From Natural Language Specifications to Program Input Parsers

Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{taolei, fanl, regina, rinard}@csail.mit.edu

Abstract

We present a method for automatically generating input parsers from English specifications of input file formats. 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]

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 (Wong and Mooney, 2007; Zettlemoyer and Collins, 2009; Poon and Domingos, 2009; Liang et al., 2011) and the successful mapping of natural language instructions into Windows command sequences (Branavan et al., 2009; Branavan et al., 2010).

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

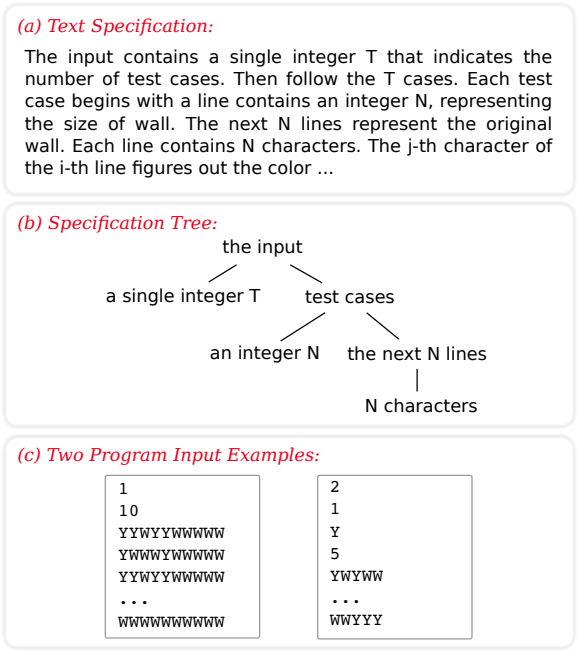

Figure 1: An example of (a) one natural language specification describing program input data; (b) the corresponding specification tree representing the program input structure; and (c) two input examples

[1]The code, data, and experimental setup for this research are available at http://groups.csail.mit.edu/rbg/code/nl2p

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 programmer 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 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 1a. Instead of directly generating code from the text specification, we first translate the specification into a *specification tree* (see Figure 1b), then map this tree into parser code (see Figure 2). We focus on the translation from the text specification to the specification tree.[2]

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 input examples. Specifically, the parser may interpret the input examples differently from the text specification. For example, the program input in Figure 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.

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

```
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,
13                   InputType* pParentLink) {
14   TestCaseType* pTestCase
15             = new TestCaseType;
16   pTestCase→pParentLink = pParentLink;
17
18   ...
19
20   return pTestCase;
21 }
22
23 InputType* ReadInput(FILE * pStream) {
24   InputType* pInput = new InputType;
25
26   pInput→T = ReadInteger(pStream);
27   for (int i = 0; i < pInput→T; ++i)  {
28     TestCaseType* pTestCase
29               = new TestCaseType;
30     pTestCase = ReadTestCase (pStream,
31               pInput);
32     pInput→lstTestCase.push_back (pTestCase);
33   }
34
35   return pInput;
36 }
```

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

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 (Clarke et al., 2010) (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

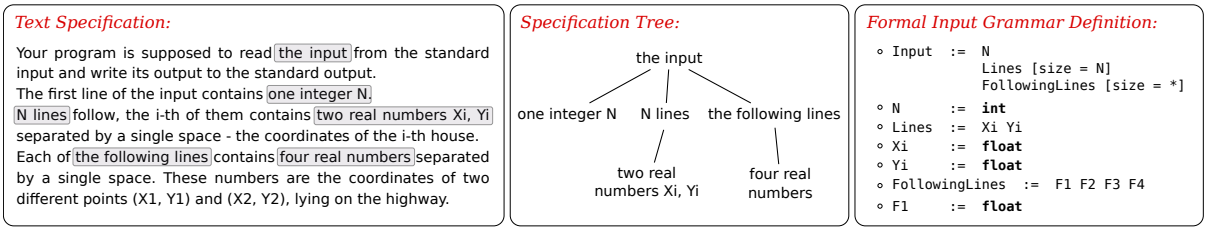

Figure 3: An example of generating input parser code from text: (a) a natural language input specification; (b) a specification tree representing the input format structure (we omit the background phrases in this tree in order to give a clear view of the input format structure); and (c) formal definition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

specification.

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 (Tang and Mooney, 2000; Zettlemoyer and Collins, 2005; Kate and Mooney, 2007; Wong and Mooney, 2007; Poon and Domingos, 2009; Liang et al., 2011; Goldwasser et al., 2011), semantic frames (Das et al., 2010; Das and Smith, 2011) and database records (Chen and Mooney, 2008; Liang et al., 2009).

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 (Branavan et al., 2009) or from executing predicted database queries (Chang et al., 2010; Clarke et al., 2010). While Branavan et al. (2009) 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 (Zhong et al., 2009; Pandita et al., 2012) and analyzing code comments to detect concurrency bugs (Tan et al., 2007; Tan et al., 2011). 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 Problem Formulation

The task of translating text specifications to input parsers consists of two steps, as shown in Figure 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. [3]

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.[4] 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 re-

[3] 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 (Johnson, 1979) can generate bottom-up parsers given such grammar.

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

ject 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 $\{w_k^i\}$. 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). 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_k^i to each noun phrase w_k^i 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 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").

4 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:[5]

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

  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

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

probability $P(\phi(w_p, w_s, w_k))$. We assume that each feature f_j is generated independently:

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

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 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:

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

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:

$$t^i \sim P(t^i|\mathbf{w}, \mathbf{t}^{-i}) \quad (1)$$

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:

$$Q(t^{i'}|\theta', w^i) \propto P(w^i|t^{i'}, \theta'). \quad (2)$$

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:

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

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 (Johnson and Griffiths, 2007; Cohn et al., 2010; Naseem and Barzilay, 2011). The algorithm pseudo code is shown in Algorithm 1.

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 (Baker, 1979), 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:

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

Here α are the Dirichlet hyper parameters and $\text{count}(f)$ 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.

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 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.

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

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

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

  Sample tree t^i for i-th document:

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

    Estimate model parameters:

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

    Sample a new specification tree from distribution Q:

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

    Generate and test code, and return feedback:

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

    Calculate accept probability r:

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

    Accept the new tree with probability r:

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

  end

end

Produce final structures:

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

Algorithm 1: The sampling framework for learning the model.

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 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").

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.

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 2: Statistics for 106 ICPC specifications.

5 Experimental Setup

Datasets: Our dataset consists of problem descriptions from ACM International Collegiate Programming Contests.[6] We collected 106 problems from ACM-ICPC training websites.[7] 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 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).

Evaluation Metrics: We evaluate the model

[6]Official Website: http://cm.baylor.edu/welcome.icpc

[7]PKU Online Judge: http://poj.org/; UVA Online Judge: http://uva.onlinejudge.org/

performance in terms of its success in generating a formal grammar that correctly represents the input format (see Figure 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 1 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 (Clarke et al., 2010).[8] 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 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 (Clarke et al., 2010) and choose a structural Support Vector Machine SVM$^{\text{struct}}$ [9] as the structure learner.

[8] 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: Average % Recall and % Precision of our model and all baselines over 20 independent runs.

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.

6 Experimental Results

Comparison with Baselines Table 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.

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

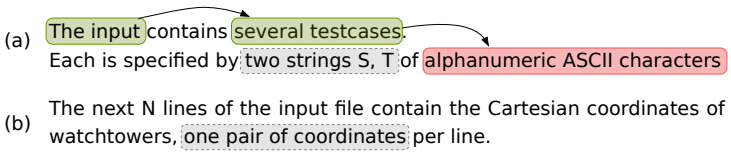

Figure 4: Examples of dependencies and key phrases predicted by our model. Green marks correct key phrases and dependencies and red marks incorrect ones. The missing key phrases are marked in gray.

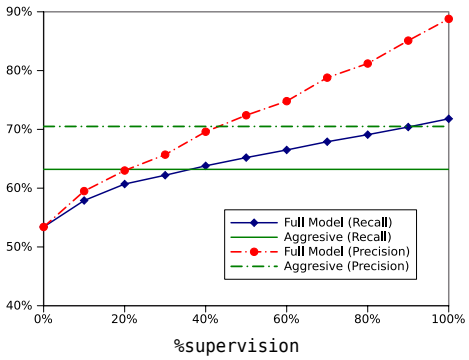

Figure 5: Precision and Recall of our model by varying the percentage of weak supervision. The green lines are the performance of *Aggressive* baseline trained with full weak supervision.

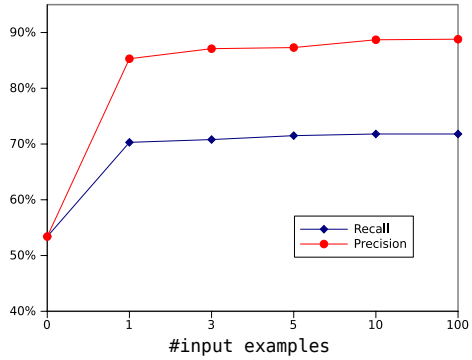

Figure 6: Precision and Recall of our model by varying the number of available input examples per text specification.

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 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 5 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 6 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.

Case Study Finally, we consider some text specifications that our model does not correctly trans-

late. In Figure 4a, 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 4b, 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 $\langle \mathbf{int}, \mathbf{int} \rangle$.

7 Conclusion

It is standard practice to write English language specifications for input formats. Programmers read the specifications, then develop source code that parses inputs in the format. Known disadvantages of this approach include development cost, parsers that contain errors, specification misunderstandings, and specifications that become out of date as the implementation evolves.

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.

8 Acknowledgements

The authors acknowledge the support of Battelle Memorial Institute (PO #300662) and the NSF (Grant IIS-0835652). Thanks to Mirella Lapata, members of the MIT NLP group and the ACL reviewers for their suggestions and comments. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors, and do not necessarily reflect the views of the funding organizations.

References

James K. Baker. 1979. 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.

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

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

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

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

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

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

Dipanjan Das and Noah A. Smith. 2011. 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.

Dipanjan Das, Nathan Schneider, Desai Chen, and Noah A. Smith. 2010. 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.

Dan Goldwasser, Roi Reichart, James Clarke, and Dan Roth. 2011. 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.

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

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

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

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

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

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

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. 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. IEEE Press.

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

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

Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. 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.

Lappoon R. Tang and Raymond J. Mooney. 2000. 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.

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

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

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

Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. 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. IEEE Computer Society.