Domain Knowledge Acquisition via Language Grounding

by

Tao Lei

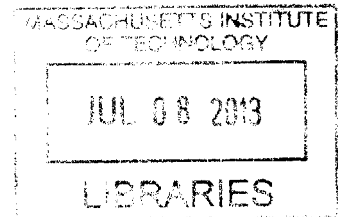
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

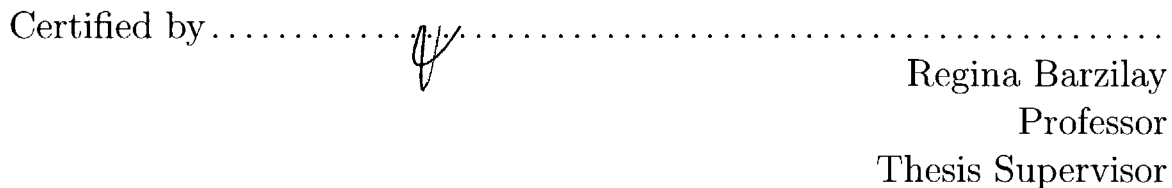
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

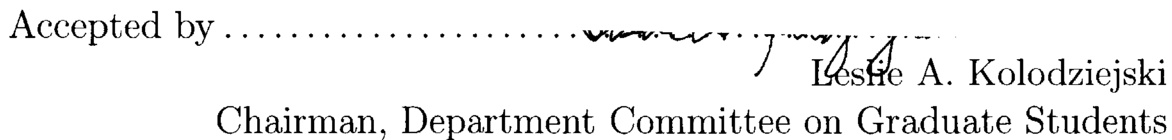


© Massachusetts Institute of Technology 2013. All rights reserved.

Author ......
Department of Electrical Engineering and Computer Science
May 22, 2013



Certified by ....
Regina Barzilay
Professor
Thesis Supervisor



Accepted by ....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Domain Knowledge Acquisition via Language Grounding

by

Tao Lei

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis addresses the language grounding problem at the level of word relation extraction. We propose methods to acquire knowledge represented in the form of relations and utilize them in two domain applications, high-level planning in a complex virtual world and input parser generation from input format specifications.

In the first application, we propose a reinforcement learning framework to jointly learn to predict precondition relations from text and to perform high-level planning guided by those relations. When applied to a complex virtual world and text describing that world, our relation extraction technique performs on par with a supervised baseline, and we show that a high-level planner utilizing these extracted relations significantly outperforms a strong, text unaware baseline.

In the second application, we use a sampling framework to predict relation trees and to generate input parser code from those trees. Our results show that our approach outperforms a state-of-the-art semantic parser on a dataset of input format specifications from the ACM International Collegiate Programming Contest, which were written in English for humans with no intention of providing support for automated processing.

Thesis Supervisor: Regina Barzilay
Title: Professor

Acknowledgments

I am grateful to my advisor Regina Barzilay for her contant support and advice in my research. Her energy and enthusiasm contributed to this work were the key to the success.

I have been very fortunate to work with many fantastic people at MIT. This work would not have been possible without the collaborations with S.R.K. Branavan, Nate Kushman, Fan Long, Martin Rinard and my advisor Regina Barzilay. I would like to thank Yuan Zhang, Yevgeni Berzak, Yoong Keok Lee, Tahira Naseem, Zach Hynes, Yonathan Belinkov, Karthik Rajagopal and Rusmin Soetjipto for their support on my research and life in general. I would also like to acknowledge the support of Battelle Memorial Institute (PO #300662) and the NFS (Grant IIS-0835652).

I dedicate this thesis to my wonderful family: my parents, Jianjun and Yueming, from whom I have learnt many things about life; my grandfather Keqiu who taught me Chinese poems, the art of language; and my grandmother, uncles, aunts and cousins.

Bibliographic Note

Portions of this thesis are based on the following papers:

> "Learning High-Level Planning from Text" by S.R.K Branavan, Nate Kushman, Tao Lei and Regina Barzilay. In the Proceedings of of the 50th Annual Meeting of the Association for Computational Linguistics (ACL)
>
> "From Natural Language Specifications to Program Input Parsers" by Tao Lei, Fan Long, Regina Barzilay and Martin Rinard. To appear in the Proceedings of the 51th Annual Meeting of the Association for Computational Linguistics (ACL)

The code and data for methods presented in this thesis are available at:

`http://groups.csail.mit.edu/rbg/code/planning`

`http://groups.csail.mit.edu/rbg/code/nl2p`

Contents

1 Introduction **10**

1.1 Learning High-Level Planning from Text 11

1.2 Generating Input Parsers from Natural Language Specifications . . . 12

1.3 Thesis Overview 12

2 Learning High-Level Planning from Text **14**

2.1 Introduction 14

2.2 Related Work 17

2.3 The Approach 19

2.3.1 Problem Formulation 19

2.3.2 Model 21

2.3.3 Applying the Model 25

2.4 Experiments 27

2.4.1 Experimental Setup 27

2.4.2 Results 30

3 Generating Program Input Parsers from Natural Language Specifications **33**

3.1 Introduction 33

3.2 Related Work 37

3.3 The Approach 38

3.3.1 Problem Formulation 38

3.3.2 Model 40

3.4 Experiments 46
3.4.1 Experimental Setup 46
3.4.2 Experimental Results 49

4 Conclusions and Future Work **52**

List of Figures

2-1 Text description of preconditions and effects (a), and the low-level actions connecting them (b). 15

2-2 A high-level plan showing two subgoals in a precondition relation. The corresponding sentence is shown above. 19

2-3 Example of the precondition dependencies present in the Minecraft domain. 25

2-4 The performance of our model and a supervised SVM baseline on the precondition prediction task. Also shown is the F-Score of the full set of *Candidate Relations* which is used unmodified by *All Text*, and is given as input to our model. Our model's F-score, averaged over 200 trials, is shown with respect to learning iterations. 30

2-5 Examples of precondition relations predicted by our model from text. Check marks (✓) indicate correct predictions, while a cross (✗) marks the incorrect one – in this case, a valid relation that was predicted as invalid by our model. Note that each pair of highlighted noun phrases in a sentence is a *Candidate Relation*, and pairs that are not connected by an arrow were correctly predicted to be invalid by our model. . . . 30

2-6 Percentage of problems solved by various models on Easy and Hard problem sets. 31

2-7 The top five positive features on words and dependency types learned by our model (above) and by SVM (below) for precondition prediction. 32

3-1 An example of (a) one natural language specification describing program input data; (b) the corresponding specification tree representing the program input structure; and (c) two input examples 34
3-2 Input parser code for reading input files specified in Figure 3-1. . . . 36
3-3 An example of generating input parser code from text: (a) a natural language input specification; (b) a specification tree representing the input format structure (we omit the background phrases in this tree in order to give a clear view of the input format structure); and (c) formal definition of the input format constructed from the specification tree, represented as a context-free grammar in Backus-Naur Form with additional size constraints. 39
3-4 Precision and Recall of our model by varying the percentage of weak supervision. The green lines are the performance of *Aggressive* baseline trained with full weak supervision. 49
3-5 Precision and Recall of our model by varying the number of available input examples per text specification. 50
3-6 Examples of dependencies and key phrases predicted by our model. Green marks correct key phrases and dependencies and red marks incorrect ones. The missing key phrases are marked in gray. 51

List of Tables

2.1 A comparison of complexity between Minecraft and some domains used in the IPC-2011 sequential satisficing track. In the Minecraft domain, the number of objects, predicate types, and actions is significantly larger. 26

2.2 Example text features. A subgoal pair $\langle x_i, x_j \rangle$ is first mapped to word tokens using a small grounding table. Words and dependencies are extracted along paths between mapped target words. These are combined with path directions to generate the text features. 26

2.3 Examples in our seed grounding table. Each predicate is mapped to one or more noun phrases that describe it in the text. 27

2.4 Percentage of tasks solved successfully by our model and the baselines. All performance differences between methods are statistically significant at $p \leq .01$. 31

3.1 Example of feature types and values. To deal with sparsity, we map variable names such as "N" and "X" into a category word "VAR" in word features. 45

3.2 Statistics for 106 ICPC specifications. 46

3.3 Average % Recall and % Precision of our model and all baselines over 20 independent runs. 48

Chapter 1

Introduction

Natural languages are the medium in which human knowledge is recorded and communicated. Today, the traditional way to infuse human knowledge into computer systems is to manually encode knowledge into heuristics, or into the model structure itself. However, with the knowledge often comes in the form of text documents in natural language, it would be both possible and important for machines to automatically access and leverage the information from the text, and effectively perform tasks that require human knowledge. For example, computers could solve traditional hard planning tasks by acquiring domain knowledge from text or generate program code by reading code specifications. Our goal is to automate machines to acquire domain knowledge from text and therefore improve their performance in various applications.

Today much research in natural language processing have focused on developing various methods for learning language semantics from text effectively [23, 10, 43, 44]. Commonly used semantic annotation schemes in these work are drived from linguistic formalism of semantics. However, these is no empirical evidence indicating which scheme is good in terms of real-world applicability – the correctness of the methods is evaluated in terms of the extraction and classification of textual entities (e.g. pairs of words/phrases) that realize the semantic relations, rather than the performance and correctness of an application system that uses such knowledge. The grounding of language in real applications, however, allows us to define, predict and evaluate semantic representations with respect to the performance of the application, and

therefore provides us a natural notion of language semantics. In fact, some recent work have had success grounding linguistic analysis in various applications [51, 12, 55, 34, 14, 33], making it possible to automatically execute Windows commands [6, 9] and to better play PC games [7] by reading text.

In this thesis, we investigate statistical models for acquiring domain knowledge (in the form of relations) and its applications in two new domains, high-level planning and input parser code generation. In contrast to traditional work in semantics, the knowledge we extract is utilized in the corresponding application and improves the application performance. In addition, we aim to learn and evaluate our model based on the application feedback without any human annotations on the text. Our work supplements previous work which grounds words to objects only, while in our case we aim to predict abstract pragmatic relations from their expressions in natural language. Our work also extends previous work which assumes the information is expressed in a single sentence, while in the parser code generation problem we predict a semantic structure at the level of the whole document that consists of multiple sentences.

1.1 Learning High-Level Planning from Text

We first address the problem of planning in a complex virtual world, in which the goal is to search for and execute a sequence of operations to complete certain planning tasks. Comprehending action preconditions and effects is important for this problem as it is the essential step in modeling the dynamics of the world. We express the semantics of precondition relations extracted from text in terms of planning operations. The challenge of modeling this connection is to ground language at the level of relations. This type of grounding enables us to create high-level plans based on language abstractions. Our model jointly learns to predict precondition relations from text and to perform high-level planning guided by those relations. We implement this idea in the reinforcement learning framework using feedback automatically obtained from plan execution attempts. When applied to a complex virtual world and text describing that world, our relation extraction technique performs on par with a

supervised baseline, yielding an F-measure of 66% compared to the baseline's 65%. Additionally, we show that a high-level planner utilizing these extracted relations significantly outperforms a strong, text unaware baseline – successfully completing 80% of planning tasks as compared to 69% for the baseline.

1.2 Generating Input Parsers from Natural Language Specifications

In the second problem, we present a method for automatically generating input parsers from English specifications of input file formats. The need to automate this task arises because input format specifications are almost described in natural languages, with these specifications then manually translated by a programmer into the code that reads the corresponding input. Our method can eliminate such development overhead by automating this process. We use a Bayesian generative model to capture relevant natural language phenomena and translate the English specification into a specification tree, which is then translated into a C++ input parser. We model the problem as a joint dependency parsing and semantic role labeling task. Our method is based on two sources of information: (1) the correlation between the text and the specification tree and (2) noisy supervision as determined by the success of the generated C++ parser in reading input examples. Our results show that our approach achieves 80.0% F-Score accuracy compared to an F-Score of 66.7% produced by a state-of-the-art semantic parser on a dataset of input format specifications from the ACM International Collegiate Programming Contest (which were written in English for humans with no intention of providing support for automated processing).

1.3 Thesis Overview

The remainder of the thesis is organized as follows: In Chapter 2, we provide details of our model to high-level planning and our experimental results. In Chapter 3 we describe our method of generating input parsers from natural language specifications

and present the empirical results. Chapter 4 concludes with the main ideas and the contributions of this work.

Chapter 2

Learning High-Level Planning from Text

2.1 Introduction

Understanding action preconditions and effects is a basic step in modeling the dynamics of the world. For example, having *seeds* is a precondition for growing *wheat*. Not surprisingly, preconditions have been extensively explored in various sub-fields of AI. However, existing work on action models has largely focused on tasks and techniques specific to individual sub-fields with little or no interconnection between them. In NLP, precondition relations have been studied in terms of the linguistic mechanisms that realize them, while in classical planning, these relations are viewed as a part of world dynamics. In this paper, we bring these two parallel views together, grounding the linguistic realization of these relations in the semantics of planning operations.

The challenge and opportunity of this fusion comes from the mismatch between the abstractions of human language and the granularity of planning primitives. Consider, for example, text describing a virtual world such as Minecraft[1] and a formal description of that world using planning primitives. Due to the mismatch in granularity, even the simple relations between *wood*, *pickaxe* and *stone* described in the

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

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

(a)

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

(b)

Figure 2-1: Text description of preconditions and effects (a), and the low-level actions connecting them (b).

sentence in Figure 2-1a results in dozens of low-level planning actions in the world, as can be seen in Figure 2-1b. While the text provides a high-level description of world dynamics, it does not provide sufficient details for successful plan execution. On the other hand, planning with low-level actions does not suffer from this limitation, but is computationally intractable for even moderately complex tasks. As a consequence, in many practical domains, planning algorithms rely on manually-crafted high-level abstractions to make search tractable [22, 31].

The central idea of our work is to express the semantics of precondition relations extracted from text in terms of planning operations. For instance, the precondition relation between pickaxe and stone described in the sentence in Figure 2-1a indicates that plans which involve obtaining stone will likely need to first obtain a pickaxe. The novel challenge of this view is to model grounding at the level of relations, in contrast to prior work which focused on object-level grounding. We build on the intuition that the validity of precondition relations extracted from text can be informed by the execution of a low-level planner.[2] This feedback can enable us to learn these relations

[2]If a planner can find a plan to successfully obtain stone after obtaining a pickaxe, then a pickaxe is likely a precondition for stone. Conversely, if a planner obtains stone without first obtaining a pickaxe, then it is likely not a precondition.

proposed a number of techniques for generating them [29, 54, 35, 3]. In general, these techniques use static analysis of the low-level domain to induce effective high-level abstractions. In contrast, our focus is on learning the abstraction from natural language. Thus our technique is complementary to past work, and can benefit from human knowledge about the domain structure.

2.3 The Approach

2.3.1 Problem Formulation

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

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

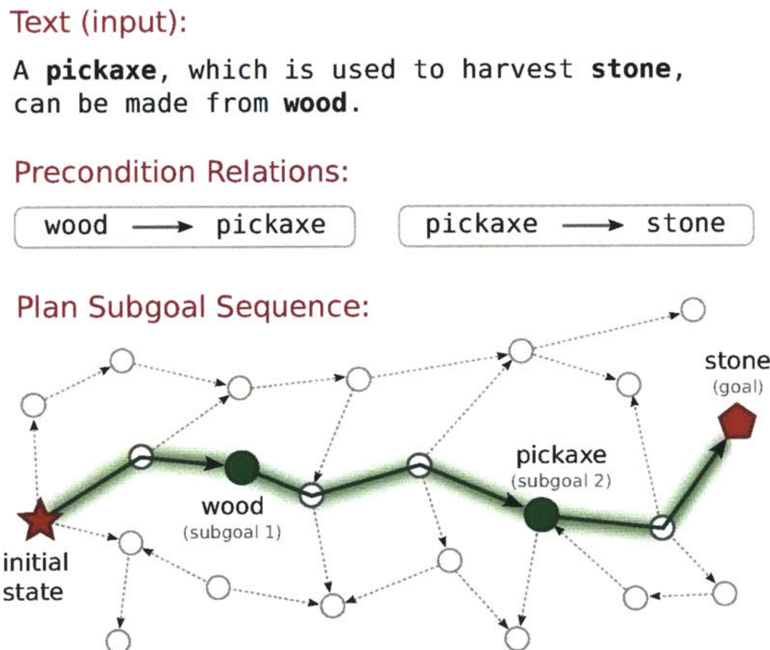


Figure 2-2: A high-level plan showing two subgoals in a precondition relation. The corresponding sentence is shown above.

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

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

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

We evaluate our algorithm in the Minecraft virtual world, using a large collection of user-generated on-line documents as our source of textual information. Our results demonstrate the strength of our relation extraction technique – while using planning feedback as its only source of supervision, it achieves a precondition relation extraction accuracy on par with that of a supervised SVM baseline. Specifically, it yields an F-score of 66% compared to the 65% of the baseline. In addition, we show that these extracted relations can be used to improve the performance of a high-level planner. As baselines for this evaluation, we employ the Metric-FF planner [25],[3] as well as a text-unaware variant of our model. Our results show that our text-driven high-level planner significantly outperforms all baselines in terms of completed planning tasks – it successfully solves 80% as compared to 41% for the Metric-FF planner and 69% for the text unaware variant of our model. In fact, the performance of our method approaches that of an oracle planner which uses manually-annotated preconditions.

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

2.2 Related Work

Extracting Event Semantics from Text The task of extracting preconditions and effects has previously been addressed in the context of lexical semantics [43, 44]. These approaches combine large-scale distributional techniques with supervised learning to identify desired semantic relations in text. Such combined approaches have also been shown to be effective for identifying other relationships between events, such as causality [23, 10, 5, 4, 19].

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

Learning Semantics via Language Grounding Our work fits into the broad area of grounded language acquisition, where the goal is to learn linguistic analysis from a situated context [40, 45, 56, 20, 36, 37, 8, 34, 52]. Within this line of work, we are most closely related to the reinforcement learning approaches that learn language by interacting with an external environment [8, 9, 52, 7].

The key distinction of our work is the use of grounding to learn abstract pragmatic relations, i.e. to learn linguistic patterns that describe relationships between objects in the world. This supplements previous work which grounds words to objects in the world [8, 52]. Another important difference of our setup is the way the textual information is utilized in the situated context. Instead of getting step-by-step instructions from the text, our model uses text that describes general knowledge about the domain structure. From this text, it extracts relations between objects in the world which hold independently of any given task. Task-specific solutions are then constructed by a planner that relies on these relations to perform effective high-level planning.

Hierarchical Planning It is widely accepted that high-level plans that factorize a planning problem can greatly reduce the corresponding search space [39, 1]. Previous work in planning has studied the theoretical properties of valid abstractions and

tion of the world state, preconditions and effects are merely single term predicates, x_i, in this world state. We assume that we are given a seed mapping between a predicate x_i, and the word types in the document that reference it (see Table 2.3 for examples). Thus, for each predicate pair $\langle x_k, x_l \rangle$, we want to utilize the text to predict whether x_k is a precondition for x_l; i.e., $x_k \rightarrow x_l$. For example, from the text in Figure 2-2, we want to predict that possessing a pickaxe is a precondition for possessing stone. Note that this relation implies the reverse as well, i.e. x_l can be interpreted as the effect of an action sequence performed on state x_k.

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

2.3.2 Model

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

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

We formally define these steps below.

Modeling Precondition Relations Given a document d, and a set of subgoal pairs $\langle x_i, x_j \rangle$, we want to predict whether subgoal x_i is a precondition for x_j. We assume that precondition relations are generally described within single sentences. We first use our seed grounding in a preprocessing step where we extract all predicate pairs where both predicates are mentioned in the same sentence. We call this set the *Candidate Relations*. Note that this set will contain many invalid relations since co-occurrence in a sentence does not necessarily imply a valid precondition relation.[4] Thus for each sentence, $\vec{w}_k$, associated with a given *Candidate Relation*, $x_i \rightarrow x_j$, our

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

```
    Input: A document d, Set of planning tasks G,
1          Set of candidate precondition relations C_all,
2          Reward function r(), Number of iterations T
3   Initialization: Model parameters θ_x = 0 and θ_c = 0.
4   for i = 1···T do
5     Sample valid preconditions:
6     C ← ∅
7     foreach ⟨x_i, x_j⟩ ∈ C_all do
8       foreach Sentence w⃗_k containing x_i and x_j do
9         v ~ p(x_i → x_j | w⃗_k, q_k; θ_c)
10        if v = 1 then C = C ∪ ⟨x_i, x_j⟩
11      end
12    end
13    Predict subgoal sequences for each task g.
14    foreach g ∈ G do
15      Sample subgoal sequence x⃗ as follows:
16      for t = 1···n do
17        Sample next subgoal:
18        x_t ~ p(x | x_{t-1}, s_0^g, s_f^g, C; θ_x)
19        Construct low-level subtask from x_{t-1} to x_t
20        Execute low-level planner on subtask
21      end
22      Update subgoal prediction model using Eqn. 2.2
23    end
24    Update text precondition model using Eqn. 2.3
25  end
```

Algorithm 1: A policy gradient algorithm for parameter estimation in our model.

task is to predict whether the sentence indicates the relation. We model this decision via a log linear distribution as follows:

$$p(x_i \to x_j \mid \vec{w}_k, q_k; \theta_c) \propto e^{\theta_c \cdot \phi_c(x_i, x_j, \vec{w}_k, q_k)}, \tag{2.1}$$

where θ_c is the vector of model parameters. We compute the feature function ϕ_c using the seed grounding, the sentence $\vec{w}_k$, and a given dependency parse q_k of the sentence. Given these per-sentence decisions, we predict the set of all valid precondition relations, C, in a deterministic fashion. We do this by considering a precondition $x_i \to x_j$ as valid if it is predicted to be valid by at least one sentence.

Modeling Subgoal Sequences Given a planning goal g, defined by initial and final goal states s_0^g and s_f^g, our task is to predict a sequence of subgoals $\vec{x}$ which will achieve the goal. We condition this decision on our predicted set of valid preconditions C, by modeling the distribution over sequences $\vec{x}$ as:

$$p(\vec{x} \mid s_0^g, s_f^g, C; \theta_x) = \prod_{t=1}^{n} p(x_t \mid x_{t-1}, s_0^g, s_f^g, C; \theta_x),$$

$$p(x_t \mid x_{t-1}, s_0^g, s_f^g, C; \theta_x) \propto e^{\theta_x \cdot \phi_x(x_t, x_{t-1}, s_0^g, s_f^g, C)}.$$

Here we assume that subgoal sequences are Markovian in nature and model individual subgoal predictions using a log-linear model. Note that in contrast to Equation 2.1 where the predictions are goal-agnostic, these predictions are goal-specific. As before, θ_x is the vector of model parameters, and ϕ_x is the feature function. Additionally, we assume a special stop symbol, $x_\emptyset$, which indicates the end of the subgoal sequence.

Parameter Update Parameter updates in our model are done via reinforcement learning. Specifically, once the model has predicted a subgoal sequence for a given goal, the sequence is given to the low-level planner for execution. The success or failure of this execution is used to compute the reward signal r for parameter estimation. This predict-execute-update cycle is repeated until convergence. We assume that our reward signal r strongly correlates with the correctness of model predictions. Therefore, during learning, we need to find the model parameters that maximize expected future reward [46]. We perform this maximization via stochastic gradient ascent, using the standard policy gradient algorithm [53, 47].

We perform two separate policy gradient updates, one for each model component. The objective of the text component of our model is purely to predict the validity of preconditions. Therefore, subgoal pairs $\langle x_k, x_l \rangle$, where x_l is reachable from x_k, are given positive reward. The corresponding parameter update, with learning rate α_c,

takes the following form:

$$\Delta\theta_c \leftarrow \alpha_c\, r \Bigg[\phi_c(x_i, x_j, \vec{w}_k, q_k) - \\ \mathbb{E}_{p(x_{i'} \rightarrow x_{j'}|\cdot)} \left[\phi_c(x_{i'}, x_{j'}, \vec{w}_k, q_k) \right] \Bigg]. \tag{2.2}$$

The objective of the planning component of our model is to predict subgoal sequences that successfully achieve the given planning goals. Thus we directly use plan-success as a binary reward signal, which is applied to each subgoal decision in a sequence. This results in the following update:

$$\Delta\theta_x \leftarrow \alpha_x\, r \sum_t \Bigg[\phi_x(x_t, x_{t-1}, s_0^g, s_f^g, C) - \\ \mathbb{E}_{p(x'_t|\cdot)} \left[\phi_x(x'_t, x_{t-1}, s_0^g, s_f^g, C) \right] \Bigg], \tag{2.3}$$

where t indexes into the subgoal sequence and α_x is the learning rate.

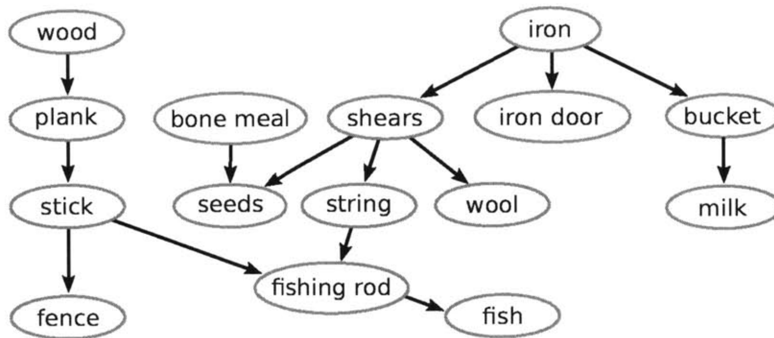


Figure 2-3: Example of the precondition dependencies present in the Minecraft domain.

2.3.3 Applying the Model

We apply our method to Minecraft, a grid-based virtual world. Each grid location represents a tile of either land or water and may also contain resources. Users can freely move around the world, harvest resources and craft various tools and objects from these resources. The dynamics of the world require certain resources or tools as prerequisites for performing a given action, as can be seen in Figure 2-3. For example, a user must first craft a *bucket* before they can collect *milk*.

Defining the Domain In order to execute a traditional planner on the Minecraft domain, we define the domain using the Planning Domain Definition Language (PDDL) [21]. This is the standard task definition language used in the International Planning Competitions (IPC).[5] We define as predicates all aspects of the game state – for example, the location of resources in the world, the resources and objects possessed by the player, and the player's location. Our subgoals x_i and our task goals s_f^g map directly to these predicates. This results in a domain with significantly greater complexity than those solvable by traditional low-level planners. Table 2.1 compares the complexity of our domain with some typical planning domains used in the IPC.

Low-level Planner As our low-level planner we employ Metric-FF [25], the state-of-the-art baseline used in the 2008 International Planning Competition. Metric-FF is a forward-chaining heuristic state space planner. Its main heuristic is to simplify the

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

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

Table 2.1: A comparison of complexity between Minecraft and some domains used in the IPC-2011 sequential satisficing track. In the Minecraft domain, the number of objects, predicate types, and actions is significantly larger.

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

Table 2.2: Example text features. A subgoal pair $\langle x_i, x_j \rangle$ is first mapped to word tokens using a small grounding table. Words and dependencies are extracted along paths between mapped target words. These are combined with path directions to generate the text features.

task by ignoring operator delete lists. The number of actions in the solution for this simplified task is then used as the goal distance estimate for various search strategies.

Features The two components of our model leverage different types of information, and as a result, they each use distinct sets of features. The text component features ϕ_c are computed over sentences and their dependency parses. The Stanford parser [18] was used to generate the dependency parse information for each sentence. Examples of these features appear in Table 2.2. The sequence prediction component takes as input both the preconditions induced by the text component as well as the planning state and the previous subgoal. Thus ϕ_x contains features which check whether two subgoals are connected via an induced precondition relation, in addition to features which are simply the Cartesian product of domain predicates.

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

Table 2.3: Examples in our seed grounding table. Each predicate is mapped to one or more noun phrases that describe it in the text.

2.4 Experiments

2.4.1 Experimental Setup

Datasets As the text description of our virtual world, we use documents from the Minecraft Wiki,[6] the most popular information source about the game. Our manually constructed seed grounding of predicates contains 74 entries, examples of which can be seen in Table 2.3. We use this seed grounding to identify a set of 242 sentences that reference predicates in the Minecraft domain. This results in a set of 694 *Candidate Relations*. We also manually annotated the relations expressed in the text, identifying 94 of the *Candidate Relations* as valid. Our corpus contains 979 unique word types and is composed of sentences with an average length of 20 words.

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

Evaluation Metrics We use our manual annotations to evaluate the type-level ac-

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

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

Baselines To evaluate the performance of our relation extraction, we compare against an SVM classifier[7] trained on the *Gold Relations*. We test the SVM baseline in a leave-one-out fashion.

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

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

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

Experimental Details All experimental results are averaged over 200 independent runs for both our model as well as the baselines. Each of these trials is run for 200 learning iterations with a maximum subgoal sequence length of 10. To find a low-level plan between each consecutive pair of subgoals, our high-level planner internally uses Metric-FF. We give Metric-FF a one-minute timeout to find such a low-level plan. To ensure that the comparison between the high-level planners and the FF baseline is fair, the FF baseline is allowed a runtime of 2,000 minutes. This is an upper bound on the time that our high-level planner can take over the 200 learning

[7]SVM^{light} [26] with default parameters.

iterations, with subgoal sequences of length at most 10 and a one minute timeout. Lastly, during learning we initialize all parameters to zero, use a fixed learning rate of 0.0001, and encourage our model to explore the state space by using the standard ϵ-greedy exploration strategy [46].

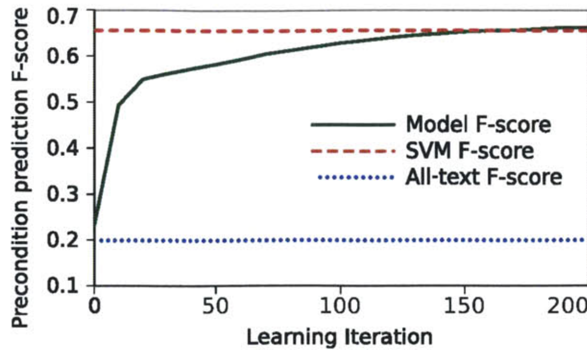


Figure 2-4: The performance of our model and a supervised SVM baseline on the precondition prediction task. Also shown is the F-Score of the full set of *Candidate Relations* which is used unmodified by *All Text*, and is given as input to our model. Our model's F-score, averaged over 200 trials, is shown with respect to learning iterations.

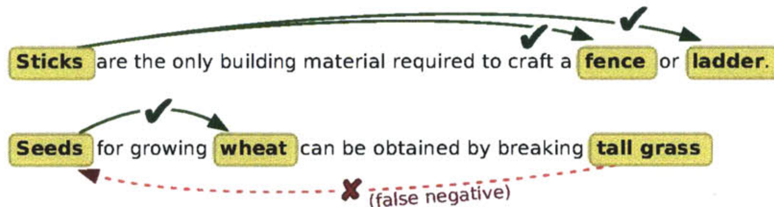


Figure 2-5: Examples of precondition relations predicted by our model from text. Check marks (✓) indicate correct predictions, while a cross (✗) marks the incorrect one – in this case, a valid relation that was predicted as invalid by our model. Note that each pair of highlighted noun phrases in a sentence is a *Candidate Relation*, and pairs that are not connected by an arrow were correctly predicted to be invalid by our model.

2.4.2 Results

Relation Extraction Figure 2-4 shows the performance of our method on identifying preconditions in text. We also show the performance of the supervised SVM baseline. As can be seen, after 200 learning iterations, our model achieves an F-Measure of 66%, equal to the supervised baseline. These results support our hypothesis that planning feedback is a powerful source of supervision for analyzing a given text corpus. Figure 2-5 shows some examples of sentences and the corresponding extracted relations.

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

Table 2.4: Percentage of tasks solved successfully by our model and the baselines. All performance differences between methods are statistically significant at $p \leq .01$.

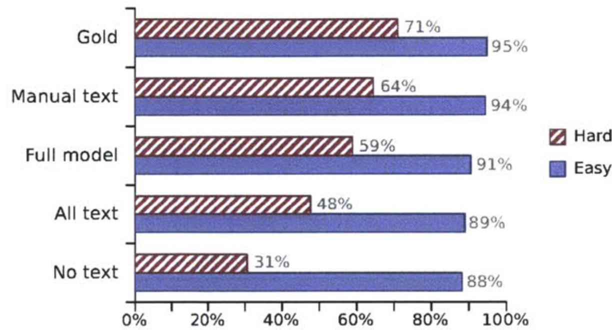


Figure 2-6: Percentage of problems solved by various models on Easy and Hard problem sets.

Planning Performance As shown in Table 2.4 our text-enriched planning model outperforms the text-free baselines by more than 10%. Moreover, the performance improvement of our model over the *All Text* baseline demonstrates that the accuracy of the extracted text relations does indeed impact planning performance. A similar conclusion can be reached by comparing the performance of our model and the *Manual Text* baseline.

The difference in performance of 2.35% between *Manual Text* and *Gold* shows the importance of the precondition information that is missing from the text. Note that *Gold* itself does not complete all tasks – this is largely because the Markov assumption made by our model does not hold for all tasks.[8]

Figure 2-6 breaks down the results based on the difficulty of the corresponding planning task. We measure problem complexity in terms of the low-level steps needed

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

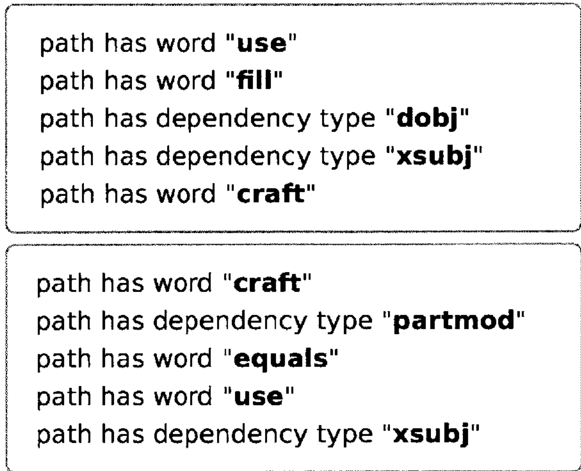


Figure 2-7: The top five positive features on words and dependency types learned by our model (above) and by SVM (below) for precondition prediction.

to implement a manually constructed high-level plan. Based on this measure, we divide the problems into two sets. As can be seen, all of the high-level planners solve almost all of the easy problems. However, performance varies greatly on the more challenging tasks, directly correlating with planner sophistication. On these tasks our model outperforms the *No Text* baseline by 28% and the *All Text* baseline by 11%.

Feature Analysis Figure 2-7 shows the top five positive features for our model and the SVM baseline. Both models picked up on the words that indicate precondition relations in this domain. For instance, the word *use* often occurs in sentences that describe the resources required to make an object, such as "bricks are items used to craft brick blocks". In addition to lexical features, dependency information is also given high weight by both learners. An example of this is a feature that checks for the direct object dependency type. This analysis is consistent with prior work on event semantics which shows lexico-syntactic features are effective cues for learning text relations [5, 4, 19].

Chapter 3

Generating Program Input Parsers from Natural Language Specifications

3.1 Introduction

The general problem of translating natural language specifications into executable code has been around since the field of computer science was founded. Early attempts to solve this problem produced what were essentially verbose, clumsy, and ultimately unsuccessful versions of standard formal programming languages. In recent years however, researchers have had success addressing specific aspects of this problem. Recent advances in this area include the successful translation of natural language commands into database queries [55, 58, 42, 33] and the successful mapping of natural language instructions into Windows command sequences [6, 9].

In this paper we explore a different aspect of this general problem: the translation of natural language input specifications into executable code that correctly parses the input data and generates data structures for holding the data. The need to automate this task arises because input format specifications are almost always described in natural languages, with these specifications then manually translated by a program-

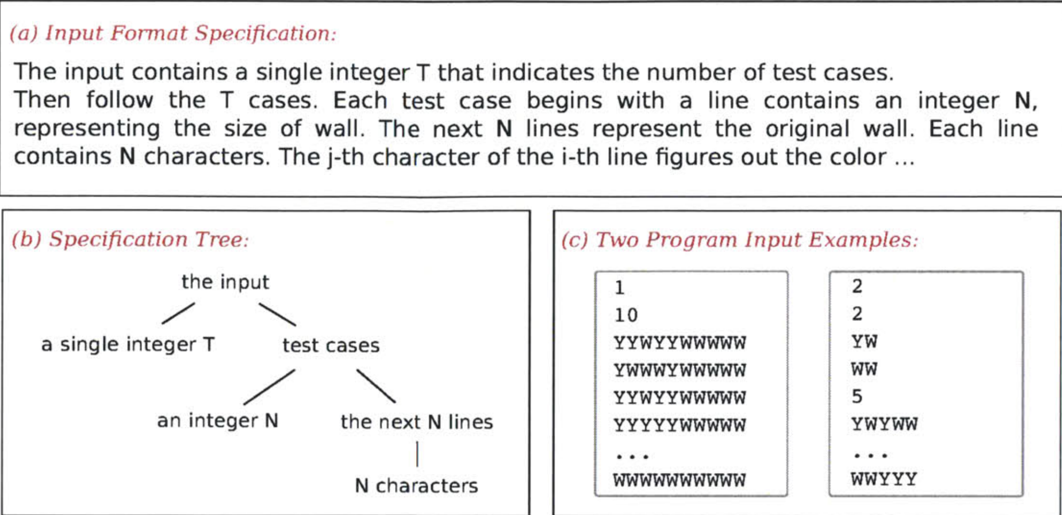


Figure 3-1: An example of (a) one natural language specification describing program input data; (b) the corresponding specification tree representing the program input structure; and (c) two input examples

mer into the code for reading the program inputs. Our method highlights potential to automate this translation, thereby eliminating the manual software development overhead.

Consider the text specification in Figure 3-1a. If the desired parser is implemented in C++, it should create a C++ class whose instance objects hold the different fields of the input. For example, one of the fields of this class is an integer, i.e., "a single integer T" identified in the text specification in Figure 3-1a. Instead of directly generating code from the text specification, we first translate the specification into a *specification tree* (see Figure 3-1b), then map this tree into parser code (see Figure 3-2). We focus on the translation from the text specification to the specification tree.[1]

We assume that each text specification is accompanied by a set of input examples that the desired input parser is required to successfully read. In standard software development contexts, such input examples are usually available and are used to test the correctness of the input parser. Note that this source of supervision is noisy — the generated parser may still be incorrect even when it successfully reads all of the

[1]During the second step of the process, the specification tree is deterministically translated into code.

input examples. Specifically, the parser may interpret the input examples differently from the text specification. For example, the program input in Figure 3-1c can be interpreted simply as a list of strings. The parser may also fail to parse some correctly formatted input files not in the set of input examples. Therefore, our goal is to design a technique that can effectively learn from this weak supervision.

We model our problem as a joint dependency parsing and role labeling task, assuming a Bayesian generative process. The distribution over the space of specification trees is informed by two sources of information: (1) the correlation between the text and the corresponding specification tree and (2) the success of the generated parser in reading input examples. Our method uses a joint probability distribution to take both of these sources of information into account, and uses a sampling framework for the inference of specification trees given text specifications. A specification tree is rejected in the sampling framework if the corresponding code fails to successfully read all of the input examples. The sampling framework also rejects the tree if the text/specification tree pair has low probability.

We evaluate our method on a dataset of input specifications from ACM International Collegiate Programming Contests, along with the corresponding input examples. These specifications were written for human programmers with no intention of providing support for automated processing. However, when trained using the noisy supervision, our method achieves substantially more accurate translations than a state-of-the-art semantic parser [14] (specifically, 80.0% in F-Score compared to an F-Score of 66.7%). The strength of our model in the face of such weak supervision is also highlighted by the fact that it retains an F-Score of 77% even when only one input example is provided for each input specification.

```
1  struct TestCaseType {
2     int N;
3     vector<NLinesType*> lstLines;
4     InputType* pParentLink;
5  }
6
7  struct InputType {
8     int T;
9     vector<TestCaseType*> lstTestCase;
10 }
11
12 TestCaseType* ReadTestCase(FILE * pStream, InputType* pParentLink) {
13    TestCaseType* pTestCase = new TestCaseType;
14    pTestCase→pParentLink = pParentLink;
15
16    ...
17
18    return pTestCase;
19 }
20
21 InputType* ReadInput(FILE * pStream) {
22    InputType* pInput = new InputType;
23
24    pInput→T = ReadInteger(pStream);
25    for (int i = 0; i < pInput→T; ++i) {
26       TestCaseType* pTestCase = new TestCaseType;
27       pTestCase = ReadTestCase (pStream, pInput);
28       pInput→lstTestCase.push_back (pTestCase);
29    }
30
31    return pInput;
32 }
```

Figure 3-2: Input parser code for reading input files specified in Figure 3-1.

3.2 Related Work

Learning Meaning Representation from Text Mapping sentences into structural meaning representations is an active and extensively studied task in NLP. Examples of meaning representations considered in prior research include logical forms based on database query [50, 57, 30, 55, 42, 33, 24], semantic frames [16, 17] and database records [13, 32].

Learning Semantics from Feedback Our approach is related to recent research on learning from indirect supervision. Examples include leveraging feedback available via responses from a virtual world [6] or from executing predicted database queries [11, 14]. While [6] formalize the task as a sequence of decisions and learns from local rewards in a Reinforcement Learning framework, our model learns to predict the whole structure at a time. Another difference is the way our model incorporates the noisy feedback. While previous approaches rely on the feedback to train a *discriminative* prediction model, our approach models a *generative process* to guide structure predictions when the feedback is noisy or unavailable.

NLP in Software Engineering Researchers have recently developed a number of approaches that apply natural language processing techniques to software engineering problems. Examples include analyzing API documents to infer API library specifications [59, 41] and analyzing code comments to detect concurrency bugs [48, 49]. This research analyzes natural language in documentation or comments to better understand existing application programs. Our mechanism, in contrast, automatically generates parser programs from natural language input format descriptions.

3.3 The Approach

3.3.1 Problem Formulation

The task of translating text specifications to input parsers consists of two steps, as shown in Figure 3-3. First, given a text specification describing an input format, we wish to infer a parse tree (which we call a *specification tree*) implied by the text. Second, we convert each specification tree into formal grammar of the input format (represented in Backus-Naur Form) and then generate code that reads the input into data structures. In this paper, we focus on the NLP techniques used in the first step, i.e., learning to infer the specification trees from text. The second step is achieved using a deterministic rule-based tool. [2]

As input, we are given a set of text specifications $\mathbf{w} = \{w^1, \cdots, w^N\}$, where each w^i is a text specification represented as a sequence of noun phrases $\{w^i_k\}$. We use UIUC shallow parser to preprocess each text specificaton into a sequence of the noun phrases.[3] In addition, we are given a set of input examples for each w^i. We use these examples to test the generated input parsers to reject incorrect predictions made by our probabilistic model.

We formalize the learning problem as a dependency parsing and role labeling problem. Our model predicts specification trees $\mathbf{t} = \{t^1, \cdots, t^N\}$ for the text specifications, where each specification tree t^i is a dependency tree over noun phrases $\{w^i_k\}$. In general many program input formats are nested tree structures, in which the tree root denotes the entire chunk of program input data and each chunk (tree node) can be further divided into sub-chunks or primitive fields that appear in the program input (see Figure 3-3). Therefore our objective is to predict a dependency tree that correctly represents the structure of the program input.

In addition, the role labeling problem is to assign a tag z^i_k to each noun phrase w^i_k

[2]Specifically, the specification tree is first translated into the grammar using a set of rules and seed words that identifies basic data types such as **int**. Our implementation then generates a top-down parser since the generated grammar is simple. In general, standard techniques such as Bison and Yacc [28] can generate bottom-up parsers given such grammar.

[3]http://cogcomp.cs.illinois.edu/demo/shallowparse/?id=7

in a specification tree, indicating whether the phrase is a *key phrase* or a *background phrase*. Key phrases are named entities that identify input fields or input chunks appear in the program input data, such as "the input" or "the following lines" in Figure 3-3b. In contrast, background phrases do not define input fields or chunks. These phrases are used to organize the document (e.g., "your program") or to refer to key phrases described before (e.g., "each line").

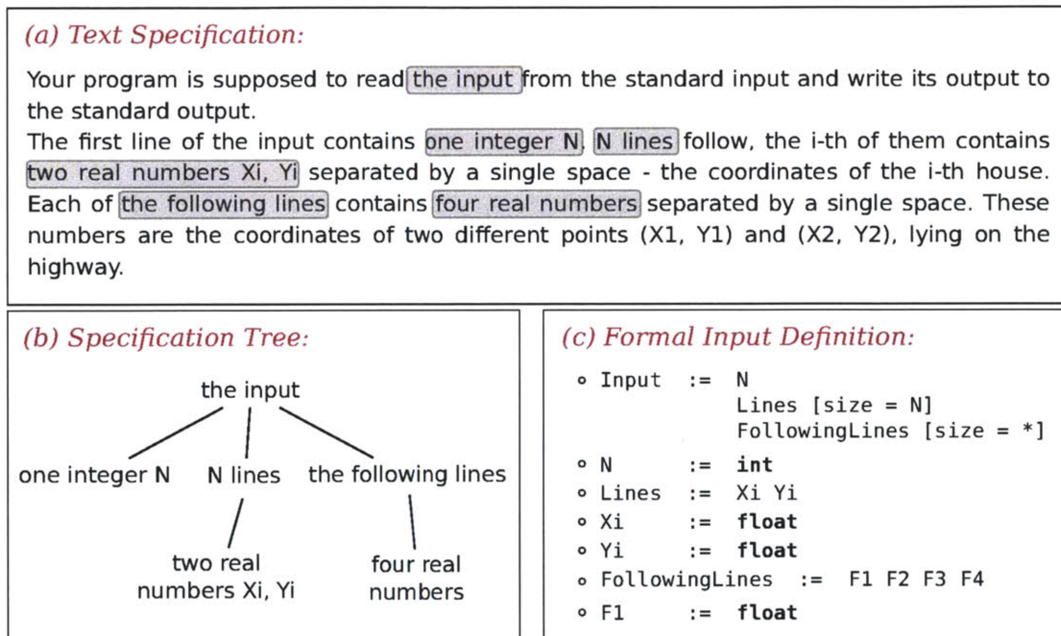


Figure 3-3: An example of generating input parser code from text: (a) a natural language input specification; (b) a specification tree representing the input format structure (we omit the background phrases in this tree in order to give a clear view of the input format structure); and (c) formal definition of the input format constructed from the specification tree, represented as a context-free grammar in Backus-Naur Form with additional size constraints.

3.3.2 Model

We use two kinds of information to bias our model: (1) the quality of the generated code as measured by its ability to read the given input examples and (2) the features over the observed text w^i and the hidden specification tree t^i (this is standard in traditional parsing problems). We combine these two kinds of information into a Bayesian generative model in which the code quality of the specification tree is captured by the prior probability $P(\mathbf{t})$ and the feature observations are encoded in the likelihood probability $P(\mathbf{w}|\mathbf{t})$. The inference jointly optimizes these two factors:

$$P(\mathbf{t}|\mathbf{w}) \propto P(\mathbf{t}) \cdot P(\mathbf{w}|\mathbf{t}).$$

Modeling the Generative Process. We assume the generative model operates by first generating the model parameters from a set of Dirichlet distributions. The model then generates text specification trees. Finally, it generates natural language feature observations conditioned on the hidden specification trees.

The generative process is described formally as follows:

- **Generating Model Parameters:** For every pair of feature type f and phrase tag z, draw a multinomial distribution parameter θ_f^z from a Dirichlet prior $P(\theta_f^z)$. The multinomial parameters provide the probabilities of observing different feature values in the text.

- **Generating Specification Tree:** For each text specification, draw a specification tree t from all possible trees over the sequence of noun phrases in this specification. We denote the probability of choosing a particular specification tree t as $P(t)$.

  Intuitively, this distribution should assign high probability to good specification trees that can produce C++ code that reads all input examples without errors,

we therefore define $P(t)$ as follows:[4]

$$P(t) = \frac{1}{Z} \cdot \begin{cases} 1 & \text{the input parser of } t \text{ reads all} \\ & \text{input examples without error} \\ \epsilon & \text{otherwise} \end{cases}$$

where Z is a normalization factor and ϵ is empirically set to 10^{-6}. In other words, $P(\cdot)$ treats all specification trees that pass the input example test as equally probable candidates and inhibits the model from generating trees which fail the test. Note that we do not know this distribution a priori until the specification trees are evaluated by testing the corresponding C++ code. Because it is intractable to test all possible trees and all possible generated code for a text specification, we never explicitly compute the normalization factor $1/Z$ of this distribution. We therefore use sampling methods to tackle this problem during inference.

- **Generating Features:** The final step generates lexical and contextual features for each tree. For each phrase w_k associated with tag z_k, let w_p be its parent phrase in the tree and w_s be the non-background sibling phrase to its left in the tree. The model generates the corresponding set of features $\phi(w_p, w_s, w_k)$ for each text phrase tuple (w_p, w_s, w_k), with probability $P(\phi(w_p, w_s, w_k))$. We assume that each feature f_j is generated independently:

$$\begin{aligned} P(w|t) &= P(\phi(w_p, w_s, w_k)) \\ &= \prod_{f_j \in \phi(w_p, w_s, w_k)} \theta_{f_j}^{z_k} \end{aligned}$$

where $\theta_{f_j}^{z_k}$ is the j-th component in the multinomial distribution $\theta_f^{z_k}$ denoting the probability of observing a feature f_j associated with noun phrase w_k labeled with tag z_k. We define a range of features that capture the correspondence between the input format and its description in natural language. For example, at the

[4]When input examples are not available, $P(t)$ is just uniform distribution.

unigram level we aim to capture that noun phrases containing specific words such as "cases" and "lines" may be key phrases (correspond to data chunks appear in the input), and that verbs such as "contain" may indicate that the next noun phrase is a key phrase.

The full joint probability of a set $\mathbf{w}$ of N specifications and hidden text specification trees $\mathbf{t}$ is defined as:

$$\begin{aligned} P(\theta, \mathbf{t}, \mathbf{w}) &= P(\theta) \prod_{i=1}^{N} P(t^i) P(w^i | t^i, \theta) \\ &= P(\theta) \prod_{i=1}^{N} P(t^i) \prod_{k} P(\phi(w_p^i, w_s^i, w_k^i)). \end{aligned}$$

Learning the Model During inference, we want to estimate the hidden specification trees $\mathbf{t}$ given the observed natural language specifications $\mathbf{w}$, after integrating the model parameters out, i.e.

$$\mathbf{t} \sim P(\mathbf{t}|\mathbf{w}) = \int_{\theta} P(\mathbf{t}, \theta | \mathbf{w}) d_{\theta}.$$

We use Gibbs sampling to sample variables $\mathbf{t}$ from this distribution. In general, the Gibbs sampling algorithm randomly initializes the variables and then iteratively solves one subproblem at a time. The subproblem is to sample only one variable conditioned on the current values of all other variables. In our case, we sample one hidden specification tree t^i while holding all other trees $\mathbf{t}^{-i}$ fixed:

$$t^i \sim P(t^i | \mathbf{w}, \mathbf{t}^{-i}) \tag{3.1}$$

where $\mathbf{t}^{-i} = (t^1, \cdots, t^{i-1}, t^{i+1}, \cdots, t^N)$.

However directly solving the subproblem (1) in our case is still hard, we therefore use a Metropolis-Hastings sampler that is similarly applied in traditional sentence parsing problems. Specifically, the Hastings sampler approximates (1) by first drawing a new $t^{i'}$ from a tractable proposal distribution Q instead of $P(t^i|\mathbf{w}, \mathbf{t}^{-i})$. We choose

Q to be:

$$Q(t^{i'}|\theta', w^i) \propto P(w^i|t^{i'}, \theta'). \tag{3.2}$$

Then the probability of accepting the new sample is determined using the typical Metropolis Hastings process. Specifically, $t^{i'}$ will be accepted to replace the last t^i with probability:

$$\begin{aligned} R(t^i, t^{i'}) &= \min\left\{1, \frac{P(t^{i'}|\mathbf{w}, \mathbf{t}^{-i})\ Q(t^i|\theta', w^i)}{P(t^i|\mathbf{w}, \mathbf{t}^{-i})\ Q(t^{i'}|\theta', w^i)}\right\} \\ &= \min\left\{1, \frac{P(t^{i'}, \mathbf{t}^{-i}, \mathbf{w})P(w^i|t^i, \theta')}{P(t^i, \mathbf{t}^{-i}, \mathbf{w})P(w^i|t^{i'}, \theta')}\right\}, \end{aligned}$$

in which the normalization factors $1/Z$ are cancelled out. We choose θ' to be the parameter expectation based on the current observations, i.e. $\theta' = E\left[\theta|\mathbf{w}, \mathbf{t}^{-i}\right]$, so that the proposal distribution is close to the true distribution. This sampling algorithm with a changing proposal distribution has been shown to work well in practice [27, 15, 38]. The algorithm pseudo code is shown in Algorithm 2.

To sample from the proposal distribution (2) efficiently, we implement a dynamic programming algorithm which calculates marginal probabilities of all subtrees. The algorithm works similarly to the inside algorithm [2], except that we do not assume the tree is binary. We therefore perform one additional dynamic programming step that sums over all possible segmentations of each span. Once the algorithm obtains the marginal probabilities of all subtrees, a specification tree can be drawn recursively in a top-down manner.

Calculating $P(\mathbf{t}, \mathbf{w})$ in $R(t, t')$ requires integrating the parameters θ out. This has a closed form due to the Dirichlet-multinomial conjugacy:

$$\begin{aligned} P(\mathbf{t}, \mathbf{w}) &= P(\mathbf{t}) \cdot \int_\theta P(\mathbf{w}|\mathbf{t}, \theta) P(\theta) d_\theta \\ &\propto P(\mathbf{t}) \cdot \prod \mathrm{Beta}\left(\mathrm{count}(f) + \alpha\right) \ . \end{aligned}$$

Input: Set of text specification documents $\mathbf{w} = \{w^1, \cdots, w^N\}$,
Number of iterations T

1 Randomly initialize specification trees $\mathbf{t} = \{t^1, \cdots, t^N\}$

2 **for** $iter = 1 \cdots T$ **do**

3   *Sample tree t^i for i-th document:*

4   **for** $i = 1 \cdots N$ **do**

5   *Estimate model parameters:*

6     $\theta' = E\left[\theta' | \mathbf{w}, \mathbf{t}^{-\mathbf{i}}\right]$

7   *Sample a new specification tree from distribution Q:*

8     $t' \sim Q(t'|\theta', w^i)$

9   *Generate and test code, and return feedback:*

10     $f' = \text{CodeGenerator}(w^i, t')$

11   *Calculate accept probability r:*

12     $r = R(t^i, t')$

13   *Accept the new tree with probability r:*

14   With probability $r : t^i = t'$

15   **end**

16 **end**

17 *Produce final structures:*

18 **return** { t^i if t^i gets positive feedback }

Algorithm 2: The sampling framework for learning the model.

Here α are the Dirichlet hyper parameters and count(f) are the feature counts observed in data $(\mathbf{t}, \mathbf{w})$. The closed form is a product of the Beta functions of each feature type.

Model Implementation: We define several types of features to capture the correlation between the hidden structure and its expression in natural language. For example, verb features are introduced because certain preceding verbs such as "contains" and "consists" are good indicators of key phrases. There are 991 unique features in total in our experiments. Examples of features appear in Table 3.1.

We use a small set of 8 seed words to bias the search space. Specifically, we require each leaf key phrase to contain at least one seed word that identifies the C++ primitive data type (such as "integer", "float", "byte" and "string").

Feature Type	Description	Feature Value
Word	each word in noun phrase w_k	lines, VAR
Verb	verbs in noun phrase w_k and the verb phrase before w_k	contains
Distance	sentence distance between w_k and its parent phrase w_p	1
Coreference	w_k share duplicate nouns or variable names with w_p or w_s	True

Table 3.1: Example of feature types and values. To deal with sparsity, we map variable names such as "N" and "X" into a category word "VAR" in word features.

We also encourage a phrase containing the word "input" to be the root of the tree (for example, "the input file") and each coreference phrase to be a background phrase (for example, "each test case" after mentioning "test cases"), by initially adding pseudo counts to Dirichlet priors.

3.4 Experiments

3.4.1 Experimental Setup

Datasets: Our dataset consists of problem descriptions from ACM International Collegiate Programming Contests.[5] We collected 106 problems from ACM-ICPC training websites.[6] From each problem description, we extracted the portion that provides input specifications. Because the test input examples are not publicly available on the ACM-ICPC training websites, for each specification, we wrote simple programs to generate 100 random input examples.

Table 3.2 presents statistics for the text specification set. The data set consists of 424 sentences, where an average sentence contains 17.3 words. The data set contains 781 unique words. The length of each text specification varies from a single sentence to eight sentences. The difference between the average and median number of trees is large. This is because half of the specifications are relatively simple and have a small number of possible trees, while a few difficult specifications have over thousands of possible trees (as the number of trees grows exponentially when the text length increases).

Statistic	Value
Total # of words	7330
Total # of noun phrases	1829
Vocabulary size	781
Avg. # of words per sentence	17.29
Avg. # of noun phrase per document	17.26
Avg. # of possible trees per document	52K
Median # of possible trees per document	79
Min # of possible trees per document	1
Max # of possible trees per document	2M

Table 3.2: Statistics for 106 ICPC specifications.

Evaluation Metrics: We evaluate the model performance in terms of its success in generating a formal grammar that correctly represents the input format (see Figure 3-

[5]Official Website: http://cm.baylor.edu/welcome.icpc
[6]PKU Online Judge: http://poj.org/; UVA Online Judge: http://uva.onlinejudge.org/

3c). As a gold annotation, we construct formal grammars for all text specifications. Our results are generated by automatically comparing the machine-generated grammars with their golden counterparts. If the formal grammar is correct, then the generated C++ parser will correctly read the input file into corresponding C++ data structures.

We use Recall and Precision as evaluation measures:

$$\text{Recall} = \frac{\#\text{ correct structures}}{\#\text{ text specifications}}$$
$$\text{Precision} = \frac{\#\text{ correct structures}}{\#\text{ produced structures}}$$

where the produced structures are the positive structures returned by our framework whose corresponding code successfully reads all input examples (see Algorithm 2 line 18). Note the number of produced structures may be less than the number of text specifications, because structures that fail the input test are not returned.

Baselines: To evaluate the performance of our model, we compare against four baselines.

The *No Learning* baseline is a variant of our model that selects a specification tree without learning feature correspondence. It continues sampling a specification tree for each text specification until it finds one which successfully reads all of the input examples.

The second baseline *Aggressive* is a state-of-the-art semantic parsing framework [14].[7] The framework repeatedly predicts hidden structures (specification trees in our case) using a structure learner, and trains the structure learner based on the execution feedback of its predictions. Specifically, at each iteration the structure learner predicts the most plausible specification tree for each text document:

$$t^i = \text{argmax}_t\ f(w^i, t).$$

Depending on whether the corresponding code reads all input examples successfully

[7]We take the name Aggressive from this paper.

Model	Recall	Precision	F-Score
No Learning	52.0	57.2	54.5
Aggressive	63.2	70.5	66.7
Full Model	**72.5**	**89.3**	**80.0**
Full Model (Oracle)	72.5	100.0	84.1
Aggressive (Oracle)	80.2	100.0	89.0

Table 3.3: Average % Recall and % Precision of our model and all baselines over 20 independent runs.

or not, the (w^i, t^i) pairs are added as an positive or negative sample to populate a training set. After each iteration the structure learner is re-trained with the training samples to improve the prediction accuracy. In our experiment, we follow [14] and choose a structural Support Vector Machine SVM$^{\text{struct}}$ [8] as the structure learner.

The remaining baselines provide an upper bound on the performance of our model. The baseline *Full Model (Oracle)* is the same as our full model except that the feedback comes from an oracle which tells whether the specification tree is correct or not. We use this oracle information in the prior $P(t)$ same as we use the noisy feedback. Similarly the baseline *Aggressive (Oracle)* is the *Aggressive* baseline with access to the oracle.

Experimental Details: Because no human annotation is required for learning, we train our model and all baselines on all 106 ICPC text specifications (similar to unsupervised learning). We report results averaged over 20 independent runs. For each of these runs, the model and all baselines run 100 iterations. For baseline *Aggressive*, in each iteration the SVM structure learner predicts one tree with the highest score for each text specification. If two different specification trees of the same text specification get positive feedback, we take the one generated in later iteration for evaluation.

[8] www.cs.cornell.edu/people/tj/svm_light/svm_struct.html

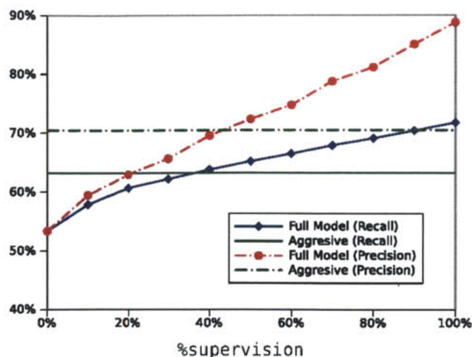


Figure 3-4: Precision and Recall of our model by varying the percentage of weak supervision. The green lines are the performance of *Aggressive* baseline trained with full weak supervision.

3.4.2 Experimental Results

Comparison with Baselines Table 3.3 presents the performance of various models in predicting correct specification trees. As can be seen, our model achieves an F-Score of 80%. Our model therefore significantly outperforms the *No Learning* baseline (by more than 25%). Note that the *No Learning* baseline achieves a low Precision of 57.2%. This low precision reflects the noisiness of the weak supervision - nearly one half of the parsers produced by *No Learning* are actually incorrect even though they read all of the input examples without error. This comparison shows the importance of capturing correlations between the specification trees and their text descriptions. Because our model learns correlations via feature representations, it produces substantially more accurate translations.

While both the *Full Model* and *Aggressive* baseline use the same source of feedback, they capitalize on it in a different way. The baseline uses the noisy feedback to train features capturing the correlation between trees and text. Our model, in contrast, combines these two sources of information in a complementary fashion. This combination allows our model to filter false positive feedback and produce 13% more correct translations than the *Aggressive* baseline.

Clean versus Noisy Supervision To assess the impact of noise on model accuracy, we compare the *Full Model* against the *Full Model (Oracle)*. The two versions

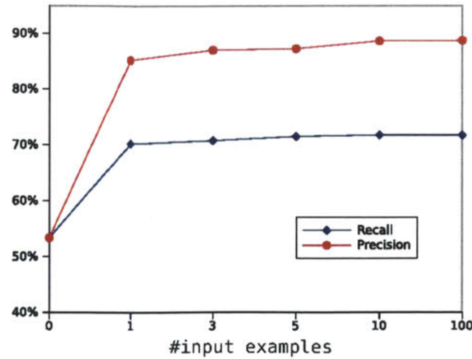


Figure 3-5: Precision and Recall of our model by varying the number of available input examples per text specification.

achieve very close performance (80% v.s 84% in F-Score), even though *Full Model* is trained with noisy feedback. This demonstrates the strength of our model in learning from such weak supervision. Interestingly, *Aggressive (Oracle)* outperforms our oracle model by a 5% margin. This result shows that when the supervision is reliable, the generative assumption limits our model's ability to gain the same performance improvement as discriminative models.

Impact of Input Examples Our model can also be trained in a fully unsupervised or a semi-supervised fashion. In real cases, it may not be possible to obtain input examples for all text specifications. We evaluate such cases by varying the amount of supervision, i.e. how many text specifications are paired with input examples. In each run, we randomly select text specifications and only these selected specifications have access to input examples. Figure 3-4 gives the performance of our model with 0% supervision (totally unsupervised) to 100% supervision (our full model). With much less supervision, our model is still able to achieve performance comparable with the *Aggressive* baseline.

We also evaluate how the number of provided input examples influences the performance of the model. Figure 3-5 indicates that the performance is largely insensitive to the number of input examples — once the model is given even one input example, its performance is close to the best performance it obtains with 100 input examples. We attribute this phenomenon to the fact that if the generated code is incorrect, it is unlikely to successfully parse any input.

(a) The input contains several testcases.
Each is specified by two strings S, T of alphanumeric ASCII characters

(b) The next N lines of the input file contain the Cartesian coordinates of watchtowers, one pair of coordinates per line.

Figure 3-6: Examples of dependencies and key phrases predicted by our model. Green marks correct key phrases and dependencies and red marks incorrect ones. The missing key phrases are marked in gray.

Case Study Finally, we consider some text specifications that our model does not correctly translate. In Figure 3-6a, the program input is interpreted as a list of character strings, while the correct interpretation is that the input is a list of string pairs. Note that both interpretations produce C++ input parsers that successfully read all of the input examples. One possible way to resolve this problem is to add other features such as syntactic dependencies between words to capture more language phenomena. In Figure 3-6b, the missing key phrase is not identified because our model is not able to ground the meaning of "pair of coordinates" to two integers. Possible future extensions to our model include using lexicon learning methods for mapping words to C++ primitive types for example "coordinates" to $\langle \mathbf{int}, \mathbf{int} \rangle$.

Chapter 4

Conclusions and Future Work

In this work we presented novel techniques to learning semantic relations in the context of two different domain applications. By acquiring semantic relations from text, our methods make it possible for machines to leverage such knowledge and then perform tasks that are intractable and generally require human involvement.

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

In the second code synthesis domain, we presented a sampling method for translating natural language specifications that describe input formats into the actual parser code that read them. Our results show that taking both the correlation between the text and the specification tree and the success of the generated C++ parser in reading input examples into account enables our method to correctly generate C++ parsers for 72.5% of our natural language specifications.

This work opens several possible directions for future research. In the high-level planning domain our method is limited to a single level of abstraction. Learning

an abstraction hierarchy can enable the induction of planning hierarchies from text. Moreover, such hierarchies are also important in light of text describing complex domains, where multiple levels of abstraction are essential for compactly describing the domain. In addition, a more general scenario in code synthesis problems is to generate code that can read input data and correctly produce expected output, i.e. programming by input-output examples. This research can potentially reduce much human effort in developing softwares.

Bibliography

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

[2] James K. Baker. Trainable grammars for speech recognition. In DH Klatt and JJ Wolf, editors, *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pages 547–550, 1979.

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

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

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

[6] S. R. K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2009.

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

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

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

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

[11] Mingwei Chang, Vivek Srikumar, Dan Goldwasser, and Dan Roth. Structured output learning with indirect supervision. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

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

[13] David L. Chen and Raymond J. Mooney. Learning to sportscast: A test of grounded language acquisition. In *Proceedings of 25th International Conference on Machine Learning (ICML-2008)*, 2008.

[14] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, 2010.

[15] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *Journal of Machine Learning Research*, 11, 2010.

[16] Dipanjan Das, Nathan Schneider, Desai Chen, and Noah A. Smith. Probabilistic frame-semantic parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 948–956, 2010.

[17] Dipanjan Das and Noah A. Smith. Semi-supervised frame-semantic parsing for unknown predicates. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 1435–1444, 2011.

[18] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC 2006*, 2006.

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

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

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

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

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

[24] Dan Goldwasser, Roi Reichart, James Clarke, and Dan Roth. Confidence driven unsupervised semantic parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, 2011.

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

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

[27] Mark Johnson and Thomas L. Griffiths. Bayesian inference for pcfgs via markov chain monte carlo. In *Proceedings of the North American Conference on Computational Linguistics (NAACL '07)*, 2007.

[28] Stephen C. Johnson. Yacc: Yet another compiler-compiler. *Unix Programmer's Manual*, vol 2b, 1979.

[29] Anders Jonsson and Andrew Barto. A causal approach to hierarchical decomposition of factored mdps. In *Advances in Neural Information Processing Systems, 13:10541060*, page 22. Press, 2005.

[30] Rohit J. Kate and Raymond J. Mooney. Learning language semantics from ambiguous supervision. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, AAAI'07, 2007.

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

[32] P. Liang, M. I. Jordan, and D. Klein. Learning semantic correspondences with less supervision. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, 2009.

[33] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2011.

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

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

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

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

[38] Tahira Naseem and Regina Barzilay. Using semantic cues to learn syntax. In *Proceedings of the 25th National Conference on Artificial Intelligence (AAAI)*, 2011.

[39] A. Newell, J.C. Shaw, and H.A. Simon. *The processes of creative thinking.* Paper P-1320. Rand Corporation, 1959.

[40] James Timothy Oates. *Grounding knowledge in sensors: Unsupervised learning for language and planning.* PhD thesis, University of Massachusetts Amherst, 2001.

[41] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 815–825, Piscataway, NJ, USA, 2012. IEEE Press.

[42] Hoifung Poon and Pedro Domingos. Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1*, EMNLP '09, 2009.

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

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

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

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

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

[48] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[49] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE11)*, May 2011.

[50] Lappoon R. Tang and Raymond J. Mooney. Automated construction of database interfaces: integrating statistical and relational learning for semantic parsing. In *Proceedings of the conference on Empirical Methods in Natural Language Processing*, EMNLP '00, 2000.

[51] Cynthia A. Thompson and Raymond J. Mooney. Acquiring word-meaning mappings for natural language interfaces. *JAIR*, 18:1–44, 2003.

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

[53] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

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

[55] Yuk Wah Wong and Raymond J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL*, 2007.

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

[57] Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of UAI*, pages 658–666, 2005.

[58] Luke S. Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2009.

[59] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 307–318, Washington, DC, USA, 2009. IEEE Computer Society.