LEARNING ADAPTIVE LANGUAGE INTERFACES THROUGH INTERACTION

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

Sida I. Wang
August 2017

This dissertation is online at: http://purl.stanford.edu/xh214jb2032

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christopher Manning, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Noah Goodman

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Percy Liang

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christopher Potts

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Natural language interfaces have the potential to complement GUIs and programming for many tasks, and enable all of us to better communicate with computers. However, until computers think like humans, they may not be able to satisfactorily understand human language, and we might have to settle for *adaptive language interfaces* where human users have to partially adapt to the capabilities of computers as computers adapt to human communication preferences. Because static datasets do not account for system capabilities, such adaptation must be part of an interactive learning process.

In this thesis, we describe two extremes of interactive language learning—learning starting from scratch, and learning by "naturalizing" a programming language. In starting from scratch, the human can use arbitrary languages. The computer does not understand any language initially, but it has the ability to learn based on what the human selects as the correct interpretation. In naturalizing a programming language, the human must use a programming language initially. Through the naturalization process, a community of users can collectively teach the computer to understand languages that they prefer more. With a starting programming language, users can provide much stronger definitional supervision beyond simple selection of the correct answer.

To start from scratch, we introduce a new language learning setting relevant to building adaptive natural language interfaces. It is inspired by Wittgenstein's language games: a human wishes to accomplish some task (e.g., achieving a certain configuration of blocks), but can only communicate with a computer, who performs the actual actions (e.g., removing all red blocks). The computer initially knows nothing about language and therefore must learn it from scratch through interaction, while the human adapts to the computer's capabilities. We created a game called SHRDLURN in a blocks world and collected interactions from 100 people playing it. First, we analyze the humans' strategies, showing that using compositionality and avoiding synonyms correlates positively with task performance. Second, we compare computer strategies, showing that modeling pragmatics on a semantic parsing model achieves a higher accuracy for more strategic players.

On the other extreme, we seed the system with a core programming language and allow users to "naturalize" the core language incrementally by defining alternative syntax and increasingly complex concepts in terms of compositions of simpler ones. Starting with a core language allows the computer to go beyond learning from simple supervision signals like selecting the right interpretation to learning from complex definitions that the users provide. In a voxel world, we show that a community of users can simultaneously teach one system a diverse language and use it to build 240 complex voxel structures. Over the course of three days, these users went from using only the core language to using the naturalized language in 85.9% of the last 10K utterances.

Experimentally, we show that our system has sufficient interactive learning ability so that the users and the system can negotiate their language through interaction and reach a language that is natural to the user and understandable by the system; and the resulting languages are diverse and adapted to the task but can deviate from standard English. We hope such adaptive language interfaces can improve human-computer communication even when computers capabilities differ significantly from humans.

Acknowledgments

My six years as a PhD student in the Stanford NLP group is an unforgettable and formative experience. First, I am extremely grateful to be co-advised by Chris Manning and Percy Liang.

Chris is always very supportive of my research. While he had a keen sense of research direction, he also gave me a lot of freedom to pursue my ideas. He was generous with frank and high-quality advice, such as "work in a productive area for an extended period of time", but it often took me a while to realize his wisdom and listen. Besides good advice, he had an uncanny sense of experiments and corpora—often spotting issues simply by looking at a table dense in numbers or a list of words. Outside of research, he always had my interests in mind, and lent me his valuable support.

Percy started right after I joined the NLP group. Almost instantly, he connected our previous work on dropout to real statisticians, and we made immediate progress. Percy gave detailed feedback of incredible quality and quantity on my code, papers, and talks—one of my SEMPRE pull request had over 100 comments. Much of the work described in this thesis not only builds on Percy's research, but also builds on his code. From his code, I learned more about semantic parsing, computational experiments, working with data, building software, and how to relieve the boredom of making slides using `sfig`. I am deeply indebted to Percy for teaching me how to do better research, write better papers and code.

During my undergraduate years, Geoff Hinton's class inspired me to study machine learning. I thank Geoff for the opportunity and for mentoring me at both Toronto and Google.

I thank Chris Potts, Noah Goodman, and Michael Frank—whose work on pragmatics and communication is deeply inspiring—for serving on my committee.

Research was made more enjoyable by some amazing collaborators. In particular, working with Arun Chaganty on method of moments was probably my most intellectually stimulating project. Helping Spence with interactive machine translation might have nudged me to continue thinking about interactive learning. I learned a lot from Sam Ginn, who built voxelurn with great speed and skills. I am grateful for the opportunity to collaborate with Roy Frostig, Stefan Wager, Mengqiu

Wang, Will Fithian and Ziang Xie, and to mentor Sam Ginn, Nadav Lidor, Megha Srivastava, and Philip Hwang. Lastly, I thank Ice, Robin, Kelvin and Megha for joining me in the plotting of the future.

I thank Danqi for keeping me more physically fit and more informed in the 4 years when we shared an office; Jiwei for playing basketball with me under his usual intensity; Ice for helping me learn more about semantic parsing with great clarity; Will Monroe for the discussions and feedback on pragmatics; Jacob Steinhardt for all the math problems and teaching me to boulder; Stephen Miller and Andrej Karpathy for the many dinner discussions; Mengqiu Wang for showing me around early conferences; Philipp Krähenbühl and Vladlen Koltun for mentoring my rotation project. Jon Gauthier for his insightful comments and encouraging feedback on this work.

Finally, I thank my parents for their hands-off yet strong and unconditional support. Shortly after the tech crash in 2001, I wanted dad to help me Java. He was initially reluctant and predicted that Java will be obsolete when I finish school. However, mom was encouraging and dad obliged—most of the code for this thesis was written in Java.

Contents

Chapter 1

Introduction

1.1 Overview

Natural language is a powerful method of communication among humans. Getting computers to understand natural language has been a goal and benchmark of AI since the celebrated Turing test (Turing, 1950). Usually, natural language refers the language used by fluent native speakers in their daily lives. However, if the computer has different capabilities from the human, then the ideal language of communication might also differ from the standard natural language (e.g. English with standard vocabularies) that is adapted to the human action space (i.e. what humans do in our daily lives). One reason is that most utterances in a natural language fall outside of the computer action space, and thus are meaningless to the computer. For example, a guest might give the following instruction to the hotel concierge:

(1) "*please give me a wakeup call at 7am for my flight tomorrow morning, thanks*"

This instruction uses "*please*" and "*thanks*", to express politeness, which makes a human more likely to oblige. Redundantly including both "*7am*" and "*morning*" make it less likely misunderstood or misremembered. "*for my flight*" can imply that more calls are appreciated should the first one fail, as opposed to "*for breakfast*". However, current computer systems do not have these human features, such as being sensitive to politeness, forgetting information, and performing common sense reasoning. If all that the system does is setting alarms then the language use might adapt accordingly:

(1) "*set alarm at 7am*"

(2) "*alarm 7am*"

On the other hand, the computer has many useful capabilities not usually performed by a human, (e.g. searching the internet, or running a particular program with a particular setting) and the natural language might not have the appropriate vocabulary and precision to express these meanings unless terms and conventions are specifically invented. For example, terms such as *email*, *download*, and *google* are part of English because the corresponding computer capabilities became important enough to be incorporated in natural language. For another example, programming languages are elaborately designed and refined to express meanings in the computer action space precisely. Consider the following bash command and its English translation

(1) `unzip -j backup.zip *.txt -x image/`

(2) "*extract all files ending with "txt" contained in backup.zip to the current directory, but exclude those files in the image folder and remove internal directory structures.*"

(3) unzip *.txt in backup.zip exclude image and flatten

The corresponding English is much longer because it must explicitly contain some information already implied by the context of `unzip`. For example, "*image*" is a folder in the context, but using "*image*" to refer to a folder is unconventional in standard English. In context, words like *.txt, flatten, and unzip are shorter than their English counterparts and less ambiguous. While English (2) might make more sense to some users than the adapted utterance (3), the benefit might be marginal because they still need to know the meaning of "*extraction*" and "*internal directory structures*". Long natural language utterances like (2) are likely harder to process by the computer than (3), which is shorter and only specifies more essential information. While (1) is even shorter than (3), users have to remember the options `j` and `x` as well as the order of the arguments if they want to produce the command (1). In contrast, the adapted utterance (3) is more human-friendly than the bash command (1), and more concise and easier for the computer to understand than English (2). In this example, even proper English would not make sense to people with no knowledge of the computer action space, and few people can produce the correct computer commands effortlessly. A tradeoff is present here—a user with no trouble recalling all bash options might as well produce a precise command that is fully adapted to the computer action space, and a user who has no idea what the computer does might find it easier to produce English. Despite this tradeoff, the more adapted language (3) might have an edge in usability over both English and command.

This example illustrate that the goal of communicating with computers in English could be misguided. Besides requiring major advances so the computer can become as capable as humans, this

goal also excludes many useful computer actions that are unintuitive to humans, and unnatural to express in natural language (Kushman and Barzilay, 2013; Gulwani and Marron, 2014; Lin et al., 2017 2017). Even when the action space is intuitive (Zelle and Mooney, 1996; Berant et al., 2013; Tellex et al., 2011; Campagna et al., 2017), the computer does not perform these actions in a similar way as humans. Due to such mismatch, improving human/computer communication might require the human and the computer to adapt to each other's capabilities and languages. Adaptation is common in human communication—changes happen constantly in natural language and are driven by both predictable forces and randomness (Campbell, 1998; Ullmann, 1962). Many domains have specific sublanguages that are very different from the base natural language (Kittredge and Lehrberger, 1982). People also adapt to communication channels such as texting and telephone. Since adaptation is natural to humans, they are capable of adapting to the computer as well. For example, people are capable of learning a programming language and an API. Even when the system is intended to understand natural language, people adapt by using more terse Google-ese and virtual assistant commands so these systems are less likely to be distracted and are more likely to work. However, human adaptation requires significant learning efforts especially if computers continue to gain new abilities. In contrast, current computers do not adapt to humans automatically. In the standard approach, where we first collect a static dataset and then train a system, adaptation is unlikely because the dataset is not affected by the system capabilities.

Thesis. If the computer can learn to communicate by interacting with humans, then it can adapt to human preferences in an automatic way, and perhaps human/computer communication would improve. Building systems that have interactive learning abilities, and studying how human users communicate with these adaptive systems is the subject of this thesis. In this work, we give the computer *some* ability to learn language through interaction using semantic parsing and machine learning techniques. Importantly, while the computer adapts to human preferences, the human also learns to adapt to the computer through interaction. This way, the human can accommodate the action space and language ability of the computer, and we have hopes to improve communication even if the computer has limited language learning and understanding abilities. In contrast, aiming directly at standard natural language with a limited system and without adapting to the computer results in suboptimal language and user deception.

In the body of this thesis, we describe two extremes of interactive language learning—learning starting from scratch, and learning by "naturalizing" a programming language. In starting from

scratch, the human can use arbitrary language initially, and the computer learns from human selection. In "naturalizing" a programming language, the human must use a programming language initially. Through the "naturalization" process, the community of users can collectively teach the computer to understand a language that they prefer more. With a starting programming language, users can provide much stronger definitional supervision beyond simple selection of the correct answer.

Our first extreme (Chapter 4) is to start from scratch—we introduce a new language learning setting relevant to building adaptive natural language interfaces. It is inspired by Wittgenstein's language games (Wittgenstein, 1953): a human wishes to accomplish some task (e.g., achieving a certain configuration of blocks), but can only communicate with a computer, who performs the actual actions (e.g., removing all red blocks). The computer initially knows nothing about language and therefore must learn it from scratch through interaction, while the human adapts to the computer's capabilities. We created a game called SHRDLURN in a blocks world and collected interactions from 100 people playing it. First, we analyze the humans' strategies, showing that using compositionality and avoiding synonyms correlates positively with task performance. Second, we compare computer strategies, showing that modeling pragmatics on a semantic parsing model accelerates learning for more strategic players.

On the other extreme (Chapter 5), we seed the system with a core programming language and allow users to "naturalize" the core language incrementally by defining alternative syntax and increasingly complex concepts in terms of compositions of simpler ones. In a voxel world, we show that a community of users can simultaneously teach one system a diverse language and use it to build 240 complex voxel structures. Over the course of three days, these users went from using only the core language to using the naturalized language in 85.9% of the last 10K utterances.

In both settings, learning language through interaction and adaptive communication are central considerations. In order to build systems that better support interactive learning, we develop new settings, model pragmatics in an online fashion, and propose a grammar induction method that take advantage of interaction. Experimentally, we see evidence that

1. for the interactive learning ability that we provide, users and the system can negotiate their language through interaction and reach a point that is natural to the user, and understandable by the system;

2. the resulting languages are diverse, adapted to the task, and can deviate significantly from standard English.

Programming language	Natural language
precise and powerful	ambiguous, powerful as well
easy for computer to understand	easy for humans to produce
clear but uncompromising syntax	ambiguous but flexible syntax
fixed, but changes with human-in-the-loop	adapt through use

Table 1.1: A contrast of programming language vs. natural language

In the rest of this thesis, we start by discussing criteria and existing approaches for natural language interfaces in Chapter 2, and background on semantic parsing in Chapter 3. Then we dive into learning language games in Chapter 4 and naturalizing a programming language in Chapter 5.

1.2 Natural language vs. programming language

In this section, we want to contrast NLIs with programming languages and discuss why each might be preferable. For many of the applications we are interested in, programming languages are successful and powerful of communicating with computers. Despite recent divergence, linguistics influenced programming languages since its early days. Context-free grammars, a formalism developed for modeling natural language, is backing most modern programming languages. An obvious and important reason to prefer NLIs is that people are already skilled at natural language, and learning to program requires effort. While particular languages are relatively stable, systems and interfaces change much more frequently and people must keep learning and adapting. More fundamentally, natural language makes extensive use of context, so utterances can be more concise when useful information are inferred from the context.

It is often debated if natural language is an appropriate medium (Dijkstra, 1978; Androutsopoulos et al., 1995) for communicating with computers since programming languages are fully adapted to the computer action space, where all meanings can be expressed precisely. Since all programming languages are precise, precision clearly does not determine if a particular programming language is suitable for a particular task. Instead, programming languages change and adapt over time in a process that involves many humans in the loop, who get experience using a programming language, and then use this experience to improve the language. While strongly-typed, strict languages where the compiler checks for many potential errors might be preferable when building large software systems, more permissive languages can be more convenient for ad hoc tasks. Verbosity can make a big difference in the user experience even when little semantic difference exists (markdown vs. HTML, JSON vs. XML, etc.)

> Some still seem to equate "the ease of programming" with the ease of making undetected mistakes. ... The "naturalness" with which we use our native tongues boils down to the ease with which we can use them for making statements the nonsense of which is not obvious. ... From one gut feeling I derive much consolation: I suspect that machines to be programmed in our native tongues——be it Dutch, English, American, French, German, or Swahili— are as damned difficult to make as they would be to use.
>
> (Dijkstra (1978))

Although ambiguities are eventually resolved, programming languages have many features that use *potential* ambiguities to improve human usability. Additional rules such as precedence and associativity are used to resolve order of operation related ambiguity that is awkward to specify using the grammar rules. These simple rules merely allow programming languages to avoid some parenthesis, but that already improves human usability. Scoping allows the same symbol to take on different meanings depending on the context. '-' is either unary or binary and may take different types of operands depending on context. In addition, features such as polymorphism, operator overloading, and type inference use context to determine the semantics in a fairly general way. These features are comparable to pragmatic reasoning in natural language. By leaving inferable details unspecified, programs can be shorter and written at an appropriate generality, which is appreciated by human users.

However, most programming languages are deliberately backed by an unambiguous context-free grammar of some type for fast parsing and analysis. These restrictions forbid more powerful methods for resolving potential ambiguities using context other than a few precise rule-based methods. If we relax the precise syntax and semantics, then we can have even more freedom to make languages more human friendly than if we stay within the confines of programming languages. While going to a natural language is one approach, there is also a big area in between where we can relax standard programming language restrictions without immediately using a standard natural language.

Exploring this middle area means that we lose systematic ways for resolving ambiguities and we need users to resolve the remaining ambiguities. A simple proposal is to stay in domains where it is easier to inspect potential ambiguities rather than making the language precise. This would be the case when it is easier for the user to visually look at all the candidates and determine if they are desired than to read programs. Ambiguity propagates when ambiguous statements are used in other statements and unresolved ambiguities multiply when combined in higher level definitions—we deal with this problem in Chapter 5.

1.3 Blocks world

Most of our experiments are done in a blocks world domain, which has a long history since Winograd (1972). We argue why this is a still a relevant setting to do experiments, and note some of its limitations.

Recently, blocks world has been identified by the DARPA Communicating with Computers program as an unsolved problem that is useful to tackle again with recent advances in AI/ML. While blocks world is a toy domain, it is intuitive, easily crowdsourced and yet captures many difficulties of language understanding. Because the language is grounded, goal-oriented and visual, failure cases tend to be very apparent compared to most dialogue tasks. For the purpose of testing adaptive language interfaces, it is appealing to tackle the simplest unsolved problem that still has the essential features of interests. In this setting, we can study reasoning about entities, properties, and the emergence of abstract, higher-order concepts from groups of blocks. For example, voxel-based games like Minecraft created highly complex worlds out of blocks. Here are some examples in blocks world

- Arbitrary world knowledge: "*add the binary encoding of the first row to the second, put the result in the third*", "*add the least amount of blue blocks so the scene looks like a cat*"
- Vagueness: "*connect the red and yellow blocks*", "*add some red to the top cube*"
- Context dependence and coreference: "*add a green block top, add another one*", "*add a dog, and another one besides it*"
- Complex actions: "*move up 10 spaces and left 5 spaces and fill the path with yellow*", "*red cube size 5*", "*rotate the chair 90 degrees*", "*chair with red seat and yellow legs*"

Limitations. Restricting to blocks world also many limitations. First, users might have a variety of intents, especially for virtual assistants, where the capabilities of the interface might be opaque to the users and identifying user intent is already challenging. Second, while rich in compositional phenomena, the lexical semantics is impoverished in blocks world. While compositional semantics is arguably more interesting, lexical semantics can be very challenging in other tasks like interfacing with an open domain databases where there are many entities and relations, large vocabulary sizes, and a hierarchical type system. Third, interfacing with unintuitive computer systems such as bash, data analysis and plotting might require different kinds of adaptations than in blocks world, where

humans have strong intuitions on how blocks can be manipulated and referred to. Even though users have strong intuitions on the "physics" of blocks world, computer simulations might derivate from that. People also have intuitions related to blocks that is not realized in the computer action space. For example, people tend to think in terms of objects they can build with blocks such as chairs and cats. Our system in Chapter 5, however, only supports procedural descriptions of how to build objects instead of declarative descriptions of objects. Finally, blocks world is fully-observable, deterministic, turn-based, and discrete, which excludes some interesting phenomena.

1.4 Related work

Grounding. Our work connects with a broad body of work on grounded language, in which language is used in some environment as a means towards some goal. Traditional work like Winograd (1972) had sophisticated narrow domain understanding, but no learning. More recent examples include playing games (Branavan et al., 2009, 2010; Reckman et al., 2010) interacting with robotics (Tellex et al., 2011, 2014), and following instructions (Vogel and Jurafsky, 2010; Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013) The main framework we use, semantic parsing, is often concerned with grounded language (Kollar et al., 2010; Matuszek et al., 2012; Artzi and Zettlemoyer, 2013).

Role of interactivity in human language learning. Examining language acquisition research, there is considerable evidence suggesting that human children require interactions to learn language, as opposed to passively absorbing language, such as when watching TV (Kuhl, 2004; Sachs et al., 1981). However, these works focus on attention and social interactions rather than the the available learning signal.

Krashen (1982) suggests that when learning a language, rather than consciously analyzing increasingly complex linguistic structures (e.g. sentence forms, word conjugations), humans advance their linguistic ability through meaningful interactions.

In contrast, the standard dataset setting has no interaction. The feedback stays the same and does not depend on the state of the system or the actions taken. We think that interactivity is important, and that an interactive language learning setting will enable adaptive and customizable systems, especially for resource-poor languages and new domains where starting from close to scratch is unavoidable.

Learning from instructions. Azaria et al. (2016) presents Learning by Instruction Agent (LIA), which also advocates learning from users. They argue that developers cannot anticipate all the actions that users want, and that the system cannot understand the corresponding natural language even if the desired action is built-in. Like Jia et al. (2017), Azaria et al. (2016) starts with an ad-hoc set of initial slot-filling commands in natural language as the basis of further instructions—our approach starts with a more expressive core PL designed to interpolate with natural language. Compared to previous work, this work studied interactive learning in a shared community setting and hierarchical definitions resulting in more complex concepts.

Rapidly extensible language. Rapidly Extensible Language (Dostert and Thompson, 1969a,b; Thompson and Thompson, 1975) has a similar goal as ours in Chapter 5. They were particularly interested in *a language that is natural for a specific user*, rather than a general natural language. They strongly argued for having idiosyncratic semantics for each use case. Similar to us, they emphasized the importance of language use in context, and realized that the key to improve communication is to make use of more context, rather than aiming at the language of a *fluent native speaker*.

In order to customize terms in the language, they used definitions like

(1) def: sex ratio of "sample": (number of "sample" who are male)*100/(number of "sample" who are female)

(2) def: Mazulu crone: Mazulu female who was born before 1920

(3) What is the sex ratio of the children of Rilazulu crones?

While the core intuition and goals are highly similar to us, the available techniques and computational power is very different. In particular, we have the advantage of having better machine learning techniques and easy access to a crowd of users to gather data and test system changes.

Adaptation in human communication. In order to communicate efficiently, language should be appropriate to the subject, or action space. One reason why natural language can express an amazingly broad set of meanings is that people are very adaptive in their language use. For example, more specific languages and protocols are used to enable more precise communication in games and mathematics where ambiguities are less useful and less tolerable. Most professions have their own jargons, where existing words take on new meanings and new words are coined. The medium of communication can play a role, for example, people adapt to noisy channels like telephone by using more distinctive words to represent characters redundantly, escape censorship using alternative

characters, and adapt to 140 character messages and tweets (i.e. textese). Most importantly, people adapt to their audience in a cooperative way, giving rise to different word usage and pragmatic phenomena (Grice, 1975).

Grammar induction. Grammar induction is generally a difficult task, but necessary if we want to learn from weaker supervision. In a general context, "grammar induction" usually refers to unsupervised grammar induction, where the goal is learning a PCFG or some form of grammar from text without any labels. In semantic parsing, "grammar induction" means semantic grammar induction, where we have both text and logical form labels, but no information on how to derive the logical form from text (Kwiatkowski et al., 2010; Zettlemoyer and Collins, 2005, 2007; Kwiatkowski et al., 2010, 2011). This difference is a consequence of semantic grammars also having rules specifying the semantics in addition to the left/right sides of a CFG.

Grammar induction is the main driver of learning in Chapter 5, where the core language and interaction gives us some leverage. In particular, we have both the utterance and derivations for the body of the definitions since users had to make the body executable. In the body, the core language structure can remain apparent in many utterances, which results in a fair amount of string overlap.

Unlike the GENLEX style grammar induction (Zettlemoyer and Collins, 2005), which use all possible spans, we are very conservative and only induce high precision rules, which is due to the lower tolerance of a large number of candidates in the interactive setting.

Chapter 2

Natural language interfaces

2.1 Natural language interfaces (NLIs)

As speech recognition works better and mobile becomes more ubiquitous, natural language understanding is increasingly the limiting factor. Through commercial virtual assistants like Alexa, Siri, Google Now, Cortana, etc., many users are already embracing spoken commands for simple tasks such as setting an alarm, navigation, playing music, and interacting with the Internet of Things. However, while handling simple commands, these systems usually fail for simple compositional commands such as "*play a Beatles song for my 7am alarm*"[1] or "*set an alarm 1 hour before my interview*". While handling such utterances is the goal of semantic parsing (Chapter 3), it is typically not deployed nor solved. Currently, when complex commands work, it is usually because the developer anticipated the particular scenarios and handling them specifically. While users still benefit from always-on responsiveness and hands-free convenience, the expressive potential of a language interface is severely limited by the lack of better language understanding.

We consider the *user initiative* case where the human wants the computer to do some task, the human communicates with the computer, and the computer performs some actions based on what the human said. Some examples are analyzing and plotting data (Gulwani and Marron, 2014), querying databases (Zelle and Mooney, 1996; Berant et al., 2013), manipulating text (Kushman and Barzilay, 2013), or controlling the Internet of Things (Campagna et al., 2017) and robots (Tellex et al., 2011).

[1] Tested in June 2017 on Alexa

2.1.1 Idealized NLI

We would like to clarify some distinct goals and approaches to NLIs. To do this, we consider the *wizard of Oz* setting for user initiative NLIs, where another human, call them *the wizard*, communicates with the human user, and performs the corresponding action. Suppose that the wizard is also a computer expert who can write arbitrary programs in addition to understanding natural language. The wizard is always helpful in this case, since it can

- convert the desired action to a program that the computer can execute if the desired action is possible,
- explain why and give suggestions on alternative if the desired action is not possible,
- and request any clarifications as needed.

While such a capable system is extremely useful, it is also fairly unrealistic when the computer has a small and fixed action space. To fully understand an utterance can require knowing *how to do things* to various degrees.

(1) write an email to say no politely

(2) arrange the blocks to build a black cat

(3) go to the supermarket and buy fruit on sale

(4) add more examples to my thesis

(5) forward student emails to the appropriate TA

(6) write the show

Arbitrary knowledge and capabilities may also be required even if we just want to parse and represent utterances in the abstract. In an example given by Bar-Hillel (1964), "*The pen is in the box*" and "*The box is in the pen*", the meaning of "*pen*" is more likely to be a writing instrument and an animal enclosure, respectively. For another example, "*the patient left the operating room in good condition*" is unlikely to mean that the patient cleaned the room. *Winograd schemas* (Winograd, 1972) are more systematic examples constructed for demonstrating how arbitrary world knowledge might be needed to resolve pronouns:

(1) The city councilmen refused the demonstrators a permit because **they** [feared/advocated] violence.

(2) The trophy doesn't fit into the brown suitcase because **it** is too [large/small].

(3) Paul tried to call George on the phone, but **he** wasn't [available/successful].

(4) The drain is clogged with hair. **It** has to be [cleaned/removed].

In these examples, what the pronoun refers to depends on the choice of A vs. B in [A/B], and these cannot be resolved using the syntax of the language.

> Continued progress in limited domain and approximate approaches (including with speech). Very long term research is needed to get a handle on human-level natural language. (Terry Winograd)

2.1.2 Restricted NLIs

Today's computers still have a limited and rigid action space different from humans, and do not have sufficient common sense / world knowledge. To capture the utilitarian viewpoint of language understanding, we consider the restricted setting of communicating with a given software system or API. In this restricted setting, if the desired action described by natural language is outside of the system action space, then the computer is allowed to fail without explanation. Here are some examples which still fall within the scope of the restricted setting, but are not handled by current systems.

(1) call Bob every hour until he picks up

(2) lowercase every word in section headings of my thesis, except the first word

(3) book me the cheapest non-stop tickets from SF to NYC, that departs within 3 days after March 18, and takes off after 12noon.

(4) move my meeting with Alice to just after my meeting with Bob for the next 3 weeks

Most of these utterances are beyond the capability of current systems, but such utterances should be possible since they can be represented by a short program. The examples above can be spoken to a human assistant. and users might have strong intuitions on what human assistants can do for similar actions. Despite being intuitive to humans, computer systems may accomplish these actions in a different way from humans assistants. As long as actions are performed by a rigid computer system, the utterances still requires adaptation. As far as the computer is concerned, these utterances are similar to the utterances below, which is less intuitive to humans and the corresponding actions do not appear in the language of a fluent native speaker in their daily lives.

(1) put red cubes on all but the leftmost orange cube

(2) form a path made of yellow blocks from the top block to the red block

(3) start a new repo containing only the directory A, but keep the commit history

(4) find all files that ends with .java and print out the names of their public methods

(5) visualize mileage per gallon vs. horse power in a scatter plot

While unintuitive utterances might require more human adaptation, they are not very different from the more human-intuitive ones as far as the computer is concerned, as long as they are both convertible to short programs. For restricted NLIs, the language is restricted to the computer action space, and the users need to have some understanding of the action space in order to productively interact with the computer. Currently, these interactions are typically done via a programming language or GUIs. The popularity of `stackoverflow.com` shows that many people found it easier to write these utterances targeted at other human experts than to write the corresponding code or finding the GUI action. While still challenging, this is where semantic parsing is promising, and adaptive language interfaces can play an important role.

2.2 Approaches to NLIs

In this section, we discuss common approaches to NLIs using a dialogue from *2001: a space odyssey* as the running example.

Dave: Open the pod bay doors, HAL.
HAL: I'm sorry, Dave. I'm afraid I can't do that.
Dave: What's the problem?
HAL: I think you know what the problem is just as well as I do.
Dave: What are you talking about, HAL?
HAL: This mission is too important for me to allow you to jeopardize it.
...

2.2.1 Restricted responses

Dave: Open the pod bay doors, HAL.
HAL: press 1 for services in English, press 2 for services in Spanish
Dave: 1
HAL: Press 1 for navigation system, press 2 for life support system, . . . , press 5 for pod bays, . . . , press # for main menu

...

A simple, effective and commonly used solution is to restrict the responses that the human user is allowed to produce. This is commonly used in Interactive Voice Response (IVR) systems for telephone customer support, as well as in some text-based chat systems. By restricting the responses, we can workaround the language understanding problem altogether since the system only has a small number of states and transitions. By responding in natural language, this approach can still give the appearance of having a dialogue with computer.

2.2.2 Rule-based system

Dave: Open the pod bay doors.
Alexa: I'm sorry Dave. I'm afraid I can't do that, and we're not in space.[2]
Dave: What's the problem?

Action → pod bay doors': `respond`("I'm sorry Dave...")
Action → What's the **X**: λX.`respond`(I think you know what the **X** is just as well as I do.)
Action → open **X** : λX. X.open()

Rule-based systems are often representationally quite sophisticated and model pretty deep aspects of language. They typically use a semantic grammar that matches a predefined pattern in the utterance and specifies how this pattern should be interpreted (i.e. semantics). These rules can be compositional, and can implement elaborate features just like a programming language except the patterns follow natural language conventions more closely, and as a result are less uniform and more ambiguous than programming languages.

Early systems that were built and studied since the 60s were rule-based. For example, Bobrow (1964) solved word problems found in high school algebra books such as "*If the number of customers Tom gets is twice the square of 20% of the number of advertisements he runs, and the number of advertisements is 45, then what is the numbers of customers Tom gets?*" Woods et al. (1972) was an important early system that helped scientists query a database of moon rocks, answering questions like "*list all the rocks that contain chronite and ulvospinel*". Winograd (1972) described a system called SHRDLU, which could answer questions, perform actions, even learning from definitions in a blocksworld.

Person: DOES A STEEPLE
Computer interrupting: SORRY, I DON'T KNOW THE WORD "STEEPLE".

[2] Alexa Easter-egg

Person: A "STEEPLE" IS A STACK WHICH CONTAINS TWO GREEN CUBES AND A PYRAMID.
Computer: I UNDERSTAND.
Person: ARE THERE ANY STEEPLES NOW?
Computer: NO.
Person: BUILD ONE.
Computer: OK. (does it)
Person: CALL THE BIGGEST BLOCK "SUPERBLOCK".
Computer: OK.
Person: HAVE YOU PICKED SUPERBLOCK UP SINCE WE BEGAN?
Computer: YES.
Person: WHY DID YOU DROP IT?

These systems handled some complex linguistic phenomena that integrate syntax, semantics, and reasoning in an end-to-end application. While they implemented complex functionalities using a rule-based approach, these transcripts do not imply that similar utterances would also work. For example, in the MASQUE system "*What are the capitals of the countries bordering the Baltic and bordering Sweden?*" succeeds, while "*What are the capitals of the countries bordering the Baltic and Sweden?*" fails (Androutsopoulos et al., 1995). Such failures are less transparent and less debuggable compared to errors in programming languages. While it is tempting to think that better grammar engineering will lead to NLIs that are both powerful and have high coverage, satisfying coverage has not been realized in the half century following these early systems. Handling in-domain variations and generalizing to more open domain language remain challenging.

2.2.3 Learning-based systems

Machine learning techniques have already found success in many areas of natural language understanding such as machine translation, syntactic parsing, and semantic parsing. In this approach, we first collect a dataset of natural language utterances and their corresponding meaning annotations, then we use a learning algorithm on this dataset. However, learning from corpora is different from learning grounded language, which is the focus of this thesis.

Corpus-based

Dave: Open the pod bay doors, HAL.

HAL: I'm sorry, Dave. I'm afraid I can't do that.
Dave: What's the problem?
HAL: I think you know what the problem is just as well as I do.
Dave: What are you talking about, HAL?
HAL: **I do not know what you are talking about**

One learning setting is to collect some dialogue transcripts, and learn from the transcripts text. Recently, end-to-end deep learning has been popular for this task. For language interfaces, this is sensible for a chatterbot, where the goal is might be to keep the dialogue flowing. Recent neural models can be trained by predicting the next utterance in a corpus of dialogue data. Since it is less clear exactly what kind of information is contained in these datasets, and what kind of information can be learned by neural networks, this setting creates opportunities to obscure what the system actually understands. For example, Li et al. (2016b,a) observes several phenomena suggesting the lack of real understanding, such as a propensity to give generic response such as "*I do not know what you are talking about*" then to be stuck in a fixed loop, and inconsistent responses such as first responding "*I am 29*" then later responding "*I am 7*" when asked "*how old are you?*".

Even before any machine learning, Weizenbaum (1966) showed that it is possible to sustain conversations in a psychotherapy setting by using rule-based transformations on texts. Perhaps more apparent than in neural models, ELIZA had no hope of achieving real language understanding. However, generating natural language in response can give the appearance of language understanding and might even be helpful to human users. However, being helpful does not imply meaningful progress towards better language understanding. For example, journals, dictionaries, and text search are also helpful tools involving language, but takes no step towards language understanding.

Grounded

Crucially, *grounding* is missing if we only work with a corpus of text. For example, while the Ubuntu corpus (Lowe et al., 2015) contains many dialogue texts about technical issues in Ubuntu, it does not contain enough information to reconstruct Ubuntu or any hardware running Ubuntu that might be necessary to solve technical issues. Another example is movie subtitles, where the motion picture is missing. More importantly, world knowledge that is assumed of the audience is never explicitly mentioned in a movie. While it might be possible to infer some of such grounding information from language, the required learning capabilities probably have to exceed human—just imagine learning to solve Ubuntu problems without knowing what Ubuntu is, or learning another language by watching movies. Even if we are allowed to know about operating systems and another

human language, which helps narrowing down how to ground the language, the remaining learning task is still formidable to human learners.

In this thesis, we study the grounded setting for language learning and take a utilitarian view where understanding is solely evaluated by whether the computer can perform the action corresponding to the utterance. This setting make failures more apparent, and successes will at lead to better NLIs. In particular, we study how language understanding grounded forcibly to a particular action space where the computer has to learn and human has to adapt. Semantic parsing is our main approach and we discuss it in Chapter 3.

2.3 Utilities and dimensions of NLIs

There are a wide range of scenarios where NLIs are potentially useful. However, there are also areas where using a programming language / GUI is far superior. In this section, we break down NLIs along several dimensions in order to better discuss their utilities.

2.3.1 Human action space vs. computer action space

Common actions for humans might be rare or impossible for current computers. For example:

- require movement and vision: "*go to the supermarket and get eggs*"
- ability to socialize, human norms: "*let's catch up over coffee*"
- politeness and intention: "*please call to wake me up for my 9am flight, thanks!*"
- require human: "*send a happy birthday card to mom every year*"

In addition to physical action, a significant part of the human action space involves using language to affects the mental states of other humans.

At the same time, reference to computer actions might rarely occur in human language, or only became popular when the corresponding computer action became popular. For example, `chmod 777 blah` and `git commit -a` have no easy English equivalents; "*email Chris about lunch*" and "*google HAL 9000*" became part of human language after email and search became capabilities important to many people. NLIs is most suitable for the computer action space. especially where the actions are somewhat intuitive to humans. Targeting the human action space might be suitable for hybrid human-in-the-loop systems and chatterbots, than goal oriented settings. Affecting human

mental state may not require actual understanding, as demonstrated by ELIZA, which approximated a psychotherapist using rule-based text transformations.

2.3.2 Broad coverage vs. specific domain

Broad coverage language interfaces are best exemplified by the virtual assistants. While actual broad coverage is the goal, current virtual assistants are very limited in their ability to understand complex language, and can be extremely opaque to the users which utterances would work. Within a narrow domain, more complex language can be used, and this is exemplified by NLIDBs. Human language seem to cover both, where terms are invented and specialized for specific domains. While board coverage NLIs is more difficult and perhaps more impactful, NLIs for specific domains remain unsolved.

2.3.3 Occasional vs. volume users

Compared to a volume user, an occasional user of a computer system is not familiar with or forgetful of the functions of the system. A system that understands a standard natural language is probably most useful for the occasional user, where discoverability is important.

The volume user is familiar with the capabilities of the system, and might desire a specialized code language suitable for the domain. Given that the volume user spends a large amount of time interacting with the system, saving every keystroke and utterance is a meaningful. Programming languages designed to the capabilities of the system are likely most useful to the expert user.

An adaptive NLI could start with a less efficient but general language suitable for the occasional user, and then adapt to an efficient, perhaps code-like, language suitable for the volume user. This distinction is related to expert vs. novice users, but not quite the same because the occasional user might be a domain expert and know high-level capabilities of the system while the novice user might not even understand the system action space.

2.3.4 Ad hoc commands vs. software engineering

The dimension of ad hoc commands vs. software engineering is concerned about tolerance for ambiguity/inefficiency vs. tolerance of high human effort. One important factor is how often the action will be repeated.

In cases where the human is actively communicating with some computer system and wants context dependent actions performed, the human may issue a very specific command that will only

run once. This is different from expertise, for example, one can be a bash expert and issue many ad hoc commands in bash. It is more important to minimize human effort rather than precision when we do not need to handle all edge cases nor be valid in all context.

In contrast to ad hoc commands, complex software requires components with clean interfaces and abstractions (e.g. Comparable, Stream, map/reduce, etc.) that would allow even more complex components to be built on top of simpler components. Besides abstractions, some bottleneck subroutines that are used repeatedly can require good runtime performance—e.g. the code in Figure 2.1. It would be sensible to spend much labor to optimize these subroutines. In these situations it is worthwhile to adapt almost entirely to the computer, and pay a high human effort to optimize both the abstractions and runtime performance. This case is likely unsuitable for an adaptive language interface.

```
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;
        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;
        i  = 0x5f3759df - ( i >> 1 );
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );
        return y;
}
```

Figure 2.1: Code for fast inverse square root, which is hard to express in anything resembling English.

Chapter 3

Semantic parsing

3.1 Overview

Semantic parsing is the main approach and default way of thinking used in this thesis and we provide selected background on *executable semantic parsing* in this chapter.[1] Semantic parsing is the task of converting natural language utterances to logical forms, which is traditionally a linguistically/logically motivated representation of meaning. However, we are not too concerned about the principles behind logical representation and happy to think of the logical form as an arbitrary program that can be *executed* to yield some action (sometimes called denotation). This captures a utilitarian definition of understanding language suitable for natural language interfaces, where the system is said to understand the utterance if it produces the correct action. For example:

Context: 

Utterance: *add a cyan block to red blocks*

Program: `add(hascolor(red), cyan)`

Action: 

[1] some background material is reproduced from (Liang, 2016)

Context:	knowledge of mathematics
Utterance:	*What is the largest prime less than 10?*
Program:	$\max(\texttt{Primes} \cap (-\infty, 10))$
Action:	7
Context:	`some string to be edited`
Utterance:	*uppercase the first letter of every word*
Program:	`s/\S+/\u$&/g`
Action:	`Some String To Be Edited`

These examples require deep language understanding, and it seems difficult for a computer to directly arrive at the final answer without some form of representation that allows the computer to reason about the domain. Because ultimately computer have to execute, we can aim at producing these programs without thinking about if language is fundamentally symbolic, or if there is a general purpose semantic representation.

Semantic parsing is rooted in formal semantics, pioneered by logician Richard Montague (Montague, 1973), who famously argued that there is "no important theoretical difference between natural languages and the artificial languages of logicians." Semantic parsing, by residing in the practical realm, is more exposed to the differences between natural language and logic, but it inherits two general insights from formal semantics: *model theory* and *compositionality*.

Model theory. The first idea is *model theory*, which states that expressions (e.g., `primes`) are mere symbols which only obtain their meaning or denotation (e.g., $\{2, 3, 5, \ldots\}$) by executing the expression with respect to a model, or in our terminology, a context. This property allows us to factor out the understanding of language (semantic parsing) from world knowledge (execution). Indeed, one can understand the utterance "*Add a cyan block to top of every red block.*" or "*What is the largest prime less than 10?*" without actually computing the answer. Model theory allows for the decoupling of language understanding from execution.

Principle of compositionality. The second idea is *compositionality*, a principle often attributed to Gottlob Frege, which states that the denotation of an expression is defined recursively as a function of the denotation of its subexpressions. This compositionality is what allows us to have a succinct characterization of meaning for a combinatorial range of possible utterances. For example, `(or (color red) (row 3))` represents the union of blocks having color red and blocks in the 3rd row. `(or (color red) (row 3))` is composed of the subexpressions `(color red)`, `(row 3)`

and $\lambda XY.$ `(or X Y)`. In particular, the denotation of `or` only depends on the denotation of Y, as opposed to the text of Y `(row 3)`. So compositionality requires that

$$[\![\texttt{(or (color red) (row 3))}]\!]_w = [\![\texttt{(or (color red) (or (row 3) (row 3)))}]\!]_w.$$

In logic and programming language, compositionality is usually assumed and rarely needs to be made explicit. However, natural language utterances often violate a strict form of compositionality. In natural language semantics, Barbara (1995) states the principle of compositionality as

> The meaning of a compound expression is a function of the meanings of its parts and of the syntactic rule by which they are combined.

For example, idioms usually cannot be analyzed compositionally—e.g. kick the bucket, barking up the wrong tree. *Winograd schema* requires pronouns to be resolved based on the meaning of the whole sentence—e.g. The city councilmen refused the demonstrators a permit because they [feared/advocated] violence.

This should not be surprising, because otherwise we should be able to write down some composition rules that can process natural language. Because this messier form of compositionality, data has a useful role to play in handling ambiguous and non-compositional utterances. Even with the help of data, however, semantic parsing would work better if a compositional analysis of natural language can transfer over to the compositional logical form, which then allows for generalization to unseen data.

3.1.1 Early systems

Like programming languages, logical forms have played a foundational role in natural language understanding systems since their genesis in the 1960s. Early examples included LUNAR, a natural language interface into a database about moon rocks (Woods et al., 1972), and SHRDLU, a system that could both answer questions and perform actions in a toy blocks world environment (Winograd, 1972).

For their time, these systems were significant achievements. They were able to handle fairly complex linguistic phenomena and integrate syntax, semantics, and reasoning in an end-to-end application. For example, SHRDLU was able to process "*Find a block which is taller than the one you are holding and put it into the box.*" However, as the systems were based on hand-crafted rules, it became increasingly difficult to generalize beyond the narrow domains and handle the intricacies of general language.

3.1.2 Rise of machine learning

In the early 1990s, influenced by the successes of statistical techniques in the neighboring speech recognition community, the field of NLP underwent a statistical revolution. Machine learning offered a new paradigm: Collect *examples* of the desired input-output behavior and then fit a statistical model to these examples. The simplicity of this paradigm coupled with the increase in data and computation allowed machine learning to prevail. What fell out of favor was not only rule-based methods, but also the natural language understanding problems. In the statistical NLP era, much of the community's attention turned to tasks—documentation classification, part-of-speech tagging, and syntactic parsing—which fell short of full end-to-end understanding. Even question answering systems relied less on understanding and more on a shallower analysis coupled with a large collection of unstructured text documents (Brill et al., 2002), typified by the TREC competitions.

3.1.3 Statistical semantic parsing

The spirit of deep understanding was kept alive by researchers in statistical semantic parsing (Zelle and Mooney, 1996; Miller et al., 1996; Wong and Mooney, 2007; Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010). A variety of different semantic representations and learning algorithms were employed, but all of these approaches relied on having a labeled dataset of natural language utterances paired with annotated logical forms, for example:

Context: 

Utterance: *add a cyan block to red blocks*

Program: `add(hascolor(red), cyan)`

3.1.4 Weak supervision

Over the last few years, two exciting developments have really spurred interest in semantic parsing. The first is reducing the amount of supervision from annotated logical forms to answers (Clarke et al., 2010; Liang et al., 2011):

Context: 

Utterance: *add a cyan block to red blocks*

Action: 

This form of supervision is much easier to obtain via crowdsourcing. Although the logical forms are not observed, they are still modeled as latent variables, which must be inferred from the answer. This results in a more difficult learning problem, but Liang et al. (2011) showed that it is possible to solve it without degrading accuracy.

3.1.5 Scaling up

The second development is the scaling up of semantic parsers to more complex domains. Previous semantic parsers had only been trained on limited domains such as US geography, but the creation of broad-coverage knowledge bases such as Freebase (Bollacker et al., 2008) set the stage for a new generation of semantic parsers for question answering. Initial systems required annotated logical forms (Cai and Yates, 2013), but soon, systems became trainable from answers (Berant et al., 2013; Kwiatkowski et al., 2013; Berant and Liang, 2014). Semantic parsers have even been extended beyond fixed knowledge bases to semi-structured tables (Pasupat and Liang, 2015). With the ability to learn semantic parsers from question-answer pairs, it is easy to collect datasets via crowdsourcing. As a result, semantic parsing datasets have grown by an order of magnitude.

In addition, semantic parsers have been applied to a number of applications outside question answering: robot navigation (Tellex et al., 2011; Artzi and Zettlemoyer, 2013), identifying objects in a scene (Matuszek et al., 2012; Krishnamurthy and Kollar, 2013), converting natural language to regular expressions (Kushman and Barzilay, 2013), and many others.

3.1.6 Neural models

Like other task in NLP, neural models have been successful in semantic parsing. The scarcity of data somewhat limits their success, and some way to be more data efficient is needed for these models to work as well as previous models. For example, Jia and Liang (2016) proposed data recombination, where they used rules mirroring the logical forms to generate more fake-data; Dong and Lapata

(2016) build some compositional structure directly into the model; Iyer et al. (2017) learns from user feedback. Although their performance does not exceed the state-of-the-art yet, neural models have the advantage of being very general, where not much has to change to apply to other domains. For example, Konstas et al. (2017) achieved good performance on AMR parsing with less adaptation to the domain than Artzi and Zettlemoyer (2015).

3.1.7 Data challenge

Perhaps more important than models, data remains a major challenge for semantic parsing. From a high level, semantic parsing is similar to machine translation in that both translates from one language to another. However, compared to the 10-100 millions input-output pairs for common languages such as English/French, and English/Chinese, current semantic parsing datasets are rather small. For example, Geo880, Regexp824, and freebase917 (Zelle and Mooney, 1996; Kushman and Barzilay, 2013; Cai and Yates, 2013) all have less than 1000 labels, while ATIS and WebQuestions (Zettlemoyer and Collins, 2007; Berant et al., 2013) have a few thousand, and WikiTableQuestions (Pasupat and Liang, 2015) have 22033 question/answer pairs.

One reason is the lack of an universal semantic representation, so it is unclear how to direct effort at getting more data. For example, all these datasets use a different logical language and we would not know what labels to collect. One idea is to use general paraphrasing models to map input utterances to the "canonical utterances" of logical forms (Berant and Liang, 2014; Wang et al., 2015). This reduces semantic parsing to a text-only problem for which there is much more data and resources. One could also use domain-general logical forms that capture the basic predicate-argument structures of sentences (Kwiatkowski et al., 2013). Abstract meaning representation (AMR) (Banarescu et al., 2013) is one popular representation backed by an extension linguistic annotation effort. However, solving downstream understanding tasks still require additional work and remains to be shown how general representations are helpful in our utilitarian criteria of understanding, which requires an action to be performed based on the language.

Given that domain specific representations seem inescapable if we want executable semantic parsing, there will be many utterances outside of the scope. In Chapter 1, we argued that language use have to adapt to system capabilities. This means that datasets have to account for the system capabilities. One benefit of our interactive systems is that we were able to get relatively large, adapted datasets (30k-50k utterances labeled by logical forms) that take system capabilities into account.

$$p_\theta(z \mid x, c) \propto \exp(\phi(x, z, c) \cdot \theta)$$

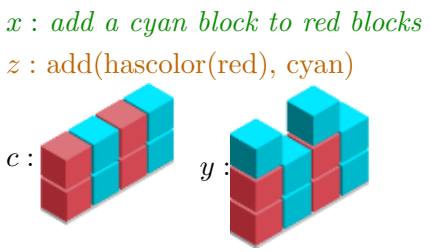


Figure 3.1: A natural language understanding problem where the goal is to map an utterance x in a context c to an action y.

3.2 Framework

3.2.1 Setup

Given an *utterance* x in a *context* c, output the desired *action* y. Figure 3.1 shows the setup for a question answering application, in which case x is a question, c is a knowledge base, and y is the answer. In blocks world, x is a command, c represents current set of blocks, and y is the desired set of blocks after x. In a robotics application, x is a command, c represents the robot's environment, and y is the desired sequence of actions to be carried by the robot (Tellex et al., 2011). To build such a system, assume that we are given a set of n examples $\{(x_i, c_i, y_i)\}_{i=1}^n$. We would like to use these examples to train a model that can generalize to new unseen utterances and contexts.

3.2.2 Semantic parsing components

We focus on a statistical semantic parsing approach to the above problem, where the key is to posit an intermediate *logical form* z that connects x and y. Specifically, z captures the semantics of the utterance x, and it also executes to the action y (in the context of c). In our running example, z would be `add(hascolor(red), cyan)`. Our semantic parsing framework consists of the following five components (see Figure 3.2):

1. **Executor**: computes the denotation (action) $y = [\![z]\!]$ given a logical form z and context c. This defines the semantic representation (logical forms along with their denotations).
2. **Grammar**: a set of rules G that produces $D(x, c)$, a set of candidate derivations of logical forms.

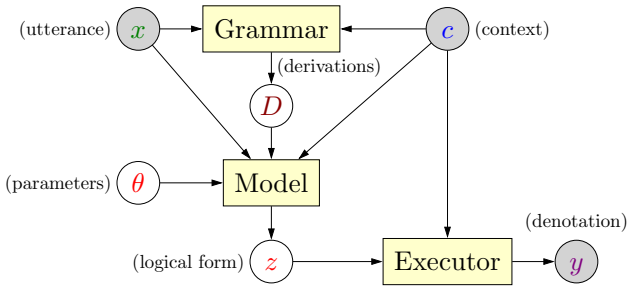


Figure 3.2: Semantic parsing framework depicting the executor, grammar, and model. The parser and learner are algorithmic components that are responsible for generating the logical form z and parameters θ, respectively.

3. **Model**: specifies a distribution $p_\theta(d \mid x, c)$ over derivations d parameterized by θ.
4. **Parser**: searches for high probability derivations d under the model p_θ.
5. **Learner**: estimates the parameters θ (and possibly rules in G) given training examples $\{(x_i, c_i, y_i)\}_{i=1}^n$.

We now instantiate each of these components for our running example: add a cyan block to red blocks

Executor

Let the semantic representation be the language of mathematics, and the executor is the standard interpretation, where the interpretations of predicates (e.g., `hascolor(red)`) are given by c and denotes the set of blocks that has color red.

Grammar

The grammar G connects utterances to possible *derivations* of logical forms. Formally, the grammar is a set of rules of the form $\alpha \Rightarrow \beta$.[2] Here is a simple grammar for our running example:

We start with the input utterance and repeatedly apply rules in G. A rule $\alpha \Rightarrow \beta$ can be applied if some span of the utterance matches α, in which case a derivation over the same span with a new syntactic category and logical form according to β is produced. Here is one possible derivation (call it d_1) for our running example:

[2] The standard way context-free grammar rules are written is $\beta \to \alpha$. Because our rules build logical forms, reversing the arrow is more natural.

Rule	Semantics	Description
Set	`all()`	(R1) all stacks
Color	`cyan|brown|red|orange`	(R2) primitive color
Color → Set	`hascolor`(c)	(R3) stacks whose top block has color c
Set → Set	`not`(s)	(R4) all stacks except those in s
Set → Set	`leftmost|rightmost`(s)	(R5) leftmost/rightmost stack in s
Set Color → Act	`add`(s,c)	(R6) add block with color c on each stack in s
Set → Act	`remove`(s)	(R7) remove the topmost block of each stack in s

Table 3.1: Example grammar for blocks world

$$\begin{array}{llllll} \textit{put} & \textit{cyan} & \textit{on} & \textit{red} & \textit{blocks} & \\ & \text{---(R2)---} & & \text{---(R2)---} & & \\ & \texttt{cyan} & & \texttt{red} & & \\ & & & \text{---------(R3)---------} & & \\ & \texttt{cyan} & & \texttt{hascolor(red)} & & \\ \multicolumn{6}{c}{\text{---------------------(R6)---------------------}} \\ \multicolumn{6}{c}{\text{ROOT}[\texttt{add(hascolor(red),cyan)}]} \end{array} \tag{3.1}$$

For example, applying (R2) produces category Color and logical form *red* then applying (R3) on the span "*red blocks*" produces category Set with logical from `hascolor(red)`. We stop when we produce the designated ROOT category over the entire utterance. In general, there could be exponentially many derivations, and multiple derivations can generate the same logical form.

Model

The model scores the set of candidate derivations generated by the grammar. A common choice used by virtually all existing semantic parsers are log-linear models (generalizations of logistic regressions). In a log-linear model, define a *feature vector* $\phi(x,c,d) \in \mathbb{R}^F$ for each possible derivation d. We can think of each feature as casting a vote for various derivations d based on some coarse property of the derivation. For example, define $F = 7$ features, each counting the number of times a given grammar rule is invoked in d, so that $\phi(x,c,d_1) = [1,1,1,1,1,0,1]$ and $\phi(x,c,d_2) = [1,1,1,1,1,0,1,1]$.

Next, let $\theta \in \mathbb{R}^F$ denote the *parameter vector*, which defines a weight for each feature representing how reliable that feature is. Their weighted combination $\text{score}(x,c,d) = \phi(x,c,d) \cdot \theta$ represents

how good the derivation is. We can exponentiate and normalize these scores to obtain a distribution over derivations:

$$p_\theta(d \mid x, c) = \frac{\exp(\text{score}(x, c, d))}{\sum_{d' \in D(x,c)} \exp(\text{score}(x, c, d'))}. \tag{3.2}$$

If $\theta = [0, 0, 0, 0, +1, -1, 0]$, then p_θ would assign probability $\frac{\exp(1)}{\exp(1)+\exp(-1)} \approx 0.88$ to d_1 and ≈ 0.12 to d_2.

Parser

Given a trained model p_θ, the parser (approximately) computes the highest probability derivation(s) for an utterance x under p_θ. Assume the utterance x is represented as a sequence of tokens (words). A standard approach is to use a *chart parser*, which recursively builds derivations for each span of the utterance. Specifically, for each category A and span $[i:j]$ (where $0 \le i < j \le \text{length}(x)$), we loop over the applicable rules in the grammar G and apply each one to build new derivations of category A over $[i:j]$. For binary rules—those of the form $BC \Rightarrow A$ such as (R4), we loop over split points k (where $i < k \le j$), recursively compute derivations $B[z_1]$ over $[i:k]$ and $C[z_2]$ over $[k:j]$, and combine them into a new derivation $A[z]$ over $[i:j]$, where z is determined by the rule; for example, $z = z_1 \cap z_2$ for (R4). The final derivations for the utterance are collected in the ROOT category over span $[0:\text{length}(x)]$.

The above procedure would generate all derivations, which could be exponentially large. Generally, we only wish to compute compute the derivations with high probability under our model p_θ. If the features of p_θ were to *decompose* as a sum over the rule applications in d—that is, $\phi(x, c, d) = \sum_{(r,i,j) \in d} \phi_{\text{rule}}(x, c, r, i, j)$, then we could use dynamic programming: For each category A over $[i:j]$, compute the highest probability derivation. However, in executable semantic parsing, feature decomposition isn't sufficient, since during learning, we also need to incorporate the constraint that the logical form executes to the true denotation ($\mathbb{I}[[\![d.z]\!] = y]$); see (3.6) below. To maintain exact computation in this setting, the dynamic programming state would need to include the entire logical form $d.z$, which is infeasible, since there are exponentially many logical forms. Therefore, *beam search* is generally employed, where we keep only the K sub-derivations with the highest model score based on only features of the sub-derivations. Beam search is not guaranteed to return the K highest scoring derivations, but it is often an effective heuristic.

Learner

While the parser turns parameters into derivations, the learner solves the inverse problem. The dominant paradigm in machine learning is to set up an objective function and optimize it. A standard principle is to maximize the likelihood of the training data $\{(x_i, c_i, y_i)\}_{i=1}^n$. An important point is that we don't observe the correct derivation for each example, but only the action y_i, so we must consider all derivations d whose logical form $d.z$ satisfy $[\![d.z]\!]_{c_i} = y_i$. This results in the log-likelihood of the observed action y_i:

$$\mathscr{O}_i(\theta) \stackrel{\text{def}}{=} \log \sum_{\substack{d \in D(x_i, c_i) \\ [\![d.z]\!]_{c_i} = y_i}} p_\theta(d \mid x_i, c_i). \tag{3.3}$$

The final objective is then simply the sum across all n training examples:

$$\mathscr{O}(\theta) \stackrel{\text{def}}{=} \sum_{i=1}^n \mathscr{O}_i(\theta), \tag{3.4}$$

The simplest approach to maximize $\mathscr{O}(\theta)$ is to use *stochastic gradient descent* (SGD), an iterative algorithm that takes multiple passes (e.g., say 5) over the training data and makes the following update on example i:

$$\theta \leftarrow \theta + \eta \nabla \mathscr{O}_i(\theta), \tag{3.5}$$

where η is a step size that governs how aggressively we want to update parameters (e.g., $\eta = 0.1$). In the case of log-linear models, the gradient has a nice interpretable form:

$$\nabla \mathscr{O}_i(\theta) = \sum_{d \in D(x_i, c_i)} (q(d) - p_\theta(d \mid x_i, c_i)) \phi(x_i, c_i, d), \tag{3.6}$$

where $q(d) \propto p_\theta(d \mid x_i, c_i) \mathbb{I}[[\![d.z]\!]_{c_i} = y_i]$ is the model distribution p_θ over derivations d, but restricted to ones consistent with y_i. The gradient pushes θ to put more probability mass on q and less on p_θ. For example, if p_θ assigns probabilities $[0.2, 0.4, 0.1, 0.3]$ to four derivations and the middle two derivations are consistent, then q assigns probabilities $[0, 0.8, 0.2, 0]$.

The objective function $\mathscr{O}(\theta)$ is not concave, so SGD is at best guaranteed to converge to a local optimum, not a global one. Another problem is that we cannot enumerate all derivations $D(x_i, c_i)$ generated by the grammar, so we approximate this set with the result of beam search, which yields K candidates (typically $K = 200$); p_θ is normalized over this set. Note that this candidate set depends

on the current parameters θ, resulting a heuristic approximation of the gradient $\nabla \mathscr{O}_i$.

We have covered the components of a semantic parsing system. Observe that the components are relatively loosely coupled: The executor is concerned purely with what we want to express independent of how it would be expressed in natural language. The grammar describes how candidate logical forms are constructed from the utterance but does not provide algorithmic guidance nor specify a way to score the candidates. The model focuses on a particular derivation and defines features that could be helpful for predicting accurately. The parser and the learner provide algorithms largely independent of semantic representations. This modularity allows us to improve each component in isolation.

3.3 Prerequisite background

Having toured the components of a semantic parsing system, we discuss some background particularly relevant to this thesis. For more general information on the components of semantic parsing, refer to section 3 of Liang (2016).

3.3.1 Lambda dependency-based semantics (DCS)

We make use of *lambda dependency-based semantics (DCS)* (Liang, 2013) in Chapter 5, which can be viewed as syntactic sugar for lambda calculus. Consider "*how many primes are less than 10?*" which can be represented as $\texttt{count}(\lambda x.\texttt{prime}(x) \wedge \texttt{less}(x, 10))$ in lambda calculus, where the λ operator can be thought of as constructing a set of all x that satisfy the condition; in symbols, $[\![\lambda x.f(x)]\!] = \{x : [\![f(x)]\!] = \texttt{true}\}$. Note that `count` is a higher-order functions that takes a function as an argument. In lambda DCS, it would be $\texttt{count}(\texttt{prime} \sqcap (\texttt{less}.10))$, where the constant 10 represent $\lambda x.(x = 10)$, the intersection operator $z_1 \sqcap z_2$ represents $\lambda x.z_1(x) \wedge z_2(x)$, and the join operator $r.z$ represents $\lambda x.\exists y.r(x,y) \wedge z(y)$.

Lambda DCS is "lifted" in the sense that operations combine functions from objects to truth values (think sets) rather than truth values. As a result, lambda DCS logical forms partially eliminate the need for variables. Noun phrases in natural language (e.g., "*prime less than 10*") also denote sets. Thus, lambda DCS arguably provides a transparent interface with natural language.

We used lambda DCS to represent sets in Chapter 5 where the core language exposes lambda DCS in a transparent way. In particular, 'has R Z' is the lambda DCS join $r.z$, which means $\lambda x.\exists y.r(x,y) \wedge z(y)$. 'R of Z' is the lambda DCS reverse join $[!r].z$, which means $\lambda x.\exists y.r(y,x) \wedge z(y)$. These joins can be chained together where explicit variables can be avoided as long as the chain of

joins is in a tree structure. This way, common set operations corresponds well with natural language, for example: 'has color red or yellow' means all voxels with color red or color yellow which is `(or (color red) (color yellow))` 'has row [col of this]' is `(row (!col this))`; 'left and not has color red' is `(and (!left this) (not (color red)))`.

3.3.2 Grammar

For Chapter 4, the grammar is just to define a set of candidate derivations for each utterance and context (see Table 3.1). Note that this is in contrast to a conventional notion of a grammar in linguistics, where the goal is to precisely characterize the set of valid sentences and interpretations. Because we will learn a statistical model over the derivations generated by the grammar anyway, the grammar can be simple and coarse.

Floating grammar rules

(3.1) shows a floating derivation using the grammar in Table 3.1. Note that grammar rules in Table 3.1 such as (R4) and (R6) only specify categories, as opposed to (R4') and (R6') which also specify terminal tokens. These floating rules (R4 and R6) merely specified which categories can be combined, and how to combine them. Order and adjacency are both ignored, whereas their "anchored" counterparts (R4' and R6') require both order and terminal tokens to be matched exactly for the rule to apply. This way, we were able to cover "*put cyan on red blocks*" in (3.1) without having needing a rule like R6".

Rule	**Semantics**	**Description**
Set $\rightarrow$ Set	$\texttt{not}(s)$	(R4) all stacks except those in s
'not' Set $\rightarrow$ Set	$\texttt{not}(s)$	(R4') all stacks except those in s
Set Color $\rightarrow$ Act	$\texttt{add}(s,c)$	(R6) add block with color c to set s
'add' Color 'to' Set $\rightarrow$ Act	$\texttt{add}(s,c)$	(R6') add block with color c to set s
'put' Color 'on' Set $\rightarrow$ Act	$\texttt{add}(s,c)$	(R6") add block with color c to set s

While this relaxation may seem linguistically blasphemous, recall that the purpose of the grammar is to merely deliver a set of logical forms, so floating rules are quite sensible provided we can keep the set of logical forms under control. Of course, since the grammar is so flexible, even more nonsensical logical forms are generated, so we must lean heavily on the features to score the derivations properly. When the logical forms are simple and we have strong type constraints, this strategy can be quite effective (Berant and Liang, 2014; Wang et al., 2015; Pasupat and Liang, 2015).

3.3.3 Learning

In learning, we are given examples of utterance-context-response triples (x, c, y). There are two aspects of learning: inducing the grammar rules and estimating the model parameters. It is important to remember that practical semantic parsers do not do everything from scratch, and often the hard-coded grammar rules are as important as the training examples. First, some lexical rules that map named entities (e.g., [*paris* $\Rightarrow$ `ParisFrance`]), dates, and numbers are generally assumed to be given (Zettlemoyer and Collins, 2005), though we need not assume that these rules are perfect (Liang et al., 2011). These rules are also often represented implicitly (Liang et al., 2011; Berant et al., 2013).

Grammar induction

How the rest of the grammar is handled varies across approaches. In CCG-style approach, inducing lexical rules is an important part of learning. In Zettlemoyer and Collins (2005), a procedure called GENLEX is used to generate candidate lexical rules from a utterance-logical form pair (x, z). A more generic induction algorithm based on higher-order unification does not require any initial grammar (Kwiatkowski et al., 2010). (Wong and Mooney, 2007) use machine translation ideas to induce a synchronous grammar (which can also be used to generate utterances from logical forms). Grammar induction handles most of learning in Chapter 5 where high-precision rules are induced from user definitions.

In approaches that learn from denotations y (Liang et al., 2011; Berant et al., 2013; Pasupat and Liang, 2015), an initial crude grammar is used to generate candidate logical forms, and rest of the work is done by the features. This approach is used in Chapter 4.

Parameter learning

As we discussed earlier, parameter estimation can be performed by stochastic gradient descent on the log-likelihood; similar objectives based on max-margin are also possible (Liang and Potts, 2015). It can be helpful to also add an L_1 regularization term $\lambda \|\theta\|_1$, which encourages feature weights to be zero, which produces a more compact model that generalizes better (Berant and Liang, 2014). In addition, one can use AdaGrad (Duchi et al., 2010), which maintains a separate step size for each feature. This can improve stability and convergence.

Discussion

Chapter 5 uses a starting programming language that allows users to define complex actions that cannot be handled by a floating grammar. This is because 10-100 rule applications are needed to reach one of these actions, and beam search will never find them. As a result, learning via grammar induction does the heavy lifting in Chapter 5 while parameter learning plays a supportive role. In contrast, lexical rules would not be flexible enough to handle the arbitrary language used in Chapter 4. Here, parameter learning with a floating grammar does the heavy lifting, and allowed us to learn starting from scratch. In general, learning via grammar induction and parameter learning complement each other.

Chapter 4

Learning language games through interaction

This chapter is based on (Wang et al., 2016), where the goal is to study a setting where learning starts from scratch and tailored to each user. It is inspired by Wittgenstein's language games: a human wishes to accomplish some task (e.g., achieving a certain configuration of blocks), but can only communicate with a computer, who performs the actual actions (e.g., removing all red blocks). The computer initially knows nothing about language and therefore must learn it from scratch through interaction, while the human adapts to the computer's capabilities. We created a game called SHRDLURN in a blocks world and collected interactions from 100 people playing it. First, we analyze the humans' strategies, showing that using compositionality and avoiding synonyms correlates positively with task performance. Second, we compare computer strategies, showing that modeling pragmatics on a semantic parsing model accelerates learning for more strategic players.

4.1 Introduction

Wittgenstein (1953) famously said that *language derives its meaning from use*, and introduced the concept of *language games* to illustrate the fluidity and purpose-orientedness of language. He described how a builder B and an assistant A can use a primitive language consisting of four words—*'block'*, *'pillar'*, *'slab'*, *'beam'*—to successfully communicate what block to pass from A to B. This is only one such language; many others would also work for accomplishing the cooperative goal.

> "2....Let us imagine a language ...The language is meant to serve for communication

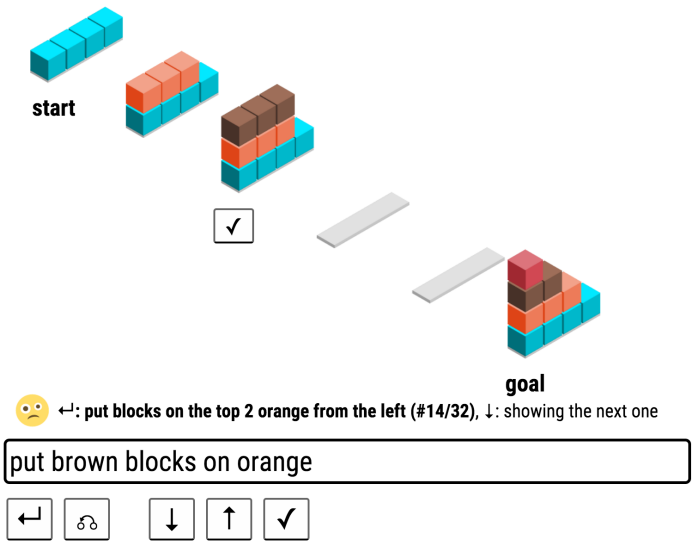


Figure 4.1: The SHRDLURN game: In this collaborative game between the human and the computer, the objective is to transform the start state into the goal state. The human types in an utterance, and the computer (which does not know the goal state) tries to interpret the utterance and perform the corresponding action. The computer initially knows nothing about the language, but through the human's feedback, learns the human's language while making progress towards the game goal. (Wittgenstein, 1953)

> between a builder A and an assistant B. A is building with building-stones; there are blocks, pillars, slabs and beams. B has to pass the stones, and that in the order in which A needs them. For this purpose they use a language consisting of the words 'block', 'pillar', 'slab', 'beam'. A calls them out; –B brings the stone which he has learnt to bring at such-and-such a call. – Conceive of this as a complete primitive language."

This chapter operationalizes and explores the idea of language games in a learning setting, which we call *interactive learning through language games* (ILLG). In the ILLG setting, the two parties do not initially speak a common language, but nonetheless need to collaboratively accomplish a goal. Specifically, we created a game called SHRDLURN,[1] in homage to the seminal work of Winograd (1972). As shown in Figure 4.1, the objective is to transform a start state into a goal state, but the only action the human can take is entering an utterance. The computer parses the utterance and produces a ranked list of possible interpretations according to its current model. The human scrolls through the list and chooses the intended one, simultaneously advancing the state of the blocks and providing feedback to the computer. Both the human and the computer wish to reach the goal state

[1] Demo: `http://shrdlurn.sidaw.xyz`

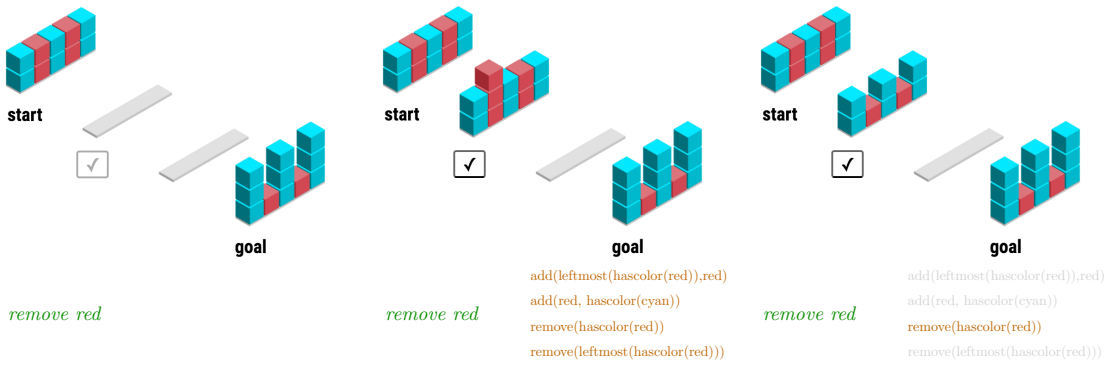


Figure 4.2: left: the human user produces an utterance, middle: the computer responds with a ranked list of actions, right: the human selects the desired action.

(only known to the human) with as little scrolling as possible. For the computer to be successful, it has to learn the human's language quickly over the course of the game, so that the human can accomplish the goal more efficiently. Conversely, the human must also accommodate the computer, at least partially understanding what it can and cannot do.

We model the computer in the ILLG as a semantic parser (Section 4.3), which maps natural language utterances (e.g., *'remove red'*) into logical forms (e.g., `remove(with(red))`). The semantic parser has no seed lexicon and no annotated logical forms, so it just generates many candidate logical forms. Based on the human's feedback, it performs online gradient updates on the parameters corresponding to simple lexical features.

During development, it became evident that while the computer was eventually able to learn the language, it was learning less quickly than one might hope. For example, after learning that *'remove red'* maps to `remove(with(red))`, it would think that *'remove cyan'* also mapped to `remove(with(red))`, whereas a human would likely use mutual exclusivity to rule out that hypothesis (Markman and Wachtel, 1988). We therefore introduce a pragmatics model in which the computer explicitly reasons about the human, in the spirit of previous work on pragmatics (Golland et al., 2010; Frank and Goodman, 2012; Smith et al., 2013). To make the model suitable for our ILLG setting, we introduce a new online learning algorithm. Empirically, we show that our pragmatic model improves the online accuracy by 8% compared to our best non-pragmatic model on the 10 most successful players (Section 4.5.3).

What is special about the ILLG setting is the real-time nature of learning, in which the human

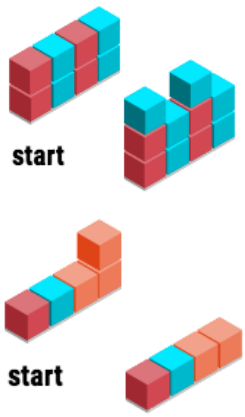
also learns and adapts to the computer. While the human can teach the computer any language—English, Arabic, Polish, a custom programming language—a good human player will choose to use utterances that the computer is more likely to learn quickly. In the parlance of communication theory, the human *accommodates* the computer (Giles, 2008; Ireland et al., 2011). Using Amazon Mechanical Turk, we collected and analyzed around 10k utterances from 100 games of SHRDLURN. We show that successful players tend to use compositional utterances with a consistent vocabulary and syntax, which matches the inductive biases of the computer (Section 4.5.2). In addition, through this interaction, many players adapt to the computer by becoming more consistent, more precise, and more concise.

On the practical side, natural language systems are often trained once and deployed, and users must live with their imperfections. We believe that studying the ILLG setting will be integral for creating adaptive and customizable systems, especially for resource-poor languages and new domains where starting from close to scratch is unavoidable.

4.2 Setting

We now describe the interactive learning of language games (ILLG) setting formally. There are two players, the human and the computer. The game proceeds through a fixed number of levels. In each level, both players are presented with a starting state $s \in \mathscr{Y}$, but only the human sees the goal state $t \in \mathscr{Y}$. (e.g. in SHRDLURN, $\mathscr{Y}$ is the set of all configurations of blocks). The human transmits an utterance x (e.g., *'remove red'*) to the computer. The computer then constructs a ranked list of candidate actions $Z = [z_1, \ldots, z_K] \subseteq \mathscr{Z}$ (e.g., `remove(with(red))`, `add(with(orange))`, etc.), where $\mathscr{Z}$ is all possible actions. For each $z_i \in Z$, it computes $y_i = [\![z_i]\!]_s$, the successor state from executing action z_i on state s. The computer returns to the human the ordered list $Y = [y_1, \ldots, y_K]$ of successor states. The human then chooses y_i from the list Y (we say the computer is *correct* if $i = 1$). The state then updates to $s = y_i$. The level ends when $s = t$, and the players advance to the next level.

Since only the human knows the goal state t and only the computer can perform actions, the only way for the two to play the game successfully is for the human to somehow encode the desired action in the utterance x. However, we assume the two players do not have a shared language, so the human needs to pick a language and teach it to the computer. As an additional twist, the human does not know the exact set of actions $\mathscr{Z}$ (although they might have some preconception of the



add(hascolor(red), cyan)

remove(rightmost(all()))
remove(rightmost(hascolor(orange)))

Figure 4.3: several different actions z might all produce the same state s

computer's capabilities).[2] Finally, the human only sees the outcomes of the computer's actions, which might be consistent with multiple actions For example, in Figure 4.3, *'remove the leftmost'* happens to be the same as *'remove the leftmost orange'*.

We expect the game to proceed as follows: In the beginning, the computer does not understand what the human is saying and performs arbitrary actions. As the computer obtains feedback and learns, the two should become more proficient at communicating and thus playing the game. Herein lies our key design principle: *language learning should be necessary for the players to achieve good game performance*.

In order to do well in this game, the player has to choose what to say, and what to accept as correct. If an example is too hard, the player can also choose to skip it and play a new one. For the simpler levels, one action can be sufficient to reach the goal, but multiple actions is allowed as well, as in Figure 4.1. The game score is based on the number of extra inspections that the player has to make. In particular, the best score is obtained if the computer performs the right action in its first attempt.

4.2.1 SHRDLURN

We explore this setting through a cooperative game called SHRDLURN where our objective is to obtain a particular set of blocks in the goal state. Only the human knows what the goal is, and only the computer can manipulate the blocks. So the way to finish this game is for the human player to teach the computer to understand a language, and it would also help if the player tries to figure out,

[2]This is often the case when we try to interact with a new software system or service before reading the manual.

Rule	Semantics	Description
Set	`all`()	all stacks
Color	`cyan|brown|red|orange`	primitive color
Color → Set	`with`(c)	stacks whose top block has color c
Set → Set	`not`(s)	all stacks except those in s
Set → Set	`leftmost|rightmost`(s)	leftmost/rightmost stack in s
Set Color → Act	`add`(s, c)	add block with color c on each stack in s
Set → Act	`remove`(s)	remove the topmost block of each stack in s

Table 4.1: The formal grammar defining the compositional action space $\mathscr{Z}$ for SHRDLURN. We use c to denote a Color, and s to denote a Set. For example, one action that we have in SHRDLURN is: *'add an orange block to all but the leftmost brown block'* $\mapsto$ `add(not(leftmost(with(brown))),orange)`.

at least approximately, what the computer is capable of.Figure 4.1

To make progress, the player first think about which action z needs to be taken in order to reach goal state, then he types an utterance x. The computer ranks actions according to the player utterance, and show the player the resulting blocks y by performing each action z on the starting state s. The player then inspects these outcome blocks y in order of model likelihood, and tells the computer what is the correct outcome he has in mind.

Let us now describe the details of our specific game, SHRDLURN. Each state $s \in \mathscr{Y}$ consists of stacks of colored blocks arranged in a line (Figure 4.1), where each stack is a vertical column of blocks. The actions $\mathscr{Z}$ are defined compositionally via the grammar in Table 4.1. Each action either adds to or removes from a set of stacks, and a set of stacks is computed via various set operations and selecting by color. For example, the action `remove`(`leftmost`(`with`(`red`))) removes the top block from the leftmost stack whose topmost block is red. The compositionality of the actions gives the computer non-trivial capabilities. Of course, the human must teach a language to harness those capabilities, while not quite knowing the exact extent of the capabilities. The actual game proceeds according to a curriculum, where the earlier levels only need simpler actions with fewer predicates.

We designed SHRDLURN in this way for several reasons. First, visual block manipulations are intuitive and can be easily crowdsourced, and it can be fun as an actual game that people would play. Second, the action space is designed to be compositional, mirroring the structure of natural language. Third, many actions z lead to the same successor state $y = [\![z]\!]_s$; e.g., the *'leftmost stack'* might coincide with the *'stack with red blocks'* for some state s and therefore an action involving either one would result in the same outcome. Since the human only points out the correct y, the computer must grapple with this indirect supervision, a reflection of real language learning. For

the computer to be successful, it has to extrapolate to many unseen commands from less than 100 examples and learn interactively. The task is impossible without compositionality and models that can take advantage of it.

4.3 Semantic parsing model

Following Zettlemoyer and Collins (2005) and most recent work on semantic parsing, we use a log-linear model over logical forms (actions) $z \in \mathscr{Z}$ given an utterance x:

$$p_\theta(z \mid x) \propto \exp(\theta^\mathsf{T} \phi(x,z)), \tag{4.1}$$

where $\phi(x,z) \in \mathbb{R}^d$ is a feature vector and $\theta \in \mathbb{R}^d$ is a parameter vector. The denotation y (successor state) is obtained by executing z on a state s; formally, $y = [\![z]\!]_s$.

Features. Our features are n-grams (including skip-grams) conjoined with tree-grams on the logical form side. Specifically, on the utterance side (e.g., *'stack red on orange'*), we use unigrams (*'stack'*, *, *), bigrams (*'red'*, *'on'*, *), trigrams (*'red'*, *'on'*, *'orange'*), and skip-trigrams (*'stack'*, *, *'on'*). On the logical form side, features corresponds to the predicates in the logical forms and their arguments. For each predicate h, let $h.i$ be the i-th argument of h. Then, we define *tree-gram* features $\psi(h,d)$ for predicate h and depth $d = 0,1,2,3$ recursively as follows:

$$\begin{aligned}
\psi(h,0) &= \{h\}, \\
\psi(h,d) &= \{(h,i,\psi(h.i,d-1)) \mid i = 1,2,3\}.
\end{aligned}$$

The set of all features is just the cross product of utterance features and logical form features. For example, if $x =$ *'enlever tout'* and $z =$ `remove(all())`, then features include:

(*'enlever'*, `all`)	(*'tout'*, `all`)
(*'enlever'*, `remove`)	(*'tout'*, `remove`)
(*'enlever'*, (`remove`, 1, `all`))	
(*'tout'*, (`remove`, 1, `all`))	

Note that we do not model an explicit alignment or derivation compositionally connecting the utterance and the logical form, in contrast to most traditional work in semantic parsing (Zettlemoyer and Collins, 2005; Wong and Mooney, 2007; Liang et al., 2011; Kwiatkowski et al., 2010; Berant et al., 2013), instead following a looser model of semantics similar to (Pasupat and Liang, 2015).

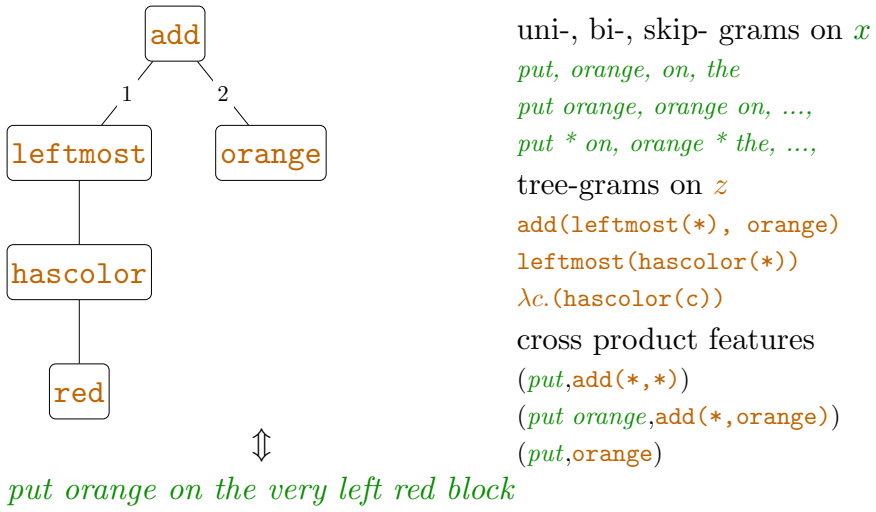


Figure 4.4: an illustration of the cross product features

brown
hascolor(brown)
leftmost(hascolor(brown))
diff(all(),leftmost(hascolor(brown))
remove(diff(all(),leftmost(hascolor(brown)))

Figure 4.5: search over logical forms in increasing length

Modeling explicit alignments or derivations is only computationally feasible when we are learning from annotated logical forms or have a seed lexicon, since the number of derivations is much larger than the number of logical forms. In the ILLG setting, neither are available. Therefore, we follow Pasupat and Liang (2015) and define features on the cross product of utterance and logical forms and hope that the spurious features such as (*'enlever'*, `all`) will be downweighted with sufficient examples.

Generation/parsing. We generate logical forms from smallest to largest using beam search. similar to the floating parser of Pasupat and Liang (2015). Specifically, for each size $n = 1, \ldots, 8$, we construct a set of logical forms of size n (with exactly n predicates) by combining logical forms of smaller sizes according to the grammar rules in Table 4.1. For each n, we keep the 100 logical forms z with the highest score $\theta^\top \phi(x, z)$ according to the current model θ. Let Z be the set of logical forms on the final beam, which contains logical forms of all sizes n. During training, due to pruning at intermediate sizes, Z is not guaranteed to contain the logical form that obtains the observed state

y. To mitigate this effect, we use a curriculum so that only simple actions are needed in the initial levels, giving the human an opportunity to teach the computer about basic terms such as colors first before moving to larger composite actions.

The system executes all of the logical forms on the final beam Z, and orders the resulting denotations y by the maximum probability of any logical form that produced it.[3]

Learning. When the human provides feedback in the form of a particular y, the system forms the following loss function:

$$\ell(\theta, x, y) = -\log p_\theta(y \mid x, s) + \lambda ||\theta||_1, \tag{4.2}$$

$$p_\theta(y \mid x, s) = \sum_{z:[\![z]\!]_s = y} p_\theta(z \mid x). \tag{4.3}$$

Then it makes a single gradient update using AdaGrad (Duchi et al., 2010), which maintains a per-feature step size.

4.4 Modeling pragmatics

In our initial experience with the semantic parsing model described in Section 4.3, we found that it was able to learn reasonably well, but lacked a reasoning ability that one finds in human learners. To illustrate the point, consider the beginning of a game when $\theta = 0$ in the log-linear model $p_\theta(z \mid x)$. Suppose that human utters *'remove red'* and then identifies $z_{\texttt{rm-red}} = \texttt{remove}(\texttt{with}(\texttt{red}))$ as the correct logical form. The computer then performs a gradient update on the loss function (4.2), upweighting features such as (*'remove'*, `remove`) and (*'remove'*, `red`).

Next, suppose the human utters *'remove cyan'*. Note that $z_{\texttt{rm-red}}$ will score higher than all other formulas since the (*'remove'*, `red`) feature will fire again. While statistically justified, this behavior fails to meet our intuitive expectations for a smart language learner. Moreover, this behavior is not specific to our model, but applies to any statistical model that simply tries to fit the data without additional prior knowledge about the specific language. While we would not expect the computer to magically guess *'remove cyan'* $\mapsto \texttt{remove}(\texttt{with}(\texttt{cyan}))$, it should at least push down the probability of $z_{\texttt{rm-red}}$ because $z_{\texttt{rm-red}}$ intuitively is already well-explained by another utterance *'remove red'*.

[3] We tried ordering based on the sum of the probabilities (which corresponds to marginalizing out the logical form), but this had the degenerate effect of assigning too much probability mass to y being the set of empty stacks, which can result from many actions.

	$z_{\texttt{rm-red}}$	$z_{\texttt{rm-cyan}}$	$z_3, z_4, \ldots$
	$p_\theta(z \mid x)$		
'remove red'	0.8	0.1	0.1
'remove cyan'	**0.6**	0.2	0.2
	$S(x \mid z)$		
'remove red'	0.57	0.33	0.33
'remove cyan'	0.43	0.67	0.67
	$L(z \mid x)$		
'remove red'	0.46	0.27	0.27
'remove cyan'	0.24	**0.38**	**0.38**

Table 4.2: **Example.** Suppose the computer saw one example of *'remove red'* $\mapsto z_{\texttt{rm-red}}$, and then the human utters *'remove cyan'*. **top**: the literal listener, $p_\theta(z \mid x)$, mistakingly chooses $z_{\texttt{rm-red}}$ over $z_{\texttt{rm-cyan}}$. **middle**: the pragmatic speaker, $S(x \mid z)$, assigns a higher probability to to *'remove cyan'* given $z_{\texttt{rm-cyan}}$; **bottom**: the pragmatic listener, $L(z \mid x)$ correctly assigns a lower probability to $z_{\texttt{rm-red}}$ where $p(z)$ is uniform.

This phenomenon, *mutual exclusivity*, was studied by Markman and Wachtel (1988). They found that children, during their language acquisition process, reject a second label for an object and treat it instead as a label for a novel object.

The pragmatic computer. To model mutual exclusivity formally, we turn to probabilistic models of pragmatics (Golland et al., 2010; Frank and Goodman, 2012; Smith et al., 2013; Goodman and Lassiter, 2015), which operationalize the ideas of Grice (1975). The central idea in these models is to treat language as a cooperative game between a speaker (human) and a listener (computer) as we are doing, but where the listener has an explicit model of the speaker's strategy, which in turn models the listener. Formally, let $S(x \mid z)$ be the speaker's strategy and $L(z \mid x)$ be the listener's strategy. The speaker takes into account the literal semantic parsing model $p_\theta(z \mid x)$ as well as a prior over utterances $p(x)$, while the listener considers the speaker $S(x \mid z)$ and a prior $p(z)$:

$$S(x \mid z) \propto \left(p_\theta(z \mid x)p(x)\right)^\beta, \tag{4.4}$$

$$L(z \mid x) \propto S(x \mid z)p(z), \tag{4.5}$$

where $\beta \geq 1$ is a hyperparameter that sharpens the distribution (Smith et al., 2013). The computer would then use $L(z \mid x)$ to rank candidates rather than p_θ. Note that our pragmatic model only affects the ranking of actions returned to the human and does not affect the gradient updates of the model p_θ.

	$z_{\texttt{rm r}}$	$z_{\texttt{add c}}$	$z_{\texttt{rm c}}$	$z_{\texttt{add r}}$
	score = # features $\times \eta$			
'remove red'	6 η	0η	1η	1η
'add cyan'	0 η	6η	1η	1η
'remove cyan'	3 η	3η	2η	2η
	$L_0(z\|x) \propto \exp(s)$			
'remove red'	0.74	0.06	0.10	0.10
'add cyan'	0.06	0.74	0.10	0.10
'remove cyan'	0.3	0.3	0.2	0.2
	$S_1(x\|z)$			
'remove red'	0.67	0.06	0.25	0.25
'add cyan'	0.06	0.67	0.25	0.25
'remove cyan'	0.27	0.27	0.5	0.5

Table 4.3: **Example.** Suppose the computer observed the two examples above the dashed line, and did a batch update with a learning rate of $\eta = \log(3/2)$, and for simplicity we used $p(z|x;\theta = 0) \approx 0$. The features used are unigrams in the utterance matched with unigrams and bigrams of predicates. So we get 6 features with non-zero weights for $u =$ *'remove'*, *'red'*, we get $u \mapsto$`Red`, $u \mapsto$`remove`, and $u \mapsto$`remove`$,1,$`Red`. **top**: the score for each utterance after the first update, **mid**: probabilities assigned by the literal listener $L_0(z|x)$, **bot**: the pragmatic speaker.

Let us walk through a simple example to see the effect of modeling pragmatics. Table 4.2 shows that the literal listener $p_\theta(z \mid x)$ assigns high probability to $z_{\texttt{rm-red}}$ for both *'remove red'* and *'remove cyan'*. Assuming a uniform $p(x)$ and $\beta = 1$, the pragmatic speaker $S(x \mid z)$ corresponds to normalizing each column of p_θ. Note that if the pragmatic speaker wanted to convey $z_{\texttt{rm-cyan}}$, there is a decent chance that they would favor *'remove cyan'*. Next, assuming a uniform $p(z)$, the pragmatic listener $L(z \mid x)$ corresponds to normalizing each row of $S(x \mid z)$. The result is that conditioned on *'remove cyan'*, $z_{\texttt{rm-cyan}}$ is now more likely than $z_{\texttt{rm-red}}$, which is the desired effect. Table 4.3 show another example, where we consider a specific features and the third utterance.

The pragmatic listener models the speaker as a cooperative agent who behaves in a way to maximize communicative success. Certain speaker behaviors such as avoiding synonyms (e.g., not *'delete cardinal'*) and using a consistent word ordering (e.g, not *'red remove'*) fall out of the game theory.[4] For speakers that do not follow this strategy, our pragmatic model is incorrect, but as we get more data through game play, the literal listener $p_\theta(z \mid x)$ will sharpen, so that the literal listener and the pragmatic listener will coincide in the limit.

[4] Of course, synonyms and variable word order occur in real language. We would need a more complex game compared to SHRDLURN to capture this effect.

$\forall z, C(z) \leftarrow 0$
$\forall z, Q(z) \leftarrow \varepsilon$
repeat
  receive utterance x from human $L(z \mid x) \propto \frac{P(z)}{Q(z)} p_\theta(z \mid x)^\beta$
  send human a list Y ranked by $L(z \mid x)$
  receive $y \in Y$ from human
  $\theta \leftarrow \theta - \eta \nabla_\theta \ell(\theta, x, y)$
  $Q(z) \leftarrow Q(z) + p_\theta(z \mid x)^\beta$
  $C(z) \leftarrow C(z) + p_\theta(z \mid x, [\![z]\!]_s = y)$
  $P(z) \leftarrow \frac{C(z)+\alpha}{\sum_{z':C(z')>0}\left(C(z')+\alpha\right)}$
until *game ends*

Algorithm 1: Online learning algorithm that updates the parameters of the semantic parser θ as well as counts C, Q required to perform pragmatic reasoning.

Online learning with pragmatics. To implement the pragmatic listener as defined in (4.5), we need to compute the speaker's normalization constant $\sum_x p_\theta(z \mid x)p(x)$ in order to compute $S(x \mid z)$ in (4.4). This requires parsing all utterances x based on $p_\theta(z \mid x)$. To avoid this heavy computation in an online setting, we propose Algorithm 1, where some approximations are used for the sake of efficiency. First, to approximate the intractable sum over all utterances x, we only use the examples that are seen to compute the normalization constant $\sum_x p_\theta(z \mid x)p(x) \approx \sum_i p_\theta(z \mid x_i)$. Then, in order to avoid parsing all previous examples again using the current parameters for each new example, we store $Q(z) = \sum_i p_{\theta_i}(z \mid x_i)^\beta$, where θ_i is the parameter after the model updates on the i^{th} example x_i. While θ_i is different from the current parameter θ, $p_\theta(z \mid x_i) \approx p_{\theta_i}(z \mid x_i)$ for the relevant example x_i, which is accounted for by both θ_i and θ.

In Algorithm 1, the pragmatic listener $L(z \mid x)$ can be interpreted as an importance-weighted version of the sharpened literal listener p_θ^β, where it is downweighted by $Q(z)$, which reflects which z's the literal listener prefers, and upweighted by $P(z)$, which is just a smoothed estimate of the actual distribution over logical forms $p(z)$. By construction, Algorithm 1 is the same as (4.4) except that it uses the normalization constant Q based on stale parameters θ_i after seeing example, and it uses samples to compute the sum over x. Following (4.5), we also need $p(z)$, which is estimated by $P(z)$ using add-α smoothing on the counts $C(z)$. Note that $Q(z)$ and $C(z)$ are updated *after* the model parameters are updated for the current example.

Lastly, there is a small complication due to only observing the denotation y and not the logical form z. We simply give each consistent logical form $\{z \mid [\![z]\!]_s = y\}$ a pseudocount based on the

model: $C(z) \leftarrow C(z) + p_\theta(z \mid x, [\![z]\!]_s = y)$ where $p_\theta(z \mid x, [\![z]\!]_s = y) \propto \exp(\theta^\top \phi(x,z))$ for $[\![z]\!]_s = y$ (0 otherwise).

Compared to prior work where the setting is specifically designed to require pragmatic inference, pragmatics arises naturally in ILLG. We think that this form of pragmatics is the most important during learning, and becomes less important if we had more data. Indeed, if we have a lot of data and a small number of possible zs, then $L(z|x) \approx p_\theta(z|x)$ as $\sum_x p_\theta(z|x)p(x) \to p(z)$ when $\beta = 1$.[5] However, for semantic parsing, we would not be in this regime even if we have a large amount of training data. In particular, we are nowhere near that regime in SHRDLURN, and most of our utterances / logical forms are seen only once, and the importance of modeling pragmatics remains.

4.5 Experiments

4.5.1 Setting

Data. Using Amazon Mechanical Turk (AMT), we paid 100 workers 3 dollars each to play SHRDLURN. In total, we have 10223 utterances along with their starting states s. Of these, 8874 utterances are labeled with their denotations y; the rest are unlabeled, since the player can try any utterance without accepting an action. 100 players completed the entire game under identical settings. We deliberately chose to start from scratch for every worker, so that we can study the diversity of strategies that different people used in a controlled setting.

Each game consists of 50 blocks tasks divided into 5 levels of 10 tasks each, in increasing complexity. Each level aims to reach an end goal given a start state. Each game took on average 89 utterances to complete.[6] It only took 6 hours to complete these 100 games on AMT and each game took around an hour on average according to AMT's *work time* tracker (which does not account for multi-tasking players). The players were provided minimal instructions on the game controls. Importantly, we gave no example utterances in order to avoid biasing their language use. Around 20 players were confused and told us that the instructions were not clear and gave us mostly spam utterances. Fortunately, most players understood the setting and some even enjoyed SHRDLURN as reflected by their optional comments:

- *That was probably the most fun thing I have ever done on mTurk.*

[5] Technically, we also need p_θ to be *well-specified*.

[6] This number is not 50 because some block tasks need multiple steps and players are also allowed to explore without reaching the goal.

- *Wow this was one mind bending games [sic].*
- *This is SO SO cool. I wish there were a way I could better contribute because this research seems to be just insanely interesting and worthwhile.*
- *That was very fun, please email me if you have any other hits like this in the future :)*

Metrics. We use the *number of scrolls* as a measure of game performance for each player. For each example, the number of scrolls is the position in the list Y of the action selected by the player. It was possible to complete this version of SHRDLURN by scrolling (all actions can be found in the first 125 of Y)—22 of the 100 players failed to teach an actual language, and instead finished the game mostly by scrolling. Let us call them *spam players*, who usually typed single letters, random words, digits, or random phrases (e.g. *'how are you'*). Overall, spam players had to scroll a lot: 21.6 scrolls per utterance versus only 7.4 for the non-spam players.

4.5.2 Human strategies

Some example utterances can be found in Table 4.4. Most of the players used English, but vary in their adherence to conventions such as use of determiners, plurals, and proper word ordering. 5 players invented their own language, which are more precise, more consistent than general English. One player used Polish, and another used Polish notation (bottom of Table 4.4).

Overall, we find that many players adapt in ILLG by becoming more consistent, less verbose, and more precise, even if they used standard English at the beginning. For example, some players became more consistent over time (e.g. from using both *'remove'* and *'discard'* to only using *'remove'*). In terms of verbosity, removing function words like determiners as the game progresses is a common adaptation. In each of the following examples from different players, we compare an utterance that appeared early in the game to a similar utterance that appeared later: *'Remove the red ones'* **became** *'Remove red.'*; *'add brown on top of red'* **became** *'add orange on red'*; *'add red blocks to all red blocks'* **became** *'add red to red'*; *'dark red'* **became** *'red'*; one player used *'the'* in all of the first 20 utterances, and then never used *'the'* in the last 75 utterances.

Players also vary in precision, ranging from overspecified (e.g. *'remove the orange cube at the left'*, *'remove red blocks from top row'*) to underspecified or requiring context (e.g. *'change colors'*, *'add one blue'*, *'Build more blocus'*, *'Move the blocks fool'*,*'Add two red cubes'*). We found that some players became more precise over time, as they gain a better understanding of ILLG.

Most players use utterances that actually do not match our logical language in Table 4.1, even the successful players. In particular, numbers are often used. While some concepts always have

Most successful players (1st–20th)		
rem cy pos 1, stack or blk pos 4, rem blk pos 2 thru 5, rem blk pos 2 thru 4, stack bn blk pos 1 thru 2, fill bn blk, stack or blk pos 2 thru 6, rem cy blk pos 2 fill rd blk **(3.01)**	remove the brown block, remove all orange blocks, put brown block on orange blocks, put orange blocks on all blocks, put blue block on leftmost blue block in top row **(2.78)**	Remove the center block, Remove the red block, Remove all red blocks, Remove the first orange block, Put a brown block on the first brown block, Add blue block on first blue block **(2.72)**
Average players (21th–50th)		
reinsert pink, take brown, put in pink, remove two pink from second layer, Add two red to second layer in odd intervals, Add five pink to second layer, Remove one blue and one brown from bottom layer **(9.17)**	remove red, remove 1 red, remove 2 4 orange, add 2 red, add 1 2 3 4 blue, emove 1 3 5 orange, add 2 4 orange, add 2 orange, remove 2 3 brown, add 1 2 3 4 5 red, remove 2 3 4 5 6, remove 2, add 1 2 3 4 6 red **(8.37)**	move second cube, double red with blue, double first red with red, triple second and fourth with orange, add red, remove orange on row two, add blue to column two, add brown on first and third **(7.18)**
Least successful players (51th–)		
holdleftmost, holdbrown, holdleftmost, blueonblue, brownonblue1, blueonorange, holdblue, holdorange2, blueonred2 , holdends1, holdrightend, hold2, orangeonorangerightmost **(14.15)**	'add red cubes on center left, center right, far left and far right', 'remove blue blocks on row two column two, row two column four', remove red blocks in center left and center right on second row **(12.6)**	laugh with me, red blocks with one aqua, aqua red alternate, brown red red orange aqua orange, red brown red brown red brown, space red orange red, second level red space red space red space **(14.32)**
Spam players ($\sim$ 85th–100)		
next, hello happy, how are you, move, gold, build goal blocks, 23,house, x, run, xav, d, j, xcv, dulicate goal (21.7)		
Most interesting		
usuń brązowe klocki, postaw pomarańczowy klocek na pierwszym klocku, postaw czerwone klocki na pomarańczowych, usuń pomarańczowe klocki w górnym rzędzie	rm scat + 1 c, + 1 c, rm sh, + 1 2 4 sh, + 1 c, - 4 o, rm 1 r, + 1 3 o, full fill c, rm o, full fill sh, - 1 3, full fill sh, rm sh, rm r, + 2 3 r, rm o, + 3 sh, + 2 3 sh, rm b, - 1 o, + 2 c,	mBROWN,mBLUE,mORANGE RED+ORANGEˆORANGE, BROWN+BROWNm1+BROWNm3, ORANGE +BROWN +ORANGEˆm1+ ORANGEˆm3 + BROWNˆˆ2 + BROWNˆˆ4

Table 4.4: Example utterances, along with the average number of scrolls for that player in parentheses. Success is measured by the number of scrolls, where the more successful players need less scrolls. 1) The 20 most successful players tend to use consistent and concise language whose semantics is similar to our logical language. 2) Average players tend to be slightly more verbose and inconsistent (left and right), or significantly different from our logical langauge (middle). 3) Reasons for being unsuccessful vary. Left: no tokenization, middle: used a coordinate system and many conjunctions; right: confused in the beginning, and used a language very different from our logical language.

the same effect in our blocks world (e.g. *'first block'* means `leftmost`), most are different. More concretely, of the top 10 players, 7 used numbers of some form and only 3 players matched our logical language. Some players who did not match the logical language performed quite well nevertheless. One possible explanation is because the action required is somewhat constrained by the logical language and some tokens can have unintended interpretations. For example, the computer can correctly interpret numerical positional references, as long as the player only refers to the leftmost and rightmost positions. So if the player says *'rem blk pos 4'* and *'rem blk pos 1'*, the computer can interpret *'pos'* as `rightmost` and interpret the bigram (*'pos'*, *'1'*) as `leftmost`. On the other hand, players who deviated significantly by describing the desired state declaratively (e.g. *'red orange red'*, *'246'*) rather than using actions, or a coordinate system (e.g. *'row two column two'*) performed poorly. Although players do not have to match our logical language exactly to perform well, being similar is definitely helpful.

Compositionality. As far as we can tell, all players used a compositional language; no one invented unrelated words for each action. Interestingly, 3 players did not put spaces between words. Since we assume monomorphemic words separated by spaces, they had to do a lot of scrolling as a result (e.g., 14.15 with utterances like *'orangeonorangerightmost'*).

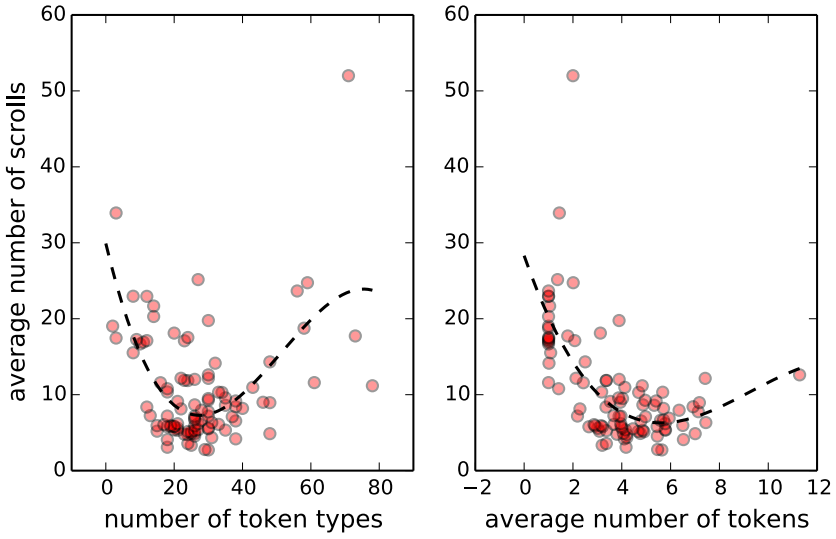


Figure 4.6: the number of scrolls needed by token types and token counts.

More quantitatively, we can consider the number of token types used, and the average number of tokens per utterance. They are somewhat predictive of the game performance as measured by

number of scrolls. Figure 4.6 shows a very noisy picture of this. We find that players who used 20-40 different token types, and who use utterances of 3-6 tokens long tend to perform better. This number of token types is comparable to the 12 predicates in Table 4.1 if we account for functional words and plural forms.

4.5.3 Computer strategies

We now present quantitative results on how quickly the computer can learn, where our goal is to achieve high accuracy on new utterances as we make just a single pass over the data. The number of scrolls used to evaluate player is sensitive to outliers and not as intuitive as accuracy. Instead, we consider *online accuracy*, described as follows. Formally, if a player produced T utterances $x^{(j)}$ and labeled them $y^{(j)}$, then

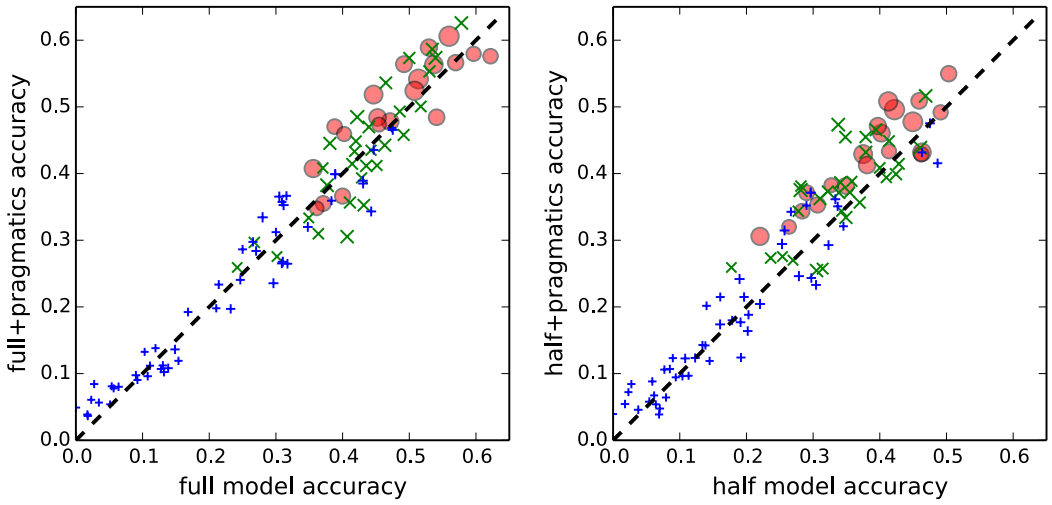
$$\text{online accuracy} \stackrel{\text{def}}{=} \frac{1}{T}\sum_{j=1}^{T} \mathbb{I}\left[y^{(j)} = [\![z^{(j)}]\!]_{s^{(j)}}\right],$$

where $z^{(j)} = \arg\max_z p_{\theta^{(j-1)}}(z|x^{(j)})$ is the model prediction based on the previous parameter $\theta^{(j-1)}$. Note that the online accuracy is defined with respect to the player-reported labels, which only corresponds to the actual accuracy if the player is precise and honest. This is not true for most spam players.

	players ranked by # of scrolls			
Method	top 10	top 20	top 50	all 100
memorize	25.4	24.5	22.5	17.6
half model	38.7	38.4	36.0	27.0
half + prag	43.7	42.7	39.7	29.4
full model	48.6	47.8	44.9	33.3
full + prag	52.8	49.8	45.8	33.8

Table 4.5: Average online accuracy under various settings. memorize: featurize entire utterance and logical form non-compositionally; half model: featurize the utterances with unigrams, bigrams, and skip-grams but conjoin with the entire logical form; full model: the model described in Section 4.3; +prag: the models above, with our online pragmatics algorithm described in Section 4.4. Both compositionality and pragmatics improve accuracy.

Compositionality. To study the importance of compositionality, we consider two baselines. First, consider a non-compositional model (*memorize*) that just remembers pairs of complete utterance and logical forms. We implement this using indicator features on (x,z), e.g., (‘*remove all the red*



(a) (b)

Figure 4.7: Pragmatics improve online accuracy. In these plots, each marker is a player. red o: players who ranked 1–20 in terms of minimizing number of scrolls, green x: players 20–50; blue +: lower than 50 (includes spam players). Marker sizes correspond to player rank, where better players are depicted with larger markers. **4.7a**: online accuracies with and without pragmatics on the full model; **4.7b**: same for the half model.

blocks', $z_{\texttt{rm-red}}$), and use a large learning rate. We do this because we still need to deal with denotation labels and conflicting player labels even if we just want to memorize examples. Second, we consider a model (*half*) that treats utterances compositionally with unigrams, bigrams, and skip-trigrams features, but the logical forms are regarded as non-compositional, so we have features such as $(\textit{'remove'}, z_{\texttt{rm-red}})$, $(\textit{'red'}, z_{\texttt{rm-red}})$, etc.

Our full model has double the online accuracy compared to memorization (Table 4.5).

Table 4.5 shows that the full model (Section 4.3) significantly outperforms both the *memorize* and *half* baselines. The learning rate $\eta = 0.1$ is selected via cross validation, and we used $\alpha = 1$

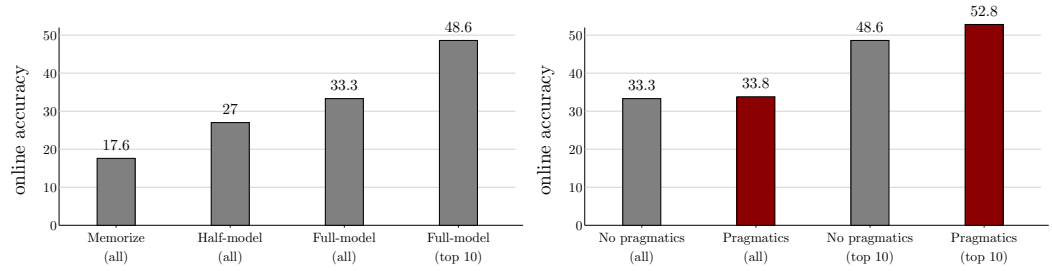


Figure 4.8: left: comparison with baselines, right: pragmatics help top players

and $\beta = 3$ following Smith et al. (2013).

Pragmatics. Next, we study the effect of pragmatics on online accuracy. Figure 4.7 shows that modeling pragmatics helps successful players (e.g., top 10 by number of scrolls) who use precise and consistent languages. Interestingly, our pragmatics model did not help and can even hurt the less successful players who are less precise and consistent. This is expected behavior: the pragmatics model assumes that the human is cooperative and behaving rationally. For the bottom half of the players, this assumption is not true, in which case the pragmatics model is not useful.

4.6 Related Work and Discussion

Our work connects with a broad body of work on grounded language, in which language is used in some environment as a means towards some goal. Examples include playing games (Branavan et al., 2009, 2010; Reckman et al., 2010) interacting with robotics (Tellex et al., 2011, 2014), and following instructions (Vogel and Jurafsky, 2010; Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013) Semantic parsing utterances to logical forms, which we leverage, plays an important role in these settings (Kollar et al., 2010; Matuszek et al., 2012; Artzi and Zettlemoyer, 2013).

What makes this work unique is our new interactive learning of language games (ILLG) setting, in which a model has to learn a language from *scratch* through interaction. While online gradient descent is frequently used, for example in semantic parsing (Zettlemoyer and Collins, 2007; Chen, 2012), we using it in a truly online setting, taking one pass over the data and measuring online accuracy (Cesa-Bianchi and Lugosi, 2006).

To speed up learning, we leverage computational models of pragmatics (Jäger, 2008; Golland et al., 2010; Frank and Goodman, 2012; Smith et al., 2013; Vogel et al., 2013). The main difference is these previous works use pragmatics with a trained base model, whereas we learn the model online. Monroe and Potts (2015) uses learning to improve the pragmatics model. In contrast, we use pragmatics to speed up the learning process by capturing phenomena like mutual exclusivity (Markman and Wachtel, 1988). We also differ from prior work in several details. First, we model pragmatics in the online learning setting where we use an online update for the pragmatics model. Second, unlikely the reference games where pragmatic effects plays an important role by design, SHRDLURN is not specifically designed to require pragmatics. The improvement we get is mainly due to players trying to be consistent in their language use. Finally, we treat both the utterance and the logical forms as featurized compositional objects. Smith et al. (2013) treats utterances (i.e.

words) and logical forms (i.e. objects) as categories; Monroe and Potts (2015) used features, but also over flat categories; (Goodman and Lassiter, 2015) used pragmatics with compositional semantics.

Looking forward, we believe that the ILLG setting is worth studying and has important implications for natural language interfaces. Today, these systems are trained once and deployed. If these systems could quickly adapt to user feedback in real-time as in this work, then we might be able to more readily create systems for resource-poor languages and new domains, that are customizable and improve through use.

4.7 Appendix

In the year following the initial release, the online demo of SHRDLURN received 26k+ examples in 1599 sessions through 130k interactions. We show some examples from these, divided into English-like, code-like, and other languages.

English-like

(1) add brown on the top unless the rightmost

(2) add a brown block on top of the right-most red block

(3) move all blocks but middle

(4) Not the brown block!

(5) add red on top of first brown,

(6) add blue blocks on top of left 3 blocks

(7) drop orange 1

(8) drop orange not left not right

Code-like languages

(1) add blo 1 bro

(2) - 1 br - 4 br - 6 br

(3) lift 1 3 5

(4) + 1 2 3 4 5 r

(5) Add x x o x o x red block

(6) rem ora blo

(7) add blo 6 pin

(8) add blo 134 bl

(9) + 1 2 3 4 5 r

(10) smaz 1 a 2 a 3 a 5

Other natural languages.

(1) 一番奥にオレンジを置く
(2) 一番右の赤を消す
(3) 只保留桔黄色的方块
(4) 去掉 蓝色 方块
(5) 在蓝色块上面加一层橙色块
(6) quita el primer bloque por la derecha
(7) છોડો વાદળી 0 1
(8) બધા વાદળી દૂર
(9) છોડો નારંગી 1 4
(10) retire les blocs bleus
(11) quitar ultimo cubo rojop
(12) ostav na kazhdiy goluboy blok vo vtorom ryadu po korichnevomu bloku

Reproducibility

All code, data, and experiments for this chapter are available on the CodaLab platform:
`https://worksheets.codalab.org/worksheets/0x9fe4d080bac944e9a6bd58478cb05e5e`
The client side code is here:
`https://github.com/sidaw/shrdlurn/tree/acl16-demo`
and a demo: `http://shrdlurn.sidaw.xyz`

Chapter 5

Naturalizing a programming language via interactive learning

This chapter is based on Wang et al. (2017), which proposes a more powerful way to learn through interaction compared to Chapter 4. Due to the exponential growth of the number of logical forms, learning from selecting alternatives cannot scale well to a more complex action space. One possible solution is to use richer supervision such as demonstrations or instructions. In particular, we can start with a usual programming language. This way, the system is always usable—with some additional user effort learning parts of the language—and we can let users teach the system by providing definitions in the programming language and existing definitions. Having this programming language also gives all the users some common ground, in contrast to Chapter 4, where each user taught the system a private language, the entire user community shares one language in this chapter.

As before, our goal is to create a convenient natural language interface for performing well-specified but complex actions such as analyzing data, manipulating text, and querying databases. However, existing natural language interfaces for such tasks are quite primitive compared to the power one wields with a programming language. To bridge this gap, we start with a core programming language and allow users to "naturalize" the core language incrementally by defining alternative, more natural syntax and increasingly complex concepts in terms of compositions of simpler ones. In a voxel world, we show that a community of users can simultaneously teach a common system a diverse language and use it to build hundreds of complex voxel structures. Over the course of three days, these users went from using only the core language to using the naturalized language in 85.9% of the last 10K utterances.

5.1 Introduction

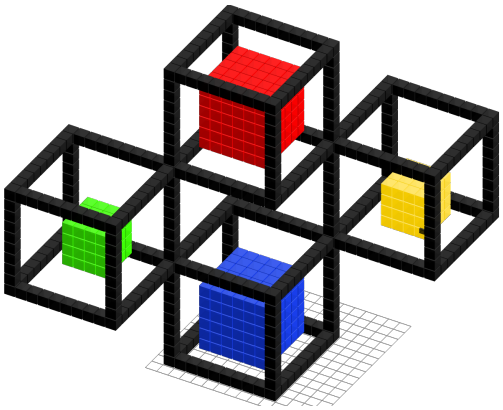
In tasks such as analyzing and plotting data (Gulwani and Marron, 2014), querying databases (Zelle and Mooney, 1996; Berant et al., 2013), manipulating text (Kushman and Barzilay, 2013), or controlling the Internet of Things (Campagna et al., 2017) and robots (Tellex et al., 2011), people need computers to perform well-specified but complex actions. To accomplish this, one route is to use a programming language, but this is inaccessible to most and can be tedious even for experts because the syntax is uncompromising and all statements have to be precise. Another route is to convert natural language into a formal language, which has been the subject of work in semantic parsing (Zettlemoyer and Collins, 2005; Artzi and Zettlemoyer, 2011; Liang et al., 2011; Berant et al., 2013; Artzi and Zettlemoyer, 2013; Pasupat and Liang, 2015). However, the capability of semantic parsers is still quite primitive compared to the power one wields with a programming language. This gap is increasingly limiting the potential of both text and voice interfaces as they become more ubiquitous and desirable.

In this chapter, we propose bridging this gap with an interactive language learning process which we call *naturalization*. Before any learning, we seed a system with a core programming language that is always available to the user. As users instruct the system to perform actions, they augment the language by *defining* new utterances—e.g., the user can explicitly tell the computer that X means Y. Through this process, users gradually and interactively teach the system to understand the language that they *want to use*, rather than the core language that they are forced to use initially. While the first users have to learn the core language, later users can make use of everything that is already taught. This process accommodates both users' preferences and the computer action space, where the final language is both interpretable by the computer and easier to produce by human users.

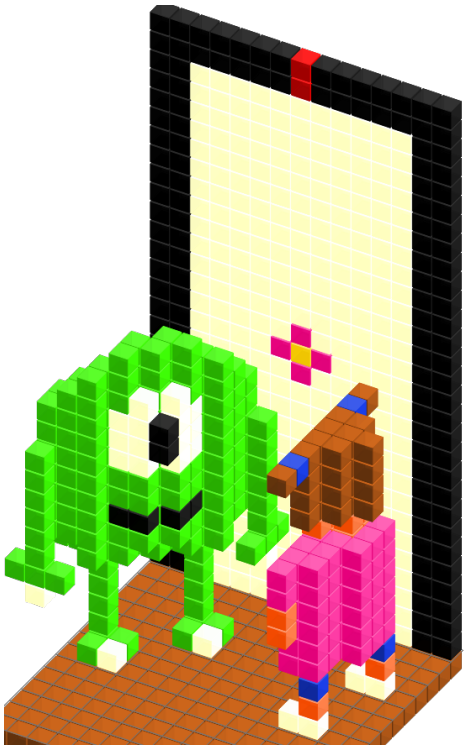
Compared to the interactive language learning with weak denotational supervision (Chapter 4), definitions are critical for learning complex actions (Figure 5.1). Definitions equate a novel utterance to a sequence of utterances that the system already understands. For example, `go left 6` and `go front` might be defined as `repeat 6 [go left]; go front`, which eventually can be traced back to the expression `repeat 6 [select left of this]; select front of this` in the core language. Unlike function definitions in programming languages, the user writes concrete values rather than explicitly declaring arguments. The system automatically extracts arguments and learns to produce the correct generalizations. For this, we propose a grammar induction algorithm tailored to the learning from definitions setting. Compared to standard machine learning, say from demonstrations, definitions provide a much more powerful learning signal: the system is told directly that `a 3 by 4 red`

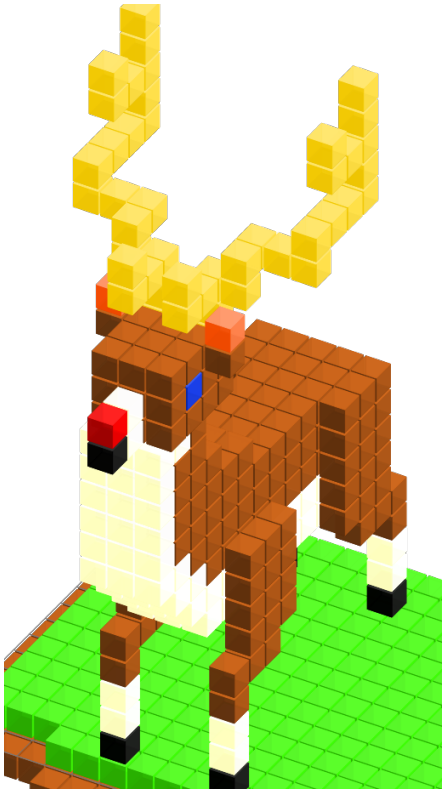
square is 3 red columns of height 4, and does not have to infer how to generalize from observing many structures of different sizes.

We implemented a system called Voxelurn, which is a command language interface for a voxel world initially equipped with a programming language supporting conditionals, loops, and variable scoping etc. We recruited 70 users from Amazon Mechanical Turk to build 230 voxel structures using our system. All users teach the system at once, and what is learned from one user can be used by another user. Thus a *community* of users evolves the language to becomes more efficient over time, in a distributed way, through interaction. We show that the user community defined many new utterances—short forms, alternative syntax, and also complex concepts such as add green monster, add yellow plate 3 x 3. As the system learns, users increasingly prefer to use the naturalized language over the core language: 85.9% of the last 10K accepted utterances are in the naturalized language.



Cubes: initial – select left 6 – select front 8 – black 10x10x10 frame – black 10x10x10 frame – move front 10 – red cube size 6 – move bot 2 – blue cube size 6 – green cube size 4 – (some steps are omitted)





Monsters, Inc: initial – move forward – add green monster – go down 8 – go right and front – add brown floor – add girl – go back and down – add door – add black column 30 – go up 9 – finish door – (some steps for moving are omitted)

Deer: initial – bird's eye view – deer head; up; left 2; back 2; { left antler }; right 2; {right antler} – down 4; front 2; left 3; deer body; down 6; {deer leg front}; back 7; {deer leg back}; left 4; {deer leg back}; front 7; {deer leg front} – (some steps omitted)

Figure 5.1: Some examples of users building structures using a naturalized language in Voxelurn.

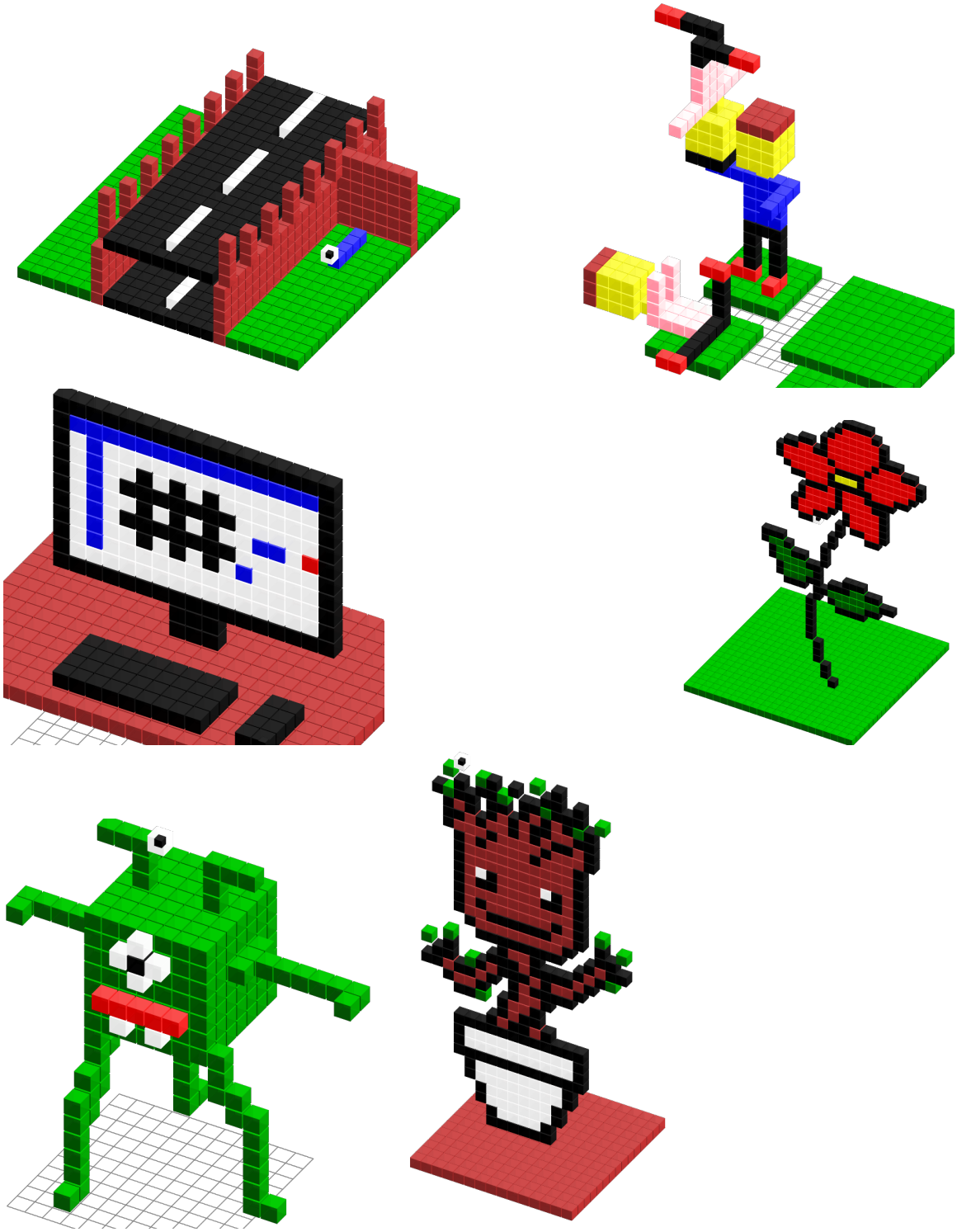


Figure 5.2: More example structures in Voxelurn

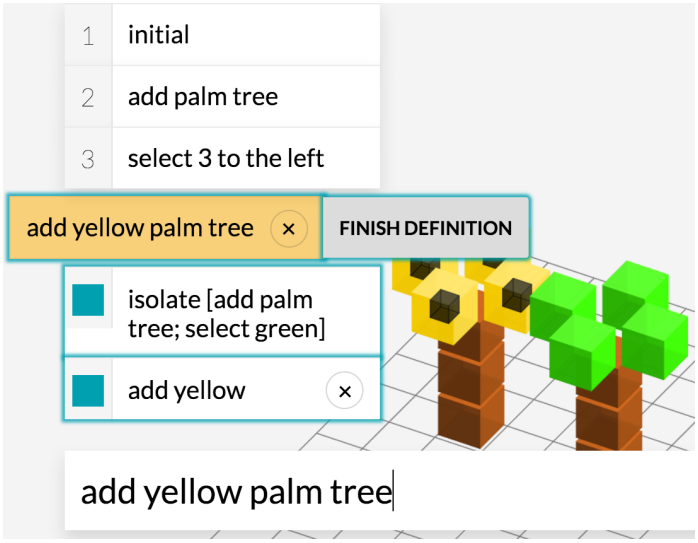


Figure 5.3: Interface used by users to enter utterances and create definitions.

5.2 Voxelurn

World. A world state in Voxelurn contains a set of voxels, where each voxel has relations 'row', 'col', 'height', and 'color'. There are two domain-specific actions, 'add' and 'move', one domain-specific relation 'direction'. In addition, the state contains a selection, which is a set of positions. While our focus is Voxelurn, we can think more generally about the world as a set of objects equiped with relations — events on a calendar, cells of a spreadsheet, or lines of text.

Core language. The system is born understanding a core language called Dependency-based Action Language (DAL), which we created (see Table 5.1 for an overview).

The language composes actions using the usual but expressive control primitives such as 'if', 'foreach', 'repeat', etc. Actions usually take sets as arguments, which are represented using lambda dependency-based compositional semantics (lambda DCS) expressions (Liang, 2013). Besides standard set operations like union, intersection and complement, lambda DCS leverages the tree dependency structure common in natural language: for the relation 'color', 'has color red' refers to the set of voxels that have color red, and its reverse 'color of has row 1' refers to the set of colors of voxels having row number 1. Tree-structured joins can be chained without using any variables, e.g., 'has color [yellow or color of has row 1]'.

Rule(s)	Example(s)	Description
$A \rightarrow A$; A	select left; add red	perform actions sequentially
$A \rightarrow$ repeat N A	repeat 3-1 add red top	repeat action N times
$A \rightarrow$ if SA	if has color red [select origin]	action if $S$is non-empty
$A \rightarrow$ while SA	while not has color red [select left of this]	action while $S$is non-empty
$A \rightarrow$ foreach SA	foreach this [remove has row row of this]	action for each item in S
$A \rightarrow [A]$	[select left or right; add red; add red top]	group actions for precedence
$A \rightarrow \{A\}$	{select left; add red}	scope only selection
$A \rightarrow$ isolate A	isolate [add red top; select has color red]	scope voxels and selection
$A \rightarrow$ select S	select all and not origin	set the selection
$A \rightarrow$ remove S	remove has color red	remove voxels
$A \rightarrow$ update RS	update color [color of left of this]	change property of selection
S	this	current selection
S	all | none | origin	all voxels, empty set, $(0,0)$
$R$of S | has RS	has color red or yellow | has row [col of this]	lambda DCS joins
not S | $S$and S | Sor S	this or left and not has color red	set operations
N | N+N | N-N	1,...,10 | 1+2 | row of this + 1	numbers and arithmetic
argmax RS | argmin RS	argmax col has color red	superlatives
R	color | row | col | height | top | left | $\cdots$	voxel relations
C	red | orange | green | blue | black | $\cdots$	color values
D	top | bot | front | back | left | right	direction values
$S \rightarrow$ very D of S	very top of very bot of has color green	syntax sugar for argmax
$A \rightarrow$ add $C[D]$ | move D	add red | add yellow bot | move left	add voxel, move selection

Table 5.1: Grammar of the core language (DAL), which includes actions (A), relations (R), and sets of values (S). The grammar rules are grouped into four categories. From top to bottom: domain-general action compositions, actions using sets, lambda DCS expressions for sets, and domain-specific relations and actions.

We protect the core language from being redefined so it is always precise and usable.[1] In addition to expressivity, the core language *interpolates* well with natural language. We avoid explicit variables by using a *selection*, which serves as the default argument for most actions.[2] For example, 'select has color red; add yellow top; remove' adds yellow on top of red voxels and then removes the red voxels.

To enable the building of more complex structures in a more modular way, we introduce a notion of *scoping*. Suppose one is operating on one of the palm trees in Figure 5.3. The user might want to use 'select all' to select only the voxels in that tree rather than all of the voxels in the scene. In general, an action A can be viewed as taking a set of voxels v and a selection s, and producing an updated set of voxels v' and a modified selection s'. The default scoping is '$[A]$', which is the same as 'A' and returns (v', s'). There are two constructs that alter the flow: First, '$\{A\}$' takes (v, s) and returns (v', s), thus restoring the selection. This allows A to use the selection as a temporary variable without affecting the rest of the program. Second, 'isolate $[A]$' takes (v, s), calls A with (s, s) (restricting the set of voxels to just the selection) and returns (v'', s), where v'' consists of voxels in v' and voxels in v that occupy empty locations in v'. This allows A to focus only on the selection (e.g., one of the palm trees). Although scoping can be explicitly controlled via '[]', 'isolate', and '{}', it is an unnatural concept for non-programmers. Therefore when the choice is not explicit, the parser generates all three possible scoping interpretations, and the model learns which is intended based on the user, the rule, and potentially the context.

5.3 Learning interactively from definitions

The goal of the user is to build a structure in Voxelurn. In Chapter 4, the user provided interactive supervision to the system by selecting from a list of candidates. This is practical when there are less than tens of candidates, but is completely infeasible for a complex action space such as Voxelurn. Roughly, 10 possible colors over the $3 \times 3 \times 4$ box containing the palm tree in Figure 5.3 yields 10^{36} distinct denotations, and many more programs. Obtaining the structures in Figure 5.1 by selecting candidates alone would be infeasible.

This work thus uses *definitions* in addition to selecting candidates as the supervision signal. Each definition consists of a *head* utterance and a *body*, which is a sequence of utterances that the system understands. One use of definitions is paraphrasing and defining alternative syntax, which

[1] Not doing so resulted in ambiguities that propagated uncontrollably, e.g., once red can mean many different colors.
[2] The selection is like the turtle in LOGO, but can be a set.

helps naturalize the core language (e.g., defining add brown top 3 times as 'repeat 3 add brown top'). The second use is building up complex concepts hierarchically. In Figure 5.3, add yellow palm tree is defined as a sequence of steps for building the palm tree. Once the system understands an utterance, it can be used in the body of other definitions. For example, Figure 5.4 shows the full definition tree of 'add palm tree'. Unlike function definitions in a programming language, our definitions do not specify the exact arguments; the system has to learn to extract arguments to achieve the correct generalization.

```
def: add palm tree
    def: brown trunk height 3
        def: add brown top 3 times
            repeat 3 [add brown top]
    def: go to top of tree
        select very top of has color brown
    def: add leaves here
        def: select all sides
            select left or right or front or back
        add green
```

Figure 5.4: Defining add palm tree, tracing back to the core language (utterances without **def:**).

begin execute x:
    if *x does not parse* **then** define x;
    if *user rejects all parses* **then** define x;
    execute user choice
begin define x:
    repeat starting with $X \leftarrow [\,]$
        user enters x';
        if x' *does not parse* **then** define x';
        if *user rejects all* x' **then** define x';
        $X \leftarrow [X; x']$;
    until *user accepts X as the def'n of x*;

Figure 5.5: When the user enters an utterance, the system tries to parse and execute it, or requests that the user define it.

The interactive definition process is described in Figure 5.5. When the user types an utterance x, the system parses x into a list of candidate programs. If the user selects one of them (based on its denotation), then the system executes the resulting program. If the utterance is unparsable or the user rejects all candidate programs, the user is asked to provide the definition body for x. Any

utterances in the body not yet understood can be defined recursively. Alternatively, the user can first execute a sequence of commands X, and then provide a head utterance for body X.

When constructing the definition body, users can type utterances with multiple parses; e.g., move forward could either modify the selection ('select front') or move the voxel ('move front'). Rather than propagating this ambiguity to the head, we force the user to commit to one interpretation by selecting a particular candidate. Note that we are using interactivity to control the exploding ambiguity.

5.4 Model and learning

Let us turn to how the system learns and predicts. This section contains prerequisites before we describe definitions and grammar induction in Section 5.5.

Semantic parsing. Our system is based on a semantic parser that maps utterances x to programs z, which can be executed on the current state s (set of voxels and selection) to produce the next state $s' = [\![z]\!]_s$. Our system is implemented as the interactive package in SEMPRE (Berant et al., 2013); see Liang (2016) for a gentle exposition.

A *derivation* d represents the process by which an utterance x turns into a program $z = \text{prog}(d)$. More precisely, d is a tree where each node contains the corresponding span of the utterance $(\text{start}(d), \text{end}(d))$, the grammar rule $\text{rule}(d)$, the grammar category $\text{cat}(d)$, and a list of child derivations $[d_1, \ldots, d_n]$.

Following Zettlemoyer and Collins (2005), we define a log-linear model over derivations d given an utterance x produced by the user u:

$$p_\theta(d \mid x, u) \propto \exp(\theta^\top \phi(d, x, u)), \tag{5.1}$$

where $\phi(d, x, u) \in \mathbb{R}^p$ is a feature vector and $\theta \in \mathbb{R}^p$ is a parameter vector. The user u does not appear in previous work on semantic parsing, but we use it to personalize the semantic parser trained on the community.

We use a standard chart parser to construct a chart. For each chart cell, indexed by the start and end indices of a span, we construct a list of partial derivations recursively by selecting child derivations from subspans and applying a grammar rule. The resulting derivations are sorted by model score and only the top K are kept. We use $\text{chart}(x)$ to denote the set of all partial derivations across all chart cells. The set of grammar rules starts with the set of rules for the core language

Feature	Description
Rule.ID	ID of the rule
Rule.Type	core?, used?, used by others?
Social.Author	ID of author
Social.Friends	(ID of author, ID of user)
Social.Self	rule is authored by user?
Span	(left/right token(s), category)
Scope	type of scoping for each user

Table 5.2: Summary of features.

(Table 5.1), but grows via grammar induction when users add definitions (Section 5.5). Rules in the grammar are stored in a trie based on the right-hand side to enable better scalability to a large number of rules.

Features. Derivations are scored using a weighted combination of features. There are three types of features, summarized in Table 5.2.

Rule features fire on each rule used to construct a derivation. ID features fire on specific rules (by ID). Type features track whether a rule is part of the core language or induced, whether it has been used again after it was defined, if it was used by someone other than its author, and if the user and the author are the same ($5 +$ #rules features).

Social features fire on properties of rules that capture the unique linguistic styles of different users and their interaction with each other. Author features capture the fact that some users provide better, and more generalizable definitions that tend to be accepted. Friends features are cross products of author ID and user ID, which captures whether rules from a particular author are systematically preferred or not by the current user, due to stylistic similarities or differences (#users $+$ #users $\times$ #users features).

Span features include conjunctions of the category of the derivation and the leftmost/rightmost token on the border of the span. In addition, span features include conjunctions of the category of the derivation and the 1 or 2 adjacent tokens just outside of the left/right border of the span. These capture a weak form of context-dependence that is generally helpful ($<\approx V^4 \times$ #cats features for a vocabulary of size V).

Scoping features track how the community, as well as individual users, prefer each of the 3 scoping choices (none, selection only $\{\mathsf{A}\}$, and voxels+selection $\mathsf{isolate}\ \{\mathsf{A}\}$), as described in Section 5.2. 3 global indicators, and 3 indicators for each user fire every time a particular scoping choice is made ($3 + 3 \times$ #users features).

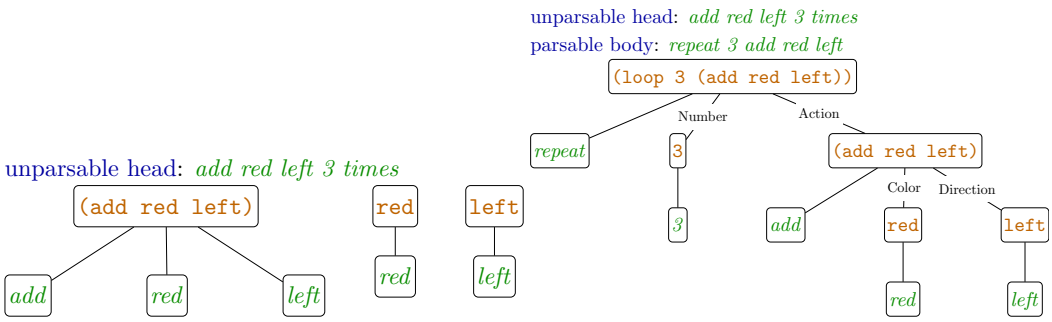


Figure 5.6: unparsable head and parsable body as the input to grammar induction

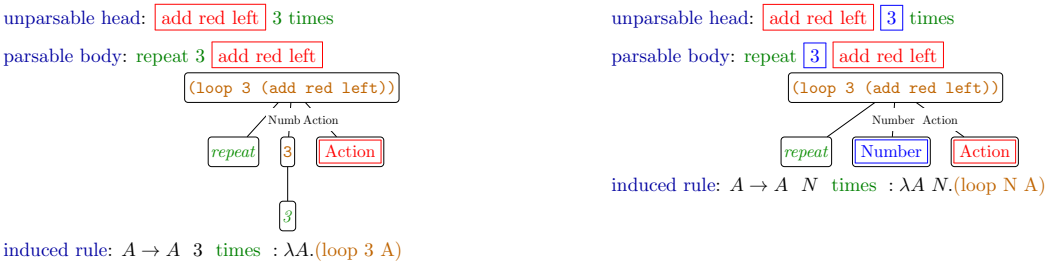


Figure 5.7: Matches and induced rules. The corresponding matches are shown in boxes of matching colors. **left:** only abstracting the action; **right:** abstracting both the number and the action

Parameter estimation. When the user types an utterance, the system generates a list of candidate next states. When the user chooses a particular next state s' from this list, the system performs an online AdaGrad update (Duchi et al., 2010) on the parameters θ according to the gradient of the following loss function:

$$-\log \sum_{d:[\![\text{prog}(d)]\!]_s=s'} p_\theta(d \mid x,u) + \lambda ||\theta||_1,$$

which attempts to increase the model probability on derivations whose programs produce the next state s'.

5.5 Grammar induction

Recall that the main form of supervision is via user definitions, which allows creation of user-defined concepts. In this section, we show how to turn these definitions into new grammar rules that can be

abstract action:

add red left | 3 | times

repeat | 3 | add red left |

$A \rightarrow A \;\; N \;\; \text{times} \;\; : \lambda A \; N.(\text{loop N A})$

abstract number and color:

add | red | left | 3 | times

repeat | 3 | add | red | left |

$A \rightarrow \text{add } C \; D \; N \; \text{times} \;\; : \lambda C \; D \; N.(\text{loop N (add C D)})$

Figure 5.8: Two sets of distinct abstractions. The corresponding matches are shown in boxes of matching colors. **top:** abstract the action and number; **bottom:** abstract color, direction and numbers.

used by the system to parse new utterances.

Previous systems of grammar induction for semantic parsing were given utterance-program pairs (x,z). Both the GENLEX (Zettlemoyer and Collins, 2005) and higher-order unification (Kwiatkowski et al., 2010) algorithms over-generate rules that liberally associate parts of x with parts of z. Though some rules are immediately pruned, many spurious rules are undoubtedly still kept. In the interactive setting, we must keep the number of candidates small to avoid a bad user experience, which means a higher precision bar for new rules.

Fortunately, the structure of definitions makes the grammar induction task easier. Rather than being given an utterance-program (x,z) pair, we are given a definition, which consists of an utterance x (head) along with the body $X = [x_1, \ldots, x_n]$, which is a sequence of utterances. The body X is fully parsed into a derivation d, while the head x is likely only partially parsed. These partial derivations are denoted by chart(x).

Since the system had to parse the defintion, we have access to the derivation d, rather than just $z = \text{prog}(d)$. d has a tree structure that limits how z can be generated. By only considering those ways of generating z that is consistent with d, we gain higher precision, while possibly losing some recall. In the interactive setting, a wrong and productive rule forces everyone to consider more options, so perhaps high precision is desirable. In addition to d, we have utterances X used to form the definition, which we make use by aligning it to x (Section 5.5.2).

A definition consists of the *head* utterance x, and the *body* of the definition $X = [x_1, \ldots, x_n]$

which is a sequence of utterances defining x that all parsed in a way consistent with the derivation. Typically, the definition is given when the system cannot parse x entirely, or parses x incorrectly according to the current user.

At a high-level, we find *matches*—partial derivations in chart(x) of the head x that also occur in the full derivation d of the body X. A grammar rule is produced by substituting any set of non-overlapping matches by their categories. As an example, suppose the user defines

add red top times 3 as 'repeat 3 [add red top]'.

Then we would be able to induce the following two grammar rules:

$$
\begin{aligned}
A \rightarrow\ & \text{add } C\ D \text{ times } N\ : \\
& \lambda CDN.\text{repeat } N\ [\text{add } C\ D] \\
A \rightarrow\ & A \text{ times } N : \\
& \lambda AN.\text{repeat } N\ [A]
\end{aligned}
$$

The first rule substitutes primitive values (red, top, and 3) with their respective pre-terminal categories (C, D, N). The second rule contains compositional categories like actions (A), which require some care. One might expect that greedily substituting the largest matches or the match that covers the largest portion of the body would work, but the following example shows that this is not the case:

$$
\underbrace{\overbrace{\text{add red}}^{A_1}\ \text{left}}_{A_2}\ \text{and here} = \underbrace{\overbrace{\text{add red}}^{A_1}\ \text{left;}}_{A_2}\ \overbrace{\text{add red}}^{A_1}
$$

Here, both the highest coverage substitution (A_1: add red, which covers 4 tokens of the body), and the largest substitution available (A_2: add red left) would generalize incorrectly. The correct grammar rule only substitutes the primitive values (red, left).

5.5.1 Highest scoring abstractions

We now propose a grammar induction procedure that optimizes a more global objective and uses the learned semantic parsing model to choose substitutions. More formally, let M be the set of partial

derivations of the head whose programs appear in the derivation d_X of the body X:

$$M \stackrel{\text{def}}{=} \{d \in \text{chart}(x) : \\ \exists d' \in \text{desc}(d_X) \wedge \text{prog}(d) = \text{prog}(d')\},$$

where $\text{desc}(d_X)$ are the descendant derivations of d_X. Our goal is to find a *packing* $P \subseteq M$, which is a set of derivations corresponding to non-overlapping spans of the head. We say that a packing P is maximal if no other derivations may be added to it without creating an overlap.

Let $\text{packings}(M)$ denote the set of maximal packings, we can frame our problem as finding the maximal packing that has the highest score under our current semantic parsing model:

$$P_L^* = \underset{P \in \text{packings}(M);}{\text{argmax}} \sum_{d \in P} \text{score}(d). \tag{5.2}$$

Finding the highest scoring packing can be done using dynamic programming on P_i^* for $i = 0, 1, \ldots, L$, where L is the length of x and $P_0^* = \emptyset$. Since $d \in M$, $\text{start}(d)$ and $\text{end}(d)$ (exclusive) refer to span in the head x. To obtain this dynamic program, let D_i be the highest scoring maximal packing containing a derivation ending *exactly* at position i (if it exists):

$$D_i = \{d_i\} \cup P^*_{\text{start}(d_i)}, \tag{5.3}$$

$$d_i = \underset{d \in M; \text{end}(d) = i}{\text{argmax}} \text{score}(d \cup P^*_{\text{start}(d)}). \tag{5.4}$$

Then the maximal packing of up to i can be defined recursively as

$$P_i^* = \underset{D \in \{D_{s(i)+1}, D_{s(i)+2}, \ldots, D_i\}}{\text{argmax}} \text{score}(D) \tag{5.5}$$

$$s(i) = \max_{d: \text{end}(d) \leq i} \text{start}(d), \tag{5.6}$$

where $s(i)$ is the largest index such that $D_{s(i)}$ is no longer maximal for the span $(0, i)$ (i.e. there is a $d \in M$ on the span $\text{start}(d) \geq s(i) \wedge \text{end}(d) \leq i$.

Once we have a packing $P^* = P_L^*$, we can go through $d \in P^*$ in order of $\text{start}(d)$, as in Algorithm 2. This generates one high precision rule per packing per definition. In addition to the highest scoring packing, we also use a "simple packing", which includes only primitive values (in Voxelurn, these are colors, numbers, and directions). Unlike the simple packing, the rule induced from the highest scoring packing does not always generalize correctly. However, a rule that often

Input : x, d_X, P^*
Output: rule
$r \leftarrow x$;
$f \leftarrow d_X$;
for $d \in P^*$ **do**
    $r \leftarrow r[\text{cat}(d)/\text{span}(d)]$ $f \leftarrow \lambda\, \text{cat}(d).f[\text{cat}(d)/d]$
return *rule* $(\text{cat}(d_X) \rightarrow r : f)$

Algorithm 2: Extract a rule r from a derivation d_X of body X and a packing P^*. Here, $f[t/s]$ means substituting s by t in f, with the usual care about names of bound variables.

generalizes incorrectly should be down-weighted, along with the score of its packings. As a result, a different rule might be induced next time, even with the same definition.

5.5.2 Extending the chart via alignment

Algorithm 2 yields high precision rules, but fails to generalize in some cases. Suppose that move up is defined as move top, where up does not parse, and does not match anything. We would like to infer that up means top. To handle this, we leverage a property of definitions that we have not used thus far: the utterances themselves. If we align the head and body, then we would intuitively expect aligned phrases to correspond to the same derivations. Under this assumption, we can then transplant these derivations from d_X to $\text{chart}(x)$ to create new matches. This is more constrained than the usual alignment problem (e.g., in machine translation) since we only need to consider spans of X which corresponds to derivations in $\text{desc}(d_X)$.

Input : x, X, d_X
for $d \in \text{desc}(d_X)$, $x' \in \text{spans}(x)$ **do**
    if $\text{aligned}(x', d, (x, X))$ **then**
        $d' \leftarrow d$;
        $\text{start}(d') \leftarrow \text{start}(x')$;
        $\text{end}(d') \leftarrow \text{end}(x')$;
        $\text{chart}(x) \leftarrow \text{chart}(x) \cup d'$
    end
end

Algorithm 3: Extending the chart by alignment: If d is aligned with x' based on the utterance, then we pretend that x' should also parse to d, and d is transplanted to $\text{chart}(x)$ as if it parsed from x'.

Algorithm 3 provides the algorithm for extending the chart via alignments. The aligned function

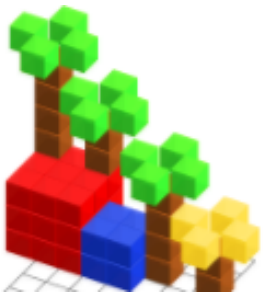


Figure 5.9: The target used for the qualifier.

is implemented using the following two heuristics:

- **exclusion**: if all but 1 pair of short spans (1 or 2 tokens) are matched, the unmatched pair is considered aligned.
- **projectivity**: if $d_1, d_2 \in \text{desc}(d_X) \cap \text{chart}(x)$, then $\text{ances}(d_1, d_2)$ is aligned to the corresponding span in x.

With the extended chart, we can run the algorithm from Section 5.5.1 to induce rules. The transplanted derivations (e.g., up) might now form new matches which allows the grammar induction to induce more generalizable rules. We only perform this extension when the body consists of one utterance, which tends to be a paraphrase. Bodies with multiple utterances tend to be new concepts (e.g., add green monster), for which alignment is impossible. Because users have to select from candidates parses in the interactive setting, inducing low precision rules that generate many parses degrade the user experience. Therefore, we induce alignment-based rules conservatively—only when all but 1 or 2 tokens of the head aligns to the body and vice versa.

5.6 Experiments

Setup. Our ultimate goal is to create a community of users who can build interesting structures in Voxelurn while naturalizing the core language. We created this community using Amazon Mechanical Turk (AMT) in two stages. First, we had *qualifier* tasks, in which an AMT worker was instructed to replicate a fixed target exactly (Figure 5.9), ensuring that the initial users are familiar with at least some of the core language, which is the starting point of the naturalization process.

Next, we allowed the workers who qualified to enter the second *freebuilding* task, in which they were asked to build any structure they wanted in 30 minutes. This process was designed to give users freedom while ensuring quality. The analogy of this scheme in a real system is that early users

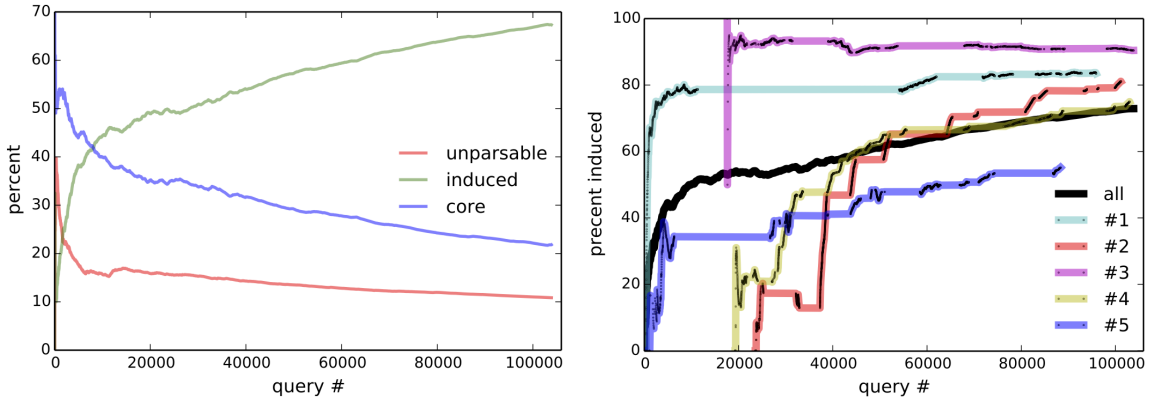


Figure 5.10: Learning curves. **Left:** cumulative percentage of all utterances that are part of the *core* language, the *induced* language, or *unparsable* by the system. **Right:** cumulative percentage of accepted utterances belonging to the induced language, overall and for the 5 heaviest users.

(or a small portion of expert users) have to make some learning investment, so the system can learn and become easier for other users.

Statistics. 70 workers passed the qualifier task, and 42 workers participated in the final free-building experiment. They built 230 structures. There were over 103,000 queries consisting of 5,388 distinct token types. Of these, 64,075 utterances were tried and 36,589 were accepted (so an action was performed). There were 2,495 definitions combining over 15,000 body utterances with 6.5 body utterances per head on average (96 max). From these definitions, 2,817 grammar rules were induced, compared to less than 100 core rules. Over all queries, there were 8.73 parses per utterance on average (starting from 1 for core).

Is naturalization happening? The answer is yes according to Figure 5.10 and Figure 5.11, which plots the cummulative percentage of utterances that are core, induced, or unparsable. To rule out that more induced utterances are getting rejected, we consider only accepted utterances in the left of Figure 5.10, which plots the percentage of induced rules among accepted utterances for the entire community, as well as for the 5 heaviest users. Since unparsable utterances cannot be accepted, accepted core (which is not shown) is the complement of accepted induced. At the conclusion of the experiment, 72.9% of all accepted utterances are induced—this becomes 85.9% if we only consider the final 10,000 accepted utterances.

Three modes of naturalization are outlined in Table 5.3. For very common operations, like moving the selection, people found `select left` too verbose and shorterned this to `|, left, >, sel |`.

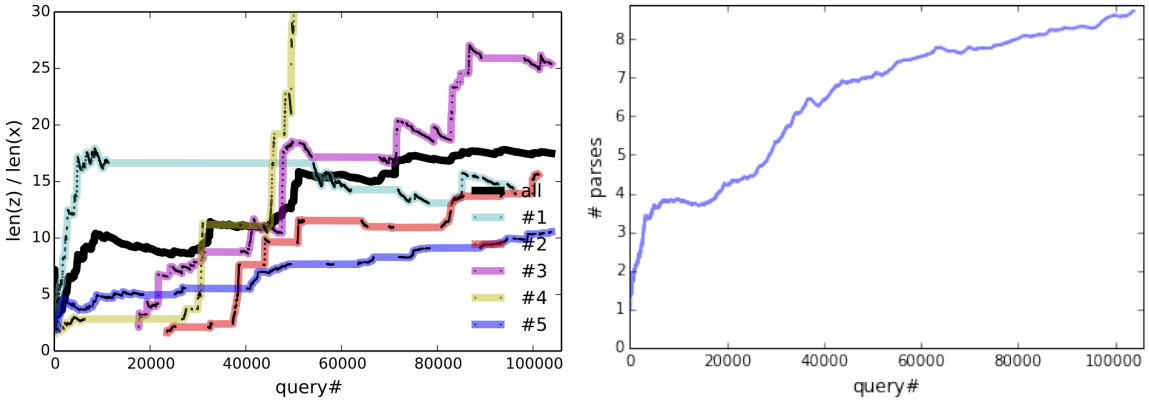


Figure 5.11: **Left:** expressiveness measured by the cumulative average ratio of the length of the program to the length of the corresponding utterance. **Right:** cumulative average of the number of derivations per utterance, as a measure of ambiguity.

Short forms
left, l, mov left, go left, <, sel left
br, black, blu, brn, orangeright, left3
add row brn left 5 := add row brown left 5
Alternative syntax
go down and right := go down; go right
select orange := select has color orange
add red top 4 times := repeat 4 [add red top]
l white := go left and add white
mov up 2 := repeat 2 [select up]
go up 3 := go up 2; go up
Higher level
add red plate 6 x 7, green cube size 4,
add green monster, black 10x10x10 frame,
flower petals, deer leg back, music box, dancer

Table 5.3: Example definitions. See CodaLab worksheet for the full leaderboard.

One user preferred go down and right instead of select bot; select right in core and defined it as go down; go right. Definitions for high-level concepts tend to be whole objects that are not parameterized (e.g., dancer). The bottom plot of Figure 5.10 suggests that users are defining and using higher level concepts, since programs become longer relative to utterances over time.

As a result of the automatic but implicit grammar induction, some concepts do not generalize correctly. In definition head 3 tall 9 wide white tower centered here, 3 and 9 did not appear in the body; for black 10x10x10 frame, we failed to tokenize.

Learned parameters. Training using L1 regularization, we obtained 1713 features with non-zero parameters. One user defined many concepts consisting of a single short token, and the Social.Author feature for that user has the most negative weight overall. With user compatibility (Social.Friends), some pairs have large positive weights and others large negative weights. The isolate scoping choice (which allows easier hierarchical building) received the most positive weights, both overall and for many users. The 2 highest scoring induced rules correspond to add row red right 5 and select left 2.

Incentives. Having complex structures shows that the actions in Voxelurn are expressive and that hierarchical definitions are useful. To incentivize this behavior, we created a leaderboard which ranked structures based on recency and upvotes (like Hacker News). Over the course of 3 days, we picked three prize categories to be released daily. The prize categories for each day were bridge, house, animal; tower, monster, flower; ship, dancer, and castle.

To incentivize more definitions, we also track *citations*. When a rule is used in an accepted utterance by another user, the rule (and its author) receives a citation. We pay bonuses to top users according to their h-index. Most cited definitions are also displayed on the leaderboard. Our qualitative results should be robust to the incentives scheme, because the users do not overfit to the incentives—e.g., around 20% of the structures are not in the prize categories and users define complex concepts that are rarely cited.

5.7 Related work and discussion

This work is an evolution of Wang et al. (2016), but differs crucially in several ways: While Wang et al. (2016) starts from scratch and relies on selecting candidates, this work starts with a programming language (PL) and additionally relies on definitions, allowing us to scale. Instead of having a private language for each user, the user community in this work shares one language.

Azaria et al. (2016) presents Learning by Instruction Agent (LIA), which also advocates learning from users. They argue that developers cannot anticipate all the actions that users want, and that the system cannot understand the corresponding natural language even if the desired action is built-in. Like Jia et al. (2017), Azaria et al. (2016) starts with an ad-hoc set of initial slot-filling commands in natural language as the basis of further instructions—our approach starts with a more expressive core PL designed to interpolate with natural language. Compared to previous work, this work studied interactive learning in a shared community setting and hierarchical definitions resulting in

more complex concepts.

Androutsopoulos et al. (1995) analyzed alternatives to natural language interface such as a query language, form-based interface and GUI. They point out that the action space is not well-defined with an NLI and that users confuse linguistic failures with conceptual failures. They sugggest that NL is not an appropriate medium for communicating with a computer system that has a precise action space. In this work, the core language is precise and defines the action space, and the naturalization process happens on top of that. Each new user can ease into the process and take advantage of the experience of previous users.

Allowing ambiguity and a flexible syntax is a key reason why natural language is easier to produce—this cannot be achieved by PLs such as Inform and COBOL which look like natural language. In this work, we use semantic parsing techniques that can handle ambiguity (Zettlemoyer and Collins, 2005, 2007; Kwiatkowski et al., 2010; Liang et al., 2011; Pasupat and Liang, 2015). In semantic parsing, the semantic representation and action space is usually designed to accommodate the natural language that is considered constant. In contrast, the action space is considered constant in the naturalizing PL approach, and the language adapts to be more natural while accommodating the action space.

Our work demonstrates that interactive definitions is a strong and usable form of supervision. In the future, we wish to test these ideas in more domains, naturalize a real PL, and handle paraphrasing and implicit arguments. In the process of naturalization, both data and the semantic grammar have important roles in the evolution of a language that is easier for humans to produce while still parsable by computers.

Reproducibility. All code, data, and experiments for this chapter are available on the CodaLab platform:
`https://worksheets.codalab.org/worksheets/0xbf8f4f5b42e54eba9921f7654b3c5c5d`
and a demo: `http://www.voxelurn.com`

Chapter 6

Conclusions

6.1 Issues

We summarize some problems and difficulties we encountered in Chapter 4 and Chapter 5.

6.1.1 Issues in learning language games

Selection does not scale. In Chapter 4, the action space can expand to tens of thousands of relatively short logical forms of 6-8 predicates if we allow binary set operations such as union, intersection and difference. With tens of thousands of logical forms, learning from denotation cannot reduce this space to a small enough space for users to inspect. Under the interactive setting, users cannot inspect hundreds of candidates to select the correct one. As a result, we reduced the richness of the action space until learning from denotation can give sufficient reduction to around 100 candidates, which becomes manageable. In principle, selection is a weak and unscalable form of supervision because user effort grows linearly as the action space grow exponentially. This is the main reason that prompted us to leverage definitions in Chapter 5. Other potential solutions include multi-step selection on parts of the logical form, and effective curriculum learning.

Need for off-policy evaluations. Due to user adaptation, evaluation can be tricky for interactive systems. For example, a scientifically pure evaluation comparing the pragmatic and regular setting would require A/B testing that randomly direct users to each setting. However, progress would be slow and expensive if every decision require that level of rigor. Instead, we just used data collected in a normal session, and model that is developed on that to test the pragmatic setting. This way, at least any gain in the pragmatic setting is probably real, while it is also likely to not detect gains that

would otherwise exist had we done the A/B test. This is not a problem unique to us—any settings (such as learning to rank and recommend) where the data collected depends on system capability have this issue, and research on counter-factual evaluations seek to address it. However, language use can be particularly adaptive and exacerbate the problem under our setting.

Pragmatics. While pragmatics helped top users, we expect that as the action space expands, and more shared language among users, our form of pragmatics will play a less important role. In addition, probabilistic pragmatics introduces an extra hyper-parameter, and tuning it can affect the accuracy. I did not carry out the statistical analysis on this.

Ultimately, I was more interested in building better systems rather than testing scientific hypothesis because there just seem to be too many factors to control for once the behavior of a complex computer system can affect the result. At this moment, at some risk of deceiving myself, there seem to be enough obvious shortcomings to improve that not every decision have to be justified by a rigorous test.

Insufficient feedback. If the computer is able to talk back to the user in a similar language that the user is teaching, then the user can potentially adapt better. In this setting, all the feedback is through the result list, and the user receive no linguistic feedback. As a result, the language use were not as adaptive as we hoped. Most likely, users just stuck with their own interpretation of the action space, and never adapted to a better one. Instances of simplifying the language is way more common than changing the semantics to adapt to the action space.

6.1.2 Issues in naturalizing a programming language

Rigid dependencies. Motivated by programming languages, we used definitions with inferred arguments to expand the language. An annoying consequence is that definitions have a rigid dependency structure on each other, much like in regular programming languages. If the definitions are shuffled, then they would have different generalization which is usually worse than the original. For example, even the randomization due to threading can have a negative effect, and we had to make sure definitions are received in exactly the same order as in the initial experiment.

Unpredictability. After thousands of rules are induced (compared to less than 100 initially), I found it difficult to predict what the system would do on a particular utterance. Since the grammar

induction method use model scores to determine what to abstract, I could not predict how each definition would generalize.

Learning from denotation gives little value. While a reasonable number of logical forms executed to the same denotation in the language games setting, the denotation in Chapter 5 consists of hundreds to tens of thousands of voxels, which is arguable more complex than the programs that generated them. In particular, random programs are unlikely to have the same denotation. The only place where learning denotation helped was for scoping constructions.

Lack of discoverability. While a community of users built on each other, and taught the computer various language, a new user coming in did not have a systematic way of learning about this. Our attempt was to use the leaderboard to show new users some example utterances that they can try. Another method we considered was to use an autocomplete function. We did not deploy autocomplete because we wanted to encourage our users to try new utterances and define more instead of just use what's already in the system. This is related to the **lack of system-initiative**, and a more conversational system might be in a better position to address this. For the computer to have more to contribute, **partial observability and stochasticity** might be helpful, which we completely ignored in our work.

Lack of abstract generalization. Our initial hope is if many users taught the computer, then all reasonable utterances can be taught with enough users, and the computer would understand most future utterances. This would be the case if all future utterances are similar to existing utterances, and a reasonable grammar induction algorithm can eventually achieve very high coverage. This turned out to be far from the case, and it is relatively easy to come up with utterances that does not generalize after learning on over 60k commands.

Consider `red cube size 5` where there are 32 ways of permuting and choosing between positional, keyword and omitted arguments:

$$\pi(\{\varepsilon, \texttt{red}, \texttt{color red}\} \times \{\varepsilon, 5, \texttt{size 5}\} \times \{\texttt{cube}\}) \tag{6.1}$$

$$= \{\texttt{cube}, \texttt{red cube}, \texttt{color red cube}, \ldots, \texttt{cube size 5 color red}\}. \tag{6.2}$$

Our high precision syntax-based grammar induction treat all these as different from each other.

Procedural rather than declarative. This is related to the ability to generalize. Suppose we can build a tree, and now consider "*a big tree*" and "*make tree 2 times bigger*". Our precise procedural action space does not easily allow for declarative generalizations and vague statements.

6.2 Applications

We discuss some criteria on what might be promising applications of adaptive language interfaces, and we briefly describe some of our work on real-world applications.

6.2.1 Criteria on applications

Because we introduce unresolved ambiguities into the language, there has to be some way to eventually resolve them. A simple method is just to show all the potential results to the user, and the user pick out the correct one. This is beneficial when the results are easier to verify than to specify precisely. For example, if we just display programs, then the programs have to be easier to read than to write from scratch. Many programming languages are like this—for example, bash (especially if options are in words) and Java (heavy on syntax), but perhaps not very terse languages like regex or Perl.

Better yet, we can just display the denotations if the users can quickly and effectively inspect the them. This was somewhat true in blocks world, where it is easier to determine if a cat is added than writing the program to add it. As discussed in Section 2.3, it would help if the task is more like ad hoc commands rather than software engineering.

Two particular applications that we did some work on are data visualization and calendar.

6.2.2 Calendar

Event scheduling is a common yet unsolved task: while several available calendar programs allow limited natural language input, in our experience they all fail as soon as they are given something slightly complicated, such as 'Move all the tuesday afternoon appointments back an hour'. We think interactive learning can give us a better NLI for calendars, which has more real world impact than blocks world. Furthermore, aiming to expand our learning methodology from definition to demonstration, we chose this domain as most users are already familiar with the common calendar GUI with an intuition for its manual manipulation. Additionally, as calendar NLIs are already deployed, particularly on mobile, we hoped users will naturally be inclined to use natural language style phrasing rather than a more technical language as we saw in the blocks world domain.

6.2.3 Data visualization

5 research systems using natural language for data analysis and visualization are examined and compared in Srinivasan and Stasko (2017). Data visualization is fairly interactive by nature, and charts are designed to convey a lot of information visually. In popular plotting libraries such as ggplot, matplotlib and vega, commands tend be easier to interpret than to write. However, using these libraries to produce charts requires frequent references to documentations, as well as trial and errors. As a result, there seem to be an opportunity to use a more adaptive language interface to visualize data.

6.3 Final remarks

NLIs have the potential to complement GUIs and programming for many tasks, and doing so can bridge the great digital divide of skills and enable all of us to better make use of computers. However, until computers think like humans, they may not be able to satisfactorily understand human language, and we might have to settle for *adaptive language interfaces* where humans users have to partially adapt to the capabilities of computers as computers adapt to human communication preferences. Because static datasets does not account for system capabilities, such adaptation needs to be part of an interactive learning process.

In this thesis, we studied two extremes interactive language learning settings, starting from scratch and starting from a programming language. To learn from scratch, the human can use any language, but have to adapt to computer capabilities as the computer learns their language. Starting from programming language is the opposite—the human have to use a language the computer already understands, and they then teach the computer to understand languages that they prefer more. The vast space between standard natural languages and rigid programming languages is where adaptive language interface can potentially improve human-computer communication when computers capabilities differ significantly from humans.

Appendix A

Samples from Voxelurn

A.1 Leaderboard

The full leaderboard is linked in the CodaLab worksheet for Wang et al. (2017):
`https://worksheets.codalab.org/rest/bundles/0x1d0b5ff13ab541d5abf4d37fd63ce9d3/contents/blob/static_leaderboard.html`

The original CodaLab worksheet is here:
`https://worksheets.codalab.org/worksheets/0xbf8f4f5b42e54eba9921f7654b3c5c5d/`

We include just a few examples and their corresponding utterances in Figure A.1 and Figure A.2.

A.2 Citations

We show users with the most number of citations in Figure A.3, and the breakdown according to utterances.

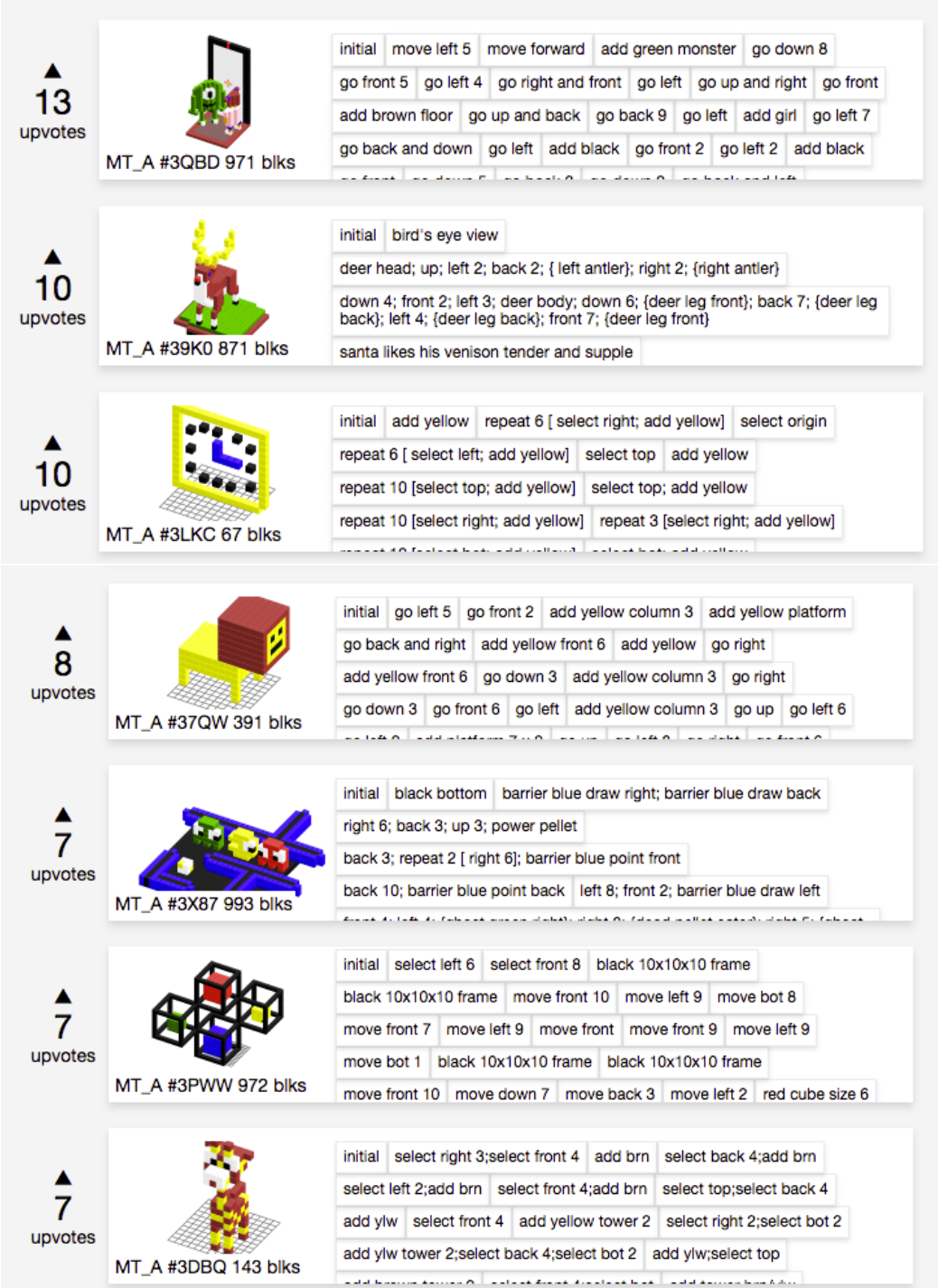


Figure A.1: Selected leaderboard entries and the sequence of commands producing them.

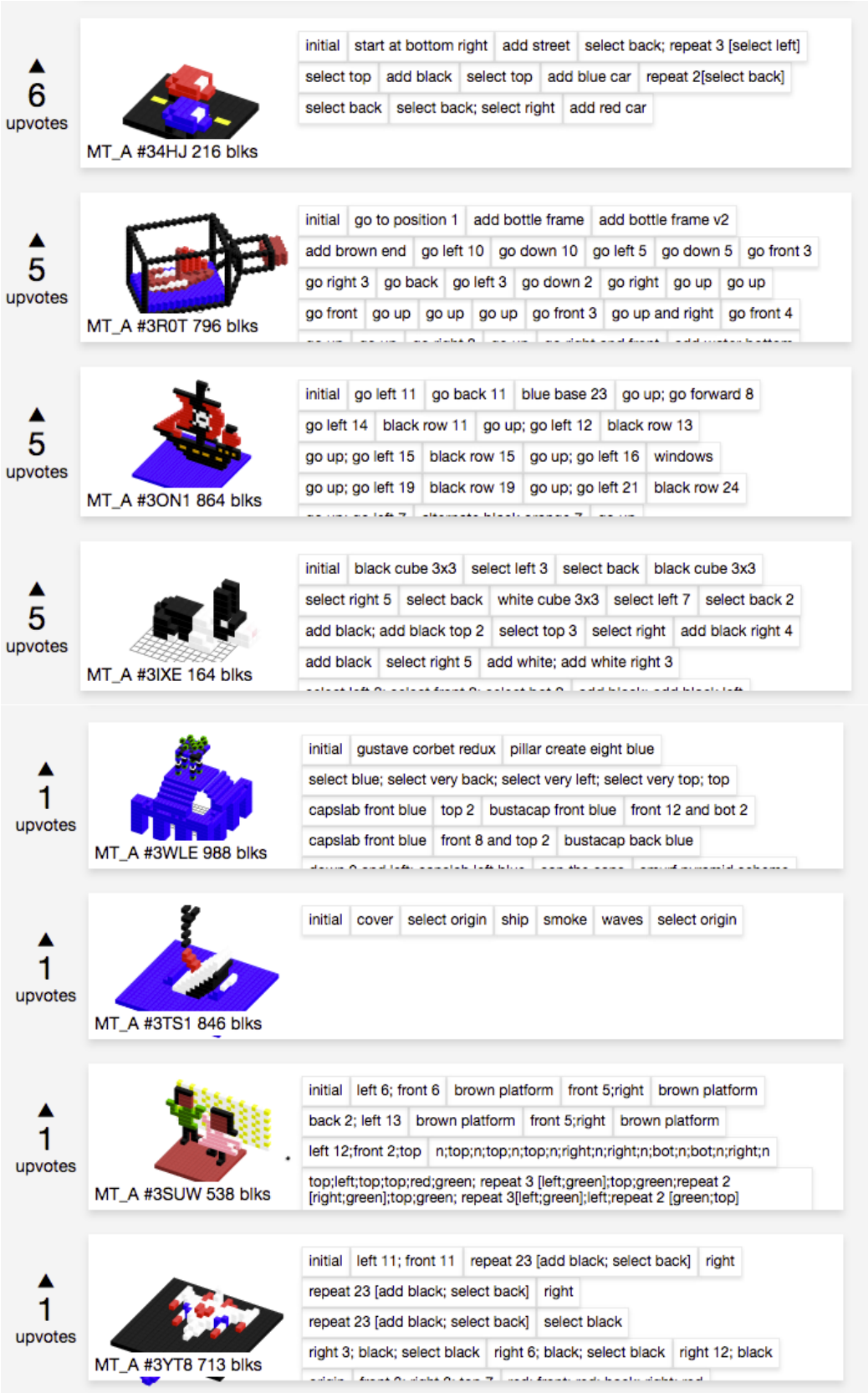


Figure A.2: Selected leaderboard entries and the sequence of commands producing them.

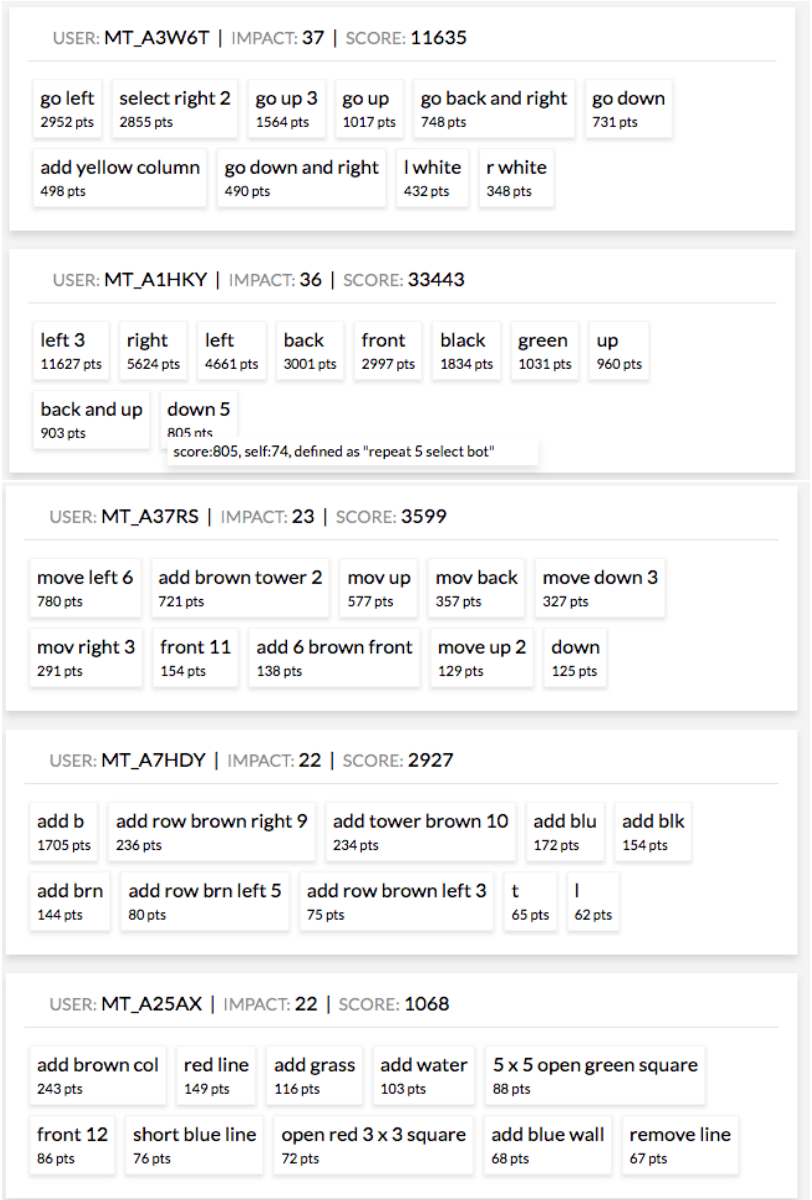


Figure A.3: Leaderboard of citations.

Bibliography

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

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

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

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

A. Azaria, J. Krishnamurthy, and T. M. Mitchell. 2016. Instructable intelligent personal agent. In *Association for the Advancement of Artificial Intelligence (AAAI)*. pages 2681–2689.

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

Y. Bar-Hillel. 1964. *Language and Information: Selected Essays on Their Theory and Application*. Addison-Wesley/The Jerusalem Academic Press.

P. Barbara. 1995. Lexical semantics and compositionality. *An Invitation to Cognitive Science* 0.

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

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

D. G. Bobrow. 1964. *Natural language input for a computer problem solving system*. Ph.D. thesis, Massachusetts Institute of Technology.

K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *International Conference on Management of Data (SIGMOD)*. pages 1247–1250.

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

S. Branavan, L. Zettlemoyer, and R. Barzilay. 2010. Reading between the lines: Learning to map high-level instructions to commands. In *Association for Computational Linguistics (ACL)*. pages 1268–1277.

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

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

G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam. 2017. Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant. In *World Wide Web (WWW)*. pages 341–350.

L. Campbell. 1998. *Historical Linguistics: An Introduction*. Edinburgh University Press.

N. Cesa-Bianchi and G. Lugosi. 2006. *Prediction, learning, and games*. Cambridge University Press.

D. L. Chen. 2012. Fast online lexicon learning for grounded language acquisition. In *Association for Computational Linguistics (ACL)*.

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

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

E. W. Dijkstra. 1978. On the foolishness of "natural language programming". *EWD667* .

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

B. Dostert and F. B. Thompson. 1969a. REL: A rapidly extensible language system I. In *International Conference on Computational Linguistics (COLING)*.

B. Dostert and F. B. Thompson. 1969b. REL: A rapidly extensible language system II. REL English. In *International Conference on Computational Linguistics (COLING)*.

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

M. Frank and N. D. Goodman. 2012. Predicting pragmatic reasoning in language games. *Science* 336:998–998.

H. Giles. 2008. *Communication accommodation theory*. Sage Publications, Inc.

D. Golland, P. Liang, and D. Klein. 2010. A game-theoretic approach to generating spatial descriptions. In *Empirical Methods in Natural Language Processing (EMNLP)*.

N. Goodman and D. Lassiter. 2015. *Probabilistic Semantics and Pragmatics: Uncertainty in Language and Thought*. The Handbook of Contemporary Semantic Theory, 2nd Edition Wiley-Blackwell.

H. P. Grice. 1975. Logic and conversation. *Syntax and semantics* 3:41–58.

S. Gulwani and M. Marron. 2014. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data, SIGMOD*. pages 803–814.

M. E. Ireland, R. B. Slatcher, P. W. Eastwick, L. E. Scissors, E. J. Finkel, and J. W. Pennebaker. 2011. Language style matching predicts relationship initiation and stability. *Psychological Science* 22(1):39–44.

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

R. Jia, L. Heck, D. Hakkani-Tür, and G. Nikolov. 2017. Learning concepts through conversations in spoken dialogue systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*.

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

G. Jäger. 2008. Game theory in semantics and pragmatics. Technical report, University of Tübingen.

R. Kittredge and J. Lehrberger. 1982. *Sublanguage: Studies of Language in Restricted Semantic Domains*. B. Blackwell.

T. Kollar, S. Tellex, D. Roy, and N. Roy. 2010. Grounding verbs of motion in natural language commands to robots. In *International Symposium on Experimental Robotics (ISER)*.

I. Konstas, S. Iyer, M. Yatskar, Y. Choi, and L. Zettlemoyer. 2017. Neural AMR: sequence-to-sequence models for parsing and generation. *CoRR* 0.

S. Krashen. 1982. *Principles and Practice in Second Language Acquisition*. Pergamon Press.

J. Krishnamurthy and T. Kollar. 2013. Jointly learning to parse and perceive: Connecting natural language to the physical world. *Transactions of the Association for Computational Linguistics (TACL)* 1:193–206.

P. K. Kuhl. 2004. Early language acquisition: cracking the speech code. *Nature Reviews Neuroscience* .

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

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

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Empirical Methods in Natural Language Processing (EMNLP)*. pages 1223–1233.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*. pages 1512–1523.

J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan. 2016a. A persona-based neural conversation model. In *Association for Computational Linguistics (ACL)*.

J. Li, M. Galley, C. Brockett, J. Gao, and W. B. Dolan. 2016b. A diversity-promoting objective function for neural conversation models. In *Human Language Technology and North American Association for Computational Linguistics (HLT/NAACL)*.

P. Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408* .

P. Liang. 2016. Learning executable semantic parsers for natural language understanding. *Communications of the ACM* 59.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*. pages 590–599.

P. Liang and C. Potts. 2015. Bringing machine learning and compositional semantics together. *Annual Reviews of Linguistics* 1(1):355–376.

X. V. Lin, C. Wang, D. Pang, K. Vu, L. Zettlemoyer, and M. D. Ernst. 2017 2017. Program synthesis from natural language using recurrent neural networks. Technical Report 0, University of Washington.

R. Lowe, N. Pow, I. Serban, and J. Pineau. 2015. The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. *arXiv preprint arXiv:1506.08909* .

E. Markman and G. F. Wachtel. 1988. Children's use of mutual exclusivity to constrain the meanings of words. *Cognitive Psychology* 20:125–157.

C. Matuszek, N. FitzGerald, L. Zettlemoyer, L. Bo, and D. Fox. 2012. A joint model of language and perception for grounded attribute learning. In *International Conference on Machine Learning (ICML)*. pages 1671–1678.

S. Miller, D. Stallard, R. Bobrow, and R. Schwartz. 1996. A fully statistical approach to natural language interfaces. In *Association for Computational Linguistics (ACL)*. pages 55–61.

W. Monroe and C. Potts. 2015. Learning in the Rational Speech Acts model. In *Proceedings of 20th Amsterdam Colloquium*.

R. Montague. 1973. The proper treatment of quantification in ordinary English. In *Approaches to Natural Language*. pages 221–242.

P. Pasupat and P. Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Association for Computational Linguistics (ACL)*.

H. Reckman, J. Orkin, and D. Roy. 2010. Learning meanings of words and constructions, grounded in a virtual game. In *Conference on Natural Language Processing (KONVENS)*.

J. Sachs, B. Bard, and M. L. Johnson. 1981. Language learning with restricted input: Case studies of two hearing children of deaf parents. *Applied Psycholinguistics* 0.

N. J. Smith, N. D. Goodman, and M. C. Frank. 2013. Learning and using language via recursive pragmatic reasoning about other agents. In *Advances in Neural Information Processing Systems (NIPS)*.

A. Srinivasan and J. Stasko. 2017. Natural language interfaces for data analysis with visualization: Considering what has and could be asked. *EuroVis* .

S. Tellex, R. Knepper, A. Li, D. Rus, and N. Roy. 2014. Asking for help using inverse semantics. In *Robotics: Science and Systems (RSS)*.

S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. J. Teller, and N. Roy. 2011. Understanding natural language commands for robotic navigation and mobile manipulation. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

F. B. Thompson and B. H. Thompson. 1975. Practical natural language processing: The REL system as prototype. *Advances in Computers* 13:109–168.

A. M. Turing. 1950. Computing machinery and intelligence. *Mind* 49:433–460.

S. Ullmann. 1962. *Semantics: An Introduction to the Science of Meaning*. W. de Gruyter.

A. Vogel, M. Bodoia, C. Potts, and D. Jurafsky. 2013. Emergence of gricean maxims from multi-agent decision theory. In *North American Association for Computational Linguistics (NAACL)*. pages 1072–1081.

A. Vogel and D. Jurafsky. 2010. Learning to follow navigational directions. In *Association for Computational Linguistics (ACL)*. pages 806–814.

S. I. Wang, S. Ginn, P. Liang, and C. D. Manning. 2017. Naturalizing a programming language via interactive learning. In *Association for Computational Linguistics (ACL)*.

S. I. Wang, P. Liang, and C. Manning. 2016. Learning language games through interaction. In *Association for Computational Linguistics (ACL)*.

Y. Wang, J. Berant, and P. Liang. 2015. Building a semantic parser overnight. In *Association for Computational Linguistics (ACL)*.

J. Weizenbaum. 1966. ELIZA–a computer program for the study of natural language communication between man and machine. *Communications of the ACM* 9(1):36–45.

T. Winograd. 1972. *Understanding Natural Language*. Academic Press.

L. Wittgenstein. 1953. *Philosophical Investigations*. Blackwell, Oxford.

Y. W. Wong and R. J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Association for Computational Linguistics (ACL)*. pages 960–967.

W. A. Woods, R. M. Kaplan, and B. N. Webber. 1972. The lunar sciences natural language information system: Final report. Technical report, BBN Report 2378, Bolt Beranek and Newman Inc.

M. Zelle and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Association for the Advancement of Artificial Intelligence (AAAI)*. pages 1050–1055.

L. S. Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence (UAI)*. pages 658–666.

L. S. Zettlemoyer and M. Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*. pages 678–687.