*, 3:105–147.

K. M. Hermann, D. Das, J. Weston, and K. Ganchev. 2014. Semantic frame identification with distributed word representations. In *Association for Computational Linguistics (ACL)*.

J. Herzig and J. Berant. 2018. Decoupling structure and lexicon for zero-shot semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*.

D. A. Hudson and C. D. Manning. 2019. GQA: A new dataset for real-world visual reasoning and compositional question answering. In *Computer Vision and Pattern Recognition (CVPR)*.

S. Iyer, A. Cheung, and L. Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086*.

S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Association for Computational Linguistics (ACL)*.

S. Iyer, I. Konstas, A. Cheung, and L. S. Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Empirical Methods in Natural Language Processing (EMNLP)*.

M. Iyyer, W. Yih, and M. Chang. 2017. Search-based neural structured learning for sequential question answering. In *Association for Computational Linguistics (ACL)*.

H. Ji and R. Grishman. 2011. Knowledge base population: Successful approaches and challenges. In *Association for Computational Linguistics (ACL)*, pages 1148–1158.

R. Jia and P. Liang. 2016. Data recombination for neural semantic parsing. In *Association for Computational Linguistics (ACL)*.

J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. Girshick. 2017a. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Computer Vision and Pattern Recognition (CVPR)*.

J. Johnson, B. Hariharan, L. van der Maaten, J. Hoffman, L. Fei-Fei, C. L. Zitnick, and R. Girshick. 2017b. Inferring and executing programs for visual reasoning. In *International Conference on Computer Vision (ICCV)*.

M. Johnson, T. Griffiths, and S. Goldwater. 2006. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 641–648.

K. S. Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28.

R. J. Kate and R. J. Mooney. 2007. Learning language semantics from ambiguous supervision. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 895–900.

B. Katz, S. Felshin, D. Yuret, A. Ibrahim, J. Lin, G. Marton, A. J. McFarland, and B. Temelkuran. 2002. Omnibase: Uniform access to heterogeneous data for question answering. In *International Conference on Application of Natural Language to Information Systems*.

R. Kreuzer, J. Hage, and A. J. Feelders. 2015. A quantitative comparison of semantic web page segmentation approaches. *International Conference on Web Engineering (ICWE)*.

J. Krishnamurthy, P. Dasigi, and M. Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Empirical Methods in Natural Language Processing (EMNLP)*.

J. Krishnamurthy and T. Mitchell. 2012. Weakly supervised training of semantic parsers. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*, pages 754–765.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1097–1105.

M. Kumar, A. Paepcke, and T. Winograd. 2007. Eyepoint: practical pointing and selection using gaze and keyboard. In *Conference on Human Factors in Computing Systems (CHI)*.

R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. 2013. Webzeitgeist: design mining the web. In *Conference on Human Factors in Computing Systems (CHI)*.

R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer. 2011. Bricolage: example-based retargeting for web design. In *Conference on Human Factors in Computing Systems (CHI)*.

N. Kushman and R. Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Human Language Technology and North American Association for Computational Linguistics (HLT/NAACL)*, pages 826–836.

N. Kushmerick. 1997. *Wrapper induction for information extraction*. Ph.D. thesis, University of Washington.

T. Kwiatkowski, E. Choi, Y. Artzi, and L. Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Empirical Methods in Natural Language Processing (EMNLP)*.

T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, M. Kelcey, J. Devlin, K. Lee, K. N. Toutanova, L. Jones, M. Chang, A. Dai, J. Uszkoreit, Q. Le, and S. Petrov. 2019. Natural questions: a benchmark for question answering research. In *Association for Computational Linguistics (ACL)*.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1223–1233.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1512–1523.

I. Labutov, B. Yang, and T. Mitchell. 2018. Learning to learn semantic parsers from natural language supervision. In *Empirical Methods in Natural Language Processing (EMNLP)*.

T. Lau, P. Domingos, and D. S. Weld. 2003a. Learning programs from traces using version space algebra. In *International Conference On Knowledge Capture*, pages 36–43.

T. Lau, S. Wolfman, P. Domingos, and D. S. Weld. 2003b. Programming by demonstration using version space algebra. *Machine Learning*, 53:111–156.

Q. Li, H. Ji, and L. Huang. 2013. Joint event extraction via structured prediction with global features. In *Association for Computational Linguistics (ACL)*.

C. Liang, M. Norouzi, J. Berant, Q. Le, and N. Lao. 2018. Memory augmented policy optimization for program synthesis with generalization. In *Advances in Neural Information Processing Systems (NeurIPS)*.

P. Liang. 2011. *Learning Dependency-Based Compositional Semantics*. Ph.D. thesis, University of California Berkeley at Berkeley.

P. Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408*.

P. Liang, M. I. Jordan, and D. Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*, pages 639–646.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599.

G. Limaye, S. Sarawagi, and S. Chakrabarti. 2010. Annotating and searching web tables using entities, types and relationships. In *Very Large Data Bases (VLDB)*, volume 3, pages 1338–1347.

T. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Doll'ar, and C. L. Zitnick. 2014. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision (ECCV)*, pages 740–755.

X. V. Lin, C. Wang, L. S. Zettlemoyer, and M. D. Ernst. 2018. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Language Resources and Evaluation Conference (LREC)*.

W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom. 2016. Latent predictor networks for code generation. In *Association for Computational Linguistics (ACL)*, pages 599–609.

E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang. 2018. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations (ICLR)*.

N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Empirical Methods in Natural Language Processing (EMNLP)*.

R. Long, P. Pasupat, and P. Liang. 2016. Simpler context-dependent logical forms via model projections. In *Association for Computational Linguistics (ACL)*.

V. Lopez, C. Unger, P. Cimiano, and E. Motta. 2013. Evaluating question answering over linked data. In *World Wide Web (WWW)*.

M. Luong, H. Pham, and C. D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1412–1421.

M. MacMahon, B. Stankiewicz, and B. Kuipers. 2006. Walk the talk: Connecting language, knowledge, and action in route instructions. In *National Conference on Artificial Intelligence*.

A. Madaan, A. Mittal, G. Ramakrishnan, and S. Sarawagi. 2016. Numerical relation extraction with minimal supervision. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille. 2015. Deep captioning with multimodal recurrent neural networks (m-RNN). In *International Conference on Learning Representations (ICLR)*.

Mausam, M. Schmitz, R. Bart, S. Soderland, and O. Etzioni. 2012. Open language learning for information extraction. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*, pages 523–534.

J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. 1997. Lore: A database management system for semistructured data. *SIGMOD record*, 26.

G. Mesnil, Y. Dauphin, K. Yao, Y. Bengio, L. Deng, D. Hakkani-Tur, X. He, L. Heck, G. Tur, and D. Yu. 2014. Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23.

B. P. Miller, L. Fredriksen, and B. So. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44.

M. Mintz, S. Bills, R. Snow, and D. Jurafsky. 2009. Distant supervision for relation extraction without labeled data. In *Association for Computational Linguistics (ACL)*, pages 1003–1011.

D. Misra, M. Chang, X. He, and W. Yih. 2018. Policy shaping and generalized update equations for semantic parsing from denotations. In *Empirical Methods in Natural Language Processing (EMNLP)*.

T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, et al. 2015. Never-ending learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

M. Miwa and M. Bansal. 2016. End-to-end relation extraction using lstms on sequences and tree structures. In *Association for Computational Linguistics (ACL)*.

R. Montague. 1973. The proper treatment of quantification in ordinary English. In *Approaches to Natural Language*, pages 221–242.

L. Mou, Z. Lu, H. Li, and Z. Jin. 2017. Coupling distributed and symbolic execution for natural language queries. In *International Conference on Machine Learning (ICML)*.

P. K. Mudrakarta, A. Taly, M. Sundararajan, and K. Dhamdhere. 2018. It was the training data pruning too! *arXiv preprint arXiv:1803.04579*.

I. Muslea, S. Minton, and C. A. Knoblock. 2001. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1):93–114.

A. Neelakantan, Q. V. Le, M. Abadi, A. McCallum, and D. Amodei. 2017. Learning a natural language interface with neural programmer. In *International Conference on Learning Representations (ICLR)*.

A. Neelakantan, Q. V. Le, and I. Sutskever. 2016. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations (ICLR)*.

A. Neelakantan, B. Roth, and A. McCallum. 2015. Compositional vector space models for knowledge base completion. In *Association for Computational Linguistics (ACL)*.

M. Nickel, V. Tresp, and H. Kriegel. 2011. A three-way model for collective learning on multi-relational data. In *International Conference on Machine Learning (ICML)*, pages 809–816.

Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 30:574–584.

T. J. O'Donnell. 2011. *Productivity and Reuse in Language*. Ph.D. thesis, Massachusetts Institute of Technology.

A. Omari, B. Kimelfeld, E. Yahav, and S. Shoham. 2016. Lossless separation of web pages into layout code and data. In *International Conference on Knowledge Discovery and Data Mining (KDD)*.

E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. 2017. Neuro-symbolic program synthesis. In *International Conference on Learning Representations (ICLR)*.

M. Paşca. 2003. Open-domain question answering from large text collections. *Computational Linguistics*, 29.

P. Pasupat and P. Liang. 2014. Zero-shot entity extraction from web pages. In *Association for Computational Linguistics (ACL)*.

P. Pasupat and P. Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Association for Computational Linguistics (ACL)*.

P. Pasupat and P. Liang. 2016. Inferring logical forms from denotations. In *Association for Computational Linguistics (ACL)*.

R. Pieraccini, E. Levin, and C. Lee. 1991. Stochastic representation of conceptual structure in the ATIS task. In *Human Language Technology (HLT)*.

I. Polosukhin and A. Skidanov. 2018. Neural program search: Solving programming tasks from description and examples. In *International Conference on Learning Representations (ICLR)*.

H. Poon. 2013. Grounded unsupervised semantic parsing. In *Association for Computational Linguistics (ACL)*.

C. Quirk, R. J. Mooney, and M. Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Association for Computational Linguistics (ACL)*.

M. Rabinovich, M. Stern, and D. Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Association for Computational Linguistics (ACL)*.

P. Rajpurkar, R. Jia, and P. Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. In *Association for Computational Linguistics (ACL)*.

P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Empirical Methods in Natural Language Processing (EMNLP)*.

C. Raymond and G. Riccardi. 2007. Generative and discriminative algorithms for spoken language understanding. In *InterSpeech*.

S. Reddy, M. Lapata, and M. Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics (TACL)*, 2(10):377–392.

S. Reddy, O. Täckström, M. Collins, T. Kwiatkowski, D. Das, M. Steedman, and M. Lapata. 2016. Transforming dependency structures to logical forms for semantic parsing. In *Association for Computational Linguistics (ACL)*, pages 127–140.

S. Reed and N. de Freitas. 2016. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*.

S. Riedel and A. McCallum. 2011. Robust biomedical event extraction with dual decomposition and minimal domain adaptation. In *Proceedings of the BioNLP Shared Task 2011 Workshop*.

S. Riedel, L. Yao, and A. McCallum. 2010. Modeling relations and their mentions without labeled text. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 148–163.

S. Riedel, L. Yao, and A. McCallum. 2013. Relation extraction with matrix factorization and universal schemas. In *North American Association for Computational Linguistics (NAACL)*.

S. Robertson and H. Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3.

S. Sarawagi and S. Chakrabarti. 2014. Open-domain quantity queries on web tables: annotation, response, and consensus models. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 711–720.

M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. 2016. Bidirectional attention flow for machine comprehension. *arXiv*.

C. Shen and Q. Zhao. 2014. Webpage saliency. In *European Conference on Computer Vision (ECCV)*.

K. Shi, J. Steinhardt, and P. Liang. 2019. FrAngel: Component-based synthesis with control structures. In *Principles of Programming Languages (POPL)*.

T. Shi, A. Karpathy, L. Fan, J. Hernandez, and P. Liang. 2017. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning (ICML)*.

K. Soh. 2017. TagUI: RPA / CLI tool for automating user interactions. https://github.com/kelaberetiv/TagUI.

A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Programming Language Design and Implementation (PLDI)*.

R. Soricut and E. Brill. 2006. Automatic question answering using the web: Beyond the factoid. *Information Retrieval*, 9.

L. Spalteholz, K. F. Li, N. Livingston, and F. Hamidi. 2008. Keysurf: a character controlled browser for people with physical disabilities. In *World Wide Web (WWW)*.

A. Spengler and P. Gallinari. 2010. Document structure meets page layout: loopy random fields for web news content extraction. *ACM Symposium on Document Engineering*.

M. Steedman. 1996. *Surface structure and interpretation*. MIT press.

M. Steedman. 2000. *The Syntactic Process*. MIT Press.

Y. Su, H. Sun, B. M. Sadler, M. Srivatsa, I. Gur, Z. Yan, and X. Yan. 2016. On generating characteristic-rich question sets for QA evaluation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Y. Su and X. Yan. 2017. Cross-domain semantic parsing via paraphrasing. In *Empirical Methods in Natural Language Processing (EMNLP)*.

F. M. Suchanek, G. Kasneci, and G. Weikum. 2007. YAGO: a core of semantic knowledge. In *World Wide Web (WWW)*, pages 697–706.

A. Suhr and Y. Artzi. 2018. Situated mapping of sequential instructions to actions with single-step reward observation. In *Association for Computational Linguistics (ACL)*.

A. Suhr, M. Lewis, J. Yeh, and Y. Artzi. 2017. A corpus of natural language for visual reasoning. In *Association for Computational Linguistics (ACL)*.

A. Suhr, S. Zhou, I. F. Zhang, H. Bai, and Y. Artzi. 2018. A corpus for reasoning about natural language grounded in photographs. *arXiv preprint arXiv:1811.00491*.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*.

A. Talmor and J. Berant. 2018. The web as knowledge-base for answering complex questions. In *North American Association for Computational Linguistics (NAACL)*.

L. R. Tang and R. J. Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *European Conference on Machine Learning (ECML)*, pages 466–477.

M. Tapaswi, Y. Zhu, R. Stiefelhagen, A. Torralba, R. Urtasun, and S. Fidler. 2016. Movieqa: Understanding stories in movies through question-answering. In *Computer Vision and Pattern Recognition (CVPR)*, pages 4631–4640.

S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. J. Teller, and N. Roy. 2011. Understanding natural language commands for robotic navigation and mobile manipulation. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

A. M. Turing. 1950. Computing machinery and intelligence. *Mind*, 49:433–460.

C. Unger, L. Bühmann, J. Lehmann, A. Ngonga, D. Gerber, and P. Cimiano. 2012. Template-based question answering over RDF data. In *World Wide Web (WWW)*, pages 639–648.

C. Unger and P. Cimiano. 2011. Pythia: compositional meaning construction for ontology-based question answering on the semantic web. In *Proceedings of the 16th international conference on Natural language processing and information systems*, pages 153–160.

P. Venetis, A. Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, and C. Wu. 2011. Recovering semantics of tables on the web. In *Very Large Data Bases (VLDB)*, volume 4, pages 528–538.

E. M. Voorhees and D. Harman. 1999. Overview of the eight text retreival conference (TREC-8). In *TREC-8*.

C. Wang, K. Tatwawadi, M. Brockschmidt, P. Huang, Y. Mao, O. Polozov, and R. Singh. 2018. Robust text-to-SQL generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100*.

C. Wang, N. Xue, and S. Pradhan. 2015a. A transition-based algorithm for AMR parsing. In *Association for Computational Linguistics (ACL)*.

X. Wang, I. Dillig, and R. Singh. 2017. Program synthesis using abstraction refinement. In *Principles of Programming Languages (POPL)*.

Y. Wang, J. Berant, and P. Liang. 2015b. Building a semantic parser overnight. In *Association for Computational Linguistics (ACL)*.

J. Weston, S. Chopra, and A. Bordes. 2015. Memory networks. In *International Conference on Learning Representations (ICLR)*.

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

T. Winograd. 1972. *Understanding Natural Language*. Academic Press.

Y. W. Wong and R. J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Association for Computational Linguistics (ACL)*, pages 960–967.

Y. W. Wong, D. Widdows, T. Lokovic, and K. Nigam. 2009. Scalable attribute-value extraction from semi-structured text. In *IEEE International Conference on Data Mining Workshops*, pages 302–307.

N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Programming Language Design and Implementation (PLDI)*.

M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. 2012. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *International Conference on Management of Data (SIGMOD)*.

Y. Yang, W. Yih, and C. Meek. 2015. WikiQA: A challenge dataset for open-domain question answering. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 2013–2018.

Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. W. Cohen, R. Salakhutdinov, and C. D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Empirical Methods in Natural Language Processing (EMNLP)*.

X. Yao, J. Berant, and B. Van-Durme. 2014. Freebase QA: Information extraction or semantic parsing. In *Workshop on Semantic parsing*.

X. Yao and B. Van-Durme. 2014. Information extraction over structured data: Question answering with Freebase. In *Association for Computational Linguistics (ACL)*.

T. Yeh, T. Chang, and R. Miller. 2009. Sikuli: using GUI screenshots for search and automation. In *User Interface Software and Technology (UIST)*.

W. Yih, M. Richardson, C. Meek, M. Chang, and J. Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Association for Computational Linguistics (ACL)*.

P. Yin, Z. Lu, H. Li, and B. Kao. 2016. Neural enquirer: Learning to query tables with natural language. *arXiv*.

P. Yin and G. Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Association for Computational Linguistics (ACL)*, pages 440–450.

T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. R. Radev. 2018a. SyntaxSQLNet: Syntax tree networks for complex and cross-domaintext-to-SQL task. In *Empirical Methods in Natural Language Processing (EMNLP)*.

T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Empirical Methods in Natural Language Processing (EMNLP)*.

W. Yu, R. Kuber, E. Murphy, P. Strain, and G. McAllister. 2005. A novel multimodal interface for improving visually impaired people's web accessibility. *Virtual Reality*, 9.

Y. Yue, T. Finley, F. Radlinski, and T. Joachims. 2007. A support vector method for optimizing average precision. In *ACM Special Interest Group on Information Retreival (SIGIR)*.

M. Zavershynskyi, A. Skidanov, and I. Polosukhin. 2018. NAPS: Natural program synthesis dataset. In *Workshop on Neural Abstract Machines & Program Induction (NAMPI)*.

M. Zelle and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1050–1055.

D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao. 2014. Relation classification via convolutional deep neural network. In *International Conference on Computational Linguistics (COLING)*.

L. S. Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence (UAI)*, pages 658–666.

L. S. Zettlemoyer and M. Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*, pages 678–687.

L. S. Zettlemoyer and M. Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*.

C. Zhang. 2015. *DeepDive: a data management system for automatic knowledge base construction*. Ph.D. thesis, University of Wisconsin-Madison.

Y. Zhang, P. Pasupat, and P. Liang. 2017. Macro grammars and holistic triggering for efficient semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Z. Zhang, K. Q. Zhu, H. Wang, and H. Li. 2013. Automatic extraction of top-k lists from the web. In *International Conference on Data Engineering*.

V. Zhong, C. Xiong, and R. Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.