Generating Computer Programs from Natural Language Descriptions

by

Nate Kushman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author ..... Signature redacted
Department of Electrical Engineering and Computer Science
August 28, 2015

Certified by ..... Signature redacted
Regina Barzilay
Professor
Thesis Supervisor

Accepted by ..... Signature redacted
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Generating Computer Programs from Natural Language Descriptions

by

Nate Kushman

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis addresses the problem of learning to translate natural language into pre-existing programming languages supported by widely-deployed computer systems. Generating programs for existing computer systems enables us to take advantage of two important capabilities of these systems: computing the semantic equivalence between programs, and executing the programs to obtain a result. We present probabilistic models and inference algorithms which integrate these capabilities into the learning process. We use these to build systems that learn to generate programs from natural language in three different computing domains: text processing, solving math problems, and performing robotic tasks in a virtual world. In all cases the resulting systems provide significant performance gains over strong baselines which do not exploit the underlying system capabilities to help interpret the text.

Thesis Supervisor: Regina Barzilay
Title: Professor

Dedicated to my wife Deb,
who has supported me, loved me,
and most importantly been my friend
throughout this entire process.

Acknowledgments

I had the unique opportunity (at least in the current day and age) of working with two different female advisors at MIT: Regina Barzilay and Dina Katabi. They are two of the most effective and productive researchers that I have ever met, and I am honored to have had the opportunity to work with both of them. Without Regina, none of the research in this thesis would have happened. She gave me the freedom to pursue my crazy ideas, but at the same time always pushed me hard to find the nugget of innovation at the core of what I was doing. Dina advised me during the early years of my time as graduate student, and first taught me the fundamentals of doing great research. She also drilled into me the importance of not just generating good ideas, but communicating them effectively as well. I am forever indebted to both Regina and Dina for everything they have done for me over the years.

I am also very fortunate to have had the opportunity to work with Luke Zettlemoyer. Luke not only served on my thesis committee, but also directly collaborated with me on the math word problems work, and served as a great sounding board for me for the last couple of years of my graduate studies.

Beyond my thesis committee, three other people collaborated with me on the work that ended up in this thesis: S.R.K. Branavan, Tao Lei and Yoav Artzi. I especially want to thank Branavan, who initially sparked my interest in NLP, cemented that interest through our papers together, and inspired the general research direction pursed in this thesis. I also had the pleasure of collaborating with many other people during my time as a graduate student. This list includes Fadel Adib, Micah Brodsky, Farinaz Edalat, Oren Etzioni, Hannaneh Hajishirzi, Hariharan Rahul, Javad Hosseini, Srikanth Kandula, Hooria Komal, Kate Lin, Sam Madden, Bruce Maggs, Martin Rinard, and Charles Sodini. I learned a lot from each of you about how to do great research, and I consider our collaborations to be some of my most productive and enjoyable times as a graduate student. I especially want to thank Hariharan Rahul who provided significant help with an early version of the presentation which was eventually used during my thesis defense.

Beyond my direct collaborators, I sat near and worked with an amazing group of people in my time as a member of both *RBG* and *DOGS*. While in *RBG* I benefited tremendously from interactions with Yonatan Belinkov, Edward Benson, Yevgeni Berzak, Aria Haghighi, Harr Chen, Jacob Eisenstein, Zach Hynes, Mirella Lapata, Yoong Keok Lee, Chengtao Li, David Alvarez Melis, Karthik Narasimhan, Tahira Naseem, Neha Patki, Roi Reichart, Christy Sauper, Elena Sergeeva, Ben Snyder and Yuan Zhang. I especially want to thank David Alvarez Melis who painstakingly corrected the notational mistakes in Appendix B, and Neha Patki who helped produce the finance dataset used in Chapter 3. In my time with *DOGS*, I enjoyed discussions and chance encounters with Omid Abari, Nabeel Ahmad, Tural Badirkhanli, Shyam Gollakota, Haitham Hassanieh, Wenjun Hu, Szymon Jakubczak, Mike Jennings, Sachin Katti, Swarun Kumar, Samuel Perli, Lixin Shi, Jue Wang, and Grace Woo. Additionally, both Marcia Davidson and Mary McDavitt have generously provided both great administrative support as well as much appreciated words of encouragement over the years.

Beyond my immediate research groups, I had fruitful discussions with many other people from both MIT as well as the larger research community. Discussions with Stefanie Tellex and Kai-yuh Hsiao cemented my interest in the general area of grounded natural language, while discussions with Ray Mooney furthered my thinking on ideas related to mapping natural language to computer programs. I gained significant insight from many people in Josh Tenenbaum's CCS group, including especially Timothy O'Donnell, Eyal Dechter and Josh himself. Rishabh Singh, David Karger and Michel Goemans helped with early versions of the algorithms used in the system for solving math word problems, and working with Michel at Akamai inspired my decision to return to graduate school in the first place. I also had productive chance encounters with Andrei Barbu, Hari Balakrishnan, Andreea Gane, Jim Glass, Amir Globerson, Tommi Jaakola, Fan Long, Paresh Malalur, Armando Solar-Lezama and Yu Xin.

I want to thank my friends and family for all their support throughout the years. Claire Flood graciously agreed to read through this thesis multiple times in order to

correct my many grammatical errors. Greg and Angela Christiana were always up for long philosophical discussions that helped me to fit my research into the larger picture of my life. My in-laws, Cathy and Yih-huei, were always willing to help out at home while I was working long hours before deadlines (including the one for this thesis). My mom has always been enthusiastically supportive of my work even if she did not always understand what it is I actually do. My dad and Valerie always provided me a welcoming place to visit or call even if I did not always hold up my end of the bargain. My brother, Tim, has never given up on me, despite my frequent lapses in communication.

Finally, last but not least I want to thank my wife, Deb. Her patience, love, and support over the years is more than I could have asked for, and certainly more than I deserve.

Bibliographic Note

The main ideas in this thesis have all been previously published in peer-reviewed conferences. The list of publications by chapter is as follows:

Chapter 2: Generating Regular Expressions

- **Using Semantic Unification to Generate Regular Expressions from Natural Language**
  Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL), 2013.

Chapter 3: Automatically Solving Math Word Problems

- **Learning to Automatically Solve Algebra Word Problems**
  Proceedings of the Annual Meeting of the Association for Computations Lingusitics (ACL), 2014.

Chapter 4: Learning to Generate Plans from Text

- **Learning High-Level Planning from Text**
  Proceedings of the Annual Meeting of the Association for Computations Lingusitics (ACL), 2012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last several decades, the dramatic increase in the capabilities of computer systems has enabled them to transform from special purpose devices locked away in our offices, to general computing devices that are integrated into our every day lives. This ubiquity has generated a desire for lay users to be able to easily perform ever more complicated and specialized tasks. Despite significant automation, many such tasks still require writing some form of computer program which fully specifies every detail in an arcane formal language. Writing such programs, however, requires specialized skills possessed by only a fraction of the population.

In contrast, most humans are comfortable with and adept at using natural language to express their desires. For this reason, human to human interaction happens primarily through natural language, and both systems developers as well as users have generated extensive natural language documentation describing how to use computer systems. In fact, with the rise of the Internet, most tasks that a user would like to perform have been described on web forums, wikis or some other form of natural language content. Even the design of computer systems themselves is often described in detail in a natural language specification before it is built using a formal programming language.

The ultimate goal of the work in this thesis is to enable lay users to program computer systems through the use of natural language. The idea of programming with natural language has been around since the early days of computer program-

Natural Language	What is the highest mountain in Alaska?	
Logical Program	`(answer (highest (mountain (loc_2 (stateid alaska:e)))))`	
Alignment	What is	`answer`
	the highest	`highest`
	moutain	`mountain`
	in	`loc_2`
	Alaska	`Alaska`

Figure 1-1: Example from the GeoQuery domain of natural language mapped to a logical programming language which was specifically designed to syntactically align with natural language.

ming [114]. Early work in this area tried to handle open domain programming, but focused on rule-based techniques which could handle only very specific natural language constructions. This resulted in what were essentially verbose and ultimately unsuccessful formal languages which looked more like English [7, 53]. Consequently, these early systems generated negative criticism from the community [36], and discouraged continued research in this direction. More recent work has achieved success by focusing on using modern machine learning techniques to tackle constrained natural language domains. The most well-known work is the successful mapping of natural language to knowledge-base queries [143, 83], but other domains have seen success as well [16, 127]. To achieve this success, most of the existing work has not only constrained the domain of the natural language, but has also constrained the space of programming languages to those designed specifically to align very closely to the natural language. These purpose-built programming languages are typically based on predicate or first-order logic where nouns map to logical entities, and verbs and/or other syntactic connectives in the natural language map to logical functions. An example of such a language can be seen in Figure 1-1, with additional examples in Appendix A.

This thesis moves beyond languages like these by presenting techniques and systems to map natural language to preexisting programming languages that were not specifically designed to align closely to natural language. In order to maintain problem tractability, we do not attempt to handle completely open-domain programs. Instead we aim to find a middle ground – constrained programming domains which are

Natural Language	An amusement park sells 2 kinds of tickets. *Tickets for children cost $1.50. Adult tickets cost $4.* On a certain day, 278 people entered the park. *On that same day the admission fees collected totaled $792.* How many children were admitted on that day? How many adults were admitted?
Computer Program	$x + y = 278$ $1.5x + 4y = 792$

(a)

Natural Language	Find all lines containing three letter word starting with 'a'.
Computer Program	`\ba[A-Za-z]{2}\b`

(b)

Natural Language	A pickaxe is used to harvest stone and can be made from wood.
Computer Program	step 1: move from (0,0) to (2,0) step 2: chop tree at: (2,0) step 3: get wood at: (2,0) step 4: craft plank from wood step 5: craft stick from plank step 6: craft pickaxe from plank and stick ... step N-1: pickup tool: pickaxe step N: harvest stone with pickaxe at: (5,5)

(c)

Figure 1-2: Examples of three different natural language task descriptions and their associated computer programs: (a) a natural language finance question which can be solved with a computer program consisting of a system of mathematical equations, (b) a text processing query and a regular expression which embodies its meaning, and (c) a partial description of a virtual world, and a robotic program to perform a task based on the world description.

broad enough to be widely applicable. Examples of the three programming language domains we consider can be seen in Figure 1-2. These three domains: regular expressions, math equations, and robotic actions are all constrained enough to be tractable and yet general purpose enough to be widely deployed. The primary motivation in developing techniques to work with existing off-the-shelf programming languages is to enable extension to new domains without requiring a linguistics or Natural Language Processing (NLP) expert to design a new language for that domain. To this end, we would also like to learn from resources generated by untrained users of these systems, rather than by linguistics/NLP experts and/or workers following extensive annotation guidelines written by such experts [8]. Thus an additional advantage of existing programming languages is the ready availability of many such resources, including both textual resources describing the domains and how to perform tasks in these domains, as well as programmatic resources consisting of existing programs already written for the domain. Furthermore, we can easily generate additional resources by taking advantage of the many programmers who are already familiar with programming on such systems.

Our work seeks to make progress on both a short term agenda, and a long term agenda. In the short term, we present systems that map natural language to widely deployed programming languages such as regular expressions, or math equations which are each supported by a variety of different computer applications. Further refinement of our techniques and datasets may lead directly to systems which can be practically deployed. We also make progress towards a long term vision of generating programs in general-purpose Turing-complete languages such as *Java* or *C* from a somewhat broad (although probably never completely open) set of natural language domains. Attaining this long term goal would not only enable lay non-programmers to generate sophisticated custom computer programs in such domains, but also allow expert programmers to quickly prototype new applications in these domains based on natural language specifications. Achieving such a vision is likely an AI-hard problem [140], however, and thus well beyond the scope of a single thesis. Nonetheless, we are able to make progress towards it by working with existing off-the-shelf pro-

gramming languages since this setup exhibits some of the core challenges presented by the more general problem, while avoiding the need to fully solve it.

1.1 Challenges

Mapping natural language to existing programming languages poses two major challenges which we must overcome: (1) the mismatch in abstraction between the natural language and the resulting programs and (2) the lack of labeled training data.

1.1.1 Mismatch in Abstraction

Since existing programming languages are not designed to map directly to natural language, there is often a mismatch in the abstractions used by the natural language, and those used by the computer programs. Consider, for example, the simple finance word question in Figure 1-2(a), which we would like to translate into the associated program consisting of a system of mathematical equations. Unlike the programming languages considered by the prior work, as shown in Figure 1-1 and Appendix A, there is no clear connection between the syntactic structure of the natural language and the structure in the resulting program. For example, generating just the first equation requires information from three different non-contiguous sentences in the original text. Furthermore, the second equation models an implicit semantic relationship not explicitly stated in the text, namely that the children and adults admitted are non-intersecting subsets of the set of people who entered the park.

More generally, the mismatch in abstraction between the natural language and the underlying programs manifests itself in three different ways:

- **Lack of Syntactic Alignment:** There is often no clear alignment between the syntax of the natural language, and the syntax of the resulting computer programs. For example, in Figure 1-3 we can see that there is no syntactic alignment between the natural language and associated program from Figure 1-2(b). In contrast, Figure 1-1 shows how the logical programming languages

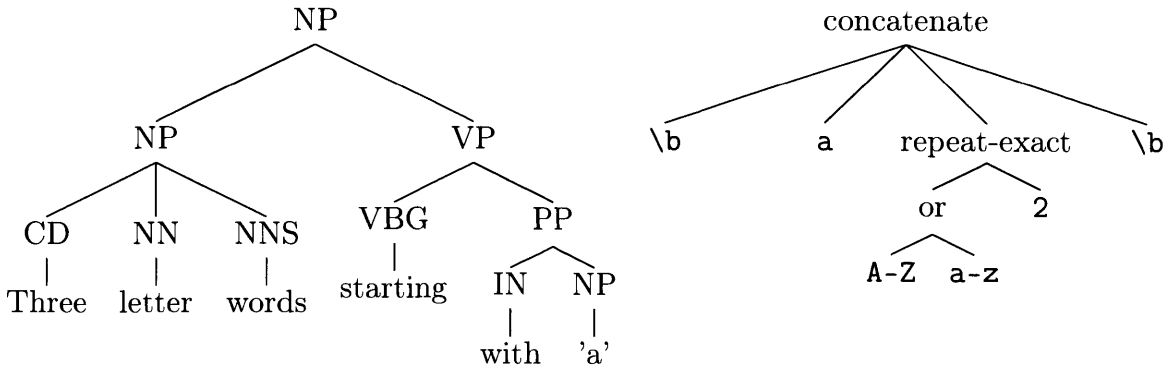


Figure 1-3: This shows the syntax trees for the sentence "Three letter words starting with 'a"' and the associated regexp `\ba[A-Za-z]{2}\b` from Figure 2-1. We can see that the two trees do not have any shared structure and thus cannot be syntactically aligned.

of the past work have been designed to ensure a parallel syntactic structure between the natural language and the programmatic representation.

- **Implied Cross-Sentence Relationships:** In order to maintain conciseness, natural language often does not state explicitly all information required for understanding. For example, understanding the finance problem in Figure 1-2(a) requires recognizing that the total set of 792 tickets is made up of the non-overlapping sets of adult tickets and child tickets. This information is never stated explicitly in the problem, and yet the problem cannot be answered correctly without it.

- **Lack of Sufficient Detail:** Natural language often describes a domain at much higher abstraction level than the programmatic primitives. Consider, for example, the text in Figure 1-2(c) describing a virtual world. This one sentence maps to dozens of low-level programmatic primitives. While the text provides a high-level description of the dynamics of the domain, it does not provide sufficient details for successful task execution.

1.1.2 Lack of Labeled Training Data

Traditional systems for generating programs from natural language rely on labeled training data [142, 138, 69, 108, 130, 70, 84, 45, 139, 73, 72, 5]. While crowd-sourcing techniques such as Amazon Mechanical Turk have significantly reduced the cost of generating such data [131], it is still a costly and labor intensive process. Furthermore, training data typically must be generated for each combination of programming language and natural language domain. As a result, the lack of such training data has limited the deployment of existing techniques to the small number of domains and languages for which large amounts of training data has already been generated.

In contrast, existing programming language domains have large amounts of *unlabeled* data readily available, including both natural language documentation, as well as existing computer programs. Most existing techniques cannot take advantage of this data, however, since it does not provide any mapping between the natural language and the programs.

1.2 Techniques

This thesis addresses the above challenges by taking advantage of the fact that we are generating programs backed by existing computer systems. The underlying computer systems enable two important capabilities.

- **Computing Semantic Equality:** We tackle the mismatch in abstraction by utilizing the underlying computers system's ability to compute the semantic equality between seemingly very different computer programs. Our work is the first to integrate this semantic inference capability into the learning process. Specifically, sufficiently powerful programming languages typically have many syntactically different programs which express the same semantic idea, meaning the programs will generate the same output for *any* input. The underlying computer systems often have inference engines that allow us to compute these equalities. Thus, when there is a *mismatch in abstraction* in our training data

between the natural language and the resulting programs, we can transform this program to a different, semantically equivalent program which provides abstractions which align well with the abstractions used in the natural language.

- **Computing Execution Outcomes:** We tackle the lack of labeled training data by using the underlying computer system's ability to execute potential output programs in order to generate a result. These execution outcomes can then be integrated into the learning process to augment or replace the direct supervision coming from training data. Consider, for example, the robotic task in Figure 1-2(c). A correct interpretation of the text would indicate that if we wanted to get some *stone* then we should first construct a *pickaxe*. We can provide this interpretation as a constraint to an underlying system that generates task execution programs given a particular goal. Using this constraint the underlying system may be able to successfully generate a valid plan for the task, when it otherwise it could not. We can use this success as an indication that our interpretation of the text was correct, and use this signal as a basis for learning – replacing traditional supervised data.

  We integrate the execution results into the learning process by building on past work utilizing reinforcement learning to interpret text [16]. The key distinction of our work is that the underlying system directly helps to generate pieces of the output program. This supplements the previous work which has focused on using the underlying system to execute potential output programs, or program steps for feedback [16, 133, 133, 15, 18, 19, 20].

1.3 Systems

We show the strength of the above two ideas by using them to learn to generate computer programs from natural language in three different domains:

- **Generating Regular Expressions:** We start by learning to map single-sentence natural language text queries to regular expressions which embody

their meaning. The main challenge comes from the *lack of syntactic alignment* between the natural language and the regular expressions in the training data, such as those in Figure 1-3. We tackle this challenge by using the semantic inference capabilities of the underlying computer system to find **semantically equivalent** programs which *do* syntactically align with the natural language.

- **Solving Algebra Word Problems:** We then learn to automatically solve algebra word problems by mapping them to a system of mathematical equations which can be solved to generate the numerical answer to the problem. The main additional challenge in this domain comes from *implied cross-sentence relationships* such as those seen in Figure 1-2a. We tackle this challenge by jointly predicting the system of equations and its alignment with the natural language, allowing a very general space of possible alignments. We handle the large size of the resulting search space using both the ability to determine the **semantic equality** between programs, as well as the ability to generate **execution outcomes**.

- **Generating Robotic Task Plans:** Finally, we learn to generate robotic task plans using natural language documentation that provides a general description of the domain. In this system we focus on a different aspect of the problem. We assume we have a formal description of a task goal, and we need to generate a step-by-step plan to achieve that goal. This is a specific instance of the more general problem of *program synthesis* which has been well studied in the programming languages literature [51]. The main challenge in our domain, as well as program synthesis in general, is the computational intractability of generating large plans/programs. We show that the natural language descriptions of the domain can be used to induce high-level relations which help guide the search process used by the program synthesis engine. We handle the lack of training data in this domain by learning to correctly interpret the text into high-level relations using only the **execution outcomes** of the program synthesis engine i.e. whether or not it is able to successfully synthesize a program

in a given amount of time.

In the rest of this chapter we give a brief overview of each of our three systems, highlight our contributions, and then give an outline of the rest of the thesis.

1.3.1 Generating Regular Expressions

We start by learning to map descriptions of text queries to regular expressions which embody their meaning. Despite not being Turing complete, regular expressions have proven themselves to be an extremely powerful and versatile formalism that has made its way into everything from spreadsheets to databases. However, they are still considered a dark art that even many programmers do not fully understand [42]. Thus, the ability to automatically generate regular expressions from natural language would be useful in many contexts.

Our goal is to learn to generate regular expressions from natural language, using a training set of natural language and regular expression pairs such as the one in Figure 1-2(b). The main challenge comes from the *lack of syntactic alignment* between the natural language and the resulting programs seen in many of the training examples, as shown in Figure 1-3. We tackle this challenge by utilizing the underlying regular expression engine to compute the semantic equivalence between seemingly different regular expressions. This equivalence calculation is done by converting the regular expressions to minimal deterministic finite automata (DFAs) which are guaranteed to be unique. When we encounter a syntactically misaligned training example, such as that in Figure 1-3, we use this DFA-based process to find another semantically equivalent regular expression which *is* syntactically aligned with the natural language.

Findings We find that our system generates the correct regular expression 65.5% of the time, compared to only 36.5% for a state-of-the-art semantic parsing model. Furthermore, we show that much of this gain stems from the integration of semantic equality calculations into the learning process.

1.3.2 Solving Algebra Word Problems

We next extend our techniques to translate a natural language math problem, such as that in Figure 1-2(a), into a system of equations which can be solved to generate the solution to the math problem. Mathematical equations, just like regular expressions, represent a constrained and yet highly expressive form of computer program. Many different underlying computer systems can be used to solve systems of math equations, from spreadsheets to computer algebra solvers. An enhanced version of our system could eventually be used to provide a constrained natural language interface to any of these systems.

The main additional problem that arises in the multi-sentence scenario is the implied cross-sentence relationships such as that in Figure 1-2(a). We tackle this challenge by jointly predicting the system of equations and its alignment with the natural language, allowing a very general space of possible alignments including complex multi-sentence mappings. This general model allows us to correctly predict the implied relationships, but leads to a very large space of possible systems of equations. To manage this explosion, we use the underlying symbolic math engine to compute the semantic equality between systems of equations. This allows us to avoid the duplicated computational cost of considering two different, but semantically equal, systems of equations during the inference process. Furthermore, we use the underlying math engine to solve possible full systems of equations to generate numeric solutions. These solutions are used to help determine whether or not the system of equations correctly represents the natural language question. For example if our solution indicates a fractional number of tickets, that is probably wrong, but a fractional price is perfectly reasonable. Rather than hand-writing rules like these, we include the potential correspondences as factors in our model allowing us to learn them from the training data.

Findings We find our model can correctly solve 70% of problems, which is more than twice as many as the 33% solved by a baseline which only considers the order

of the numbers in the text. Furthermore, integrating both execution outcomes, and semantic equality calculations into the learning process increases the performance by almost 15%.

1.3.3 Generating Robotic Task Plans

In some domains the challenge is not to formally represent the task in a programmatic format, but to determine the necessary steps to execute this program from this representation. Specifically, in the previous two tasks, we generated declarative computer programs, i.e. regular expressions, and mathematical equations. Declarative programs specify the goal of the computation *without* specifying the necessary steps to execute in order to achieve this goal. Instead we relied on the underlying systems, a regular expression engine and a mathematical solver, to efficiently determine the appropriate steps for execution based on the declarative program. In other domains, however, *program synthesis* like this is computationally intractable for complex programs.

Consider for example a robot performing tasks in a virtual world like *Minecraft.*[1] A typical task in this world might be to *get a stone*, which we could encode formally in the declarative program `HAVE(STONE)`. However, determining the set of necessary steps to get from a starting world state to a state where the robot has `STONE` is a well-studied NP-hard problem called classical planning [113].

We can, however, leverage natural language understanding in this task as well. Specifically, many computing domains have significant on-line natural language documentation describing the capabilities and dynamics of the domain. For example, the *Minecraft* virtual world has an expansive wiki describing the available tools, actions, and resources in the domain.[2] Our work asks the question of whether successful interpretation of this documentation can help with the NP-hard search problem.

The two main challenges in interpreting such text is the *lack of training data*, and

[1] http://www.minecraft.net/

[2] http://minecraft.gamepedia.com

the *mismatch in abstraction* between the high level natural language world descriptions and the low-level robotic programs we wish to generate. We turn this second challenge into an opportunity by utilizing an off-the-shelf robotic planner to generate the low-level programs, but using our interpretation of the text in order to induce high-level relations which can guide the search process of the low-level planner. We are able to handle the lack of training data by learning to correctly interpret the text for this guidance using only the **execution outcomes** of the low-level planner, i.e. whether or not it is able to successfully find a plan given the current guidance in a given period of time. Our technique builds on past work utilizing reinforcement learning techniques to interpret text [16, 133]. The key distinction of our work is that the underlying system directly helps to generate pieces of the output program rather than simply executing potential output programs, or program steps.

Findings Our results show that our technique for guiding the program generation process using textual information allows our system to generate correct programs 80% of tasks, compared to the 41% generated by a state-of-the-art baseline which does not utilize the textual information. Furthermore, using execution outcomes as its only source of supervision, our system can predict high-level relations on par with a directly supervised base-line.

1.4 Contributions

The main contributions of this thesis are a set of techniques and systems to enable the generation of computer programs from natural language descriptions. We make contributions both in new techniques, as well as the new capabilities enabled by those techniques.

The primary technical contributions of the thesis are to show that:

- **Semantic inference improves learning** We show that utilizing the semantic inference capabilities of the underlying domain can improve the effectiveness of learning to interpret natural language. Our work is the first to utilize semantic

inference in the learning process. We show this in both the regular expression domain and the math word problem domain.

- **Execution outcomes can be used to interpret natural language** We show that we can learn to interpret text without any supervised data by observing the outcome of executing possible programmatic interpretations of the text on the underlying computer domain. Our work is the first system for learning to interpret non-goal specific text without supervised data, using only execution outcomes. We show this in the domain of classical robotic planning.

We show the strength of these two techniques across three different systems, each representing a new capability.

- **Generating Regular Expressions** This thesis presents the first statistical system for generating regular expressions from natural language descriptions. This system is discussed in Chapter 2, and uses our semantic inference technique to enable more effective learning.

- **Solving Math Word Problems** We introduce the first fully statistical system for learning to automatically solve natural language word problems. This system is discussed in Chapter 3, and uses both semantic inference and execution outcomes in order to learn more effectively.

- **Performing Robotic Tasks** We describe the first system for utilizing non-goal specific text to help perform robotic task planning. This system is discussed in Chapter 4, and uses execution outcomes in order to learn to interpret text without any supervised data.

Finally, the machine learning algorithms we designed for each task represent a further contribution. Our model for each task includes a significant hidden component which models the relationship between the natural language and the resulting programs in a much more fine-grained manner than the supervision provided for each domain. Our algorithms integrate the semantic inference and/or execution capabilities of the

underlying domain in order to more effectively explore the space of possible hidden choices.

1.5 Outline

The remainder of this thesis is organized as follows.

- **Chapter 2** describes our system for translating single sentence natural language goal descriptions into computer programs. This section introduces the idea of semantic equality and shows how it can be integrated into the learning process.
- **Chapter 3** discusses the translation of multi-sentence natural language goal descriptions into computer programs. Our multi-sentence technique integrates both semantic equality as well as execution outcomes into the learning process.
- **Chapter 4** presents our technique for utilizing existing text resources describing a computer system in order to learn how to generate programs for that system. Our reinforcement learning technique utilizes execution outcomes in order to interpret the text without any training data.
- **Chapter 5** concludes with an overview of the main ideas and contributions of the thesis. We also discuss potential future research directions that build in the ideas presented in the rest of the thesis.

Chapter 2

Generating Regular Expressions

In order to make progress on the general problem of translating natural language descriptions into computer programs, we start with a constrained version of the problem. In this chapter, we consider only single sentence natural language descriptions of text processing tasks such as one might perform with the Unix `grep` command. We learn to map these commands to regular expressions, which are an expressive yet constrained type of computer program. Despite the constrained nature of the task, we show that the general problem of syntactic misalignment between the natural language and the computer programs arises. To tackle this challenge, we are going to utilize the first of the two important properties of computer programs – the ability to compute semantic equality. We will show that integrating this capability into the learning process significantly improves its performance.

2.1 Introduction

Despite the constrained nature of our task, the ability to automatically generate regular expressions (regexps) from natural language would be useful in many contexts. Specifically, regular expressions have proven themselves to be an extremely powerful and versatile formalism that has made its way into everything from spreadsheets to databases. However, despite their usefulness and wide availability, they are still considered a dark art that even many programmers do not fully understand [42].

Natural Language Description	Regular Expression
three letter word starting with 'X'	`\bX[A-Za-z]{2}\b`

Figure 2-1: An example natural language description and its associated regular expression.[2]

Natural Language Description	Knowledge-base Query
What is the highest mountain in Alaska?	`(answer (highest (mountain (loc_2 (stateid alaska:e)))))`

(a)

Natural Language Phrase	Knowledge-base Query Fragment
What is	`answer`
the highest	`highest`
mountain	`mountain`
in	`loc_2`
Alaska	`stateid alaska:e`

(b)

Figure 2-2: An example text description and associated database query from the GeoQuery dataset.

Our goal is to learn to generate regexps from natural language, using a training set of natural language and regular expression pairs such as the one in Figure 2-1. We do not assume that the data includes an alignment between fragments of the natural language and fragments of the regular expression. Inducing such an alignment during learning is particularly challenging because oftentimes even humans are unable to perform a *fragment-by-fragment* alignment.

On the surface, this task is similar to past work on a task called *semantic parsing* which has focused largely on mapping natural language to knowledge-base queries [69, 143, 73]. In that work, just as in our task, the goal is to map a natural language sentence to a formal logical representation, typically some form of knowledge-base query. The success of that work, however, has relied heavily on designing the knowledge-base schemas such that the resulting queries align very closely with the natural language. As a result, the datasets have three important properties:

- **Finding a Fragment-by-Fragment Alignment is Always Possible:** While

[2]Our regular expression syntax supports Perl regular expression shorthand which utilizes `\b` to represent a break (i.e. a space or the start or end of the line). Our regular expression syntax also supports intersection (&) and complement(~).

the past work did not assume the fragment-by-fragment alignment was given, they do assume that such an alignment is *possible* for all training examples. Figure 2-2(a) shows an example of a typical training sample from the GeoQuery dataset[142], one of the most commonly used datasets in this domain. Figure 2-2(b) shows the corresponding fragment-by-fragment alignment.

- **The Logical Representation Provides Strong Typing Constraints:** The entries and predicates in the knowledge-base are typically organized into ontologies which provide significant constraints on the set of valid queries. For example, the GeoQuery domain contains 10 different types, and 39 different relations, where most relations accept only one of the 10 different types for each argument. Recent work on the Freebase schema has shown that the typing constraints on this domain are strong enough to enable state-of-the-art results using techniques which build up *all possible* queries matching a set of pre-defined patterns [11].

- **Each Natural Language Word Maps to a Couple of Logical Operators:** The entities and relations in the knowledge-base schemas are chosen such that each one maps to a single word or a short natural language phrase. Only in special cases will a single natural language word map to a fragment of the query with more than one operator, and in these cases the number of operators is still typically very small [146]. For example, in Figure 2-2(b) we can see that each constant in the logical representation maps to one word in the natural language. In fact, Jones et al. (2012) were able to achieve state-of-the-art results on the GeoQuery dataset with a model which explicitly assumes that a single word cannot map to more than one logical operator.

When mapping natural language to general purpose programming languages such as regular expressions, however, we cannot hand design the logical representation to ensure that these properties are maintained. Specifically, we find that the alignment between the natural language and the regular expressions often happens at the level of the whole phrase, making a fine grained fragment-by-fragment alignment impossible.

`([A-Za-z]{3})&(\b[A-Za-z]+\b)&(X.*)`

(a)

Natural Language Phrase	Regular Expression Fragment
three letter	`[A-Za-z]{3}`
word	`\b[A-Za-z]+\b`
starting with 'X'	`X.*`

(b)

Figure 2-3: (a) shows a regexp which is semantically equivalent to that in Figure 2-1, yet admits a fragment-by-fragment mapping to the natural language. (b) shows this mapping.

Consider, for example, the regular expression in Figure 2-1. No fragment of the regexp maps directly to the phrase "three letter word". Instead, the regexp explicitly represents the fact that there are only two characters after `X`, which is not stated explicitly by the text description and must be inferred. Thus, such a training sample does not admit a fragment-by-fragment mapping similar to the one that we saw in Figure 2-2. Furthermore, the correct regular expression representation of even just the simple natural language phrase "word" is the regular expression `\b[A-Za-z]*\b` which is a combination of 10 operators in the regular expression language. This is significantly more complicated than the single word representations learned by the past work [146]. Lastly, regular expressions have relatively few typing constraints since most of the operators can accept any valid regular expression as an argument.

Key Ideas The key idea in this chapter is to utilize the ability to compute the semantic equivalence between syntactically very different programs to enable more effective learning of the meaning of the natural language. This is a departure from past work on semantic parsing which has focused on either the *syntactic* interface between the natural language and the logical form, or on execution-based equality, neither of which utilize the inference power inherent in traditional programming languages.

To see how we can take advantage of the ability to computer semantic equality, consider the regular expression in Figure 2-3(a). This regular expression is semantically equivalent to the regular expression in Figure 2-1. Furthermore, it admits a fragment-by-fragment mapping as can be seen in Figure 2-3(b). In contrast, as we

noted earlier, the regexp in Figure 2-1 does not admit such a mapping. In fact, learning can be quite difficult if our training data contains only the regexp in Figure 2-1. We can, nonetheless, use the regexp in Figure 2-3 as a stepping-stone for learning *if* we can use semantic inference to determine the equivalence between the two regular expressions. More generally, whenever the regexp in the training data does not factorize in a way that facilitates a direct mapping to the natural language description, we must find a regexp *which does factorize* and be able to compute its equivalence to the regexp we see in the training data. We compute this equivalence by converting each regexp to a minimal deterministic finite automaton (DFA) and leveraging the fact that minimal DFAs are guaranteed to be the same for semantically equivalent regexps [57].

We also depart from the past work through the use of a more effective parsing algorithm which allows us to handle the additional ambiguity stemming from both the weak typing and large number of logical operators associated with individual words. Specifically, the state-of-the-art semantic parsers [72, 83] utilize a pruned chart parsing algorithm which fails to represent many of the top parses and is prohibitively slow in the face of weak typing. In contrast, we use an n-best parser which always represents the most likely parses, and can be made very efficient through the use of the parsing algorithm from Huang and Chiang (2005).

Summary of Approach Our approach works by inducing a combinatory categorial grammar (CCG) [122]. This grammar consists of a lexicon which pairs words or phrases with lambda calculus expressions which can be combined compositionally to generate a regular expression. Our technique learns a noisy lexicon which contains both correct and incorrect entries. To choose the correct entries for a given sentence, we use a log-linear model to place a probability distribution over the space of possible choices. We integrate out ability to compute semantic equality into the process of learning parameters for this model, in order to maximize the probability of any regular expression which is semantically equal to the correct one. We learn the noisy lexicon through an iterative process which initializes the lexicon by pairing

each sentence in the training data with the full regular expression associated with it. These lexical entries are iteratively refined by considering possible ways to split the regular expression and possible ways to split the natural language phrase.

Evaluation We evaluate our technique using a dataset of sentence/regular expression pairs which we generated using Amazon Mechanical Turk [131] and oDesk [103]. We find that our model generates the correct regexp for 66% of sentences, while the state-of-the-art semantic parsing technique from Kwiatkowski et al. (2010) generates correct regexps for only 37% of sentences. The results confirm our hypothesis that leveraging the inference capabilities of general programming languages can help disambiguate natural language meaning.

2.2 Related Work

This section gives an overview of the four main areas of related work: natural language grounding, computer program induction, regular expressions and CCG parsing.

2.2.1 Natural Language Grounding

There is a large body of research learning to map natural language to some form of logical representation. Our work differs from all of this work in both our use of semantic equivalence during the learning process as well as in the domain that we consider.

Past Techniques for Computing Equivalence

Implicit in all fully supervised learning techniques is some form of equivalence calculation which is used in the learning process to determine whether or not the generated output is equal to the correct output. Each of the past supervised systems for language grounding uses one of the following three types of equivalence: string equivalence, local transformations, or execution equivalence. None of this work has utilized full semantic equivalence during the learning process as we do. There has also

Natural Language	Knowledge-base Query	Result
What is the largest city in Georgia?	`(answer (largest (city (loc_2 (stateid georgia:e)))))`	Atlanta
What is the capital of Georgia?	`(answer (capital (loc_2 (stateid georgia:e)))))`	Atlanta

Figure 2-4: An example showing the problem with using execution equivalence. These two knowledge-base queries both execute to the same result even though semantically they are very different

been some work on unsupervised language grounding where the notion of equivalence does not arise.

- **Exact String Equivalence** The traditional comparison used for logical representations is simply exact string equivalence [142, 138, 69, 108, 130, 70]. With this technique two logical representations are considered equivalent only if they generate exactly the same string when written down in a human readable format. We saw in Figure 2-1 how using this form of equivalence is problematic.

- **Local Transformation Equivalence** Given the limitations of exact string equivalence, some systems have augmented exact equivalence by allowing a heuristically chosen set of local changes on the logical form to handle minor syntactic inconsistencies [45, 139, 73, 72]. These are typically simple transformations such as reordering the arguments to an *and* operator, or performing simple tree transformations. None of these techniques, however, allow the replacement of one set of logical operators with a different, but semantically equivalent set, as is required by the regular expression domain.

- **Execution Equivalence** The last type of equivalence which has been used by the past supervised work is *execution equivalence* [16, 17, 15, 18, 29, 5, 31, 83]. This is the most similar to our use of full semantic equivalence. In this model, the final logical form is executed on some input to generate an output. The outputs are then compared for equivalence. For example, knowledge-base queries would be executed on a knowledge-base and declared equal if they generate the same result. In the case of regular expressions, we run the regular expressions on a set of input text strings, to see which strings each regexp matches. We

declare two regexps to be equal if they both match the same set of input strings. This provides a very weak form of semantic equivalence which is valid only on the provided input(s). It can recognize deeper semantic equivalences but it has the opposite problem – many semantically different logical forms will generate the same output on a given input. Consider for example the two knowledge-base queries in Figure 2-4. Both of the queries generate the same result, *Atlanta*, and so the learning process would consider these two queries to be equivalent even through they clearly represent very different semantic concepts. In contrast, our work uses an exact, theoretically sound measure of semantic equivalence that determines whether two logical representations are equivalent for all possible execution inputs. We show in Section §2.8 that utilizing full semantic equivalence results in much more effective learning.

- **Unsupervised Techniques** More recently there has been a focus on unsupervised and weakly supervised techniques which learn from alternative weaker sources of supervision besides direct training examples. For example, Poon (2013) bootstraps an unsupervised learning algorithm using lexical matching between words in the natural language and the names of relations and entities in the database to which they are grounding. Similarly, [5] learn to ground natural language to a knowledge-base by utilizing conversations between a human and a machine where they assume they know the full logical representation for the machine utterances. They design a heuristic loss function which tries to maximize the coherence between the human's utterances and the machines. Both Liang et al. (2009) and Chen and Mooney (2008) learn from unaligned sequences of natural language and actions. In all of these techniques, the notion of equivalence does not directly arise since they are not trying to match their outputs to those seen in a training dataset.

Domains Considered

Past work on mapping natural language to logical representations has largely focused on knowledge-base queries and control applications, with some work on computer programs as well.

- **Knowledge-Base Queries** The vast majority of language grounding work has focused on mapping natural language to knowledge-base queries [142, 138, 69, 108, 130, 70, 45, 139, 73, 72, 23, 22, 12, 74]. Almost all of this work has focused on the JOBS, GEO, ATIS and Freebase domains. In these domains the knowledge-base schemas were manually designed to align very closely to the syntax of natural language. As we discussed in Section §2.1, this ensures that a fragment-by-fragment alignment is always possible, and that each natural language word maps to only a couple of logical operators. Similarly, Pasupat and Liang (2015) work with tables from *Wikipedia* which have similar properties. These properties obviate the need for any kind of semantic inference during the learning process. In our work, we consider the problem of mapping natural language to more general programming languages where we cannot rely on such properties to be true.

- **Control Applications** Grounding natural language to control applications such as robots has been an area of interest in Artificial Intelligence research since Terry Winograd first built his SHRDLU system to interpret natural language instructions in a virtual blocks world [136]. While early systems such as SHRDLU were rule based [35, 59, 60], modern systems instead use machine learning algorithms which take advantage of either direct training examples [29, 5, 127, 128], or indirect weaker forms of supervision [16, 17, 15, 18].

  Similar to the work on knowledge-base queries, much of the work on control applications has relied on hand designed logical representations that closely align with the natural language. Machine learning techniques are used to map the natural language to this representation, which is then deterministically converted into actions in the underlying control application. For example, Artzi

and Zettlemoyer (2013) manually designed a set of logical predicates which cover the types of entities and action sequences seen in their dataset. Chen and Mooney (2011) designed a grammar they called "landmark plans" which models the same dataset in a different way.

In contrast, our goal is to avoid the need to hand design a new logical language for each domain by instead learning to work with more general purpose programming languages such as regular expressions. The most similar work to ours in the domain of control applications is that by Branavan et. al. where they learn to map natural language directly to actions in the underlying control application without a hand designed intermediate logical representation [16, 17]. Vogel and Jurafsky (2010) use a similar technique except they learn from training examples rather than directly from environment feedback. This work was a step in the same direction as ours, however it all utilized flat sequence models of the natural language which cannot handle the complicated compositional constructions seen in more general natural language datasets such as ours. Additionally, Tellex et al. (2011) present a formalism called Generalized Grounding Graphs, which performs situated interpretation of natural language instructions. While their system translates directly to the underlying robotic actions, their formalism is heavily oriented towards spatial commands, and thus not appropriate for general programmatic domains such as ours.

- **Computer Programs** There has been some past work on mapping natural language to computer programs. Most of the early work in this area focused on rule-based systems [92, 107, 35, 60, 43]. For example, work in the HCI community on a technique called *Keyword Programming* focused on simple word overlap techniques [85]. Work in the programming languages community has focused on rule-based techniques [77, 109, 52] that typically interact with the user to resolve ambiguities, either by providing examples [109] or by choosing from a ranked list of possible interpretations [52].

  We believe our work was the first to use training data to learn to automatically

generate a general purpose program, such as a regular expression, from natural language. Since we originally published the work discussed in this chapter, more recent work has continued in this direction, mapping natural language to other somewhat general purpose programming languages. Lei et al. (2013) learn to map natural language to program input parsers. Quirk et al. (2015) considers mapping natural language to programs on the If-This-Then-That (IFTTT) website [3]. IFTTT programs, however, do not possess any compositional structure and are thus inherently quite simple and not truly general purpose. Allamanis et al. (2015) consider the problem of mapping natural language to general purpose programs written in *C* and *Java*. Given the difficulty of this problem, they focus on the simpler task of search ranking, rather than generating the output program directly from the natural language. They find that using vector-based Bag-of-Word representations allows them to achieve reasonable performance on this simpler task. Similarly, Movshovitz-Attias and Cohen (2013) consider the program of learning to predict program comments given a large dataset of already commented code. They find it is useful to model both the distinct properties of comments and code, and well as the joint properties they share. This recent work shows the significant interest in generating more general purpose programs directly from natural language, highlighting the importance of this task.

2.2.2 Generating Programs from Other Forms of Specification

In addition to work on generating programs from natural language, there has been a significant amount of work on generating programs from other sources of specification. This includes generating programs from examples [121, 118], high-level specifications [129], and program traces [75]. Gulwani (2010) provides a good overview of this space. This general line of work complements ours by developing techniques for generating programs from some form of specification, but does not address the issue

[3] `http://ifttt.com`

of natural language interpretation which is central to our work.

2.2.3 Generating Regular Expressions

Regular expressions are based on a body of work studying regular languages and their equivalent representations as finite automata [57]. Our work utilizes these classic results which show that regular expressions can be represented as minimal DFAs and then compared for equivalency [57]. Past work has looked at generating regular expressions from natural language using rule-based techniques [107], and also at automatically generating regular expressions from examples [3, 48]. To the best of our knowledge, however, our work is the first to use training data to learn to automatically generate regular expressions from natural language.

2.2.4 CCG Parsing

Our work builds on a rich history of research using Combinatory Categorial Grammars (CCGs) to learn a mapping from natural language to logical representations. CCG is a grammar formalism introduced and developed by Steedman (1996, 2000). Early work using machine learning with CCGs [54, 30] focused largely on broad coverage syntactic parsing using the CCGBank dataset [55], a version of the Penn Treebank that is annotated with full CCG syntactic parse trees. Our work utilizes a log-linear model similar in form to the one developed for syntactic parsing by Clark and Curran (2007), however we utilize a modified parsing algorithm. More recent work has focused on domain-specific semantic parsing techniques which generate a logical representation of a given natural language sentence. For example, Zettlemoyer and Collins (2005, 2007) learn a CCG to generate knowledge-base queries; Artzi and Zettlemoyer (2013) learns a CCG to interpret natural language robotic instructions; and Lee et al. (2014) generates formal time expressions from natural language utterances. Most of this work builds the grammar completely manually [74, 78], or through a combination of manual entries and domain-specific template-based generation [143, 144]. In contrast, we build on a line of work which learns the grammar from the data using an approach

based on syntactic unification [73, 72]. However, we depart from this work, and all prior CCG work, in our use of semantic equality during the learning process, our modified parsing algorithm, and the domain that we consider.

2.3 Generating Programs Using CCGs

This section describes the components of our model for learning to generate programs from natural language using Combinatory Categorial Grammars (CCGs). This section serves mostly as background, while the following sections describe the innovative aspects of our model. We start by describing how computer programs such as regular expressions can be represented as Abstraction Syntax Trees (ASTs). We then describe how we can use Combinatory Categorial Grammars (CCGs) to generate regular expression ASTs from natural language descriptions.

2.3.1 Representing Programs as Abstract Syntax Trees

While computer programs can be represented as a string of characters, this representation obscures the structure imposed by the programming language. Specifically, programming languages, unlike natural language, are typically designed so that they can be unambiguously parsed into tree format called an Abstract Syntax Tree (AST). ASTs explicitly represent the program structure, while dropping notational details such as commas and parenthesis. For this reason, most computer program manipulation tools work on the AST rather than directly on the textual representation of the program. Figure 2-5 shows the AST for the regular expression from Figure 2-1. Each of the nonterminals in the tree is a function in the regular expression language, while the terminals are either integers (to be used for the bounds in a repeat) or some type of character class. The full set of terminals and non-terminals is shown in Figure 2-6. Throughout this chapter we will refer to the regular expression string and its AST interchangeably, however, the AST representation is used mainly in our discussion of grammar induction in Section §2.5.2.

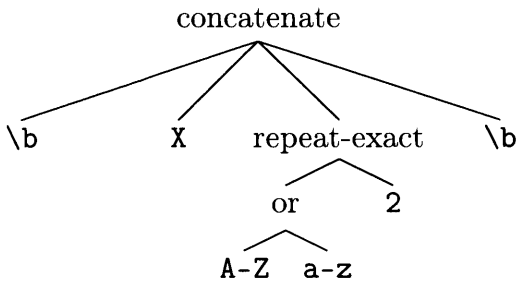


Figure 2-5: This shows the abstract syntax tree (AST) representation for the regexp `\bX[A-Za-z]{2}\b` from Figure 2-1.

function description	type signature	example regexp
concatenate	(regexp, regexp) → regexp	`ab`
and	(regexp,regexp) → regexp	`a&b`
or	(regexp, regexp) → regexp	`a|b`
not	regexp → regexp	`~a`
unbounded repeat	regexp → regexp	`a*`
fully bounded repeat	(regexp, int, int) → regexp	`(a){3,5}`
lower bounded repeat	(regexp, int) → regexp	`(a){3,}`
exact bounded repeat	(regexp, int) → regexp	`(a){3}`

(a)

terminal description	type	example regexp
character	regexp	`X`
character range	regexp	`a-z`
word boundary	regexp	`\b`
integer	int	2

(b)

Figure 2-6: This shows (a) the set of non-terminals, and (b) the set of terminals in the abstract syntax tree representation of regular expressions

2.3.2 Combinatory Categorial Grammars

Our parsing model is based on a Combinatory Categorial Grammar [123, 122]. A CCG will generate one or more possible logical representations for each sentence that can be parsed by its grammar. In CCG parsing most of the grammar complexity is contained in the lexicon, Λ, while the parser itself contains only a few simple rewrite rules called combinators.

Lexicon

A CCG lexicon, Λ, consists of a set of lexical entries that couple natural language with a lambda calculus expression. The lambda calculus expressions contain fragments of the full logical representation for a given sentence. In this thesis our focus is using CCGs to generate computer programs. So our lexical entries will use lambda calculus expressions which contain contiguous fragments of the ASTs that we discussed in Section §2.3.1. Formally, each lexical entry, l, is a tuple, $\langle l_{\vec{w}}, l_t : l_r \rangle$, where $l_{\vec{w}}$ is the natural language word or phrase, l_t is the syntactic type of the lambda calculus expression (as described below), and l_r is the lambda calculus expression itself.

Traditional semantic representations for natural language contain three basic types: e for entities, t for true values, and i for numbers [24]. For purposes of representing programs we will use only two of those types, e to represent a regular expression, and i for an integer. Thus every complete subtree of an AST has either type e or type i, and the full tree will always have type e. Incomplete tree fragments are then represented using lambda expressions with function types such as those in the following lexical entries:

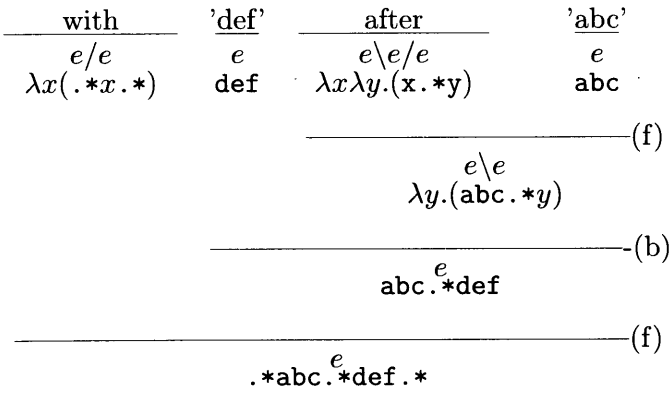
$$\langle \text{ after, } e\backslash e/e : \lambda x \lambda y.(x \,.* y) \rangle$$
$$\langle \text{ at least, } e/i/e : \lambda x \lambda y.(\texttt{(}x\texttt{)\{}y\texttt{,\}}) \rangle$$

Note that CCG syntactic types augment the traditional lambda calculus type information with a (/) or a (\) for each argument indicating whether that argument comes from the left or the right, in sentence order. Thus $e\backslash e/e$ can be read as a function which first takes an argument of type e on the right then takes another argument of type e on the left, and returns an expression of type e. [4]

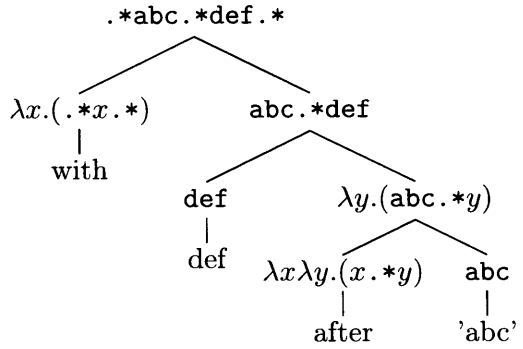
Combinators

Full CCG parses are built by combining lexical entries through the use of a set of combinators. Our parser uses only the two most basic CCG combinators, forward

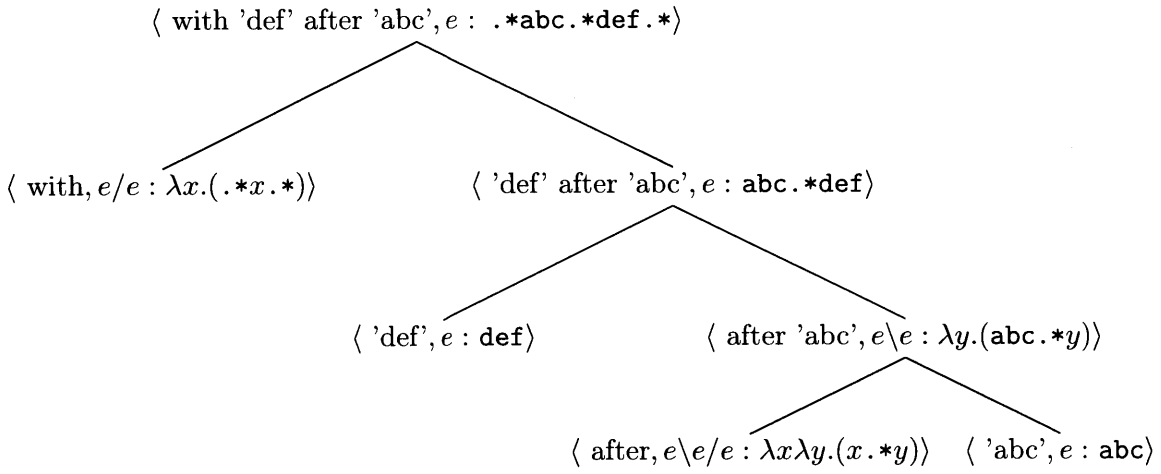
[4]Conventionally CCGs also augment the lambda calculus type information with a syntactic category (such as S/NP) for each lexical entry, which is used similar to the way part-of-speech tags are used in standard CFG parsing. Our lexical entries do not include this additional information.



(a) CCG format



(b) Simple CFG format



(c) Full CFG format

Figure 2-7: This shows a parse of the same example phrase in three different formats. (a) using traditional CCG format, (b) using traditional CFG format with just the regular expression fragment at each node of the tree, and (c) using CFG format with the full tuple $\langle \vec{w}, t : r \rangle$, at each node, where $\vec{w}$ is the natural language, t is the type, and r is the regular expression fragment.

function application and backward function application.[5] These combinators work as follows:

$$\begin{array}{llll} e/e:f & e:g & \rightarrow & e:f(g) \quad \text{(forward)} \\ e:f & e\backslash e:g & \rightarrow & e:g(f) \quad \text{(backward)} \end{array}$$

The forward combinator applies a lambda function to an argument on its right (in sentence order) when the type of the argument matches the type of the function's first argument. The backward combinator works analogously.

Parse Trees

The derivation of an AST from a natural language sentence using a CCG can be visualized using a tree, as in Figure 2-7. The leaves of the tree are the chosen set of lexical entries. At each non-terminal in the parse tree, a new lambda calculus expression is generated by the lambda function application of one child to the other using one of the combinators described earlier. Formally, a parse tree, t, consists of a choice of lexical entries, $t_l = \{l^{(i)} | i = 1...n\}$ at the leaves, as well as a set of non-terminal parse productions, $t_s = \{s^{(i)} \rightarrow s_l^{(i)} s_r^{(i)} | i = 1...m\}$ where $s^{(i)}$ is the syntactic type of the i^{th} non-terminal, and $s_l^{(i)}$ and $s_r^{(i)}$ are the types of the two children used to generate $s^{(i)}$. Note that we will allow our lexicon, Λ, to contain lexical entries with multiple natural language words to handle phrases such as "starts with" which do not naturally decompose in a compositional way. Thus the choice of lexical entries, t_l, for a given sentence, $\vec{w}$, must cover all words in the sentence without overlapping. Formally, the flattened list $\{l_{\vec{w}}^{(i)} : i = 1...n\}$ is equal to $\vec{w}$ where, as defined earlier, $l_{\vec{w}}^{(i)}$ is the natural language phrase in lexical entry $l^{(i)}$. The root of a valid parse tree must have a lambda calculus expression of type e, i.e. a valid regular expression.

[5]Technically, this choice of combinators makes our model just a Categorial Grammar instead of a CCG.

2.3.3 Probabilistic CCGs

For a given CCG lexicon, Λ, and sentence, $\vec{w}$ there will, in general, be many valid parse trees, $T(\vec{w}; \Lambda)$. To disambiguate among these, we assign a probability to each parse tree, $t \in T(\vec{w}; \Lambda)$ using a standard log-linear model with feature function ϕ and parameters θ:

$$p(t|\vec{w}; \theta, \Lambda) = \frac{e^{\theta \cdot \phi(t)}}{\sum_{t' \in T(\vec{w}; \Lambda)} e^{\theta \cdot \phi(t')}}$$

In general, for a given sentence, $\vec{w}$, there will be many different parse trees, t, which generate a given regular expression r, i.e. have r at their root. Thus in order to calculate the probability of generating r, we need to sum over the probability of all trees which have r at their root. If we define $\texttt{EVAL}(t)$ to extract the regular expression at the root of t, then:

$$p(r|\vec{w}; \theta, \Lambda) = \sum_{t \in T(\vec{w}, \Lambda) | \texttt{EVAL}(t) = r} p(t|\vec{w}; \theta, \Lambda)$$

Note that at test time, for computational efficiency reasons, we calculate only the most likely parse, t^*, and return $\texttt{EVAL}(t^*)$.

Parameter Estimation At training time, our goal is to learn from data which contains only natural language sentences paired with computer programs, i.e. regular expressions, which embody their meaning. Formally, each training sample, $s_i = \langle \vec{w}_i, r_i \rangle$, will contain a natural language sentence, $\vec{w}_i$, and a regular expression, r_i. Thus at training time, the standard objective is to maximize the regularized marginal log-likelihood of the data. Formally:

$$O = \sum_i \log \sum_{t | \texttt{EVAL}(t) = r_i} p(t|\vec{w}_i; \theta, \Lambda) - \frac{\lambda}{2} \sum_{k < |\theta|} \theta_k^2 \tag{2.1}$$

where k is the parameter index and λ is a meta-parameter controlling the level of regularization. This objective, which the past work has used, is limited, and in Section §2.4 we will discuss those limitations and how we modify the objective to

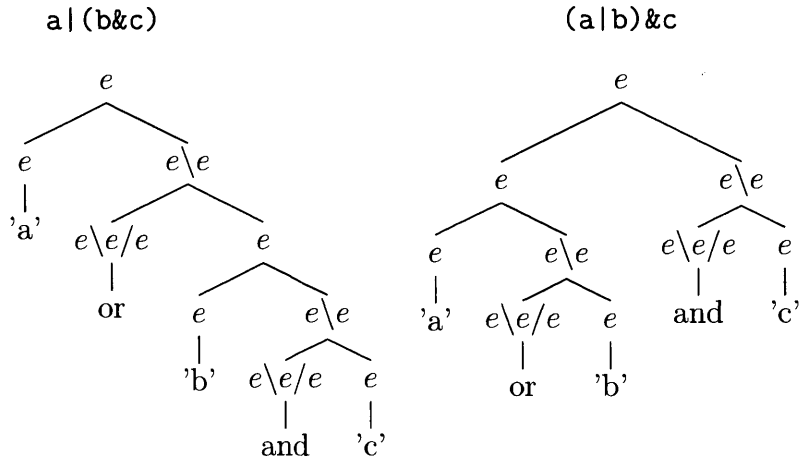


Figure 2-8: This figure shows why it is necessary to lexicalize our parsing model. These two parse trees, for the same sentence, contain exactly the same set of parse productions, so they will have the same probability, yet they generate different regular expressions.

address them.

Efficient Parsing To enable efficient inference (and thus learning) on our model, we will define our features ϕ over individual parse productions. Specifically, our ϕ can be factorized into features on the individual lexical entries, $l \in t_l$, and features on individual productions in the parse tree, $(s_p \rightarrow s_l s_r) \in t_s$. Formally:

$$\phi(t) = \sum_{l \in t_l} \phi(l) + \sum_{s_p \rightarrow s_l \ s_r \in t_s} \phi(s_p \rightarrow s_l \ s_r) \tag{2.2}$$

This factorization allows us to compute t^* using the standard CYK dynamic programming parsing algorithm [68] which runs in $O(n^3)$ time, where n is the length of $\vec{w}$. In contrast, without constraints on the features exact computation of t^* is exponential in n in the worst case. As we will discuss in Section 2.5 this choice of factorization will also have practical implications on the inference processes used during learning as well.

Lexicalized Model The model described above, as defined by 2.2 completely ignores the words themselves when generating $\phi(s_p \rightarrow s_l \ s_r)$. This leads to circumstances where two different parses for the same sentence will have exactly the same

set of features, and thus, by definition, the same probability. Figure 2-8 shows such an example. To avoid this problem, we will *lexicalize* our model, i.e. we will include some lexical information at each non-terminal in addition to just the lambda-calculus types. Specifically, we will follow standard practice and include a single head-word at each non-terminal [32]. Thus our feature function becomes:

$$\phi(t) = \sum_{l \in t_l} \phi(l) + \sum_{\langle c_p, w_p \rangle \to \langle c_l, w_l \rangle \langle c_r, w_r \rangle \in t_c} \phi(\langle c_p, w_p \rangle \to \langle c_l, w_l \rangle \langle c_r, w_r \rangle)$$

Note that this modification still allows us to use the CYK dynamic programming algorithm to calculate t^*, however the runtime increases from $O(n^3)$ to $O(n^5)$.

2.4 Integrating Semantic Equivalence into the Model

In Section §2.3.3 we discussed the standard maximum marginal log-likelihood objective used for training CCG log-linear models. This objective was:

$$O = \sum_i \log \sum_{t | \texttt{EVAL}(t) = r_i} p(t | \vec{w}_i; \theta, \Lambda) - \frac{\lambda}{2} \sum_k \theta_k^2 \tag{2.3}$$

where $\texttt{EVAL}(t)$ extracts the regular expression from the root of parse tree t.

Such an objective is problematic, however, because, as we saw in Figure 2-1, a given training sample, $\langle \vec{w}_i, r_i \rangle$, may not admit a fragment-by-fragment mapping between the regular expression r_i and the natural language, $\vec{w}_i$. The main departure of our work is to use the semantic inference capabilities of computer programs to get around this problem by finding another semantically equivalent regular expression which does admit a fragment-by-fragment mapping. Specifically, we will modify the training objective to maximize the probability of all parse trees t such that $r = \texttt{EVAL}(t)$ is *semantically equal* to r_i, instead of just those where r is exactly *syntactically* equal to r_i.

To enable such a change, we will define a procedure, $\texttt{SEM-EQ}(r_1, r_2)$ which returns true if r_1 is semantically equal to r_2, i.e. if regular expression r_1 accepts exactly the

same set of inputs as regular expression r_2. Our modified object then becomes:

$$O = \sum_i \log \sum_{t|\texttt{SEM-EQ(EVAL(t)},r_i)} p(t|\vec{w}_i; \theta, \Lambda) - \frac{\lambda}{2} \sum_k \theta_k^2 \tag{2.4}$$

In the following we discuss how we can efficiently implement `SEM-EQ`.

2.4.1 Regexp Equivalence Using Deterministic Finite Automata

The core of our model involves computing the semantic equivalence between two regular expressions. This can be done efficiently by converting the regular expressions to Deterministic Finite Automata (DFAs). Specifically, regular expressions at their core are simply a convenient format for describing a regular language, and classic work shows that they can be directly written down as non-deterministic finite automata (NFAs)[57]. Furthermore, NFAs can be converted to DFAs, and any DFA can be deterministically compiled down to a minimal DFA[57]. Finally, minimal DFAs for the same regular language are guaranteed to be equal up to a relabeling of the nodes[57]. In theory, the DFA representation of a regular expression may be exponentially larger than the the original NFA. However, past work has shown that most regular expressions do not exhibit this exponential behavior [126, 97], and the conversion process is renowned for its good performance in practice [97].

Hence, we implement the function `SEM-EQ`(r_1, r_2), defined above, by converting r_1 and r_2 to minimal DFAs and then comparing these for equality. In practice, we do this using a modified version of Møller (2010). To catch any cases where the resulting DFA might be prohibitively large, we set a timeout on this process. In our experiments we use a one second timeout, which results in timeouts on less than 0.25% of the regular expressions. Note that we can tune this timeout parameter to reduce the computational overhead at the cost of less effective learning. At the extreme, if we set the timeout to zero, then we are simply falling back to using the exact string equivalence technique employed by the past work, as defined in equation (2.5).

2.5 Learning

A probabilistic CCG is defined by a lexicon, Λ, and a parameter vector, θ. In this section we describe our algorithm to learn both of these from a training dataset, S, where each sample $s_i = \langle \vec{w}_i, r_i \rangle$ pairs a natural language sentence $\vec{w}_i$ with a regular expression, r_i.

Our algorithm uses separate techniques to learn Λ and θ. At a high level it works as follows:

- **Learning the Lexicon:** Our technique generates a noisy lexicon by considering possible fragment-by-fragment mappings between the natural language and the regular expression in each of the training samples. This noisy lexicon *will* include many incorrect entries for each word or phrase, but the goal is to ensure that it also contains the correct entry(s) for each word or phrase. We discuss this process in Section §2.5.2.

- **Learning Theta:** Since this noisy lexicon will allow many incorrect parses for each sentence, we use a stochastic gradient descent algorithm to learn values for θ such that the correct parses have high probability. We discuss this process in Section 2.5.1.

Our algorithm performs these two learning processes simultaneously using an iterative algorithm. The details of the algorithm can be seen in Algorithm 1. While our approach is similar in nature to the past work on CCG based semantic parsing [143, 73], we depart from this work in two important ways in order to effectively learn to generate computer programs:

- We utilize the ability to compute the semantic equality between computer programs, in order to effectively learn from training samples which do not admit a fragment-by-fragment mapping, such as that in Figure 2-1. Specifically, we integrate into our gradient descent updates, the DFA-based semantic equality computation described in §2.4.1.

Inputs: Training set of sentence regular expression pairs. $\{\langle \vec{w}_i, r_i \rangle \,|i = 1 \ldots n\}$

Functions:

- N-BEST$(\vec{w}; \theta, \Lambda)$ n-best parse trees for $\vec{w}$ using the algorithm from §2.5.1
- SEM-EQ(t, r) calculates the equality of the regexp from parse t and regexp r using the algorithm from §2.4.1
- SPLIT-LEX(T) splits all lexical entries used by any parse tree in set T, using the process described in §2.5.2 and outlined in Algorithm 2.

Initialization: $\Lambda = \{\langle \vec{w}_i, e : r_i \rangle \,|i = 1 \ldots n\}$

For $k = 1 \ldots K, i = 1 \ldots n$

Update Lexicon: Λ

- $T =$ N-BEST$(\vec{w}_i; \theta, \Lambda)$
- $C = \{t | t \in T \wedge$ SEM-EQ$(t, r_i)\}$
- $\Lambda = \Lambda \cup$ SPLIT-LEX(C)

Update Parameters: θ

- $T =$ N-BEST$(\vec{w}_i; \theta, \Lambda)$
- $C = \{t | t \in T \wedge$ SEM-EQ$(t, r_i)\}$
- $\Delta = E_{p(t|t \in C)}[\phi(t, \vec{w})] - E_{p(t|t \in T)}[\phi(t, \vec{w})]$
- $\theta = \theta + \alpha\Delta$

Output: The lexicon and the parameters, $\langle \Lambda, \theta \rangle$

Algorithm 1: The full learning algorithm.

- We utilize an n-best parsing algorithm based on the work of Huang and Chiang (2005) in order to handle the large number of potential parses that exist in our domain due to the weak typing and complex lexical entries. In Section §2.8.2 we show that in our domain this algorithm much more effectively represents the top parses than the pruned chart parsing algorithms used by the past work [143, 73, 83].

In the remainder of this section we discuss first the process for learning θ and then the process for generating the lexicon, Λ.

2.5.1 Estimating Theta

Recall from Section §2.4 that when estimating θ, our objective is to maximize the marginal log-likelihood of the training data. Formally,

$$O = \sum_i \log \sum_{t|\texttt{SEM-EQ}(\texttt{EVAL}(t),r_i)} p(t|\vec{w}_i;\theta,\Lambda) - \frac{\lambda}{2}\sum_{k<|\theta|}\theta_k^2 \tag{2.5}$$

We maximize this objective using stochastic gradient descent. Differentiating the objective to get the gradient of parameter θ_j for training example i, results in:

$$\frac{\partial O_i}{\partial \theta_j} = \mathbb{E}_{p(t|\texttt{SEM-EQ}(\texttt{EVAL}(t),r_i);\theta,\Lambda)}\left[\phi_j(t,\vec{w}_i)\right] - \mathbb{E}_{p(t;\theta;\Lambda)}\left[\phi_j(t,\vec{w}_i)\right] - \lambda\theta_j \tag{2.6}$$

This gives us the standard log-linear gradient, where the first term is the expected feature counts in the correct parse trees for $\vec{w}_i$, the second term is the expected feature counts in all valid parse trees for $\vec{w}_i$, and the third term is the regularization. As discussed in Section §2.3.3, we define the features in our model over individual parse productions, admitting the use of the inside-outside algorithm [68] to efficiently calculate the unconditioned expected counts in the second term. However, the first term does not factorize, since it is conditioned on generating a regular expression, r, which is semantically equivalent to the regular expression in the training data, such that $\texttt{SEM-EQ}(r, r_i)$. In fact, exact techniques for computing the first term are computationally intractable, since r may be syntactically very different from r_i, as we saw in Figure 2-1.

Thus we turn to approximate gradient calculations. We approximate the gradient by computing the marginal feature counts over the n-best full parse trees. This is in contrast to the past work which uses a beam-search based version of the inside-outside algorithm to approximate the marginal feature counts over each subtree. We discuss in more detail below how we can compute the n-best parses efficiently, as well how our algorithm contrasts with that of the past work. Formally, we utilize the n-best

parses to approximate the gradient as follows:

$$\frac{\partial O_i}{\partial \theta_j} = \mathbb{E}_{p(t|t\in T_i,\texttt{SEM-EQ}(\texttt{EVAL}(t),r_i);\theta,\Lambda)}[\phi(t,\vec{w}_i)] - \mathbb{E}_{p(t|t\in T_i;\theta,\Lambda)}[\phi(t,\vec{w}_i)] - \lambda\theta_j \qquad (2.7)$$

where T_i is the set of n-best parses for $\vec{w}_i$. This calculation simply approximates both terms of the gradient from equation (2.6) above by considering only the n-best valid parses rather than all valid parses.

Efficiently Calculating the n-Best Parses

The main advantage of our n-best based approximation comes from the fact that we can very efficiently compute exactly the n-best parse trees. To do this we use an algorithm originally developed by Jimenez and Marzal (2000), and subsequently improved by Huang and Chiang (2005). This algorithm utilizes the fact that the first best parse, t_1, makes the optimal choice at each decision point, and the 2^{nd} best parse, t_2 must make the same optimal choice at every decision point, *except for one.* To execute on this intuition, the algorithm first calculates t_1 by generating an unpruned CYK-style parse forest which includes a priority queue of possible subparses for each constituent. The set of possible 2^{nd} best parses T are those that choose the 2^{nd} best subparse for exactly one constituent of t_1 but are otherwise identical to t_1. The algorithm chooses $t_2 = \arg\max_{t\in T} p(t)$. More generally, T is maintained as a priority queue of possible n^{th} best parses. At each iteration, i, the algorithm sets $t_i = \arg\max_{t\in T} p(t)$ and augments T by all parses, $t_i^{(j)}$ which both differ from t_i at exactly one constituent c_j and choose the next best possible subparse for c_j.

Contrasting our Algorithm to the Past Work

The past work has approximated the gradient through a dynamic programming algorithm similar to the inside-outside algorithm. To understand how this differs from our algorithm consider that dynamic programming algorithms must define the space of subproblems that they are going to solve. CCG parsing works similar to CFG parsing in that these subproblems involve finding the highest scoring subparse over

some span of the sentence. These are typically stored in a matrix-like chart which has a cell for each possible span from word i to word j. Our work differs from the past work in the set of entries we store in each chart cell.

Entries in our chart are indexed by a tuple $\langle i, j, k, c \rangle$, where i and j define the span, k defines the head word, as discussed in Section §2.3.3, and c is the CCG type at the root of the subparse. Note that c contains only the type information, and not the entire lambda-calculus expression, thus each entry in the chart represents many possible lambda-calculus expressions. In theory, the space of possible CCG types can be exponential in the length of the sentence. In practice, as discussed in Section §2.6.4, we limit the number of arguments in a given lambda-expression, such that the resulting CCG grammar has at most 16 different non-terminal types. This results in a chart which has $O(|\vec{w}|^3)$ entries. The main downside of such a chart configuration is that we cannot use it to directly compute the first expectation in equation(2.6). This is because the expectation is conditioned on the regular expression at the root of the parse tree, yet the entries in our chart are not indexed by this value. Instead, use a full unpruned chart of this form to calculate the n-best parses via the algorithm described above, and approximate the expectation using these parse trees.

The past work address the problem differently, and instead used an augmented chart whose entries are indexed by a tuple $\langle i, j, k, c, e \rangle$, where i,j, and c are as above, and e is the full lambda-calculus expression at each non-terminal. This allows the conditioned expectation from equation(2.6) to be calculated directly using the inside-outside algorithm. The problem, however, is that there are an exponential number of possible regular expressions for each sentence span, and thus and exponential number of possible chart entries in each cell. To combat this explosion, the past work resorts to a beam search, and at each cell maintains only the m highest scoring entries. The resulting chart has at most $O(|\vec{w}|^2 m)$ entries.

Qualitatively, we can compare these two approximations based on the set of parse trees which they represent. Our chart represents every valid parse tree for the given sentence, but we calculate the expectation based on only the top n full parse trees. These parse trees, however, are always the n most likely full parse trees. In contrast,

the past work uses the inside-outside algorithm to calculate the expectation based on the entire parse forest represented in its chart, which can contain an exponential number of parse trees. However, since the beam search prunes the chart myopically at each chart cell, it often prunes the highest probability parse trees out of the chart. In fact, as shown in Section §2.8.2, we find that the single most likely parse tree is pruned out almost 20% of the time. Furthermore, our results show that this inability to represent the likely parses significantly impacts the overall prediction performance. It is also important to note that the runtime of our n-best algorithm scales much better. Specifically, our algorithm calculates the full chart in $O(|\vec{w}|^5)$ time, and then uses the chart to compute the n-best parses in $O(|\vec{w}|n \log n)$ time for an overall runtime of $O(|\vec{w}|^5 + |\vec{w}|n \log n)$. In contrast, the algorithm used by the past work runs in $O(|\vec{w}|^3 m^2)$. In practice, we find that the runtime of our algorithm is dominated by the $|\vec{w}|^5$ calculation up front, and changing n from 1 to 10,000 increases the runtime by less than a factor of 2. In contrast, an increase in m has a direct quadratic effect on the runtime of the beam search algorithm. In our experiments, we found that even with n set to 10,000 and m set to 200, our algorithm still ran almost 20 times faster.

2.5.2 Learning the Lexicon

This section describes our technique for learning the lexicon, Λ. The lexicon defines the space of possible CCG parse trees for a given sentence. It consists of a set of lexical entries, $\langle l_{\vec{w}}, l_t, l_r \rangle$, where $l_{\vec{w}}$ is the natural language word or phrase, l_t is the syntactic type, and l_r is the lambda-calculus expression. We learn the lexicon from a set of training samples which pair a sentence, $\vec{w}_i$, with a full regular expression, r_i. The goal of our learning algorithm is two-fold. We would like to generate a set of lexical entries such that:

1. For each training sample i there exists a parse tree, t, which will generate the correct regular expression, i.e. $\texttt{SEM-EQ}(\texttt{EVAL}(t), r_i)$

2. The generated lexical entries will generalize well to new unseen data.

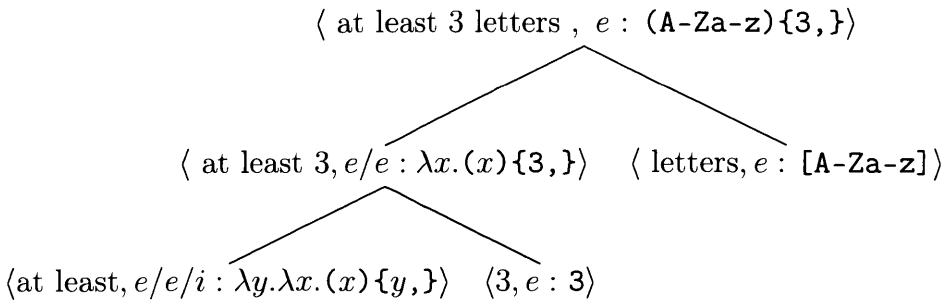


Figure 2-9: An example parse tree with lexical entries at its leaves.

Consider, for example, the parse in Figure 2-9. When given a training example containing the sentence "at least 3 letters", and the regular expression `[A-Za-z]{3,}` we would like our algorithm to generate the lexical entries:

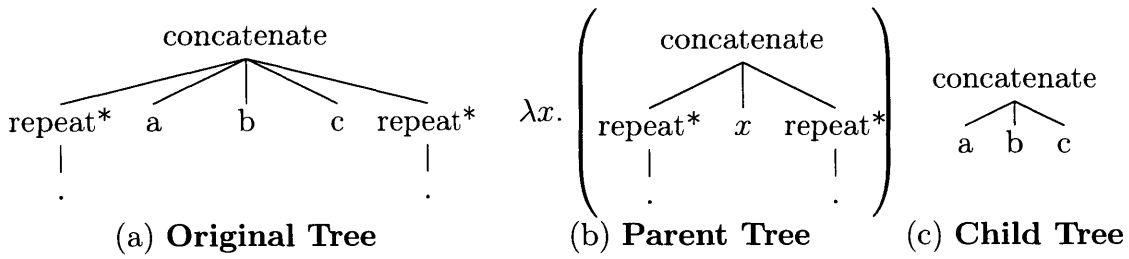
$$\langle \text{ at least, } e/i/e : \lambda x \lambda y.((x)\{y,\}) \rangle$$
$$\langle 3,\ i : 3 \rangle$$
$$\langle \text{ letters, } e : \texttt{[A-Za-z]} \rangle$$

In the process of generating the above lexical entries, our algorithm will also generate many other incorrect lexical entries such as: $\langle$ at least, $i : 3\rangle$. Thus the resulting lexicon will generate valid parse trees which evaluate to many different regular expressions, and we rely on our probabalistic model to choose the correct one at test time.

Our lexicon learning algorithm has the following three components:

- **Initialization:** We initialize the lexicon by generating a single lexical entry for each training sample which pairs the full sentence, $\vec{w}_i$, with the associated regular expression, r_i. Formally, the lexicon is initialized as: $\Lambda = \{\langle \vec{w}_i, e : r_i \rangle | i = 1 \ldots n\}$

- **Candidates for Splitting:** The initial Λ will perfectly parse the training data, however it will not generalize at all to sentences which are not exactly observed in the training data. Hence, in each iteration we refine the lexicon by splitting existing lexical entries to generate more granular lexical entries which will generalize better. Formally, the set of lexical entries to be further split is defined as: $S_i = \{t_l | t \in \texttt{N-BEST}(\vec{w}_i) \wedge \texttt{SEM-EQ}(\texttt{EVAL}(t), r_i\}$, where `N-BEST`



(a) **Original Tree** (b) **Parent Tree** (c) **Child Tree**

Figure 2-10: The tree in (a) represents the lambda expression from the lexical entry ⟨with abc, e:.*abc.*⟩. One possible split of this lexical entry generates the parent lexical entry ⟨with, $e/e : \lambda x.(.*x.*)$⟩ and the child lexical entry, ⟨abc, R :abc⟩, whose lambda expressions are represented by (b) and (c), respectively.

is the n-best algorithm described in the previous section, and SEM-EQ is the DFA-based semantic equality calculation discussed in Section §2.4.1.

- **Splitting Algorithm:** For each lexical entry to be split, we add a set of new lexical entries which contains the cross product of all binary splits of the natural language and all binary splits of the lambda-calculus expression. The details of this splitting process can be seen in Algorithm 2 and are discussed in detail in the following section.

Splitting Lexical Entries

Each lexical entry consists of a sequence of n words aligned to a typed regular expression function, $\langle w_{0:n}, T : r\rangle$. Our splitting algorithm generates new lexical entries for each possible way to split a lexical entry into two, such that they can be recombined to obtain the original lexical entry by using one of the two combinators discussed in Section §2.3.2.[6] This set is generated by taking the cross product of all binary splits of the natural language and all binary splits of the typed regular expression function. The set of binary splits of the natural language is simply $\{\langle w_{0:j}, w_{j:n}\rangle | j = 1 \ldots n-1\}$, and the set of binary splits of the lambda-calculus expression, $S(r)$ is defined in the following section. Each split $s \in S(r)$ with parent s_p and child s_c generates two pairs of lexical entries, one for forward application, and one for backward application.

[6]This process is analogous to the syntactic unification process done by Kwiatkowski et al. (2010).

Function `gen_var()`
| **return** a new unused variable name
Function `gen_entries`$(l_{\vec{w}}, s_p, s_c)$
| **return** $\{ \langle w_{0:j}, T/T_c : s_p \rangle \mid 0 \leq j \leq n-1 \} \cup$
$\{ \langle w_{j:n}, T_c : s_c \rangle \mid 0 \leq j \leq n-1 \} \cup$
$\{ \langle w_{0:j}, T_c : s_c \rangle \mid 0 \leq j \leq n-1 \} \cup$
$\{ \langle w_{j:n}, T \backslash T_c : s_p \rangle \mid 0 \leq j \leq n-1 \}$
Input: set of parse trees, T
Initialization:
$S = \{\}$
foreach $t \in T$ **do**
| **foreach** lexical entry $l \in t_l$ **do**
| | **foreach** non-terminal node n in the AST of l_r **do**
| | | $C(n)$ = children of n
| | | $c = |C(n)|$
| | | **if** $n =$ *concatenate* **then**
| | | | **foreach** $i \in 1 \ldots c$ **do**
| | | | | **foreach** $j \in i+1 \ldots c$ **do**
| | | | | | $x =$ `gen_var()`
| | | | | | $s_p = \lambda x.t$ with $C(n)[i \ldots j]$ replaced with x
| | | | | | $s_c =$ subtree rooted at n with $C(n) = C(n)[i \ldots j]$
| | | | | | $S = S \cup$ `gen_entries`$(l_{\vec{w}}, s_p, s_c)$
| | | | | **end**
| | | | **end**
| | | **else if** $n \in \{$*and*,*or*$\}$ **then**
| | | | $\mathbb{V}$ = all subsets of $C(n)$
| | | | **foreach** $V \in \mathbb{V}$ **do**
| | | | | $x =$ `gen_var()`
| | | | | $s_p = \lambda x.t$ with all nodes in V replaced with a single x
| | | | | $s_c =$ subtree rooted at n with $C(n) = V$
| | | | | $S = S \cup$ `gen_entries`$(l_{\vec{w}}, s_p, s_c)$
| | | | **end**
| | | **else**
| | | | $x =$ `gen_var()`
| | | | $s_p = \lambda x.t$ with n replaced with x
| | | | $s_c =$ subtree rooted at n
| | | | $S = S \cup$ `gen_entries`$(l_{\vec{w}}, s_p, s_c)$
| | | **end**
| | **end**
| **end**
end
Output: Set of splits, S

Algorithm 2: The algorithm for splitting existing lexical entries to generate new more fine grain lexical entries. This is the implementation of `SPLIT-LEX` from Algorithm 1.

Formally, the full set of new lexical entries generated is:

$$\begin{aligned}
&\{\langle w_{0:j}, T/T_c : s_p\rangle \,|(0 \le j \le n-1) \wedge (s \in S(r))\} \quad \cup \\
&\{\langle w_{j:n}, T_c : s_c\rangle)|(0 \le j \le n-1) \wedge (s \in S(r))\} \quad \cup \\
&\{\langle w_{0:j}, T_c : s_c\rangle\,, |(0 \le j \le n-1) \wedge (s \in S(r))\} \quad \cup \\
&\{\langle w_{j:n}, T\backslash T_c : s_p\rangle \,|(0 \le j \le n-1) \wedge (s \in S(r))\}
\end{aligned}$$

where T is the type of the original regular expressions, r, and T_c is the type of the newly generated child, s_c.

Splitting Regular Expression Lambda Functions We will explain the process for splitting a lambda-calculus expression through a modified version of the AST representation introduced in Section §2.3.1. In this description, the body of a lambda-calculus expression will be represented by an AST which has a special node type for lambda-calculus variables. Lambda function application becomes a tree substitution operation which replaces the variable node with the AST of the argument. If r is an AST lambda function, then each split, s, generates a child expression s_c and a parent expression s_p such that $r = s_p \oplus s_c$, where $\oplus$ represents lambda function application. We consider two types of splits:

- **Basic:** For each node, n, in r besides the root node, we generate a split where s_c is the subtree rooted at node n. For such splits, s_p is the lambda expression r with n replaced with a bound variable, say x.

- **Complex:** We also consider a set of more complicated splits at each node whose associated function type constant can take any number of arguments, i.e. `or`, `and`, or `cons`. If $C(n)$ are the children of node n, then we generate a split for each possible subset, $\{V|V \subset C(n)\}$. Note that since `concatenate` is order dependant, V must be contiguous for `concatenate` nodes. For each choice of V, the child tree, s_c, is a version of the tree rooted at node n pruned to contain only the children in V. Additionally, the parent tree, s_p, is generated from r by replacing all the children in V with a single bound variable, say x. Figure 2-10

shows an example of such a split. In §2.6.4 we discuss our technique to avoid generating an exponential number of splits.

In either case, we only consider splits in which s_c does not have any bound variables, so its type, T_c, is always one of the basic types (e or i). The type of s_p is then type of the original expression, T augmented by an additional argument for the child type, i.e. either T/T_c or $T\backslash T_c$.

Adding New Lexical Entries

Our model splits all lexical entries used in parses which generate correct regular expressions, i.e. in Algorithm 1 all those in C, and adds *all* of the generated lexical entries to Λ. In contrast, the previous work [143, 73] has a very conservative process for adding new lexical entries. This process relies on a good initialization of the feature weights associated with a new lexical entry. They perform this initialization using a IBM Model 1 [21] alignment of the words in the training sentences with the names of functions in the associated lambda calculus expression. Such an initialization is ineffective in our domain since it has very few primitive functions and most of the training examples use more than half of these functions. Instead, we add new lexical entries more aggressively, and rely on the n-best parser to effectively ignore any lexicon entries which do not generate high probability parses.

2.6 Applying the Model

2.6.1 Features

Recall from Section §2.3.3, that our feature function ϕ factorizes into features on individual lexical items and features over individual parse productions, as follows:

$$\phi(t) = \sum_{l \in t_l} \phi(l) + \sum_{\langle c_p, w_p \rangle \rightarrow \langle c_l, w_l \rangle \langle c_r, w_r \rangle \in t_c} \phi(\langle c_p, w_p \rangle \rightarrow \langle c_l, w_l \rangle \langle c_r, w_r \rangle)$$

For each lexical entry, $l = \langle l_{\vec{w}}, l_t, l_r \rangle$, we generate 4 types of features: (1) a feature for $\langle l_{\vec{w}}, l_t, l_r \rangle$, (2) a feature for $l_{\vec{w}}$, (3) a feature for $\langle l_t, l_r \rangle$, and (4) a set of features indicating whether $l_{\vec{w}}$ contains a string literal and whether the leaves of l_r contain any exact character matches (rather than character range matches). For each parse production, $\langle c_p, w_p \rangle \rightarrow \langle c_l, w_l \rangle \langle c_r, w_r \rangle$, we have features that combine all subsets of the head word and CCG type, of the two children and the newly generated parent, i.e. c_p,w_p,c_l,w_l, c_r and w_r.

2.6.2 Initialization

In addition to the sentence level initialization discussed in §3.3 we also initialize the lexicon, Λ, with three other sets of lexical entries.

- **Skip Word Entries:** For every individual word in our training set vocabulary, we add an identity lexical entry whose lambda expression is just a function which takes one argument and returns that argument. This allows our parser to skip semantically unimportant words in the natural language description, and ensures that it generates at least one parse for every example in the dataset. Including explicit entries in the lexicon for this skipping process allows us to learning appropriate weights on these at training time. In contrast, the past work has used a fixed, manually selected weight for word skipping [143, 73].

- **Quoted String Entries:** We add a lexical entry for each quoted string literal in the natural language entries in the training set. Thus for the phrase, "lines with 'abc' twice" we would add the lexical entry ⟨'abc', e : `abc` ⟩.

- **Number Entries:** We also add lexical entries for both numeric and word representations of numbers, such as ⟨1, e : `1`⟩ and ⟨one, e : `1`⟩.

We add these last two types of lexical entries because it does not make sense to try to learn either of them from the data. There is an unbounded number of possible quoted strings, and most of them appear only once in our dataset. Furthermore,

generating the correct lexical entries is trivial since quoted strings have a direct one-to-one correspondence with the appropriate regular expression. Learning the connection between logical numbers and their natural language representations is a well-studied problem which requires considering sub-word units, and is thus outside of the scope of this thesis.

At test time we also add both skip word lexical entries for every word in the test set vocabulary as well as lexical entries for every quoted string literal seen in the test queries. Note that the addition of these lexical entries requires only access to the test queries and does not make use of the regular expressions (i.e. labels) in the test data in any way.

2.6.3 Parameters

We initialize the weight of all lexical entry features except the identity features to a default value of 1 and initialize all other features to a default weight of 0. We regularize our log-linear model using a λ value of 0.001. We use a learning rate of $\alpha = 1.0$, set $n = 10,000$ in our n-best parser, and run each experiment with 5 random restarts and $K = 50$ iterations. We report results using the pocket algorithm technique originated by Gallant (1990). Specifically, after each training iteration we compute the fraction of the training samples which are correctly parsed using the current weights. At testing time we use the feature weights which generated the highest such score on the training data.

2.6.4 Constraints on Lexical Entry Splitting

To prevent the generation of an exponential number of splits, we constrain the lexical entry splitting process as follows:

- We only consider splits at nodes which are at most a depth of 2 from the root of the original tree.
- We limit lambda expressions to 2 arguments.

- We limit the resulting children in unordered node splits (`and` and `or`) to contain at most 4 of the arguments.

These restrictions ensure the number of splits is at most a degree-4 polynomial of the regexp size.[7]

2.6.5 Optimizing Runtime

In practice we find the n-best parser described in §2.5.1 is fast enough that the training runtime is dominated by the computation of the DFAs for the regexp of each of the n-best parse trees. To improve the runtime performance we find the m most likely regexps by summing over the n-best parse trees[8]. We calculate DFAs only for these regexps, and include only parses which evaluate to one of these regexps in the approximate gradient calculation from equation (2.7). This optimization is not strictly necessary, because even without it our parser runs much faster than the beam search parser. However we found that performing the optimization had no impact on the resulting accuracy.

2.7 Experimental Setup

2.7.1 Dataset

Our dataset consists of 824 natural language and regular expression pairs gathered using Amazon Mechanical Turk [131] and oDesk [103].[9] On Mechanical Turk we asked workers to generate their own original natural language queries to capture a subset of the lines in a file (similar to UNIX `grep`). In order to compare to example based techniques we also ask the Mechanical Turk workers to generate 5 positive and 5 negative examples for each query. On oDesk we hired a set of programmers

[7]The unification process used by Kwiatowski et al. (2010) bounded the number of splits similarly.

[8]We set m to 200 in our experiments.

[9]This is similar to the size of the datasets used by past work at the time this work was originally published.

to generate regular expressions for each of these natural language queries. For our experiments, we split the dataset into 3 sets of 275 queries each and tested using 3-fold cross validation. We tuned our parameters separately on each development set but ended up with the same values in each case.

2.7.2 Evaluation Metrics

We evaluate by comparing the generated regular expression for each sentence with the correct regular expression using our DFA equivalence technique, `SEM-EQ`. As discussed in §2.4.1 this metric is exact, indicating whether the generated regular expression is semantically equivalent to the correct regular expression. Additionally, as discussed in §2.6.2, our identity lexical entries ensure we generate a valid parse for every sentence, so we report only accuracy instead of precision and recall.

2.7.3 Baselines

We compare our work directly to **UBL**, the state-of-the-art semantic parser from Kwiatkowski et al. (2010). In consultation with the authors, we modified their publicly available code to handle the lambda-calculus format of our regular expressions.

In order to better understand the gains provided by our model, we also compare to versions of our model modified along three different axes: the parsing algorithm, the lexicon induction algorithm and the equality algorithm. We considered two different parsing algorithms:

- **NBestParse:** This is the parsing algorithm used in our full model as described in Section §2.5.1.

- **BeamParse:** This modification to our model replaces the `N-BEST` procedure from Algorithm 1 with the beam search algorithm used for parsing by past semantic parsing algorithms [143, 73, 83]. Section §2.5.1 contrasts this algorithm with our n-best parsing algorithm. We set the beam size to 200, which is equivalent to the past work [143, 73, 83]. With this setting, the slow runtime of this algorithm allowed us to run only two random restarts.

We consider two different lexical induction algorithms:

- **SplitLexAll:** This is the lexical induction algorithm used by our full model, as described in Section§2.5.2, which generates new lexical entries by splitting the lexical entries used in *all* of the correct parses.

- **SplitLexTop:** This is a modification to our model which passes only the highest probability parse in C to `SPLIT-LEX` in Algorithm 1, instead of all parses in C. This is more similar to the conservative splitting algorithm used by the past work [143, 73, 72].

We consider three different equality algorithms:

- **SemanticEq:** This is the equivalence algorithm used in our full model, as described in Section §2.4.1.

- **StringEq:** This is a modification to our model which replaces the `SEM-EQ` procedure from Algorithm 1 with exact regular expression string equality as described in Section §2.2.1.

- **ExampleEq:** This is a modification to our model which replaces `SEM-EQ` with a procedure that evaluates the regexp on all the positive and negative examples associated with the given query and returns true if all 10 are correctly classified. This represents the performance of the example based equivalence techniques used by the past work, as described in Section §2.2.1.

- **HeuristicEq:** This is a modification to our model which replaces `SEM-EQ` with a smart heuristic form of semantic equality. Our heuristic procedure first flattens the regexp trees by merging all children into the parent node if they are both of the same type and of type `or`, `and`, or `concatenate`. It then sorts all children of the `and` and `or` operators. Finally, it converts both regexps back to a flat strings and compares these strings for equivalence. This process should be much more effective than the *local transformation equivalence* used by the past work as described in Section §2.2.1.

Model	Percent Correct
UBL	36.5%
Our Full Model	**65.5%**

Table 2.1: Accuracy of our model compared to the state-of-the-art semantic parsing model from Kwiatkowski et al. (2010).

Parsing Algorithm	Equality Algorithm	Lexical Induction	Accuracy
BeamParse	HeuristicEq	SplitLexAll	9.4%
BeamParse	HeuristicEq	SplitLexTop	22.1%
NBestParse	StringEq	SplitLexAll	31.1%
NBestParse	ExampleEq	SplitLexAll	52.3%
NBestParse	HeuristicEq	SplitLexAll	56.8%
NBestParse	SemanticEq	SplitLexAll	**65.5%**

Table 2.2: Accuracy of our model as we change the parsing algorithm, the equality algorithm, and the lexical induction algorithm.

2.8 Results

We can see from Table 2.1 that our model outperforms the state-of-art baseline algorithm from Kwiatkowski et al. (2010). In fact we predict the correct regular expression almost twice as often as the baseline.

The results in Table 2.2 show that all aspects of our model are important to its performance. In particular, we can see that the use of semantic equality is critical to our technique. Our full model outperforms exact string equality (*StringEq*) by over 30%, example based equality (*ExampleEq*) by over 13% and our smart heuristic equality procedure (*HeuristicEq*) by 9%. These improvements confirm that calculating exact semantic equality during the learning process helps to disambiguate language meanings. Additionally, we can see that using the beam search parsing algorithm of the past work (*BeamParse*) significantly degrades performance, even if we change the lexical induction algorithm to split only the lexical entries used in the most probable parse (*SplitLexTop*).

Percentage of Data	15%	30%	50%	75%
NBestParse-HeuristicUnify	12.4%	26.4%	39.0%	45.4%
Our Model	29.0%	50.3%	58.7%	65.2%
Relative Gain	**2.34x**	**1.91x**	**1.51x**	**1.43x**

Table 2.3: Accuracy for varying amounts of training data. The relative gain line shows the accuracy of our model divided by the accuracy of the baseline.

2.8.1 Effect of Additional Training Data

Table 2.3 shows the change in performance as we increase the amount of training data. We see that our model provides particularly large gains when there is a small amount of training data. These gains decrease as the amount of training data increases because the additional data allows the baseline to learn new lexical entries for every special case. This reduces the need for the fine-grained lexicon decomposition which is enabled by our semantic equality algorithm. For example, our model will learn separate lexical entries for "line", "word", "starting with", and "ending with". The baseline instead will just learn separate lexical entries for every possible combination such as "line starting with", "word ending with", etc. Our model's ability to decompose, however, allows it to provide equivalent accuracy to even the best baseline with less than half the amount of training data. Furthermore, we would expect this gain to be even larger for domains with more complex mappings and a larger number of different combinations.

2.8.2 Beam Search vs. n-Best

A critical step in the training process is calculating the expected feature counts over all parses that generate the correct regular expression. In §2.5.1 we discussed the trade-off between approximating this calculation using the n-best parses, as our model does, verses the beam search model used by the past work. The effect of this trade-off can be seen clearly in Figure 2-11. The n-best parser always represents the n-best parses, which is set to 10,000 in our experiments. In contrast, the beam search algorithm fails to represent the top parse almost 20% of the time and represents less than 15% of the

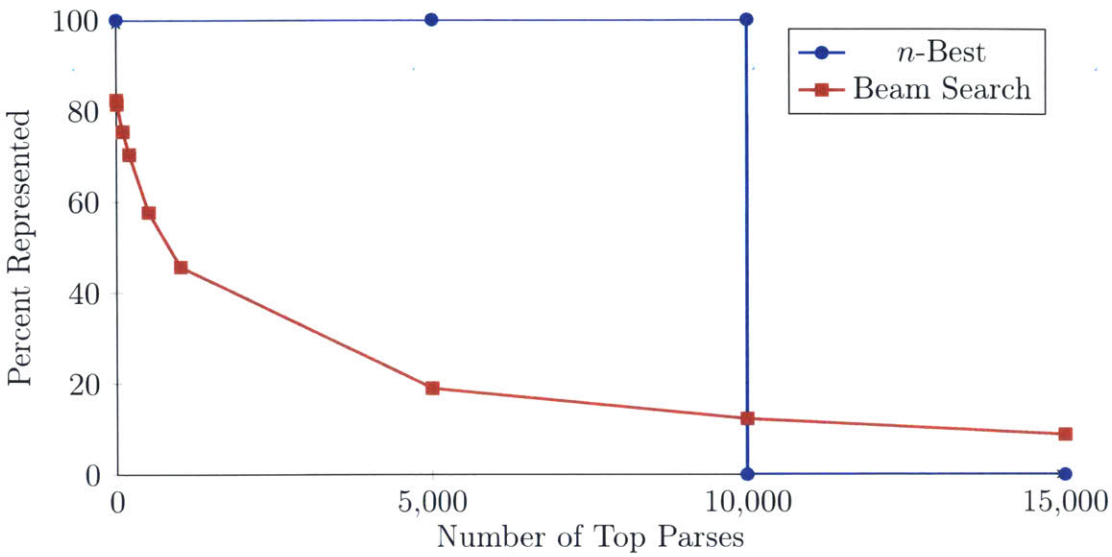


Figure 2-11: This graph compares the percentage of the top-n parses which are represented by the n-best approximation used in our model (*n-Best*) to the set of parses represented by the beam search approximation used by the past work (*Beam Search*). The n-Best algorithm does not represent any parses beyond the top 10,000, but it represents all of these parses. In contrast, the *Beam Search* algorithm is able to represent parses past the top 10,000 as the cost of missing 85% of the parses in the top 10,000.

10,000 most likely parses. This difference in representation ability is what provides the more than 30% difference in accuracy between the *BeamParse-HeuristicEq* version of our model and the *NBestParse-HeuristicEq* version of our model as shown in Table 2.2.

2.9 Conclusions and Future Work

In this chapter, we presented a technique for learning to translate single sentence descriptions of computer tasks into computer programs which can perform these tasks. We demonstrated our technique in the domain of text processing where the generated computer programs are regular expressions. The key idea behind our approach was to integrate a DFA based form of semantic equality into the learning process to handle training samples whose meaning does not decompose. Our algorithm iteratively learns both the lexicon and parameters of a probabilistic CCG which is used to perform the

translation. Experiments on a dataset of natural language regular expression pairs showed that integrating semantic equality into our model allows it to significantly outperform the prior state-of-the-art techniques.

We performed our work on the domain of regular expressions, for which semantic equality is computationally tractable. In more general domains, semantic equality is undecidable. However, as we discussed in Section §2.4.1 the performance of our technique degrades gracefully as we approximate semantic equality rather than computing it exactly. Thus, we believe our work motivates the use of semantic inference techniques for language grounding in more general domains through the use of some form of approximation or by restricting its use in some way. For example, SAT and SMT solvers have seen significant success in performing semantic inference for program induction and hardware verification despite the computational intractability of these problems in the general case.

Chapter 3

Automatically Solving Math Word Problems

In the previous chapter we tackled the problem of translating single sentence task descriptions into computer programs in the context of regular expressions. Complex tasks clearly cannot be described in a single sentence, however. So in this chapter we tackle the more complex problem of translating multi-sentence task descriptions into programs. We tackle this problem in the context of natural language math problems. Our task is to translate the natural language description of a math problem into a system of equations which can be solved to generate the solution to the problem. As with regular expressions, the system of equations is a constrained type of computer program. The main additional challenge beyond those discussed in Chapter 2 is that the natural language often implies complex cross-sentence relationships which are not stated explicitly in the text. We tackle this challenge with an algorithm which reasons jointly across sentences to construct the system of linear equations, while simultaneously recovering an alignment of the variables and numbers in these equations to the problem text. To handle the large space generated by the joint inference algorithm, we utilized both the semantic equivalence and program execution capabilities of the underlying symbolic math system. We show that using these techniques results in significantly more effective learning.

Word problem
An amusement park sells 2 kinds of tickets. Tickets for children cost \$1.50. Adult tickets cost \$4. On a certain day, 278 people entered the park. On that same day the admission fees collected totaled \$792. How many children were admitted on that day? How many adults were admitted?
Equations
$x + y = 278$ $1.5x + 4y = 792$
Solution
$x = 128 \quad y = 150$

Figure 3-1: An example algebra word problem. Our goal is to map a given problem to a set of equations representing its algebraic meaning, which are then solved to get the problem's answer.

3.1 Introduction

Algebra word problems concisely describe a world state and pose questions about it. The described state can be modeled with a system of equations whose solution specifies the questions' answers. For example, Figure 3-1 shows one such problem. The reader is asked to infer how many children and adults were admitted to an amusement park, based on constraints provided by ticket prices and overall sales. This chapter studies the task of learning to automatically solve such problems given only the natural language.

Challenge Solving these problems requires reasoning across sentence boundaries to find a system of equations that concisely models the described semantic relationships. For example, in Figure 3-1, the total ticket revenue computation in the second equation summarizes facts about ticket prices and total sales described in the second, third, and fifth sentences. Furthermore, the first equation models an implicit semantic relationship, namely that the children and adults admitted are non-intersecting subsets of the set of people who entered the park.

Summary of Approach Our model defines a joint log-linear distribution over full systems of equations and alignments between these equations and the text. The space of possible equations is defined by a set of equation templates, which we induce from the training examples, where each template has a set of slots. Number slots are filled by numbers from the text, and unknown slots are aligned to nouns. For example, the system in Figure 3-1 is generated by filling one such template with four specific numbers (1.5, 4, 278, and 792) and aligning two nouns ("Tickets" in "Tickets for children", and "tickets" in "Adult tickets"). These inferred correspondences are used to define cross-sentence features that provide global cues to the model. For instance, in our running example, the string pairs ("$1.50", "children") and ("$4","adults") both surround the word "cost," suggesting an output equation with a sum of two constant-variable products.

Key Ideas We handle the large search space resulting from our joint model by utilizing the semantic equivalence and program execution capabilities of the underlying symbolic math platform. By using the platform to detect the semantic equality between potential systems of equations, we are able to avoid the duplicated computational cost of considering two different, but semantically equal, systems during the inference process. Furthermore, using the platform to numerically solve possible full systems of equations to generate numeric solutions, enables us to use the solution to help determine whether or not the system of equations correctly represents the natural language question. For example, if our solution indicates a fractional number of tickets, that is probably wrong, but a fractional price is perfectly reasonable. Rather than hand writing rules like these, we include the potential correspondences as additional features in our log-linear model, allowing us to learn them from the training data.

Evaluation We consider learning with two different levels of supervision. In the first scenario, we assume access to each problem's numeric solution (see Figure 3-1) for most of the data, along with a small set of seed examples labeled with full equa-

tions. During learning, a solver evaluates competing hypotheses to drive the learning process. In the second scenario, we are provided with a full system of equations for each problem. In both cases, the available labeled equations (either the seed set, or the full set) are abstracted to provide the model's equation templates, while the slot filling and alignment decisions are latent variables whose settings are estimated by directly optimizing the marginal data log-likelihood.

We evaluate our approach on three different corpora: a newly gather corpus of 514 algebra word problems from Algebra.com, the arithmetic dataset from [58], and a new corpus of finance problems. On the algebra dataset our algorithm generates the correct system of equations for over 66% of the word problems. In contrast, a baseline model which does not utilize the semantic equality and execution capabilities of the underlying platform generates the correct equations for only 51% of the problems. Furthermore, on the arithmetic dataset our approach outperforms the system from [58] which was engineered specifically for that type of data.

3.2 Related Work

Our work is related to three main areas of research: automatic word problem solvers, natural language grounding, and information extraction.

3.2.1 Automatic Word Problem Solvers

The most related work to ours has also focused on solving natural language math problems. Early work in this area utilized hand-engineered rule-based techniques to solve math word problems in specific domains [99, 82]. In contrast, our work is the first to tackle the problem using statistical techniques which learn from only a training corpus of questions paired with equations or answers rather that using hand-engineered rules.

Since we originally published the work described in this chapter, interest in this area has exploded, generating much follow-on research. For example, Hosseini et al. (2014) presents a system for solving natural language addition and subtraction prob-

lems by learning to categorize the central verbs of the problem into one of seven categories such as *giving* or *taking* of items. Seo et al. (2014) grounds natural language phrases from geometry problems to visual components of the associated diagrams, but do not actually attempt to solve the geometry problems. Roy and Roth (2015) generates a single expression tree which spans all of the sentences in the problem. The tree can include all four arithmetic operators, but cannot handle multiple equations as our work can.

3.2.2 Natural Language Grounding

As discussed in Section §2.2.1 there is a large body of work mapping natural language to some form of logical representation. Our work differs from this work in both the complexity of the cross sentence relationships, and our use of the underlying platform.

Cross Sentence Relationships

Most past work on language grounding has considered only single sentences in isolation. Techniques that consider multiple sentences typically do so with either only implicit cross-sentence reasoning, or very simple explicit reasoning.

- **Single Sentences in Isolation** Much of the past work on language grounding has focused on grounding questions to knowledge-base queries, and thus is inherently single-sentence in nature [142, 138, 69, 108, 130, 70, 45, 139, 73, 72, 31, 23, 22, 12, 74]. Most other grounding work has also focused on single sentences. For example, Matuszek et al. (2012) consider single sentence descriptions of a subset of the objects in a scene. Additionally, work on grounding to control applications has utilized multi-sentence documents, but considered each sentence in isolation. For example, the techniques from Branavan et al. (2011a, 2011b, 2012) select an individual sentence from a strategy document to interpret and apply in a given situated context.

- **Implicit Cross-Sentence Reasoning** Most of the work on grounding that considers multiple sentences has done so in the context of a situated control

application, where the only cross sentence reasoning considered comes from the situated context itself. For example, Branavan et al. (2009, 2010) consider direction following in the context of the Microsoft Windows GUI. Their technique serially interprets individual sentences such that the only cross-sentence interaction comes from the changes to the GUI state generated by the actions from the preceding sentences. The techniques used by Artzi and Zettlemoyer (2013), Kim and Mooney (2012), Vogel and Jurafsky (2010) and Chen and Mooney (2011, 2012) work similarly in the context of a virtual world. Finally the technique of Tellex et al. (2014) does so in a real world robotic context.

- **Explicit Cross Sentence Reasoning** Some prior work has explicitly performed cross sentence reasoning, however this work has only considered relatively simple cross-sentence relationships. For example, Zettlemoyer and Collins (2009) consider coreference relationships by performing semantic parsing in a two step process. In the first step, they parse each sentence completely independently, but explicitly mark referring expressions in the resulting logical representation. In the second step, they resolve those referring expressions to one of the logical entities generated by a preceding sentence. Additionally, Artzi and Zettlemoyer (2011) manually define a cross sentence loss function which tries to encourage agreement between machine utterances and the corresponding human utterances. Finally, Lei et al. (2013) generate a tree structure representing a computer program input parser by assuming that the natural language describes the program inputs in serial order. In contrast, we consider cross-sentence relationships such as those in Figure 3-1 which are considerably more complex than simple coreference or tree structures.

Utilizing the Underlying Computer Platform

Our work utilizes the underlying platform to determine both semantic equivalence and generate execution outcomes. Our use of semantic equivalence is novel and has not been considered by prior work. Some prior work, however, has utilized the exe-

cution capabilities of the underlying platform. For example, Branavan et al. (2009, 2010) learns to interpret natural language instructions purely by observing the outcome of actions executed in the Windows GUI. Similarly, Branavan et al. (2011, 2012) learns to interpret a strategy guide by observing the game score received from the outcome of actions taken in a computer game. Additionally, both Berant et al. (2013) and Kwiatkowski et al. (2013) generate possible interpretations of a natural language question, execute these against a knowledge-base, and use the results of that execution in the learning process. Finally, as discussed above, much of the work in instruction interpretation utilizes the execution capabilities of the platform to generate the appropriate situated context for a given instruction during learning [5, 29, 27, 128].

3.2.3 Information Extraction

Our approach is related to work on template-based information extraction, where the goal is to identify instances of event templates in text and extract their slot fillers. Most work has focused on the supervised case, where the templates are manually defined and data is labeled with alignment information, e.g. [50, 87, 62, 110]. However, some recent work has studied the automatic induction of the set of possible templates from data [25, 111]. In our approach, systems of equations are relatively easy to specify, providing a type of template structure. However, our data does not contain the alignments, which we instead model with latent variables. Furthermore, template slots in information extraction typically have well-defined domain-specific meanings which align to a relatively constrained set of natural language embodiments. In contrast, our mathematical equations have no predefined meanings associated with them, and can align to a much more diverse space of natural language embodiments. For example a summation in an equation could refer to many different real world situations described in the natural language, such as a person being given additional goods, two different subsets of a larger set, the total income across a set of products, or many other quite different situations. Finally, mapping to a semantic representation that can be executed allows us to leverage weaker supervision during learning.

3.3 Mapping Word Problems to Equations

We map word problems to equations using a *derivation* which contains two components: (1) a template which defines the overall structure of the equation system, and (2) alignments of the slots in that template to the numbers and nouns in the text. We use the training data to both determine the space of possible derivations as well as learn a probability model to choose the best derivation for a given word problem. Specifically, the space of possible templates is induced from the data at training time. Furthermore, both the choice of template, and the alignments are highly ambiguous, and the choice of template is informed by the availability of good alignments. So we model these two choices jointly using a log-linear model.

Figure 3-2 shows both components of two different derivations. The template dictates the form of the equations in the system and the type of slots in each equation: u slots represent unknowns and n slots are for numbers that must be filled from the text. In Derivation 1, the selected template has two unknown slots, u_1 and u_2, and four number slots, n_1 to n_4. Slots can be shared between equations, for example, the unknown slots u_1 and u_2 in the example appear in both equations. Slots may have different instances, for example u_1^1 and u_1^2 are the two instances of u_1 in the example.

We align each slot instance to a word in the problem. Each number slot n is aligned to a number, and each unknown slot u is aligned to a noun. For example, Derivation 1 aligns the number 278 to n_1, 1.50 to n_2, 4 to n_3, and 792 to n_4. It also aligns both instances of u_1 (e.g., u_1^1 and u_1^2) to "Tickets", and both instances of u_2 to "tickets". In contrast, in Derivation 2, instances of the same unknown slot (e.g. u_1^1 and u_1^2) are aligned to two different words in the problem (different occurrences of the word "speed"). This allows for a tighter mapping between the natural language and the system template, where the words aligned to the first equation in the template come from the first two sentences, and the words aligned to the second equation come from the third.

Given an alignment, the template can then be instantiated: each number slot n is replaced with the aligned number, and each unknown slot u with a variable. The

Derivation 1	
Word problem	An amusement park sells 2 kinds of tickets. Tickets for children cost \$ 1.50 . Adult tickets cost \$ 4 . On a certain day, 278 people entered the park. On that same day the admission fees collected totaled \$ 792 . How many children were admitted on that day? How many adults were admitted?
Aligned template	$u_1^1 + u_2^1 - n_1 = 0$ $\quad$ $n_2 \times u_1^2 + n_3 \times u_2^2 - n_4 = 0$
Instantiated equations	$x + y - 278 = 0$ $\quad$ $1.5x + 4y - 792 = 0$
Answer	$x = 128$ $y = 150$

Derivation 2	
Word problem	A motorist drove 2 hours at one speed and then for 3 hours at another speed. He covered a distance of 252 kilometers. If he had traveled 4 hours at the first speed and 1 hour at the second speed , he would have covered 244 kilometers. Find two speeds?
Aligned template	$n_1 \times u_1^1 + n_2 \times u_2^1 - n_3 = 0$ $\quad$ $n_4 \times u_1^2 + n_5 \times u_2^2 - n_6 = 0$
Instantiated equations	$2x + 3y - 252 = 0$ $\quad$ $4x + 1y - 244 = 0$
Answer	$x = 48$ $y = 52$

Figure 3-2: Two complete derivations for two different word problems. Derivation 1 shows an alignment where two instances of the same slot are aligned to the same word (e.g., u_1^1 and u_1^2 both are aligned to "Tickets"). Derivation 2 includes an alignment where four identical nouns are each aligned to different slot instances in the template (e.g., the first "speed" in the problem is aligned to u_1^1).

output system of equations is then solved by the underlying mathematical system to generate the final answer. In the following, we formally define both the space of possible derivations, and the probability model we use to choose the best derivation.

3.3.1 Derivations

Definitions

Let $\mathcal{X}$ be the set of all word problems. A word problem $x \in \mathcal{X}$ is a sequence of k words $\langle w_1, \ldots w_k \rangle$. Also, define an *equation template* t to be a formula $A = B$, where A and B are expressions. An expression A is one of the following:

- A number constant f.
- A number slot n.
- An unknown slot u.
- An application of a mathematical relation R to two expressions (e.g., $n_1 \times u_1$).

We define a *system template* T to be a set of l equation templates $\{t_0, \ldots, t_l\}$. $\mathcal{T}$ is the set of all system templates. Unknown slots may occur more than once in a system template, to allow variables to be reused in different equations. We denote a specific instance i of a slot, u for example, as u^i. For brevity, we omit the instance index when a slot appears only once. To capture a correspondence between the text of x and a template T, we define an alignment p to be a set of pairs (w, s), where w is a token in x and s is a slot instance in T.

Given the above definitions, an equation e can be constructed from a template t where each number slot n is replaced with a real number, each unknown slot u is replaced with a variable, and each number constant f is kept as is. We call the process of turning a template into an equation *template instantiation.* Similarly, an equation system E is a set of l equations $\{e_0, \ldots, e_l\}$, which can be constructed by instantiating each of the equation templates in a system template T. Finally, an answer a is a tuple of real numbers.

An **amusement park** sells 2 **kinds** of **tickets**. **Tickets** for **children** cost $ 1.50 . Adult **tickets** cost $ 4 . On a certain **day**, 278 **people** entered the **park**. On that same **day** the **admission fees** collected totaled $ 792 . How many **children** were admitted on that **day**? How many **adults** were admitted?
$u_1^1 + u_2^1 - n_1 = 0$ $n_2 \times u_1^2 + n_3 \times u_2^2 - n_4 = 0$

Figure 3-3: The first example problem and selected system template from Figure 3-2 with all potential aligned words marked. Nouns (boldfaced) may be aligned to unknown slot instances u_i^j, and number words (highlighted) may be aligned to number slots n_i.

We define a derivation y from a word problem x to an answer as a tuple (T, p, E, a), where T is the selected system template, p is an alignment between T and x, E is the system of equations generated by instantiating T using x through p, and a is the answer generated by solving T. Note that derivations are uniquely defined by the choice of T and p, and we only include E and a for notational convenience. Finally, let $\mathcal{Y}$ be the set of all derivations.

The Space of Possible Derivations

We aim to map each word problem x to an equation system E. The space of equation systems considered is defined by the set of possible system templates $\mathcal{T}$ and the words in the original problem x, that are available for filling slots. We generate $\mathcal{T}$ from the training data, as described in Section 3.4.1. Given a system template $T \in \mathcal{T}$, we create an alignment p between T and x. The set of possible alignment pairs is constrained as follows: each number slot $n \in T$ can be aligned to any number in the text, a number word can only be aligned to a single slot n, and must be aligned to all instances of that slot. Additionally, an unknown slot instance $u \in T$ can only be aligned to a noun word. A complete derivation's alignment pairs all slots in T with words in x.

Figure 3-3 illustrates the space of possible alignments for the first problem and system template from Figure 3-2. Nouns (shown in boldface) can be aligned to any of the unknown slot instances in the selected template (u_1^1, u_1^2, u_2^1, and u_2^2 for the

template selected). Numbers (highlighted) can be aligned to any of the number slots (n_1, n_2, n_3, and n_4 in the template).

3.3.2 Probabilistic Model

Both the choice of system template and the choice of alignment are highly ambiguous, leading to many possible derivations $y \in \mathcal{Y}$ for each word problem $x \in \mathcal{X}$. We discriminate between competing analyses using a log-linear model, which has a feature function $\phi : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d$ and a parameter vector $\theta \in \mathbb{R}^d$. The probability of a derivation y given a problem x is defined as:

$$p(y|x;\theta) = \frac{e^{\theta \cdot \phi(x,y)}}{\sum_{y' \in \mathcal{Y}} e^{\theta \cdot \phi(x,y')}}$$

Section 3.7 defines the full set of features used.

At test time we consider two different metrics, generating the correct full system of equations, or generating just the correct numerical answer. These two metrics are highly correlated, since the answer is generated deterministically from the system of equations. They are not equivalent, however, since multiple systems of equations can generate the same numerical answer.

The first metric requires us to find the most likely system of equations E given a problem x, assuming the parameters θ are known:

$$f(x) = \arg\max_E p(E|x;\theta)$$

Here, the probability of the system of equations is marginalized over template selection and alignment:

$$p(E|x;\theta) = \sum_{\substack{y \in \mathcal{Y} \\ \text{s.t. } \mathrm{EQ}(y)=E}} p(y|x;\theta) \tag{3.1}$$

where $\mathrm{EQ}(y)$ extracts the system of equations E out of derivation y.

For the second metric, we find the most likely answer a by again marginalizing

over templates and alignments such that:

$$f(x) = \arg\max_{a} \sum_{\substack{y \in \mathcal{Y} \\ \text{s.t. } \mathrm{AN}(y)=a}} p(y|x;\theta)$$

where $\mathrm{AN}(y)$ extracts the answer a out of derivation y.

In both cases, the distribution over derivations y is modeled as a latent variable. We use a beam search inference procedure to approximately compute Equations 3.1 and 3.2, as described in Section 3.5.

3.4 Learning

To learn our model, we need to induce the structure of system templates in $\mathcal{T}$ and estimate the model parameters θ. This process can be driven by supervision that provides a system of equations E for all problems, or by weaker supervision that provides only an answer a for most problems.

3.4.1 Template Induction

We generate the space of possible system templates $\mathcal{T}$ from a set of n training examples $\{(x_i, E_i) : i = 1, \ldots, n\}$, where x_i is a word problem and E_i is a system of equations. Note that when some of the training data contains only answers, we generate the templates from the subset of the data labeled with full equation systems. To generalize a system of equations E to a system template T we:

1. Extract, n_x, all numbers contained in x.

2. Replace each number m in E with a number slot, if m is contained in n_x.

3. Replace each variable with an unknown slot.

In this way, numbers not mentioned in the problem text automatically remain in the template as constants. This allows us to solve problems that require numbers

Word problem
A chemist has a solution that is 18 % alcohol and one that is 50 % alcohol. He wants to make 80 liters of a 30 % solution. How many liters of the 18 % solution should he add? How many liters of the 30 % solution should he add?
Labeled equations
$18 \times 0.01 \times x + 50 \times 0.01 \times y = 30 \times 0.01 \times 80$ $x + y = 80$
Induced template system
$n_1 \times 0.01 \times u_1^1 + n_2 \times 0.01 \times u_2^1 = n_3 \times 0.01 \times n_4$ $u_1^2 + u_2^2 = n_5$

Figure 3-4: During template induction, we automatically detect the numbers in the problem (highlighted above) to generalize the labeled equations to templates. Numbers not present in the text are considered part of the induced template.

that are implied by the problem semantics rather than appearing directly in the text, such as the percent problem in Figure 3-4.

3.4.2 Parameter Estimation

For parameter estimation, we assume access to n training examples $\{(x_i, \mathcal{V}_i) : i = 1, \ldots, n\}$, each containing a word problem x_i and a validation function $\mathcal{V}_i$. The validation function $\mathcal{V} : \mathcal{Y} \to \{0, 1\}$ maps a derivation $y \in \mathcal{Y}$ to 1 if it is correct, or 0 otherwise.

We can vary the validation function to learn from different types of supervision. In Section 3.9 we will use validation functions that check whether the derivation y has either the correct system of equations E, or the correct answer a. Specifically, we will consider two scenarios:

- A fully supervised scenario where the validation function for all problems has access to the correct system of equations E.
- A semi-supervised scenario where the correct system of equations E is available for only a subset of the problems, while the validation function for the rest of

the problems has access to only the correct answer a.

Note that in neither scenario do we have access to the alignments P in the full derivation y. Thus, we estimate θ by maximizing the conditional log-likelihood of the data, marginalizing over all valid derivations:

$$O = \sum_i \sum_{\substack{y \in \mathcal{Y} \\ \text{s.t. } \mathcal{V}_i(y)=1}} \log p(y|x_i;\theta) - \frac{\lambda}{2} \sum_{v<|\theta|} \theta_v^2$$

where v is the parameter index, and λ is a meta-parameter controlling the level of regularization. We use L-BFGS [101] to optimize the parameters. The gradient of the individual parameter θ_j is given by:

$$\frac{\partial O}{\partial \theta_j} = \sum_i \mathbb{E}_{p(y|x_i,\mathcal{V}_i(y)=1;\theta)} \left[\phi_j(x_i,y)\right] - \mathbb{E}_{p(y|x_i;\theta)} \left[\phi_j(x_i,y)\right] - \lambda\theta_j \tag{3.2}$$

which is the standard log-linear gradient, where the first term is the expected feature counts in the correct derivations for x_i, the second term is the expected feature counts in all valid derivations for x_i, and the third term is the regularization. Section 3.5 describes how we approximate the first two terms of the gradient using beam search.

3.5 Inference

Computing the second expectation in Equation 3.2 requires summing over all templates and all possible ways to instantiate them. This space is exponential in the number of slots in the largest template in $\mathcal{T}$, the set of available system templates. Therefore, we approximate the computation using a beam search. We initialize the beam with all templates in $\mathcal{T}$ and iteratively align slots from the templates in the beam to words in the problem text. For each template, the next slot to be considered is selected according to a predefined canonicalized ordering for that template. After each iteration we prune the beam to keep the top-k partial derivations according to the model score. When pruning the beam, we allow at most l partial derivations for each template, to ensure that a small number of templates does not monopolize the

Input: Word problem x, the set of equation system templates $\mathcal{T}$, and the beam size k.

Definitions:

- $d = (d_T, d_{\vec{w}})$ is a partial derivation where $d_{\vec{w}}$ is a (possibly partial) sequence of words aligned to the slots in the system template d_T.

Functions:

- `COMPLETE`(d) indicates whether or not $d_{\vec{w}}$ contains alignments for every slot in d_T.
- `AUGMENT`(d, w) generates a new partial derivation by adding word w to $d_{\vec{w}}$.
- `kBEST`(B, k) scores all the partial derivations in B based on their current set of features, and then returns the k highest scoring derivations.

Initialization:

- $B = \{(T, \emptyset) \mid T \in \mathcal{T}\}$: Initialized the beam with a partial derivation for each template in $\mathcal{T}$, with an empty vector of alignments for each.

While: B contains partial derivations with unfilled slots

$B_{next} = \emptyset$

ForEach: $d \in B$

If: d has no empty slots

$B_{next} = B_{next} \cup d$

Else:

ForEach: valid word, $w \in x$ for the next unaligned slot in d_T

$B_{next} = B_{next} \cup$ `AUGMENT`(d, w)

$B =$ `kBEST`(B_{next}, k)

Output: k-Best list B

Algorithm 3: k-Best search algorithm. We run this for increasing values of f until the resulting k-best set is non-empty.

beam. We continue this process until all templates in the beam are fully instantiated. The details of the algorithm can be seen in Algorithm 3.

During learning we also compute the first term in the gradient (Equation 3.2) using our beam search approximation. Depending on the available validation function $\mathcal{V}$

(as defined in Section 3.4.2), we can accurately prune the beam for the computation of this part of the gradient. Specifically, when we have access to the labeled system of equations for a given sample, we can constrain the search to consider only partial hypotheses that could possibly be completed to produce the labeled equations.

3.6 Integrating Semantic Equality and Execution Outcomes

Performing joint inference over the choice of equation systems and alignments leads to a very large search space which can be difficult to explore efficiently. To manage this large search space, we take advantage of the capabilities of the underlying mathematical platform which allows us to compute the semantic equality between systems of equations, as well as utilize the execution outcomes resulting from solving the systems of equations.

3.6.1 Semantic Equality

While the space of possible systems of equations is very large, many syntactically different systems of equations are actually semantically equivalent. For example, the phrase "John is 3 years older than Bill" can correctly generate either the equation $j = b + 3$ or the equation $j - 3 = b$. These two equations are semantically equivalent, yet naive template generation would produce a separate template from each of these equations. To avoid this redundancy, we can utilize the inference capabilities of the underlying mathematical computer system. Specifically, during template generation, we use the mathematical solver Maxima [89] to canonicalize the templates into a normal form representation. The normal form is produced by symbolically solving for the unknown slots in terms of the number slots and the constants.

Slot Signatures

In a template like $s_1 + s_2 = s_3$, the slot s_1 is distinct from the slot s_2, but semantically these two slots are equivalent. Thus to share features between such slots, we generate a signature for each slot and slot pair, such that semantically equivalent slots have identical signatures. The signature for a slot indicates the system of equations it appears in, the specific equation it is in, and the terms of the equation it is a part of. Pairwise slot signatures concatenate the signatures for the two slots as well as indicating which terms are shared. This allows, for example, n_2 and n_3 in Derivation 1 in Figure 3-2 to have the same signature, while the pairs $\langle n_2, u_1 \rangle$ and $\langle n_3, u_1 \rangle$ have different signatures. To share features across templates, slot and slot-pair signatures are generated for both the full template, as well as for each of the constituent equations.

3.6.2 Execution Outcomes

Given a derivation, y, we can also utilize the underlying mathematical system to solve the resulting system of equations to generate a final numerical answer, a. We can utilize this answer in the learning process to help us determine whether or not the derivation is likely to be correct. For example, if y aligns a given unknown, u_1, to the word *tickets*, and the solution generates a fractional number for u_1, then the derivation is probably wrong. However, if u_1 is aligned to *price* then the derivation may be correct. Similarly, if u_1 is aligned to *price* and the solution generates a negative result, then the derivation is probably wrong, but if u_1 is aligned to *profit* the derivation may be correct. To avoid the need to hand generate rules like these, we integrate these correlations into the model as additional features similar to the rest of the features discussed in Section §3.7.

3.7 Features

The features $\phi(x, y)$ are computed for a derivation y and problem x and cover all derivation decisions, including template and alignment selection. Many of these fea-

tures are generated using part-of-speech tags, lematizations, and dependency parses computed with standard tools.[1] For each word in y aligned to a number slot, we also identify the closest noun in the dependency parse. For example, the noun for 278 in Derivation 1, Figure 3-2 would be "people." Most of the features are calculated based on these nouns, rather than the number words themselves.

In addition to the execution outcome features discussed in Section §3.6.2, we compute three other types of features: document level features, features that look at a single slot entry, and features that look at pairs of slot entries. Table 3.1 lists all the features used. Unless otherwise noted, when computing slot and slot pair features, a separate feature is generated for each of the signature types discussed in Section §3.6.1.

3.7.1 Document level features

Oftentimes the natural language in x will contain words or phrases which are indicative of a certain template, but are not associated with any of the words aligned to slots in the template. For example, the word "chemist" might indicate a template like the one seen in Figure 3-4. We include features that connect each template with the unigrams and bigrams in the word problem. We also include an indicator feature for each system template, providing a bias for its use.

3.7.2 Single Slot Features

We include three different types of features looking at individual slots.

Query Features The natural language x always contains one or more questions or commands indicating the queried quantities. For example, the first problem in Figure 3-2 asks "How many children were admitted on that day?" The queried quantities, the *number of children* in this case, must be represented by an unknown in the system of equations. We generate a set of features which look at both the word

[1] In our experiments these are generated using the Stanford parser [34]

Document level
Unigrams
Bigrams
Single slot
Has the same lemma as a question object
Is a question object
Is in a question sentence
Is equal to one or two (for numbers)
Word lemma X nearby constant
Slot pair
Dep. path contains: Word
Dep. path contains: Dep. Type
Dep. path contains: Word X Dep. Type
Are the same word instance
Have the same lemma
In the same sentence
In the same phrase
Connected by a preposition
Numbers are equal
One number is larger than the other
Equivalent relationship
Solution Features
Is solution all positive
Is solution all integer

Table 3.1: The features divided into categories.

overlap and the noun phrase overlap between slot words and the objects of a question or command sentence. We also compute a feature indicating whether a slot is filled from a word in a question sentence.

Small Number Indicators Algebra problems frequently use phrases such as "2 kinds of tickets" (e.g., Figure 3-2). These numbers do not typically appear in the equations. To account for this, we add a single feature indicating whether a number is one or two.

Features for Constants Many templates contain constants which are identifiable from words used in nearby slots. For example, in Figure 3-4 the constant 0.01 is related to the use of "%" in the text. To capture such usage, we include a set of

lexicalized features which concatenate the word lemma with nearby constants in the equation. These features do not include the slot signature.

3.7.3 Slot Pair Features

The majority of features we compute account for relationships between slot words. This includes features that trigger for various equivalence relations between the words themselves, as well as features of the dependency path between them. We also include features that look at the numerical relationship of two numbers, where the numeric values of the unknowns are generated as discussed in Section §3.6.2. This helps recognize that, for example, the total of a sum is typically larger than each of the (typically positive) summands.

Additionally, we have a single feature looking at shared relationships between pairs of slots. For example, in Figure 3-2 the relationship between "tickets for children" and "$1.50" is "cost". Similarly the relationship between "Adult tickets" and "$4" is also "cost". Since the actual nature of this relationship is not important, this feature is not lexicalized, instead it is only triggered by the equality of the representative words. We consider two cases: subject-object relationships where the intervening verb is equal, and noun-to-preposition object relationships where the intervening preposition is equal.

3.8 Experimental Setup

3.8.1 Datasets

We show results for three different datasets: Algebra, Arithmetic and Finance. The *Algebra* dataset is our main dataset and all results in Section §3.9 are for this dataset unless otherwise indicated.

Algebra We collected a new dataset of algebra word problems from Algebra.com, a crowd-sourced tutoring website. The questions were posted by students for mem-

Dataset	**Algebra**	**Arithmetic**	**Finance**
# problems	514	395	90
# sentences	1616	1137	179
# words	19357	11037	2632
Vocabulary size	2352	1429	336
Mean words per problem	37	28	29
Mean sentences per problem	3.1	2.9	2.0
Mean nouns per problem	13.4	10.4	9.9
# unique equation systems	28	9	16
Mean slots per system	7	3	4
Mean derivations per problem	4M	125	634

Table 3.2: Dataset statistics.

bers of the community to respond with solutions. Therefore, the problems are highly varied, and are taken from real problems given to students. We heuristically filtered the data to get only linear algebra questions which did not require any explicit background knowledge. From these we randomly chose a set of 1024 questions. As the questions are posted to a web forum, the posts often contained additional comments which were not part of the word problems and the solutions are embedded in long free-form natural language descriptions. To clean the data we asked Amazon Mechanical Turk workers to extract from the text: the algebra word problem itself, the solution equations, and the numeric answer. We manually verified both the equations and the numbers to ensure they were correct. To ensure each problem type is seen at least a few times in the training data, we removed the infrequent problem types. Specifically, we induced the system template from each equation system, as described in Section 3.4.1, and removed all problems for which the associated system template appeared less than 6 times in the dataset. This left us with 514 problems. Problems in this dataset utilize all four arithmetic operators.

Arithmetic We also report results for the arithmetic dataset collected by Hosseini et al. (2014) in order to compare directly to the *ARIS* system introduced in that paper. This dataset contains only addition and subtraction problems solved with a single equation.

Finance To ensure that our techniques generalize to other domains, we also collected a dataset of 90 finance problems extracted from finance class homework assignments found on-line. Problems in this dataset also include exponentiation, in addition to the four arithmetic operators.

Table 3.2 provides statistics for all three datasets. We can see that the Algebra dataset has significantly more ambiguity than the other datasets as evidenced by the much larger average number of possible derivations.

3.8.2 Baselines

While there has been significant past work on rule based techniques for automatically solving math word problems, none of the available systems could handle the type of problems in our *Algebra* dataset. So instead, we compare to two baselines which help confirm the general difficulty of the problems in the dataset:

- **Majority** This baseline always chooses the most common template, t in the training data. The unknown slots in the template are left unaligned, and the numerical slots are aligned based purely on the order of the numbers in the original text. Specifically, we find the ordered index of each number in the text, such that the index of the first number is 1, the index of the second is 2, etc. Then for each instance of a given system template T in the training data, we generate a sequence of the indexes of the numbers used to fill its number slots. For example, Derivation 1 from Figure 3-2 would generate the sequence $(4, 2, 3, 5)$. We choose the most common such sequence observed in the training data, and use this sequence to generate all alignments.

- **Correct Equation Types** This baseline always chooses the *correct* template. Note that this requires access to the labels and is more information than our system is provided. The alignment selection is performed using the same technique as the *Majority* baseline based on the order of the numbers in the original text, but with a separate order choice for each template.

On the *Arithmetic* dataset we also compare to the **ARIS** system described in Hosseini et al. (2014). This system learns to classify verbs into one of seven different categories indicating state transitions such as *giving* or *receiving* items. A rule-based technique uses these categories along with the problem text to construct a formal state-transition representation of the problem. From this representation the system deterministically generates an equation that can be solved to prove the final answer to the problem. This system only handles addition and subtraction problems so we cannot compare to on the other two datasets.

3.8.3 Evaluation Protocol

Since our model generates a solution for every problem, we report only accuracy. We report two metrics:

- **Equation Accuracy:** measures how often the system generates exactly the correct equation system. When comparing equations, we avoid spurious differences by canonicalizing the equation systems, as described in Section 3.6.1.
- **Answer Accuracy:** evaluates how often the generated numerical answer is correct. To compare answer tuples we disregard the ordering and require each number appearing in the reference answer to appear in the generated answer.

We run all our experiments using n-fold cross-validation. On the *Arithmetic* dataset we use 3 folds to be consistent with the results in Hosseini et al. (2014), and on the *Algebra* and *Finance* datasets we use 5 folds. To maintain a consistent protocol, the folds are chosen randomly for all results we report. Note that to show cross-domain effects, Hosseini et al. (2014) also reports results for a non-random choice of folds, but we do not include those results here.

3.8.4 Parameters and Solver

In our experiments we set k in our beam search algorithm (Section 3.5) to 200, and l to 20. We run the L-BFGS computation for 50 iterations. We regularize our

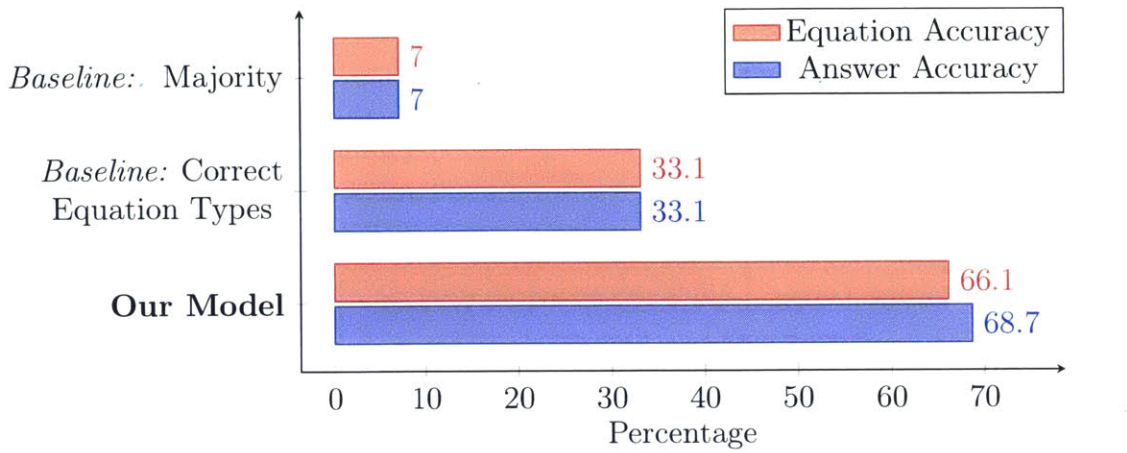


Figure 3-5: **Algebra Dataset:** Accuracy of our model relative to both baselines on the *Algebra* dataset when provided full equational supervision for all training samples. We can see that our model significantly outperforms both baselines.

learning objective using a λ value of 0.1. The set of mathematical relations supported by our implementation is $\{+, -, \times, /, power\}$. This set was chosen to handle the questions in the data, and can be easily extended to handle more relations. Our implementation uses the Gaussian Elimination function in the Efficient Java Matrix Library (EJML) [1] to generate answers given a set of equations.

3.9 Results

We evaluate our model with two different forms of supervision, either full equations for all questions (*Fully Supervised*) or only final numerical answers for most of the questions (*Semi-Supervised*).

3.9.1 Fully Supervised

We first consider the scenario where our model is given full equations for all questions. Figures 3-5, 3-6 and 3-7 show results for this scenario on the three different datasets. We can see that on all three datasets, our system significantly outperforms the *Majority* baseline, and even the *Correct Equation Types* baseline which is provided the correct template and only has to choose the correct alignment. Furthermore, on the

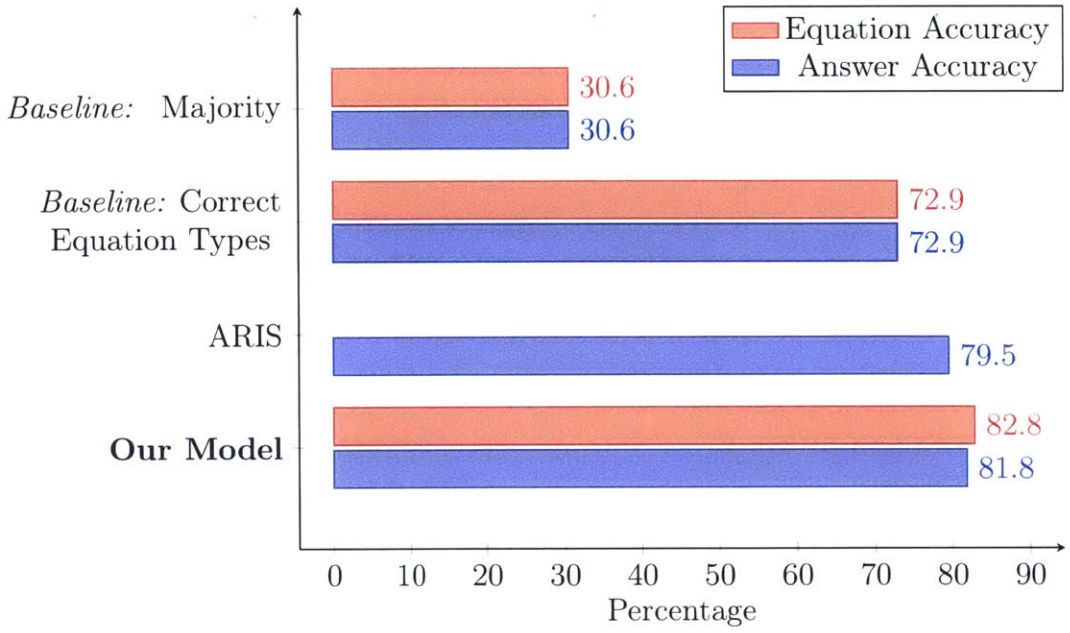


Figure 3-6: **Arithmetic Dataset:** Accuracy of our model relative to the baselines on the *Arithmetic* dataset when provided full equational supervision for all training samples. We can see that our model even outperforms the **A**RIS system which includes significant manual engineering for the type of problems seen in this dataset.

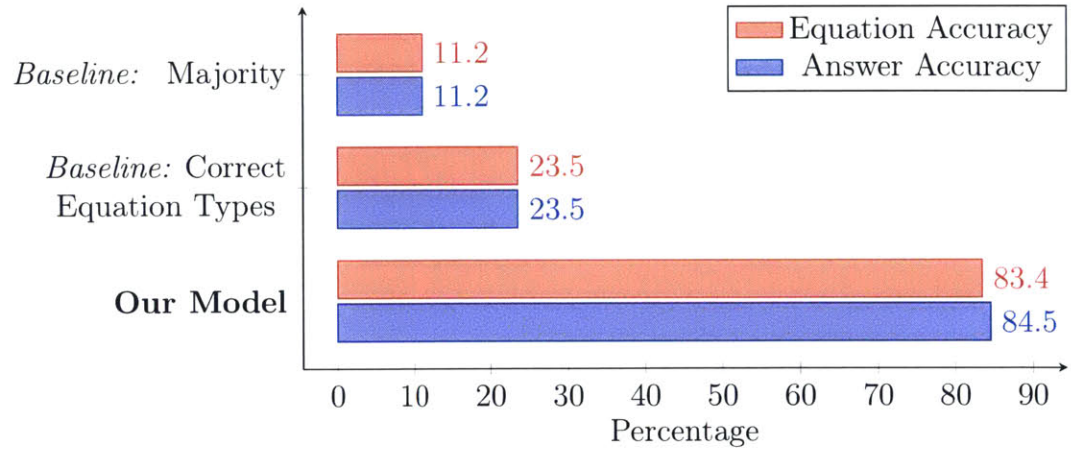


Figure 3-7: **Finance Dataset:** Accuracy of our model relative to both baselines on the *Finance* dataset when provided full equational supervision for all training samples. We can see that even in more applied domains such as finance, our model significantly outperforms both baselines.

Model	Equation Accuracy	Answer Accuracy
Pipeline Model (No Joint Model)	51.4	55.1
No Execution Outcomes or Semantic Equivalence	50.7	56.6
No Semantic Equivalence	54.7	62.3
No Execution Outcomes	61.4	63.6
Our Full Model	66.1	68.7

Table 3.3: Accuracy of model when various components are removed. This shows the importance of all three main components of our model: *Joint Inference*, *Semantic Equivalence*, and *Execution Outcomes*.

arithmetic dataset, we outperform the **ARIS** system which is engineered specifically for the type of problems seen in that dataset. The strong results across all three datasets show that our model generalizes well across various types of natural language math problems. In the rest of this subsection we evaluate our model in various ways on the *Algebra* dataset to help identify both why the model performs well, and where its weaknesses lie.

Model Ablations

We start by evaluating which aspects of the overall model are most important to its performance. Our model is distinguished by three important properties. First, we jointly model the selection of the system template and alignments, in order to handle the complex cross-sentence relationships we see in the word problems. Additionally, we use both the *semantic equivalence* technique discussed in Section §3.6.1 and the *execution outcomes* discussed in Section §3.6.2 to handle the large search space that results from the joint modeling. We can see from Figure 3.3 that all three of these properties are critical to our model's performance. First, moving from our joint modeling technique to the *Pipeline Model* reduces performance by almost 15% on equation accuracy, and 14% on answer accuracy. The *Pipeline Model* first predicts the best system template, using only the document-level features, and then predicts the best set of alignments for this template. Second, if we utilize the joint inference,

	All Features	w/o pair	w/o document
All Features	68.7	42.8	63.8
w/o single	65.9	39.6	57.6
w/o document	63.8	25.7	–
w/o pair	42.8	–	–

Table 3.4: Cross-validation accuracy results with different feature groups ablated. The first row and column show the performance when a single group is ablated, while the other entries show the performance when two groups are ablated simultaneously.

but remove both the semantic equivalence and execution outcomes components of our model (*No Execution Outcomes or Semantic Equivalence*), then performance also degrades significantly, by more than 15% and 12% respectively. Finally, we can see that the ability to compute semantic equivalence is somewhat more critical to the effectiveness of our model than the execution outcomes (*No Semantic Equivalence* vs. *No Execution Outcomes*).

Feature Ablations

We now look at how the various groups of features used by our model affect its performance. Table 3.4 shows ablation results for each feature group from Section §3.7. We can see that all of the features contribute to the overall performance, and that the *pair* features are the most important followed by the *document* features. We also see that the *pair* features can compensate for the absence of other features. For example, the performance drops only slightly when the *document* features are removed in isolation. However, the drop is much more dramatic when they are removed along with the *pair* features.

Performance and Template Frequency

We now turn to an analysis of the relationship between equation accuracy and the frequency of each equation template in the data set. Table 3.5 reports results after grouping the problems into four different frequency bins. We can see that our system correctly answers more than 85% of the question types which occur frequently while

Template Frequency	Equation Accuracy	Answer Accuracy	Percentage of Data
≤ 10	43.6	50.8	25.5
$11 - 15$	46.6	45.1	10.5
$16 - 20$	44.2	52.0	11.3
> 20	85.7	86.1	52.7

Table 3.5: Performance on different template frequencies.

(1)	A painting is 10 inches tall and 15 inches wide. A print of the painting is 25 inches tall, how wide is the print in inches?
(2)	A textbook costs a bookstore 44 dollars, and the store sells it for 55 dollars. Find the amount of profit based on the selling price.
(3)	The sum of two numbers is 85. The difference of twice of one of them and the other one is 5. Find both numbers.
(4)	The difference between two numbers is 6. If you double both numbers, the sum is 36. Find the two numbers.

Figure 3-8: Examples of problems our system does not solve correctly.

still achieving near 50% accuracy on those that occur relatively infrequently.

Qualitative Error Analysis

In order to understand the weaknesses of our system, we examined its output on one fold of the data and identified two main classes of errors. The first, accounting for approximately one-quarter of the cases, includes mistakes where more background or world knowledge might have helped. For example, Problem 1 in Figure 3-8 requires understanding the relation between the dimensions of a painting, and how this relation is maintained when the painting is printed, and Problem 2 relies on understanding concepts of commerce, including cost, sale price, and profit. While these relationships could be learned in our model with enough data, as it does for percentage problems (e.g., Figure 3-4), various outside resources, such as knowledge bases (e.g. Freebase) or distributional statistics from a large text corpus, might help us learn them with less training data.

The second category, which accounts for about half of the errors, includes mistakes that stem from compositional language. For example, the second sentence in Problem 3 in Figure 3-8 could generate the equation $2x - y = 5$, with the phrase "twice of one of them" generating the expression $2x$. Given the typical shallow nesting, it is possible to learn templates for these cases given enough data, and in the future it might also be possible to develop new, cross-sentence semantic parsers to enable better generalization from smaller datasets.

3.9.2 Semi-Supervised

The results presented in the first part of this section focused on learning from training data that contains full equations for all training samples. We now turn our attention to learning from weaker supervision, just numerical answers, for most of the data. We consider two different scenarios. In both cases, a small fraction of the training data is labeled with full equations, while the rest of the data is labeled with only numerical answers. In the first scenario, we label only five samples with full equations, but very carefully choose these five samples. In the second scenario, we vary the fraction labeled with full equations, but choose the labeled set randomly. Both results are based on the *Algebra* dataset.

Equation Labels For Only Five Samples

We can see from Figure 3-9 that our model can learn quite effectively from the relatively weak supervision provided by mostly numerical answers. Specifically, *5 Equations + Answers* shows the performance when our model is given full equations for only five samples, and just numerical answers for the rest of the training data. We chose the fully labeled samples by identifying the five most common types of questions in the data and annotating a randomly sampled question of each type. We can see that even with this limited supervision our system correctly answers almost 70% as many questions as when it is given full equations for all of the training data (*All Equations*) i.e. 68.7% vs. 46.1%. In contrast, the baseline scenario (*5 Equations*),

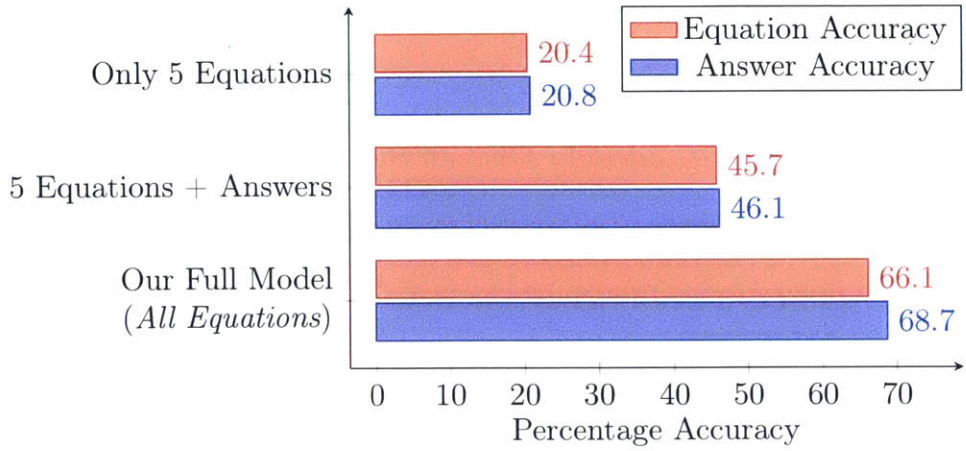


Figure 3-9: Performance evaluation of our model when provided only numerical answers for most training samples, with full equations provided for just five of the samples (*5 Equations + Answers*). This significantly outperforms a baseline version of our model which is provided only the five equations and no numerical answers (*Only 5 Equations*). It also achieves about 70% of the performance of *Our Full Model* which is provided full equations for all training samples.

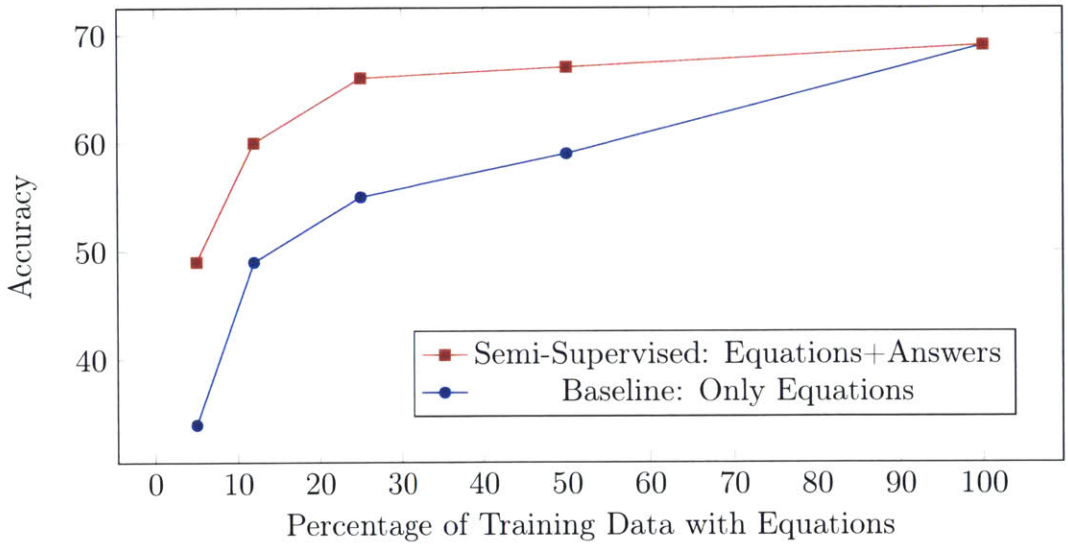


Figure 3-10: Performance evaluation of our model when we randomly choose the samples to label with full equations. *Semi-Supervised* is provided with full equation labels for a fraction of the data (with the fraction varied along the x-axis), and just numerical answers for the rest of the data. *Baseline* is provided the same subset of the training samples labeled with full equations, but does not have access to the rest of the training data.

which is provided only the five seed equation labels and does not have access to the numerical answer labels, performs much worse.

Equation Labels For Randomly Chosen Samples

The results in Figure 3-10 show that even when we randomly choose the set of samples to label with full equations, the model can learn quite effectively from only numerical answers for most of the samples. When full equations are provided for only 5% of the questions we see a relative gain of 40% for the *Semi-Supervised* model over the *Baseline* model which does not utilize any numerical answers, i.e. 49% vs. 34%. Furthermore, when full equations are provided for only 25% of the training data the *Semi-Supervised* model correctly answers 66% of the questions, just below the 69% it achieves when given full equations for all questions.

3.10 Conclusion

We presented an approach for automatically learning to solve algebra word problems. Our model jointly constructs systems of equations and aligns their variables and numbers to the problem text. To handle the large search space resulting from the joint model we utilize both the semantic equality and execution capabilities of the underlying mathematical engine. Computing the semantic equality between abstracted systems of equations allows us to avoid significant redundancy in the search process. Additionally, solving a possible equational interpretation of the text, allows us to use the resulting numerical answer to help determine whether or not the interpretation is correct. Using a newly gathered corpus we show that our joint model, our use of semantic equality and our use of execution outcomes all significantly improve the accuracy of the resulting system. To the best of our knowledge, this is the first learning result for this task.

There are many opportunities to extend the approach to related domains. The general representation of mathematics lends itself to many different domains including geometry, physics, and chemistry. Additionally, the techniques can be used to syn-

thesize even more complex structures, such as general-purpose computer programs, from natural language.

Chapter 4

Learning to Generate Plans from Text

In the first two chapters we tackled the problem of translating natural language task descriptions into computer programs which can execute those tasks. In both cases we translated the natural language description into a declarative program, i.e. a formal program which acts as a high-level specification by defining constraints on the output, but does not enumerate a sequence of step-by-step instructions which can be directly executed in order to generate such an output. To actually execute these programs, we relied on the underlying computer systems to efficiently transform these declarative programs into a sequence of executable steps, i.e. a regular expression executor, and a mathematical solver. However, for many domains this *program synthesis* process can be computationally intractable.

One such domain is that of classical robotic planning. A typical task in this domain might be to *build a pickaxe*, which we could encode formally in the declarative program `have(pickaxe)`, meaning that we would like to reach a state where we have a pickaxe. However, translating from this declarative form, into a set of steps that the robot can execute to build a pickaxe is often computationally intractable in practice and (in the general case) is an NP-Hard problem [46].

In this chapter, we show how we can leverage natural language understanding to help overcome these computational tractability issues. Specifically, many computing domains have significant on-line natural language documentation describing the capabilities and dynamics of the domain. Correct interpretation of this text can

provide a formal high-level description of the structure of the domain. These formal descriptions can be used by a low-level planner to help constrain the search process it performs during program synthesis. We show that by integrating this textual knowledge, the resulting system can successfully complete almost twice as many tasks as a state-of-the-art baseline planner which does not utilize the text.

4.1 Introduction

The idea of automatically synthesizing a program from a high-level specification has long been a goal of artificial intelligence research [49]. Early work in this area utilized techniques based on theorem-proving [86], while more recent work has focused on using version algebras [76], SAT/SMT solvers [120], and genetic programming [65]. Despite significant progress in this area over the last several years, wide-scale use of program synthesis is still hampered by the central issue of computational tractability. For general programs, the search problem is NP-hard, and so to maintain tractability current systems are forced to either heavily constrain the space of programs they allow [118], constrain the space of specifications they allow [129], manually craft high-level abstractions [46, 81], or focus on relatively short simple programming tasks such as bit-manipulation [121].

One noteworthy aspect of this situation is that many interesting programming domains have significant existing documentation describing the domain in natural language. For example, most computer applications have user manuals and help websites [91], programming APIs typically have documentation for each command as well as tutorials outlining their use [39] and games and virtual worlds often have extensive user generated wiki sites [93]. Automatic interpretation of the natural language descriptions in these documents could be used to infer the abstract structure of the programming domain, thereby enabling more efficient program synthesis. This chapter presents just such a system which utilizes existing text documentation in order to help counter the computational tractability problems of program synthesis.

We develop our system in the domain of *classical planning*, an area of program

A pickaxe, which is used to harvest stone, can be made from wood.

(a)

Low Level Actions for: wood → pickaxe → stone	
step 1:	move from (0,0) to (2,0)
step 2:	chop tree at: (2,0)
step 3:	get wood at: (2,0)
step 4:	craft plank from wood
step 5:	craft stick from plank
step 6:	craft pickaxe from plank and stick
...	
step N-1:	pickup tool: pickaxe
step N:	harvest stone with pickaxe at: (5,5)

(b)

Figure 4-1: Text description of preconditions and effects (a), and the low-level actions connecting them (b).

synthesis which has received considerable focus [134]. Work in this area has been used to control not only the humanoid robots for which it was originally designed [14] but also the actions of space satellites [135] and even self-driving cars [38]. Much progress has been made in this field over the past 20 years, leading to planners which are very fast and can work quite well in practice [38]. Despite this fact, planning is still hindered by the computational issues inherent to program synthesis in general, and so effective deployment of state-of-the-art systems in complex domains typically requires manual domain-specific design of action hierarchies and/or heuristics to ensure tractability [46, 134, 38].

Challenges In order to take advantage of natural language documentation to help overcome the computation issues in classical planning, we must tackle two primary challenges:

- **Mismatch in Abstraction:** There is a mismatch between the typical abstraction-level of human language and the granularity of planning primitives. Consider, for example, text describing a virtual world such as *Minecraft*[1] and a formal

[1]http://www.minecraft.net/

description of that world using planning primitives. Due to the mismatch in granularity, even the simple relations between *wood*, *pickaxe* and *stone* described in the sentence in Figure 4-1(a) result in dozens of low-level planning actions in the world, as can be seen in Figure 4-1(b). While the text provides a high-level description of world dynamics, it does not provide sufficient details for successful plan execution.

- **Lack of Labeled Training Data:** The main goal of our system is to avoid the need to manually generate domain specific heuristics or action hierarchies by instead utilizing existing natural language documents. However, traditional natural language grounding techniques rely on domain specific labeled training data which is typically generated manually [29, 127, 5]. If we must generate such resources for every domain of interest, then our efforts are probably better spent developing domain specific planning systems, rather than annotating natural language data.

Key Ideas To handle the mismatch in abstraction, our system utilizes the fact that natural language domain documentation typically describes the set of available actions one can perform, and how those actions relate to each other. For example, the natural language relation "is used to" from Figure 4-1(a), implies that obtaining a *pickaxe* is a precondition for obtaining *stone*. Thus, our solution grounds these linguistic relations in abstract *precondition* relations *between* states in the underlying domain, rather than directly in the entities of the domain.

To avoid the need for labeled training data our system takes advantage of the ability of the underlying system to compute execution outcomes. Specifically, we build on the intuition that the validity of precondition relations extracted from text can be informed by the execution of a low-level planner. For example, if a planner can find a plan to successfully obtain *stone* after obtaining a *pickaxe*, then a *pickaxe* is likely a precondition for *stone*. Conversely, if a planner generates a plan to obtain *stone* without first obtaining a *pickaxe*, then it is likely not a precondition. This feedback can enable us to learn to correctly ground the natural language relations

without annotated training data. Moreover, we can use the learned relations to guide a high level planner and ultimately improve planning performance.

Summary of Approach We implement these ideas in the reinforcement learning framework, wherein our model jointly learns to predict precondition relations from text and to perform high-level planning guided by those relations. For a given planning task and a set of candidate relations, our model repeatedly predicts a sequence of subgoals where each subgoal specifies an attribute of the world that must be made true. It then asks the low-level planner to find a plan between each consecutive pair of subgoals in the sequence. The observed feedback – whether the low-level planner succeeded or failed at each step – is utilized to update the policy for both text analysis and high-level planning.

Evaluation We evaluate our algorithm in the *Minecraft* virtual world, using a large collection of user-generated on-line documents as our source of textual information. Our results demonstrate the strength of our relation extraction technique – while using planning feedback as its only source of supervision, it achieves a precondition relation extraction accuracy on par with that of a supervised SVM baseline. Specifically, it yields an F-score of 66% compared to the 65% of the baseline. In addition, we show that these extracted relations can be used to improve the performance of a high-level planner. As baselines for this evaluation, we employ the Metric-FF planner [56],[2] as well as a text-unaware variant of our model. Our results show that our text-driven high-level planner significantly outperforms all baselines in terms of completed planning tasks – it successfully solves 80% as compared to 41% for the Metric-FF planner and 69% for the text unaware variant of our model. In fact, the performance of our method approaches that of an oracle planner which uses manually-annotated preconditions.

[2]The state-of-the-art baseline used in the 2008 International Planning Competition. http://ipc.informatik.uni-freiburg.de/

4.2 Related Work

Our work is related to prior work on natural language event semantics, language grounding, and hierarchical planning. The prior work on generating programs previously discussed in Sections §2.2.2 and §2.2.1 is also related.

4.2.1 Extracting Event Semantics from Text

The task of extracting preconditions and effects has previously been addressed in the context of lexical semantics [117, 116]. These approaches combine large-scale distributional techniques with supervised learning to identify desired semantic relations in text. Such combined approaches have also been shown to be effective for identifying other relationships between events, such as causality [47, 26, 13, 10, 37].

Similar to these methods, our algorithm capitalizes on surface linguistic cues to learn preconditions from text. However, our only source of supervision is the feedback provided by the planning task which utilizes the predictions. Additionally, we not only identify these relations in text, but also show they are valuable in performing an external task.

4.2.2 Learning Semantics via Language Grounding

Our work fits into the broad area of grounded language acquisition, where the goal is to learn linguistic analysis from a situated context [102, 119, 141, 40, 96, 95, 16, 84, 133]. Within this line of work, we are most closely related to the reinforcement learning approaches that learn language by interacting with an external environment [16, 17, 133, 15, 18, 19, 20].

The key distinction of our work is that the underlying planning system directly helps to generate pieces of the output program. This supplements previous work which simply uses the underlying system to execute potential output programs, or program steps for feedback [16, 133, 133, 15, 18, 19, 20]. Another important difference of our setup is the way the textual information is utilized in the situated context.

Instead of getting step-by-step instructions from the text, our model uses text that describes general knowledge about the domain structure. From this text, it extracts relations between objects in the world which hold independently of any given task. Task-specific solutions are then constructed by a planner that relies on these relations to perform effective high-level planning.

4.2.3 Hierarchical Planning

It is widely accepted that high-level plans that factorize a planning problem can greatly reduce the corresponding search space [100, 6]. Previous work in planning has studied the theoretical properties of valid abstractions and proposed a number of techniques for generating them [67, 137, 90, 9]. In general, these techniques use static analysis of the low-level domain to induce effective high-level abstractions. In contrast, our focus is on learning the abstraction from natural language. Thus our technique is complementary to past work, and can benefit from human knowledge about the domain structure.

4.3 Problem Formulation

Our task is two-fold. First, given a text document describing an environment, we wish to extract a set of precondition/effect relations implied by the text. Second, we wish to use these induced relations to determine an action sequence for completing a given task in the environment.

We formalize our task as illustrated in Figure 4-2. As input, we are given a world defined by the tuple $\langle S, A, T \rangle$, where S is the set of possible world states, A is the set of possible actions and T is a deterministic state transition function. Executing action a in state s causes a transition to a new state s' according to $T(s' \mid s, a)$. States are represented using propositional logic predicates $x \in \mathbf{x}$, where each state is simply a set of such predicates, i.e. $s \subset \mathbf{x}$.

The objective of the text analysis part of our task is to automatically extract a set of valid precondition/effect relationships from a given document d. Given our defini-

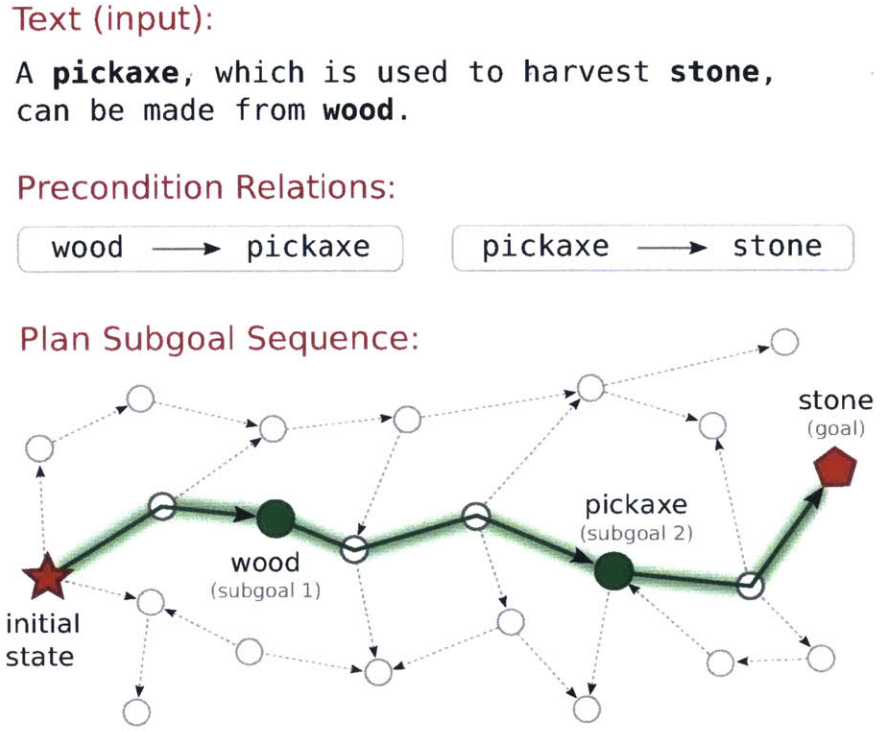


Figure 4-2: A high-level plan showing two subgoals in a precondition relation. The corresponding sentence is shown above.

tion of the world state, preconditions and effects are merely single term predicates, x, in this world state. We assume that we are given a seed mapping between a predicate x, and the word types in the document that reference it (see Table 4.4 for examples). Thus, for each predicate pair $\langle x^f, x^t \rangle$ we want to utilize the text to predict where there exists a precondition relation from x^f to x^t, i.e. whether x^f is a precondition for x^t. For example, from the text in Figure 4-2, we want to predict that possessing a pickaxe is a precondition for possessing stone. Note that this relation implies the reverse as well, i.e. predicate x^t can be interpreted as the effect of an action sequence performed on a state containing predicate x^f.

Each planning goal $g_i \in G$ is defined by a starting state $s_0^{g_i}$, and a final goal state $s_f^{g_i}$. This goal state is represented by a set of predicates which need to be made true. In the planning part of our task, our objective is to find a sequence of actions $\vec{a}$ that connect s_0^g to s_f^g. Finally, we assume document d does not contain step-by-step instructions for any individual task, but instead describes general facts about the

given world that are useful for a wide variety of tasks.

4.4 Model

The key idea behind our model is to leverage textual descriptions of preconditions and effects to guide the construction of high level plans. We define a high-level plan as a sequence of *subgoals*, where each subgoal is represented by a single-term predicate, x_i, that needs to be set in the corresponding world state – e.g. `have(wheat)=true`. Thus the set of possible subgoals is defined by the set of all possible single-term predicates in the domain. In contrast to low-level plans, the transition between these subgoals can involve multiple low-level actions. Our algorithm for textually informed high-level planning operates in four steps:

1. Use text to predict the preconditions of each subgoal. These predictions are for the entire domain and are not goal specific.
2. Given a planning goal and the induced preconditions, predict a subgoal sequence that achieves the given goal.
3. Execute the predicted sequence by giving each pair of consecutive subgoals to a low-level planner. This planner, treated as a black box, computes the low-level plan actions necessary to transition from one subgoal to the next.
4. Update the model parameters, using the low-level planner's success or failure as the source of supervision.

We formally define these steps below.

4.4.1 Modeling Precondition Relations

Given a document d, and a set of subgoal pairs $\langle x^f, x^t \rangle$, we want to predict whether subgoal x^f is a precondition for x^t. We assume that precondition relations are generally described within single sentences. We first use our seed grounding in a preprocessing step where we extract all predicate pairs where both predicates are mentioned

in the same sentence. We call this sequence, $V = \langle v_0, v_1, \cdots v_n \rangle$, the *Candidate Relations*. Note that this sequence will contain many invalid relations since co-occurrence in a sentence does not necessarily imply a valid precondition relation.[3] Thus for each candidate relationship, $v_i = \langle x_i^f, x_i^t, \vec{w}_i, q_i \rangle$, our task is to predict $c_i \in \{-1, 1\}$ which specifies whether or not sentence $\vec{w}_i$ with dependency parse q_i indicates a precondition relationship between predicates x_i^f and x_i^t. We model this decision via a log linear distribution as follows:

$$p(c_i \mid v_i; \theta_c) = \frac{e^{\theta_c \cdot \phi_c(c_i, v_i)}}{\sum_{c' \in \{-1,1\}} e^{\theta_c \cdot \phi_c(c', v_i)}} \tag{4.1}$$

where θ_c is the vector of precondition prediction parameters. We compute the feature function ϕ_c using the seed grounding, the sentence $\vec{w}_i$, and the given dependency parse q_i of the sentence. We define $C = \langle c_0, c_1, \cdots c_n \rangle$ to be a sequence of predictions for each of the candidate relationships, $v_i \in V$.

4.4.2 Modeling Subgoal Sequences

Given a planning goal $g_i \in G$, defined by initial and final goal states s_0^g and s_f^g, our task is to predict a sequence of subgoals $\vec{x}^i$ which will achieve the goal. We condition this decision on our predicted sequence of precondition relations C, by modeling the distribution over sequences $\vec{x}$ as:

$$\begin{aligned} p(\vec{x}^i \mid g_i, C; \theta_x) &= \prod_{t=1}^{n} p(x_t^i \mid x_{t-1}^i, g_i, C; \theta_x) \\ p(x_t^i \mid x_{t-1}^i, g_i, C; \theta_x) &= \frac{e^{\theta_x \cdot \phi_x(x_t^i, x_{t-1}^i, g_i, C)}}{\sum_{x' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x', x_{t-1}^i, g_i, C)}} \end{aligned}$$

Here we assume that subgoal sequences are Markovian in nature and model individual subgoal predictions using a log-linear model. Note that in contrast to Equation 4.1 where the predictions are goal-agnostic, these predictions are goal-specific. As before, θ_x is the vector of model parameters, and ϕ_x is the feature function. Additionally, we

[3]In our dataset only 11% of *Candidate Relations* are valid.

assume a special stop symbol, $x_\emptyset$, which indicates the end of the subgoal sequence.

4.5 Parameter Estimate via Execution Outcomes

Parameter estimation in our model is done using execution outcomes via reinforcement learning. Specifically, once the model has predicted a subgoal sequence for a given goal, the sequence is given to the low-level planner for execution. The success or failure of this execution is used to compute the reward signal R for parameter estimation. This predict-execute-update cycle is repeated until convergence [125]. This section discusses the details of this process.

Our goal is to estimate the optimal parameters for each component of our model, θ_x^*, and θ_c^*. For notational convenience, we will refer to the total set of parameters as θ, and the optimal such set as θ^*. To estimate the optimal parameters, we define a reward function, R which correlates well with sub-goal sequences which allow the low-level planner to successfully generate full plans. We define R based on a choice of $C = \langle c_0, c_1, \cdots c_n \rangle$, the validity of each candidate relation, as well as $X = \langle \vec{x}^0, \vec{x}^1, \cdots \vec{x}^m \rangle$, a sequence of subgoal sequences, one for each planning goal, $g_i \in G$. Given a reward function, our learning objective is to maximize the *Value function*, V_θ, which is the expected reward we will receive from a combined choice of C and X. Formally:

$$V_\theta = \mathbb{E}_{p(X,C|G,V)} \left[R(X, C) \right]$$

We discuss the details of the reward function in Section §4.5.2, but to facilitate learning, we will assume the that the reward function factorizes as follows:

$$\begin{aligned} R(X, C) &= R_x(X) + R_c(X, C) \\ &= \sum_{i=0}^{m} R_x(\vec{x}^i, i) + \sum_{i=0}^{n} R_c(X, c_i, i) \end{aligned}$$

Since computing $R(X, C)$ requires running the low-level planner on each sequence in X, we cannot compute the objective in closed form. We can, however, estimate

g_i	a planning goal defined by a starting state, $s_0^{g_i}$ and a final state $s_f^{g_i}$
G	the sequence of all planning goals, $\langle g_0, g_1, \cdots g_n \rangle$
$\vec{x}^i$	sequence of subgoal predicates,$\langle x_0^i, x_1^i \cdots x_{\lvert\vec{x}^i\rvert} \rangle$
X	a sequence of $\vec{x}^i$, $\langle \vec{x}^0, \vec{x}^1 \cdots \vec{x}^n \rangle$
$\mathcal{X}$	space of possible choices of X
$\mathbf{x}$	space of possible subgoal predicates
$\vec{\mathbf{x}}$	space of possible subgoal predicate sequences
c_i	predicts the validity of candidate precondition relationship, i, such that $c_i \in \{0, 1\}$
C	a sequence of precondition predictions, $\langle c_0, c_1, \cdots c_n \rangle$, one for each candidate precondition relationship
$\mathcal{C}$	the space of possible choices of C
v_i	a tuple $\langle x_i^f, x_i^t, \vec{w}_i, q_i \rangle$ designating a candidate precondition relationship between predicates x_i^f and x_i^t as indicated by sentence $\vec{w}_i$ with dependency parse q_i
V	a sequence of candidate precondition relationships, $\langle v_0, v_1, \cdots v_n \rangle$
θ_c	vector of parameters for precondition relation prediction
θ_x	vector of parameters for subgoal sequence prediction
θ	vector of all parameters, i.e. both θ_c and θ_x

Table 4.1: Notation used in defining our model.

the optimal parameters, θ^*, using a well-studied family of techniques called policy gradient algorithms.

4.5.1 Policy Gradient Algorithm

We choose to use a policy gradient algorithm for parameter estimation because such techniques scale effectively to large space spaces such as ours. These type of algorithms are only guaranteed to find a local optimum, but we find that in practice, they work quite well on our domain. They approximate the optimal policy parameters, θ^*, by performing stochastic gradient ascent on the value function, V_θ, such that in each iteration the algorithm computes a noisy estimate of the true gradient. To compute the gradients, we first take the derivative of the value function, V_θ with respect to the two different parameter vectors (see Appendix §B for details). With the notation as

defined in Table 4.1, this yields:

$$\frac{\partial}{\partial \theta_x} V_\theta = \mathbb{E}_{p(X,C|G,V;\theta)} \left[\sum_{i=0}^{m} \left(R_x(\vec{x}^i, i) + R_c(X, C) \right) \left[\sum_{j=0}^{|\vec{x}^i|} \phi_x(x^i_j, x^i_{j-1}, g_i, C) - \mathbb{E}_{p(x'|x^i_{j-1}, C;\theta_x)} \left[\phi_x(x', x^i_{j-1}, g_i, C) \right] \right] \right]$$

$$\frac{\partial}{\partial \theta_c} V_\theta = \mathbb{E}_{p(X,C|G,V;\theta)} \left[\sum_{i=0}^{n} \left(R_x(X) + R_c(X, c_i, i) \right) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c'|v_i) \right] \right] \right]$$

Exact computation of these derivatives is intractable since it requires summing over all possible choices of $\langle X, C \rangle$. Policy gradient algorithms instead compute a noisy estimate of the gradient using just a subset of the histories. So we will draw samples from $\langle X, C \rangle$ and compute the reward $R(X, C)$ by asking the low-level planner to compute a plan between each pair of subgoals in X. In practice, we will sample a single $\langle X, C \rangle$ in each iteration. Using this approximation, with learning rates α_x and α_c, the updates become:

$$\Delta\theta_x = \alpha_x \sum_{i=0}^{m} \left(R_x(\vec{x}^i, i) + R_c(X, C) \right) \left[\sum_{j=0}^{|\vec{x}^i|} \phi_x(x^i_j, x^i_{j-1}, g_i, C) - \mathbb{E}_{p(x'|x^i_{j-1}, g_i, C;\theta_x)} \left[\phi_x(x', x^i_{j-1}, g_i, C) \right] \right] \tag{4.2}$$

$$\Delta\theta_c = \alpha_c \sum_{i=0}^{n} \left(R_x(X) + R_c(X, c_i, i) \right) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i) \right] \right] \tag{4.3}$$

The full sampling and update procedure can be seen in Algorithm 4. In each iteration we make a single predication for each candidate relation, c, in the document, d, and predict a single subgoal sequence, $\vec{x}$ for each task in G. The resulting reward, $R(X, C)$, is then used to update both sets of parameters, θ_x and θ_c using the update equations given above. As the algorithm proceeds, the estimate of θ improves, leading to more useful samples of $R(X, C)$, which in turn leads to further improvements in the estimate of θ. We continue this process until convergence [125].

Input: A document d, Set of planning tasks G,
Set of candidate precondition relations V,
Reward function $R(X, C)$, Number of iterations T

Initialization: Model parameters $\theta_x = 0$ and $\theta_c = 0$.

for $i = 1 \cdots T$ **do**
  Sample precondition relations:
  $C \leftarrow \langle -1^n \rangle$
  for $i = 1 \cdots n$ **do**
    $c_i \sim p(c_i \mid v_i; \theta_c)$
  end
  Predict subgoal sequences for each task g.
  foreach $g \in G$ **do**
    Sample subgoal sequence $\vec{x}$ as follows:
    for $t = 1 \cdots n$ **do**
      Sample next subgoal:
      $x_t \sim p(x \mid x_{t-1}, g, C; \theta_x)$
      Construct low-level subtask from x_{t-1} to x_t
      Execute low-level planner on subtask
    end
    Update subgoal prediction model using Eqn. 4.2
  end
  Update text precondition model using Eqn. 4.3
end

Algorithm 4: A policy gradient algorithm for parameter estimation in our model.

4.5.2 Reward Function

Our reward function, $R(X, C)$ is computed using the set of predicted subgoal sequences, X, and the set of precondition predictions, C. The computation is based on feedback from the low-level planner, $f(x^f, x^t)$ indicating its ability to find a plan between subgoals, x^f and x^t. Formally:

$$f(x_j^i, x_k^i) = \begin{cases} 1 & \text{if the planner can successfully find a plan from subgoal} x_j^i \text{ to } x_k^i \text{ for task } i \\ 0 & \text{otherwise} \end{cases}$$

We would like to define R such that it correlates well with the correctness of our predictions. As discussed earlier, we factor $R(X, C)$ as follows:

$$R(X, C) = \sum_{i=0}^{m} R_x(\vec{x}^i, i) + \sum_{i=0}^{n} R_c(X, c_i, i)$$

where R_x gives reward for effective subgoal sequences, and R_c gives reward for precondition predictions which are likely to be correct.

Precondition Prediction Reward We define $R_c(X, c_i, i)$ independently for each precondition prediction, c_i, based on the consistency between c_i and the low-level planners success or failure in finding a plan between x_i^f and x_i^t. Formally, we define binary functions $S(X, i)$, indicating at least one success, and $F(X, i)$, indicating at least one failure, as follows:

$$S(X, i) = \begin{cases} 1 & \text{if } \sum_{j=0}^{m} \sum_{\substack{k=1 \\ x_{k-1}^j = x_i^f \\ x_k^j = x_i^t}}^{|\vec{x}^j|-1} f(x_{k-1}^j, x_k^j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$F(X,i) = \begin{cases} 1 & \text{if } \sum_{j=0}^{m} \sum_{\substack{k=1 \\ x_{k-1}^j = x_i^f \\ x_k^j = x_i^t}}^{|\vec{x}^j|-1} 1 - f(x_{k-1}^j, x_k^j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note, however, that success in finding a plan from x_i^f to x_i^t does not always mean that v_i is a good precondition relation, because subgoal x_i^f may be completely unnecessary, i.e. the planner may be able to successfully reach state x_i^t without first going through state x_i^f. Formally, we define a binary function $U(X,i)$, indicating subgoal x_i^f is unnecessary to reach subgoal x_i^t, as follows:

$$U(X,i) = \begin{cases} 1 & \text{if } \sum_{j=0}^{m} \sum_{\substack{k=1 \\ x_1^j = x_i^f \\ x_k^j = x_i^t}}^{|\vec{x}^j|-1} f(x_{k-1}^j, x_k^j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

We then define the reward, $R_c(X, c_i, i)$ based on whether or not the prediction c_i is consistent with the feedback from the planning successes and failures. Formally:

$$R_c(X, c_i, i) = \begin{cases} R_c^s \cdot c_i & \text{if } S(X,i) \text{ and not } U(X,i) \\ R_c^f \cdot c_i & \text{if } U(X,i) \text{ or } [F(X,i) \text{ and not } S(X,i)] \\ 0 & \text{otherwise } (x_i^f \text{ and } x_i^i \text{ do not occur as sequential states in } X) \end{cases}$$

where R_c^s and R_c^f are tunable reward parameters. We set R_c^s to a positive number and R_c^f to a negative number, such that multiplying by $c_i \in \{-1, 1\}$ generates positive reward if c_i agrees with the feedback, and negative reward if it disagrees.

Subgoal Sequence Reward We define $R_x(\vec{x}^i, i)$ based on the success of the entire sequence rather than basing it on independent rewards for each subgoal prediction, $x_j^i \in \vec{x}^i$, because if the low-level planner is unable to successfully generate a plan between two sequential subgoals, x_{j-1}^i and x_j^i, there are two possible reasons for

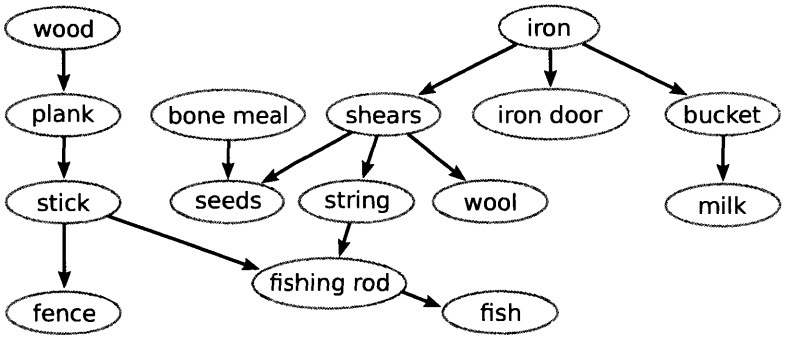


Figure 4-3: Example of the precondition dependencies present in the *Minecraft* domain.

this. The first is that subgoal x_j^i was simply a poor prediction, and replacing it with another choice would allow the planner to successfully generate plans for the entire sequence. The other possibility, however, is that some or all of the sequence leading up to x_{j-1}^i is incorrect, and there's simply no good choice for subgoal x_j^i. As a result of this ambiguity, we cannot know where in the subgoal sequence our predictions went wrong, and thus cannot provide reward based on individual subgoal selections. Instead, we calculate the reward based on the success or failure of the entire sequence. Formally:

$$R_x(\vec{x}^i, i) = \begin{cases} R_x^s & \text{if } f(s_0^{g_i}, x_0^i) f(x_{|\vec{x}^i|-1}^i, s_f^{g_i}) \prod_{j=1}^{|\vec{x}^i|} f(x_{j-1}^i, x_j^i) = 1 \\ R_x^f & \text{otherwise} \end{cases}$$

where R_x^s and R_x^f are tunable parameters.

4.6 Applying the Model

We apply our method to *Minecraft*, a grid-based virtual world. Each grid location represents a tile of either land or water and may also contain resources. Users can freely move around the world, harvest resources and craft various tools and objects from these resources. The dynamics of the world require certain resources or tools as prerequisites for performing a given action, as can be seen in Figure 4-3. For example,

Domain	#Objects	#Pred Types	#Actions
Parking	49	5	4
Floortile	61	10	7
Barman	40	15	12
Minecraft	**108**	**16**	**68**

Table 4.2: A comparison of complexity between *Minecraft* and some domains used in the IPC-2011 sequential satisficing track. In the *Minecraft* domain, the number of objects, predicate types, and actions is significantly larger.

a user must first craft a *bucket* before they can collect *milk*.

4.6.1 Defining the Domain

In order to execute a traditional planner on the *Minecraft* domain, we define the domain using the Planning Domain Definition Language (PDDL) [41]. This is the standard task definition language used in the International Planning Competitions (IPC).[4] We define as predicates all aspects of the game state – for example, the location of resources in the world, the resources and objects possessed by the player, and the player's location. Our subgoals x_i and our task goals s_f^g map directly to these predicates. This results in a domain with significantly greater complexity than those solvable by traditional low-level planners. Table 4.2 compares the complexity of our domain with some typical planning domains used in the IPC.

4.6.2 Low-level Planner

As our low-level planner we employ Metric-FF [56], the state-of-the-art baseline used in the 2008 International Planning Competition. Metric-FF is a forward-chaining heuristic state space planner. Its main heuristic is to simplify the task by ignoring operator delete lists. The number of actions in the solution for this simplified task is then used as the goal distance estimate for various search strategies.

Words
Dependency Types
Dependency Type × Direction
Word × Dependency Type
Word × Dependency Type × Direction

Table 4.3: Example text features. A subgoal pair $\langle x_i, x_j \rangle$ is first mapped to word tokens using a small grounding table. Words and dependencies are extracted along paths between mapped target words. These are combined with path directions to generate the text features.

4.6.3 Features

The two components of our model leverage different types of information, and as a result, they each use distinct sets of features. The text component features ϕ_c are computed over sentences and their dependency parses. The Stanford parser [34] was used to generate the dependency parse information for each sentence. Examples of these features appear in Table 4.3. The sequence prediction component takes as input both the preconditions induced by the text component as well as the planning state and the previous subgoal. Thus ϕ_x contains features which check whether two subgoals are connected via an induced precondition relation, in addition to features which are simply the Cartesian product of domain predicates.

4.7 Experimental Setup

4.7.1 Datasets

As the text description of our virtual world, we use documents from the *Minecraft* Wiki,[5] the most popular information source about the game. Our manually constructed seed grounding of predicates contains 74 entries, examples of which can be seen in Table 4.4. We use this seed grounding to identify a set of 242 sentences that reference predicates in the *Minecraft* domain. This results in a set of 694 *Candidate*

[4]http://ipc.icaps-conference.org/

[5]http://www.minecraftwiki.net/wiki/Minecraft_Wiki/

Domain Predicate	Noun Phrases
`have(plank)`	wooden plank, wood plank
`have(stone)`	stone, cobblestone
`have(iron)`	iron ingot

Table 4.4: Examples in our seed grounding table. Each predicate is mapped to one or more noun phrases that describe it in the text.

Relations. We also manually annotated the relations expressed in the text, identifying 94 of the *Candidate Relations* as valid. Our corpus contains 979 unique word types and is composed of sentences with an average length of 20 words.

We test our system on a set of 98 problems that involve collecting resources and constructing objects in the *Minecraft* domain – for example, fishing, cooking and making furniture. To assess the complexity of these tasks, we manually constructed high-level plans for these goals and solved them using the Metric-FF planner. On average, the execution of the sequence of low-level plans takes 35 actions, with 3 actions for the shortest plan and 123 actions for the longest. The average branching factor is 9.7, leading to an average search space of more than 10^{34} possible action sequences. For evaluation purposes we manually identify a set of *Gold Relations* consisting of all precondition relations that are valid in this domain, including those not discussed in the text.

4.7.2 Evaluation Metrics

We use our manual annotations to evaluate the type-level accuracy of relation extraction. To evaluate our high-level planner, we use the standard measure adopted by the IPC. This evaluation measure simply assesses whether the planner completes a task within a predefined time.

4.7.3 Baselines

To evaluate the performance of our relation extraction, we compare against an SVM classifier[6] trained on the *Gold Relations*. We test the SVM baseline in a leave-one-out

[6] SVM^{light} [64] with default parameters.

fashion.

To evaluate the performance of our text-aware high-level planner, we compare against five baselines. The first two baselines – *FF* and *No Text* – do not use any textual information. The *FF* baseline directly runs the Metric-FF planner on the given task, while the *No Text* baseline is a variant of our model that learns to plan in the reinforcement learning framework. It uses the same state-level features as our model, but does not have access to text.

The *All Text* baseline has access to the full set of 694 *Candidate Relations*. During learning, our full model refines this set of relations, while in contrast the *All Text* baseline always uses the full set.

The two remaining baselines constitute the upper bound on the performance of our model. The first, *Manual Text*, is a variant of our model which directly uses the links derived from manual annotations of preconditions in text. The second, *Gold*, has access to the *Gold Relations*. Note that the connections available to *Manual Text* are a subset of the *Gold* links, because the text does not specify all relations.

4.7.4 Experimental Details

All experimental results are averaged over 200 independent runs for both our model as well as the baselines. Each of these trials is run for 200 learning iterations with a maximum subgoal sequence length of 10. To find a low-level plan between each consecutive pair of subgoals, our high-level planner internally uses Metric-FF. We give Metric-FF a one-minute timeout to find such a low-level plan. To ensure that the comparison between the high-level planners and the FF baseline is fair, the FF baseline is allowed a runtime of 2,000 minutes. This is an upper bound on the time that our high-level planner can take over the 200 learning iterations, with subgoal sequences of length at most 10 and a one minute timeout. Lastly, during learning we initialize all parameters to zero, use a fixed learning rate of 0.0001, and encourage our model to explore the state space by using the standard ϵ-greedy exploration strategy [124].

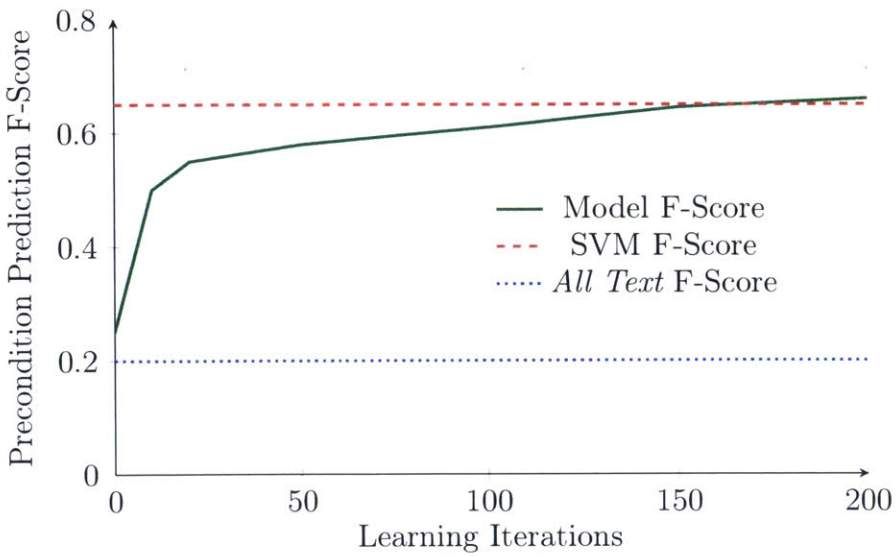


Figure 4-4: The performance of our model and a supervised SVM baseline on the precondition prediction task. Also shown is the F-Score of the full set of *Candidate Relations* which is used unmodified by *All Text*, and is given as input to our model. Our model's F-score, averaged over 200 trials, is shown with respect to learning iterations.

4.8 Results

4.8.1 Relation Extraction

Figure 4-4 shows the performance of our method on identifying preconditions in text. We also show the performance of the supervised SVM baseline. As can be seen, after 200 learning iterations, our model achieves an F-Measure of 66%, equal to the supervised baseline. These results support our hypothesis that planning feedback is a powerful source of supervision for analyzing a given text corpus. Figure 4-5 shows some examples of sentences and the corresponding extracted relations.

4.8.2 Planning Performance

As shown in Table 4.5 our text-enriched planning model outperforms the text-free baselines by more than 10%. Moreover, the performance improvement of our model over the *All Text* baseline demonstrates that the accuracy of the extracted text rela-

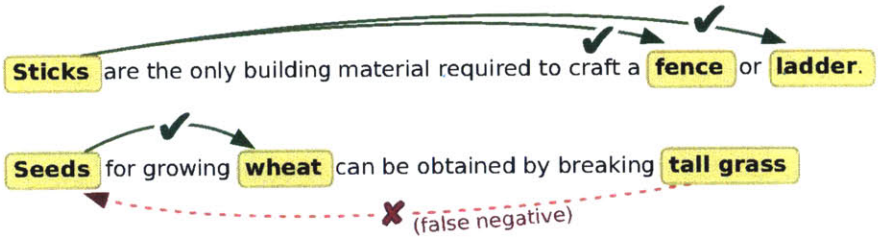


Figure 4-5: Examples of precondition relations predicted by our model from text. Check marks (✓) indicate correct predictions, while a cross (✘) marks the incorrect one – in this case, a valid relation that was predicted as invalid by our model. Note that each pair of highlighted noun phrases in a sentence is a *Candidate Relation*, and pairs that are not connected by an arrow were correctly predicted to be invalid by our model.

Method	%Plans
FF	40.8
No text	69.4
All text	75.5
Full model	**80.2**
Manual text	84.7
Gold connection	87.1

Table 4.5: Percentage of tasks solved successfully by our model and the baselines. All performance differences between methods are statistically significant at $p \leq .01$.

tions does indeed impact planning performance. A similar conclusion can be reached by comparing the performance of our model and the *Manual Text* baseline.

The difference in performance of 2.35% between *Manual Text* and *Gold* shows the importance of the precondition information that is missing from the text. Note that *Gold* itself does not complete all tasks – this is largely because the Markov assumption made by our model does not hold for all tasks.[7]

Figure 4-6 breaks down the results based on the difficulty of the corresponding planning task. We measure problem complexity in terms of the low-level steps needed to implement a manually constructed high-level plan. Based on this measure, we divide the problems into two sets. As can be seen, all of the high-level planners solve

[7]When a given task has two non-trivial preconditions, our model will choose to satisfy one of the two first, and the Markov assumption blinds it to the remaining precondition, preventing it from determining that it must still satisfy the other.

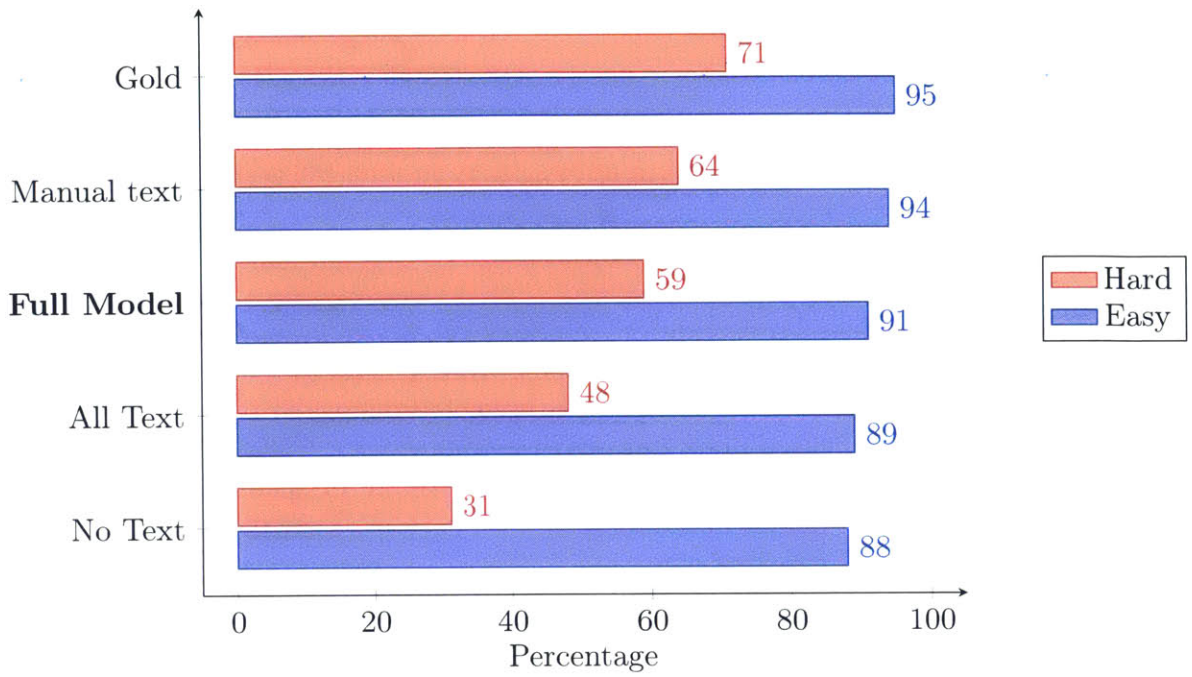


Figure 4-6: Percentage of problems solved by various models on Easy and Hard problem sets.

almost all of the easy problems. However, performance varies greatly on the more challenging tasks, directly correlating with planner sophistication. On these tasks our model outperforms the *No Text* baseline by 28% and the *All Text* baseline by 11%.

4.8.3 Feature Analysis

Figure 4-7 shows the top five positive features for our model and the SVM baseline. Both models picked up on the words that indicate precondition relations in this domain. For instance, the word *use* often occurs in sentences that describe the resources required to make an object, such as "bricks are items used to craft brick blocks". In addition to lexical features, dependency information is also given high weight by both learners. An example of this is a feature that checks for the direct object dependency type. This analysis is consistent with prior work on event semantics which shows lexico-syntactic features are effective cues for learning text relations [13, 10, 37].

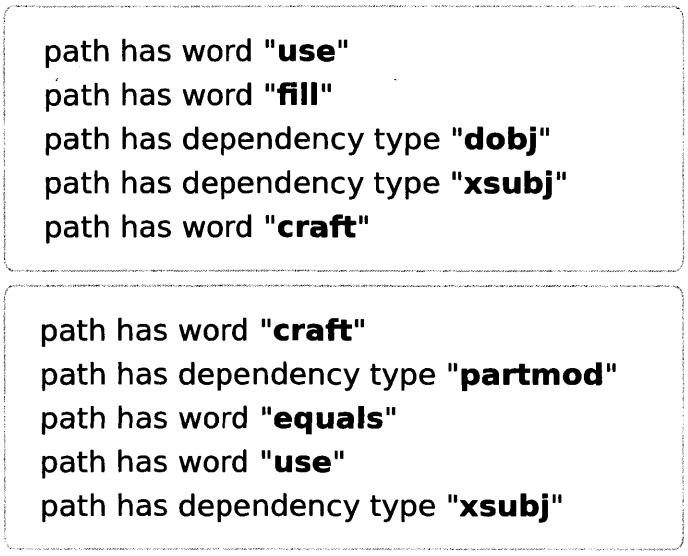


Figure 4-7: The top five positive features on words and dependency types learned by our model (above) and by SVM (below) for precondition prediction.

4.9 Conclusions

In this chapter, we presented a novel technique for inducing precondition relations from text by grounding them in the semantics of planning operations. While using planning feedback as its only source of supervision, our method for relation extraction achieves a performance on par with that of a supervised baseline. Furthermore, relation grounding provides a new view on classical planning problems which enables us to create high-level plans based on language abstractions. We show that building high-level plans in this manner significantly outperforms traditional techniques in terms of task completion.

Chapter 5

Conclusions and Future Work

In this thesis, we have introduced systems and techniques for learning to translate natural language into preexisting programming languages which are supported by widely-deployed computer systems. Our work is a departure from most past work which has focused on translating to special-purpose programming languages designed to align closely to natural language. Mapping to existing programming languages presents significant challenges stemming from the complicated relationship between the natural language and the resulting programs, however it allows us to utilize the underlying computer systems during the learning process itself. We show that integrating the capabilities of the underlying systems leads to substantially more effective learning.

We focus on two capabilities: computing semantic equivalence between programs, and executing programs to obtain a result. In Chapter 2, we utilized the semantic equivalence capabilities of the underlying system in order to handle the fact that samples in the training data often do not exhibit syntactic alignment between the natural language and the associated programs. In such cases we leveraged the underlying platform to help find a semantically equivalent program which *does* syntactically align with the natural language. In Chapter 3, we showed that this capability could be further utilized to constrain the search space of our joint inference technique. By discovering the semantic equivalences between syntactically different program templates, we were able to eliminate redundancy in the inference process, and improve

the statistical efficiency of the learning.

In Chapter 4, we utilized the execution capabilities of the underlying platform in order to learn to interpret text without labeled training data. We use our current interpretation of the text to generate a high-level program to perform a given task. We feed this program to the underlying system to generate a detailed low-level program for the task. The system's success in generating such a low-level program is used as the basis for a reinforcement learning procedure. This procedure is used to learn, without any labeled data, the parameters of a Markov Decision Process (MDP) which both interprets the text and uses the interpretation to generate high-level programs. In Chapter 3 we further utilized the ability to execute programs to learn more effectively. We executed the generated programs, and use the results of the execution, in this case the solution to a system of mathematical equations as additional features in a log-linear model.

5.1 Limitations

One of the main limitations of our work comes from the data itself. Both the system for generating regular expressions, and the system for solving algebra problems relied on supervised data. In each case the datasets only cover a small subset of the natural language inputs possible even within these two constrained domains. For example, many real-world regular expressions are written to query data which is structured in very specific ways. Our system cannot handle such queries without training examples containing such queries. Additionally, many algebra word problems require background knowledge to solve. For example, we may need to know that *profit* is the difference between *sales* and *costs*. Since our system does not directly utilize any background knowledge, we can only handle such problems if we have previously seen similar problems in the training data.

Our work on algebra word problems utilized a flat model which did not directly model compositional language. Given the typical shallow nesting in the data, it is possible to learn templates for these cases given enough data, but a system which

explicitly models this compositionality may exhibit better generalization from smaller datasets.

Finally, our model for generating high-level plans utilized a Markov model which makes predictions for the current subgoal considering only the previous subgoal, as well as the start and end states. This constraint on the model makes it challenging to handle subgoals which have multiple preconditions to predict.

5.2 Future Work

This thesis makes steps towards the long term goal of learning to translate unconstrained natural language into a fully general-purpose Turing-complete programming language. However, much future work remains. Fully realizing this vision requires solving hard problems beyond those discussed in this thesis, such as learning and representing common-sense knowledge. There are however multiple possible next steps which are direct extensions of the ideas discussed here.

- **Extending Semantic Equivalence to Turing-complete Languages**

  We showed the use of semantic equivalence in the limited context of regular expressions and math equations. Working in constrained domains enabled the efficient calculation of semantic equivalence. When extending this idea to Turing-complete languages, exact semantic equivalence calculations will no longer be computationally tractable, requiring the consideration of approximate techniques. However, SAT and SMT solvers have seen significant success in performing semantic inference for program induction and hardware verification despite the computational intractability of these problems in the general case. Exploring their use in our context would be an intriguing direction for future research.

- **Utilizing additional semantic inference capabilities**

  This thesis focused on utilizing both the execution and semantic equivalence capabilities of the underlying platform. Computer programming systems, how-

ever, make available a wide variety of other capabilities which we have not exploited. One particularly intriguing capability is the abstract interpretation techniques that have been used quite successfully in the programming languages and compiler communities [33]. These techniques can be used to prove that two programs share various semantic properties beyond strict semantic equality. For example, they could be used to show that two programs will generate the same output on a constrained set of inputs, rather than for all inputs. Given that natural language is often ambiguous and context specific, integrating such additional properties into the learning process could prove to be very fruitful.

- **Probabilistically Modeling Programs**

  Long term, one of the fundamental challenges in learning to generate programs from natural language is the limited amount of labeled training data. This challenge becomes even greater as we move to general purpose programming languages which enable many more possible implementations of the same basic idea. Unlabeled data can be easily obtained, however. Specifically, a significant advantage of working with off-the-shelf programming languages is that large open-source computer program repositories provide us easy access to almost unbounded volumes of code in many such languages. While we cannot use this data to directly learn the mapping between natural language and code, we can use it to learn a probabilistic model of the higher-level abstract structures present within typical computer programs. The resulting higher-level structures can then be used as candidates for natural language groundings, enabling us to learn more complex mappings with significantly less labeled data.

Appendix A

Generating Regular Expressions

Domain	**GeoQuery**	
Natural Language	What is the highest mountain in Alaska?	
Logical Program	`(answer (highest (mountain (loc_2 (stateid alaska:e)))))`	
Alignment	What is	`answer`
	the highest	`highest`
	moutain	`mountain`
	in	`loc_2`
	Alaska	`Alaska`

Domain	**SAIL**	
Natural Language	move forward twice to the chair	
Logical Program	$\lambda a.move(a) \wedge dir(a, forward) \wedge len(a, 2) \wedge to(a, ix.chair(x))$	
Alignment	move	$move(a)$
	forward	$dir(a, forward)$
	twice	$len(a, 2)$
	to	$to(y, x)$
	the chair	$ix.chair(x)$

Domain	**Freebase**	
Natural Language	What college did Obama go to?	
Logical Program	`Type.University` $\sqcap$ `Education.BarakObama`	
Alignment	college	`Type.University` $\sqcap$ `Education`
	Obama	`BarakObama`

Domain	**RoboCup**	
Natural Language	Purple goalie turns the ball over to Pink8	
Logical Program	`turnover(PurplePlayer1, PinkPlayer8)`	
Alignment	Purple goalie	`PurplePlayer1`
	turns the ball over to	`turnover(x,y)`
	Pink8	`PinkPlayer8`

Domain	**ATIS**	
Natural Language	On May 4th Atlanta to Denver Delta flight 257	
Logical Program	$\lambda x.month(x, may) \wedge day\_number(x, fourth) \wedge from(x, atlanta) \wedge$ $to(x, denver) \wedge airline(x, delta\_air\_lines) \wedge flight(x) \wedge$ $flight\_number(x, 257)$	
Alignment	May	$month(x, may)$
	4th	$day\_number(x, fourth)$
	Atlanda	$atlanta$
	to	$from(x, y) \wedge to(x, z)$
	Denver	$denver$
	Delta	$delta\_air\_lines$
	flight	$flight(x)$
	257	$flight\_number(x, 257)$

Figure A-1: Examples of four different domains considered by the past work. In each case, the natural language is mapped to a logical programming language which was specifically designed to syntactically align with natural language.

Appendix B

Learning to Generate Plans from Text

B.1 Lemmas

This section proves a few lemmas that are used in the succeeding derivations.

Lemma B.1.1.

$$\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)\sum_{i=0}^{n} Q(X,c_i,i) = \sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{c_i\in\{-1,1\}} Q(X,c_i,i)p(X|c_i,G,V;\theta)p(c_i|v_i;\theta_c)$$

Proof.

$$\begin{aligned}
&\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)\sum_{i=0}^{n} Q(X,c_i,i) = \\
&\quad\sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{C\in\mathcal{C}} Q(X,c_i,i)p(X|C,G;\theta)\prod_{j=0}^{n} p(c_j|v_j;\theta_c) \\
&= \sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{c_i'\in\{-1,1\}} Q(X,c_i',i)\sum_{\{C\in\mathcal{C}|c_i=c_i'\}} p(X|C,G;\theta)\prod_{j=0}^{n} p(c_j|v_j;\theta_c) \\
&= \sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{c'\in\{-1,1\}} Q(X,c',i)p(c_i=c'|v_i;\theta_c)\sum_{\{C\in\mathcal{C}|c_i=c'\}} p(X|C,G;\theta)\prod_{\substack{j=0\\ j\neq i}}^{n} p(c_j|v_j;\theta_c) \\
&= \sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{c'\in\{-1,1\}} Q(X,c',i)p(c_i=c'|v_i;\theta_c)p(X|c_i=c',G,V;\theta)
\end{aligned}$$

Slightly simplifying the notation results in the summation in the lemma:

$$= \sum_{X \in \mathcal{X}} \sum_{i=0}^{n} \sum_{c_i \in \{-1,1\}} Q(X, c_i, i) p(X|c_i, G, V; \theta) p(c_i|v_i; \theta_c)$$

□

Lemma B.1.2.

$$\sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) \sum_{i=0}^{m} Q(\vec{x}^i, i) = \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{\vec{x} \in \vec{\mathbf{x}}} Q(\vec{x}^i, i) p(\vec{x}^i|g_i, C; \theta_x)$$

Proof.

$$\sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) \sum_{i=0}^{m} Q(\vec{x}^i, i) =$$

$$\sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{X \in \mathcal{X}} p(X|C, G; \theta_x) \sum_{i=0}^{m} Q(\vec{x}^i, i)$$

$$= \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{X \in \mathcal{X}} p(X|C, G; \theta_x) Q(\vec{x}^i, i)$$

$$= \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{\vec{x}' \in \vec{\mathbf{x}}} \sum_{\{X \in \mathcal{X} | \vec{x}^i = \vec{x}'\}} p(X|C, G; \theta_x) Q(\vec{x}', i)$$

$$= \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{\vec{x}' \in \vec{\mathbf{x}}} Q(\vec{x}', i) \sum_{\{X \in \mathcal{X} | \vec{x}^i = \vec{x}'\}} p(X|C, G; \theta_x)$$

$$= \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{\vec{x}' \in \vec{\mathbf{x}}} Q(\vec{x}', i) p(\vec{x}'|g_i, C; \theta_x) \sum_{\{X \in \mathcal{X} | \vec{x}^i = \vec{x}'\}} p(X|\vec{x}^i = \vec{x}', C, G; \theta_x)$$

Since the last summation equals 1, we can drop it

$$= \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{\vec{x}' \in \vec{\mathbf{x}}} Q(\vec{x}', i) p(\vec{x}'|g_i, C; \theta_x)$$

Slightly simplifying the notation results in the summation in the lemma:

$$= \sum_{C \in \mathcal{C}} p(C|V; \theta_c) \sum_{i=0}^{m} \sum_{\vec{x} \in \vec{\mathbf{x}}} Q(\vec{x}, i) p(\vec{x}|g_i, C; \theta_x)$$

□

Lemma B.1.3.

$$\frac{\partial}{\partial \theta_x} p(\vec{x}|g, C; \theta_x) = p(\vec{x}|g, C; \theta_x) \left[\sum_{i=0}^{|\vec{x}|} \phi_x(x_i, x_{i-1}, g, C) - \mathbb{E}_{p(x'|x_{i-1}, g, C; \theta_x)} \left[\phi_x(x', x_{i-1}, g, C) \right] \right]$$

Proof.

$$\frac{\partial}{\partial \theta_x} p(\vec{x}|g, C; \theta_x) = \frac{\partial}{\partial \theta_x} \prod_{i=0}^{|\vec{x}|} p(x_i|x_{i-1}, g, C; \theta_x)$$

Using the derivative product rule:

$$= \sum_{i=0}^{|\vec{x}|} \left[\left(\frac{\partial}{\partial \theta_x} p(x_i|x_{i-1}, g, C; \theta_x) \right) \prod_{\substack{j=0 \\ j \neq i}}^{|\vec{x}|} p(x_j|x_{j-1}, g, C; \theta_x) \right]$$

$$= \sum_{i=0}^{|\vec{x}|} \left[\frac{\frac{\partial}{\partial \theta_x} p(x_i|x_{i-1}, g, C; \theta_x)}{p(x_i|x_{i-1}, g, C; \theta_x)} \prod_{j=0}^{|\vec{x}|} p(x_j|x_{j-1}, g, C; \theta_x) \right]$$

$$= p(\vec{x}|g, C; \theta_x) \sum_{i=0}^{|\vec{x}|} \left[\frac{\frac{\partial}{\partial \theta_x} p(x_i|x_{i-1}, g, C; \theta_x)}{p(x_i|x_{i-1}, g, C; \theta_x)} \right]$$

$$= p(\vec{x}|g, C; \theta_x) \sum_{i=0}^{|\vec{x}|} \frac{\partial}{\partial \theta_x} \log p(x_i|x_{i-1}, g, C; \theta_x)$$

To compute the inner derivative, we note that we have defined $p(x_i|x_{i-1}, g, C; \theta_x)$ as a log-linear distribution. Thus:

$$p(x_i|x_{i-1}, g, C; \theta_x) = \frac{e^{\theta_x \cdot \phi_x(x_i, x_{i-1}, g, C)}}{\sum_{x' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x', x_{i-1}, g, C)}}$$

$$\log p(x_i|x_{i-1}, g, C; \theta_x) = \theta_x \cdot \phi_x(x_i, x_{i-1}, g, C) - \log \sum_{x' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x_i, x_{i-1}, g, C)}$$

Therefore:

$$\frac{\partial}{\partial \theta_x} \log p(x_i|x_{i-1}, g, C; \theta_x) = \frac{\partial}{\partial \theta_x} \left[\theta_x \cdot \phi_x(x_i, x_{i-1}, g, C) - \log \sum_{x' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x_i, x_{i-1}, g, C)} \right]$$

$$= \phi_x(x_i, x_{i-1}, g, C) - \left(\frac{\frac{\partial}{\partial \theta_x} \sum_{x' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x', x_{i-1}, g, C)}}{\sum_{x'' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x'', x_{i-1}, g, C)}} \right)$$

$$= \phi_x(x_i, x_{i-1}, g, C) - \left(\frac{\sum_{x' \in \mathbf{x}} \frac{\partial}{\partial \theta_x} e^{\theta_x \cdot \phi_x(x', x_{i-1}, g, C)}}{\sum_{x'' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x'', x_{i-1}, g, C)}} \right)$$

$$= \phi_x(x_i, x_{i-1}, g, C) - \left(\frac{\sum_{x' \in \mathbf{x}} \phi_x(x', x_{i-1}, g, C) e^{\theta_x \cdot \phi_x(x', x_{i-1}, g, C)}}{\sum_{x'' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x'', x_{i-1}, g, C)}} \right)$$

$$= \phi_x(x_i, x_{i-1}, g, C) - \sum_{x' \in \mathbf{x}} \phi_x(x', x_{i-1}, g, C) \left(\frac{e^{\theta_x \cdot \phi_x(x', x_{i-1}, g, C)}}{\sum_{x'' \in \mathbf{x}} e^{\theta_x \cdot \phi_x(x'', x_{i-1}, g, C)}} \right)$$

$$= \phi_x(x_i, x_{i-1}, g, C) - \sum_{x' \in \mathbf{x}} \phi_x(x', x_{i-1}, g, C) p(x' | x_{i-1}, g, C)$$

$$= \phi_x(x_i, x_{i-1}, g, C) - \mathbb{E}_{p(x' | x_{i-1}, g, C; \theta_x)} \left[\phi_x(x', x_{i-1}, g, C) \right]$$

Plugging this back into the earlier expression yields the result:

$$= p(\vec{x} | g, C; \theta_x) \left[\sum_{i=0}^{|\vec{x}|} \phi_x(x_i, x_{i-1}, g, C) - \mathbb{E}_{p(x' | x_{i-1}, g, C; \theta_x)} \left[\phi_x(x', x_{i-1}, g, C) \right] \right]$$

□

Lemma B.1.4.

$$\frac{\partial}{\partial \theta} p(c | \theta) = p(c | \theta) \left(\phi(c) - \mathbb{E}_{p(c')} \left[\phi(c') \right] \right)$$

Proof.

$$\frac{\partial}{\partial \theta} p(c | \theta) = \frac{\partial}{\partial \theta} \frac{e^{\theta \cdot \phi(c)}}{\sum_{c' \in \{-1, 1\}} e^{\theta \cdot \phi(c')}}$$

Applying the quotient rule:

$$= \frac{\frac{\partial}{\partial \theta} \left(e^{\theta \cdot \phi(c)} \right) \sum_{c' \in \{-1, 1\}} e^{\theta \cdot \phi(c')} - e^{\theta \cdot \phi(c)} \frac{\partial}{\partial \theta} \left(\sum_{c' \in \{-1, 1\}} e^{\theta \cdot \phi(c')} \right)}{\left(\sum_{c' \in \{-1, 1\}} e^{\theta \cdot \phi(c')} \right)^2}$$

$$
\begin{aligned}
&= \frac{\left(e^{\theta\cdot\phi(c)}\phi(c)\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}\right) - \left(e^{\theta\cdot\phi(c)}\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}\phi(c')\right)}{(\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')})^2} \\
&= \left(\frac{e^{\theta\cdot\phi(c)}}{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}}\right)\left(\frac{\left(\phi(c)\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}\right) - \left(\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}\phi(c')\right)}{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}}\right) \\
&= p(c|\theta)\left(\frac{\phi(c)\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}}{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}} - \frac{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}\phi(c')}{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}}\right) \\
&= p(c|\theta)\left(\phi(c) - \frac{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}\phi(c')}{\sum_{c'\in\{-1,1\}} e^{\theta\cdot\phi(c')}}\right) \\
&= p(c|\theta)\left(\phi(c) - \sum_{c'\in\{-1,1\}} \frac{e^{\theta\cdot\phi(c')}}{\sum_{c''\in\{-1,1\}} e^{\theta\cdot\phi(c'')}}\phi(c')\right) \\
&= p(c|\theta)\left(\phi(c) - \sum_{c'\in\{-1,1\}} p(c')\phi(c')\right) \\
&= p(c|\theta)\left(\phi(c) - \mathbb{E}_{p(c')}\left[\phi(c')\right]\right)
\end{aligned}
$$

□

B.2 Derivation of update for θ_x

$$
\begin{aligned}
&\frac{\partial}{\partial\theta_x}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R(X,C)\right] = \frac{\partial}{\partial\theta_x}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_x(X) + R_c(X,C)\right] \\
&= \frac{\partial}{\partial\theta_x}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_x(X)\right] + \frac{\partial}{\partial\theta_x}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_c(X,C)\right]
\end{aligned}
$$

We'll start by computing the first derivative:

$$
\begin{aligned}
&\frac{\partial}{\partial\theta_x}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_x(X)\right] = \frac{\partial}{\partial\theta_x}\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)R_x(X) \\
&= \frac{\partial}{\partial\theta_x}\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)\sum_{i=0}^{m} R_x(\vec{x}^i,i)
\end{aligned}
$$

Applying Lemma B.1.2:

$$= \frac{\partial}{\partial \theta_x} \sum_{C \in \mathcal{C}} p(C|V;\theta_c) \sum_{i=0}^{m} \sum_{\vec{x} \in \vec{\mathbf{x}}} R_x(\vec{x}^i, i) p(\vec{x}^i | g_i, C; \theta_x)$$

$$= \sum_{C \in \mathcal{C}} p(C|V;\theta_c) \sum_{i=0}^{m} \sum_{\vec{x} \in \vec{\mathbf{x}}} R_x(\vec{x}^i, i) \frac{\partial}{\partial \theta_x} p(\vec{x}^i | g_i, C; \theta_x)$$

Applying Lemma B.1.3:

$$= \sum_{C \in \mathcal{C}} p(C|V;\theta_c) \sum_{i=0}^{m} \sum_{\vec{x} \in \vec{\mathbf{x}}} R_x(\vec{x}^i, i) p(\vec{x}^i | g_i, C; \theta_x) \left[\sum_{j=0}^{|\vec{x}^i|} \phi_x(x_j^i, x_{j-1}^i, g_i, C) - \mathbb{E}_{p(x'|x_{j-1}^i, g_i, C; \theta_x)} \left[\phi_x(x', x_{j-1}^i, g_i, C) \right] \right]$$

Applying Lemma B.1.2 in reverse:

$$\frac{\partial}{\partial \theta_x} \mathbb{E}_{p(X,C|G,V;\theta)} \left[R_x(X) \right] =$$

$$\sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) \sum_{i=0}^{m} R_x(\vec{x}^i, i) \left[\sum_{j=0}^{|\vec{x}^i|} \phi_x(x_j^i, x_{j-1}^i, g_i, C) - \mathbb{E}_{p(x'|x_{j-1}^i, g_i, C; \theta_x)} \left[\phi_x(x', x_{j-1}^i, g_i, C) \right] \right]$$

We compute the derivative of the second term similarly:

$$\frac{\partial}{\partial \theta_x} \mathbb{E}_{p(X,C|G,V;\theta)} \left[R_c(X, C) \right] = \frac{\partial}{\partial \theta_x} \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) R_c(X, C)$$

$$= \frac{\partial}{\partial \theta_x} \sum_{C \in \mathcal{C}} p(C|V;\theta_c) \sum_{X \in \mathcal{X}} R_c(X, C) p(X|C, G; \theta_x)$$

$$= \sum_{C \in \mathcal{C}} p(C|V;\theta_c) \sum_{X \in \mathcal{X}} R_c(X, C) \frac{\partial}{\partial \theta_x} p(X|C, G; \theta_x)$$

Applying a slightly modified version of Lemma B.1.3:

$$= \sum_{C \in \mathcal{C}} p(C|V;\theta_c) \sum_{X \in \mathcal{X}} R_c(X, C) p(X|C, G; \theta_x) \left[\sum_{i=0}^{m} \sum_{j=0}^{|\vec{x}^i|} \phi_x(x_j^i, x_{j-1}^i, g_i, C) - \mathbb{E}_{p(x'|x_{j-1}^i, g_i, C; \theta_x)} \left[\phi_x(x', x_{j-1}^i, g_i, C) \right] \right]$$

$$= \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) R_c(X, C) \left[\sum_{i=0}^{m} \sum_{j=0}^{|\vec{x}^i|} \phi_x(x_j^i, x_{j-1}^i, g_i, C) - \mathbb{E}_{p(x'|x_{j-1}^i, g_i, C; \theta_x)} \left[\phi_x(x', x_{j-1}^i, g_i, C) \right] \right]$$

Combining together the two derivatives:

$$\frac{\partial}{\partial \theta_x} \mathbb{E}_{p(X,C|G,V;\theta)} \left[R(X, C) \right] =$$

$$\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)\sum_{i=0}^{m} R_x(\vec{x}^i,i)\left[\sum_{j=0}^{|\vec{x}^i|}\phi_x(x^i_j,x^i_{j-1},g_i,C)-\mathbb{E}_{p(x'|x^i_{j-1},g_i,C;\theta_x)}\left[\phi_x(x',x^i_{j-1},g_i,C)\right]\right]+$$
$$\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)R_c(X,C)\left[\sum_{i=0}^{m}\sum_{j=0}^{|\vec{x}^i|}\phi_x(\vec{x}^i_j,x^i_{j-1},g_i,C)-\mathbb{E}_{p(x'|x^i_{j-1},g_i,C;\theta_x)}\left[\phi_x(x',x^i_{j-1},g_i,C)\right]\right]$$
$$=\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)\sum_{i=0}^{m}\left(R_x(\vec{x}^i,i)+R_c(X,C)\right)\left[\sum_{j=0}^{|\vec{x}^i|}\phi_x(x^i_j,x^i_{j-1},g_i,C)-\mathbb{E}_{p(x'|x^i_{j-1},g_i,C;\theta_x)}\left[\phi_x(x',x^i_{j-1},g_i,C)\right]\right]$$
$$=\mathbb{E}_{p(X,C|G,V;\theta)}\left[\sum_{i=0}^{m}\left(R_x(\vec{x}^i,i)+R_c(X,C)\right)\left[\sum_{j=0}^{|\vec{x}^i|}\phi_x(x^i_j,x^i_{j-1},g_i,C)-\mathbb{E}_{p(x'|x^i_{j-1},g_i,C;\theta_x)}\left[\phi_x(x',x^i_{j-1},g_i,C)\right]\right]\right]$$

So then if we sample an X and a C the approximation of the derivative becomes:

$$\frac{\partial}{\partial\theta_x}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R(X,C)\right]=$$
$$\sum_{i=0}^{m}\left(R_x(\vec{x}^i,i)+R_c(X,C)\right)\left[\sum_{j=0}^{|\vec{x}^i|}\phi_x(x^i_j,x^i_{j-1},g_i,C)-\mathbb{E}_{p(x'|x^i_{j-1},g_i,C;\theta_x)}\left[\phi_x(x',x^i_{j-1},g_i,C)\right]\right]$$

B.3 Derivation of update for θ_c

The derivative for θ_c follows very similarly to the derivative for θ_x:

$$\frac{\partial}{\partial\theta_c}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R(X,C)\right]=\frac{\partial}{\partial\theta_c}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_x(X)\right]+\frac{\partial}{\partial\theta_c}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_c(X,C)\right]$$

We'll start by computing the derivative of the second term:

$$\frac{\partial}{\partial\theta_c}\mathbb{E}_{p(X,C|G,V;\theta)}\left[R_c(X,C)\right]=\frac{\partial}{\partial\theta_c}\sum_{C\in\mathcal{C}}\sum_{X\in\mathcal{X}} p(X,C|G,V;\theta)\sum_{i=0}^{n}R(X,c_i,i)$$

Applying Lemma B.1.1:

$$=\frac{\partial}{\partial\theta_c}\sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{c_i\in\{-1,1\}} R(X,c_i,i)p(X|c_i,G,V;\theta)p(c_i|v_i;\theta_c)$$

To simplify the update calculation, we do not backpropagate the effect of changes to $p(X|\cdot)$ into the θ_c updates, and thus do not include it in the derivative.

$$=\sum_{X\in\mathcal{X}}\sum_{i=0}^{n}\sum_{c_i\in\{-1,1\}} R(X,c_i,i)p(X|c_i,G,V;\theta)\frac{\partial}{\partial\theta_c}p(c_i|v_i;\theta_c)$$

Applying Lemma B.1.4:

$$= \sum_{X \in \mathcal{X}} \sum_{i=0}^{n} \sum_{c_i \in \{-1,1\}} R(X, c_i, i) p(X|c_i, G, V; \theta) p(c_i|v_i; \theta_c) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right]$$

Applying Lemma B.1.1 in reverse:

$$= \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) \sum_{i=0}^{n} R(X, c_i, i) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right]$$

We compute the derivative of the first term similarly:

$$\frac{\partial}{\partial \theta_c} \mathbb{E}_{p(X,C|G,V;\theta)} \left[R_x(X)\right] = \frac{\partial}{\partial \theta_c} \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) R_x(X)$$

$$= \frac{\partial}{\partial \theta_c} \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X|C, G; \theta_x) R_x(X) p(C|V; \theta_c)$$

$$= \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X|C, G; \theta_x) R_x(X) \frac{\partial}{\partial \theta_c} p(C|V; \theta_c)$$

applying a variant of Lemma B.1.3:

$$= \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X|C, G; \theta_x) R_x(X) p(C|V; \theta_c) \left[\sum_{i=0}^{n} \phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right]$$

$$= \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) R_x(X) \left[\sum_{i=0}^{n} \phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right]$$

Combining together the two derivatives yeilds:

$$\frac{\partial}{\partial \theta_c} \mathbb{E}_{p(X,C|G,V;\theta)} \left[R(X, C)\right] =$$

$$\sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) R_x(X) \left[\sum_{i=0}^{n} \phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right] +$$

$$\sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) \sum_{i=0}^{n} R(X, c_i, i) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right]$$

$$= \sum_{C \in \mathcal{C}} \sum_{X \in \mathcal{X}} p(X, C|G, V; \theta) \sum_{i=0}^{n} \left(R_x(X) + R_c(X, c_i, i)\right) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta_c)} \left[\phi_c(c', v_i)\right]\right]$$

$$= \mathbb{E}_{p(X,C|G,V;\theta_c)} \left[\sum_{i=0}^{n} \left(R_x(X) + R_c(X, c_i, i) \right) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i;\theta)} \left[\phi_c(c', v_i) \right] \right] \right]$$

So then if we sample an X and a C the approximation of the derivative becomes:

$$\frac{\partial}{\partial \theta_c} \mathbb{E}_{p(X,C|G,V;\theta)} \left[R(X, C) \right] = \sum_{i=0}^{n} \left(R_x(X) + R_c(X, c_i, i) \right) \left[\phi_c(c_i, v_i) - \mathbb{E}_{p(c'|v_i)} \left[\phi_c(c', v_i) \right] \right]$$

Bibliography

[1] Peter Abeles. Efficient java matrix library, 2014. `https://code.google.com/p/efficient -java-matrix-library/`.

[2] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 2123–2132, 2015.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[4] Yoav Artzi and Luke Zettlemoyer. Bootstrapping semantic parsers from conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 421–432. Association for Computational Linguistics, 2011.

[5] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 2013.

[6] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intell.*, 71(1):43–100, 1994.

[7] Bruce W Ballard and Alan W Biermann. Programming in natural language:"NLC" as a prototype. In *Proceedings of the 1979 annual conference*, pages 228–237. ACM, 1979.

[8] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation (amr) 1.2 specification.

[9] Jennifer L. Barry, Leslie Pack Kaelbling, and Tomas Lozano-Perez. DetH*: Approximate hierarchical solution of large markov decision processes. In *IJCAI'11*, pages 1928–1935, 2011.

[10] Brandon Beamer and Roxana Girju. Using a bigram event model to predict causal potential. In *Proceedings of CICLing*, pages 430–441, 2009.

[11] Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *Proceedings of ACL*, volume 7, page 92, 2014.

[12] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2013.

[13] Eduardo Blanco, Nuria Castell, and Dan Moldovan. Causal relation extraction. In *Proceedings of the LREC'08*, 2008.

[14] Mario Bollini, Jennifer Barry, and Daniela Rus. Bakebot: Baking cookies with the pr2. In *The PR2 workshop: results, challenges and lessons learned in advancing robots with a common platform, IROS*, 2011.

[15] S. R. K. Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a monte-carlo framework. In *Proceedings of ACL*, pages 268–277, 2011.

[16] S.R.K Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of ACL*, pages 82–90, 2009.

[17] S.R.K Branavan, Luke Zettlemoyer, and Regina Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *Proceedings of ACL*, pages 1268–1277, 2010.

[18] S.R.K. Branavan, David Silver, and Regina Barzilay. Non-linear monte-carlo search in civilization ii. In *Proceedings of IJCAI*, 2011.

[19] S.R.K. Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. Learning high-level planning from text. In *Proceedings of ACL*, 2012.

[20] S.R.K. Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a monte-carlo framework. *Journal of Artificial Intelligence Research*, 43:661–704, 2012.

[21] Peter F Brown, Vincent J Della Pietra, Stephen A Della Pietra, and Robert L Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

[22] Qingqing Cai and Alexander Yates. Semantic parsing freebase: Towards open-domain semantic parsing. In *Proceedings of the Joint Conference on Lexical and Computational Semantics.*, 2013.

[23] Qingqing Cai and Alexander Yates. Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2013.

[24] Bob Carpenter. *Type-logical semantics.* MIT press, 1997.

[25] Nathanael Chambers and Dan Jurafsky. Template-based information extraction without the templates. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2011.

[26] Du-Seong Chang and Key-Sun Choi. Incremental cue phrase learning and bootstrapping method for causality extraction using cue phrase and word pair probabilities. *Inf. Process. Manage.*, 42(3):662–678, 2006.

[27] David Chen. Fast online lexicon learning for grounded language acquisition. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2012.

[28] David L. Chen and Raymond J. Mooney. Learning to sportscast: a test of grounded language acquisition. In *Proceedings of ICML*, 2008.

[29] David L Chen and Raymond J Mooney. Learning to interpret natural language navigation instructions from observations. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-2011)*, pages 859–865, 2011.

[30] Stephen Clark and James R Curran. Wide-coverage efficient statistical parsing with ccg and log-linear models. *Computational Linguistics*, 33(4):493–552, 2007.

[31] J. Clarke, D. Goldwasser, M.W. Chang, and D. Roth. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, pages 18–27. Association for Computational Linguistics, 2010.

[32] Michael Collins. *Head-driven statistical models for natural language parsing.* PhD thesis, University of Pennsylvania, 1999.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[34] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Conference on Language Resources and Evaluation*, 2006.

[35] Nikolaos Mavridis Deb Roy, Kai-Yuh Hsiao. Coversational robots: Building blocks for grounding word meaning. In *Proceedings of the HLT-NAACL Workshop on Learning Word Meaning from Non-linguistic Data*, volume 6, pages 70–77. Association for Computational Linguistics, 2003.

[36] Edsger W Dijkstra. On the foolishness of "natural language programming". In *Program Construction*, pages 51–53. Springer, 1979.

[37] Q. Do, Y. Chan, and D. Roth. Minimally supervised event causality identification. In *EMNLP*, 7 2011.

[38] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Path planning for autonomous vehicles in unknown semi-structured environments. *The International Journal of Robotics Research*, 29(5):485–501, 2010.

[39] Dropbox. Dropbox core api, 2015. URL `http://www.dropbox.com/developers/core/docs`.

[40] Michael Fleischman and Deb Roy. Intentional context in situated natural language learning. In *Proceedings of CoNLL*, pages 104–111, 2005.

[41] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20: 2003, 2003.

[42] Jeffrey Friedl. *Mastering Regular Expressions*. OReilly, 2006.

[43] Boris Galitsky and Daniel Usikov. Programming spatial algorithms in natural language. In *Proceedings of the AAAI Workshop on Spatial and Temporal Reasoning*, 2008.

[44] Steven I. Gallant. Perceptron-based learning algorithms. *Neural Networks, IEEE Transactions on*, 1(2):179–191, 1990.

[45] Ruifang Ge and Raymond Mooney. A statistical semantic parser that integrates syntax and semantics. In *Proceedings of CoNLL*, 2005.

[46] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

[47] Roxana Girju and Dan I. Moldovan. Text mining for causal relations. In *Proceedigns of FLAIRS*, pages 360–364, 2002.

[48] R.C. Gonzalez and M.G. Thomason. Syntactic pattern recognition: An introduction. 1978.

[49] Cordell Green. Application of theorem proving to problem solving. Technical report, DTIC Document, 1969.

[50] Ralph Grishman, David Westbrook, and Adam Meyers. NYUâĂŹs English ACE 2005 System Description. In *Proceedings of the Automatic Content Extraction Evaluation Workshop*, 2005.

[51] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

[52] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 803–814. ACM, 2014.

[53] George E Heidorn. Automatic programming through natural language dialogue: A survey. *IBM Journal of Research and Development*, 20(4):302–313, 1976.

[54] Julia Hockenmaier and Mark Steedman. Generative models for statistical parsing with combinatory categorial grammar. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 335–342. Association for Computational Linguistics, 2002.

[55] Julia Hockenmaier and Mark Steedman. Ccgbank: a corpus of ccg derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, 33(3):355–396, 2007.

[56] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[57] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 2. Addison-wesley Reading, MA, 1979.

[58] Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. Learning to solve arithmetic word problems with verb categorization. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–533, 2014.

[59] Kai-yuh Hsiao, Nikolaos Mavridis, and Deb Roy. Coupling perception and simulation: Steps towards conversational robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 928–933. IEEE, 2003.

[60] Kai-yuh Hsiao, Stefanie Tellex, Soroush Vosoughi, Rony Kubat, and Deb Roy. Object schemas for grounding language in a responsive robot. *Connection Science*, 20(4):253–276, 2008.

[61] Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64. Association for Computational Linguistics, 2005.

[62] Heng Ji and Ralph Grishman. Refining event extraction through cross-document inference. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2008.

[63] Victor M. Jimenez and Andres Marzal. Computation of the n best parse trees for weighted and stochastic context-free grammars. *Advances in Pattern Recognition*, pages 183–192, 2000.

[64] Thorsten Joachims. Advances in kernel methods. chapter Making large-scale support vector machine learning practical, pages 169–184. MIT Press, 1999.

[65] Colin G Johnson. Genetic programming with fitness based on model checking. In *Genetic Programming*, pages 114–124. Springer, 2007.

[66] Bevan K. Jones, Mark Johnson, and Sharon Goldwater. Semantic parsing with bayesian tree transducers. In *Proceedings of ACL*, 2012.

[67] Anders Jonsson and Andrew Barto. A causal approach to hierarchical decomposition of factored mdps. In *Advances in Neural Information Processing Systems, 13:1054-1060*, page 22. Press, 2005.

[68] Daniel Jurafsky and James H Martin. *Speech and Language Processing*. Pearson, 2014.

[69] R.J. Kate and R.J. Mooney. Using string-kernels for learning semantic parsers. In *Association for Computational Linguistics*, volume 44, page 913, 2006.

[70] R.J. Kate, Y.W. Wong, and R.J. Mooney. Learning to transform natural to formal languages. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1062. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[71] Joohyun Kim and Raymond Mooney. Unsupervised pcfg induction for grounded language learning with highly ambiguous supervision. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2012.

[72] T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. Lexical generalization in ccg grammar induction for semantic parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523. Association for Computational Linguistics, 2011.

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

[74] Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of Empirical Methods in Natural Language Processing*, 2013.

[75] Tessa Lau, Pedro Domingos, and Daniel S Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd international conference on Knowledge capture*, pages 36–43. ACM, 2003.

[76] Tessa A Lau, Pedro Domingos, and Daniel S Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[77] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile syStems, Applications, and Services*, pages 193–206. ACM, 2013.

[78] Kenton Lee, Yoav Artzi, Jesse Dodge, and Luke Zettlemoyer. Context-dependent semantic parsing for time expressions. In *Proceedings of the Conference of the Association for Computational Linguistics*, pages 1437–1447, 2014.

[79] Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. From natural language specifications to program input parsers. In *Proceeding of the Association for Computational Linguistics.*, 2013.

[80] Tao Lei, Fan Long, Regina Barzilay, and Martin C Rinard. From natural language specifications to program input parsers. Association for Computational Linguistics (ACL), 2013.

[81] Marián Lekavý and Pavol Návrat. Expressivity of strips-like and htn-like planning. *Lecture Notes in Artificial Intelligence*, 4496:121–130, 2007.

[82] Iddo Lev, Bill MacCartney, Christopher Manning, and Roger Levy. Solving logic puzzles: From robust processing to precise semantics. In *Proceedings of the Workshop on Text Meaning and Interpretation.* Association for Computational Linguistics, 2004.

[83] P. Liang, M.I. Jordan, and D. Klein. Learning dependency-based compositional semantics. *Computational Linguistics*, pages 1–94, 2011.

[84] Percy Liang, Michael I. Jordan, and Dan Klein. Learning semantic correspondences with less supervision. In *Proceedings of ACL*, pages 91–99, 2009.

[85] Greg Little and Robert C. Miller. Keyword programming in java. *Automated Software Engineering*, 16(1):37–71, 2009.

[86] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2 (1):90–121, 1980.

[87] Mstislav Maslennikov and Tat-Seng Chua. A multi-resolution framework for information extraction from free text. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2007.

[88] Cynthia Matuszek, Nicholas FitzGerald, Luke Zettlemoyer, Liefeng Bo, and Dieter Fox. A joint model of language and perception for grounded attribute learning. In *Proceedings of the International Conference on Machine Learning*, 2012.

[89] Maxima. Maxima, a computer algebra system. version 5.32.1, 2014. URL `http://maxima.sourceforge.net/`.

[90] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic discovery and transfer of maxq hierarchies. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 648–655, 2008.

[91] Microsoft. Microsoft support, 2015. URL `http://support.microsoft.com`.

[92] R. Mihalcea, H. Liu, and H. Lieberman. Nlp (natural language processing) for nlp (natural language programming). *Computational Linguistics and Intelligent Text Processing*, pages 319–330, 2006.

[93] MinecraftWiki. Minecraft wiki, 2015. URL http://www.minecraftwiki.net.

[94] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. http://www.brics.dk/automaton/.

[95] Raymond J. Mooney. Learning to connect language and perception. In *Proceedings of AAAI*, pages 1598–1601, 2008.

[96] Raymond J. Mooney. Learning language from its perceptual context. In *Proceedings of ECML/PKDD*, 2008.

[97] N. Moreira and R. Reis. Implementation and application of automata. 2012.

[98] Dana Movshovitz-Attias and William W Cohen. Natural language models for predicting programming comments. 2013.

[99] Anirban Mukherjee and Utpal Garain. A review of methods for automatic understanding of natural language mathematical problems. *Artificial Intelligence Review*, 29(2), 2008.

[100] A. Newell, J.C. Shaw, and H.A. Simon. *The Processes of Creative Thinking.* Paper P-1320. Rand Corporation, 1959. URL http://books.google.com/books?id=dUIkPAAACAAJ.

[101] Jorge Nocedal and Stephen Wright. Numerical optimization, series in operations research and financial engineering. *Springer, New York*, 2006.

[102] James Timothy Oates. *Grounding Knowledge in Sensors: Unsupervised Learning for Language and Planning.* PhD thesis, University of Massachusetts Amherst, 2001.

[103] oDesk, 2013. http://odesk.com/.

[104] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Annual Meeting for the Association for Computational Linguistics (ACL)*, 2015.

[105] Hoifung Poon. Grounded unsupervised semantic parsing. In *Proceeding of the Annual Meeting of the North American Chapter of the Association for Computational Linguistics*, 2013.

[106] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, Beijing, China, July 2015.

[107] Aarne Ranta. A multilingual natural-language interface to regular expressions. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 79–90. Association for Computational Linguistics, 1998.

[108] R.G. Raymond and R. J. Mooney. Discriminative reranking for semantic parsing. In *Proceedings of the COLING/ACL poster sessions*, pages 263–270. Association for Computational Linguistics, 2006.

[109] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples.

[110] Roi Reichart and Regina Barzilay. Multi-event extraction guided by global constraints. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics*, 2012.

[111] Alan Ritter, Mausam, Oren Etzioni, and Sam Clark. Open domain event extraction from twitter. In *Proceedings of the Conference on Knowledge Discovery and Data Mining*, 2012.

[112] S. Roy and D. Roth. Solving general arithmetic word problems. In *EMNLP*, 2015.

[113] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach, 2010.

[114] Jean E Sammet. The use of english as a programming language. *Communications of the ACM*, 9(3):228–230, 1966.

[115] Min Joon Seo, Hannaneh Hajishirzi, Ali Farhadi, and Oren Etzioni. Diagram understanding in geometry questions. In *The AAAI Conference on Artificial Intelligence (AAAI-2014), Québec City, Québec, Canada*, 2014.

[116] Avirup Sil and Alexander Yates. Extracting STRIPS representations of actions and events. In *Recent Advances in Natural Language Learning (RANLP)*, 2011.

[117] Avirup Sil, Fei Huang, and Alexander Yates. Extracting action and event semantics from web text. In *AAAI 2010 Fall Symposium on Commonsense Knowledge (CSK)*, 2010.

[118] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751, 2012.

[119] Jeffrey Mark Siskind. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *Journal of Artificial Intelligence Research*, 15:31–90, 2001.

[120] Armando Solar-Lezama. *Program synthesis by sketching.* ProQuest, 2008.

[121] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. *ACM SIGPLAN Notices*, 40(6):281–294, 2005.

[122] M. Steedman. *The syntactic process.* MIT press, 2000.

[123] Mark Steedman. Surface structure and interpretation. 1996.

[124] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[125] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in NIPS*, pages 1057–1063, 2000.

[126] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 396–411. Springer, 2005.

[127] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R Walter, Ashis Gopal Banerjee, Seth J Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*, 2011.

[128] Stefanie Tellex, Pratiksha Thaker, Joshua Joseph, and Nicholas Roy. Learning perceptually grounded word meanings from unaligned parallel data. *Machine Learning*, 94(2):151–167, 2014.

[129] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[130] C.A. Thompson and R.J. Mooney. Acquiring word-meaning mappings for natural language interfaces. *Journal of Artificial Intelligence Research*, 18(1):1–44, 2003.

[131] Mechanical Turk, 2013. `http://mturk.com/`.

[132] Adam Vogel and Dan Jurafsky. Learning to follow navigational directions. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2010.

[133] Adam Vogel and Daniel Jurafsky. Learning to follow navigational directions. In *Proceedings of the ACL*, pages 806–814, 2010.

[134] David E. Wilkins and Marie des Jardins. A call for knowledge-based planning. *AI magazine*, 22(1):99, 2001.

[135] Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *IJCAI*, volume 97, pages 1178–1185, 1997.

[136] Terry Winograd. Understanding natural language. *Cognitive psychology*, 3(1): 1–191, 1972.

[137] Alicia P. Wolfe and Andrew G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *In Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 816–823, 2005.

[138] Y.W. Wong and R.J. Mooney. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Annual Conference of the North American Chapter of the Association of Computational Linguistics on Human Language Technology*, pages 439–446. Association for Computational Linguistics, 2006.

[139] Y.W. Wong and R.J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *Annual Meeting for the Association for Computational Linguistics (ACL)*, volume 45, page 960, 2007.

[140] Roman V Yampolskiy. Ai-complete, ai-hard, or ai-easy–classification of problems in ai. In *The 23rd Midwest Artificial Intelligence and Cognitive Science Conference*. Citeseer, 2012.

[141] Chen Yu and Dana H. Ballard. On the integration of grounding language and learning objects. In *Proceedings of AAAI*, pages 488–493, 2004.

[142] J.M. Zelle and R.J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1050–1055, 1996.

[143] L.S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. 2005.

[144] L.S. Zettlemoyer and M. Collins. Online learning of relaxed ccg grammars for parsing to logical form. In *In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL-2007*. Citeseer, 2007.

[145] L.S. Zettlemoyer and M. Collins. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 976–984. Association for Computational Linguistics, 2009.

[146] Luke Zettlemoyer. *Learning to Map Sentences To Logical Form*. PhD thesis, Massachusetts Institute of Technology, 2009.