Using Semantic Unification to Generate Regular Expressions from Natural Language

Using Semantic Unification to Generate Regular Expressions from Natural Language

Nate Kushman **Regina Barzilay**
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{nkushman, regina}@csail.mit.edu

Abstract

We consider the problem of translating natural language text queries into regular expressions which represent their meaning. The mismatch in the level of abstraction between the natural language representation and the regular expression representation make this a novel and challenging problem. However, a given regular expression can be written in many semantically equivalent forms, and we exploit this flexibility to facilitate translation by finding a form which more directly corresponds to the natural language. We evaluate our technique on a set of natural language queries and their associated regular expressions which we gathered from Amazon Mechanical Turk. Our model substantially outperforms a state-of-the-art semantic parsing baseline, yielding a 29% absolute improvement in accuracy.[1]

1 Introduction

Regular expressions (regexps) have proven themselves to be an extremely powerful and versatile formalism that has made its way into everything from spreadsheets to databases. However, despite their usefulness and wide availability, they are still considered a dark art that even many programmers do not fully understand (Friedl, 2006). Thus, the ability to automatically generate regular expressions from natural language would be useful in many contexts.

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

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

Figure 1: An example text description and its associated regular expression.[3]

We can think of this task as an instance of grounded semantic parsing, similar to the work done in the domain of database queries (Kate and Mooney, 2006; Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010). However, the current success in semantic parsing relies on two important properties of the data. First, while the past work did not assume the alignment was given, they did assume that finding a fine grained fragment-by-fragment alignment was possible. Secondly, the semantic domains considered in the past were strongly typed. This typing provides constraints which significantly reduce the space of possible parses, thereby greatly reducing the ambiguity.

However, in many interesting domains these two properties may not hold. In our domain, the alignment between the natural language and the regular expressions often happens at the level of the whole phrase, making fragment-by-fragment alignment impossible. For example, in Figure 1 no fragment of the regexp maps clearly to the phrase "three letter". Instead, the regexp explicitly represents the fact that there is only two characters after `X`, which is not stated explicitly by the text description and must be inferred. Furthermore, regular expressions have

[1] The dataset used in this work is available at http://groups.csail.mit.edu/rbg/code/regexp/

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

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

three letter	`[A-Za-z]{3}`
word	`\b[A-Za-z]+\b`
starting with 'X'	`X.*`

(b)

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

relatively few type constraints.

The key idea of our work is to utilize semantic unification in the logical domain to disambiguate the meaning of the natural language. Semantic unification utilizes an inference engine to determine the semantic equality of two syntactically divergent expressions. This is a departure from past work on semantic parsing which has largely focused on the *syntactic* interface between the natural language and the logical form, and on example-based semantic equality, neither of which utilize the inference power inherent in many symbolic domains.

To see how we can take advantage of semantic unification, consider the regular expression in Figure 2(a). This regular expression is semantically equivalent to the regular expression in Figure 1. Furthermore, it admits a fragment-by-fragment mapping as can be seen in Figure 2(b). In contrast, as we noted earlier, the regexp in Figure 1 does not admit such a mapping. In fact, learning can be quite difficult if our training data contains only the regexp in Figure 1. We can, nonetheless, use the regexp in Figure 2 as a stepping-stone for learning *if* we can use semantic inference to determine the equivalence between the two regular expressions. More generally, whenever the regexp in the training data does not factorize in a way that facilitates a direct mapping to the natural language description, we must find a regexp *which does factorize* and be able to compute its equivalence to the regexp we see in the training data. We compute this equivalence by converting each regexp to a minimal deterministic finite automaton (DFA) and leveraging the fact that minimal DFAs are guaranteed to be the same for semantically equivalent regexps (Hopcroft et al., 1979).

We handle the additional ambiguity stemming from the weak typing in our domain through the use of a more effective parsing algorithm. The state of the art semantic parsers (Kwiatkowski et al., 2011; Liang et al., 2011) utilize a pruned chart parsing algorithm which fails to represent many of the top parses and is prohibitively slow in the face of weak typing. In contrast, we use an n-best parser which always represents the most likely parses, and can be made very efficient through the use of the parsing algorithm from Jimenez and Marzal (2000).

Our approach works by inducing a combinatory categorial grammar (CCG) (Steedman, 2001). This grammar consists of a lexicon which pairs words or phrases with regular expression functions. The learning process initializes the lexicon by pairing each sentence in the training data with the full regular expression associated with it. These lexical entries are iteratively refined by considering all possible ways to split the regular expression and all possible ways to split the phrase. At each iteration we find the n-best parses with the current lexicon, and find the subset of these parses which are correct using DFA equivalence. We update the weights of a log-linear model based on these parses and the calculated DFA equivalence.

We evaluate our technique using a dataset of sentence/regular expression pairs which we generated using Amazon Mechanical Turk (Turk, 2013). We find that our model generates the correct regexp for 66% of sentences, while the state-of-the-art semantic parsing technique from Kwiatkowski et al. (2010) generates correct regexps for only 37% of sentences. The results confirm our hypothesis that leveraging the inference capabilities of the semantic domain can help disambiguate natural language meaning.

2 Related Work

Generating Regular Expressions Past work has looked at generating regular expressions from natural language using rule based techniques (Ranta, 1998), and also at automatically generating regular expressions from examples (Angluin, 1987). To the best of our knowledge, however, our work is the first to use training data to learn to automatically generate regular expressions from natural language.

Language Grounding There is a large body of research mapping natural language to some form of meaning representation (Kate and Mooney, 2006; Kate et al., 2005; Raymond and Mooney, 2006; Thompson and Mooney, 2003; Wong and Mooney,

2006; Wong and Mooney, 2007; Zelle and Mooney, 1996; Branavan et al., 2009; Mihalcea et al., 2006; Poon and Domingos, 2009). In some of the considered domains the issue of semantic equivalence does not arise because of the way the data is generated. The most directly related work in these domains, is that by Kwiatkowski et al. (2010 and 2011) which is an extension of earlier work on CCG-based semantic parsing by Zettlemoyer and Collins (2005). Similar to our work, Kwiatkowski et al. utilize unification to find possible ways to decompose the logical form. However, they perform only *syntactic unification*. Syntactic unification determines equality using only variable substitutions and does not take advantage of the inference capabilities available in many semantic domains. Thus, syntactic unification is unable to determine the equivalence of two logical expressions which use different lexical items, such as "$.\star$" and "$.\star.\star$". In contrast, our DFA based technique can determine the equivalence of such expressions. It does this by leveraging the equational inference capabilities of the regular expression domain, making it a form of *semantic unification*. Thus, the contribution of our work is to show that using semantic unification to find a deeper level of equivalence helps to disambiguate language meanings.

In many other domains of interest, determining semantic equivalence is important to the learning process. Previous work on such domains has focused on either heuristic or example-driven measures of semantic equivalence. For example, Artzi and Zettlemoyer (2011) estimate semantic equivalence using a heuristic loss function. Other past work has executed the logical form on an example world or in a situated context and then compared the outputs. This provides a very weak form of semantic equivalence valid only in that world/context (Clarke et al., 2010; Liang et al., 2009; Liang et al., 2011; Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013). In contrast, our work uses an exact, theoretically sound measure of semantic equivalence that determines whether two logical representations are equivalent in *any* context, i.e. on any input string.

3 Background

3.1 Finding Regexp Equivalence Using DFAs

Regular expressions can be equivalently represented as minimal DFAs, which are guaranteed to be equal for the same regular language (Hopcroft et al., 1979). The DFA representation of a regular expression may be exponentially larger than the the original regular expression. However, past work has shown that most regular expressions do not exhibit this exponential behavior (Tabakov and Vardi, 2005; Moreira and Reis, 2012), and the conversion process is renowned for its good performance in practice (Moreira and Reis, 2012). Hence, we compare the equivalence of two regular expressions by converting them to minimal DFAs and comparing the DFAs. We do this using a modified version of Møller (2010).[4]

function sig.	regexp	function signature	regexp
cons(R,R,...)	`ab`	rep*(R)	`a*`
and(R,R,...)	`[a-b]&[b-c]`	repminmax(I,I,R)	`a{3,5}`
or(R,R,...)	`a|b`	repmin(I,R)	`a{3,}`
not(R)	`~(a)`	repexact(I,R)	`a{3}`

Figure 3: This shows the signatures of all functions in our lambda calculus along with their regexp syntax.

3.2 Lambda Calculus Representation

To take advantage of the inherent structure of regular expressions, we deterministically convert them from a flat string representation into simply typed lambda calculus expressions. The full set of functions available in our lambda calculus can be seen in Figure 3. As can be seen from the figures, our lambda calculus is very weakly typed. It has only two primitive types, integer (I) and regexp (R), with most arguments being of type R.

3.3 Parsing

Our parsing model is based on a Combinatory Categorial Grammar. In CCG parsing most of the grammar complexity is contained in the lexicon, Λ, while the parser itself contains only a few simple rewrite rules called combinators.

Lexicon The lexicon, Λ, consists of a set of lexical entries that couple natural language with a lambda calculus expression. Our lexical entries contain words or phrases, each of which is associated with a function from the lambda calculus we described in §3.2. For example:

[4]We set a timeout on this process to catch any cases where the resulting DFA might be prohibitively large. We use a one second timeout in our experiments, which results in timeouts on less than 0.25% of the regular expressions.

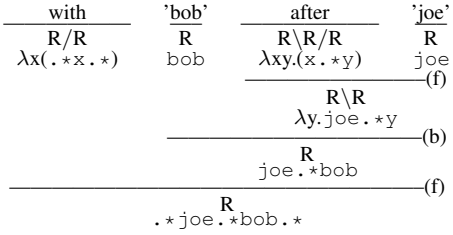


Figure 4: This shows an example parse.

$\langle$ after, R\R/R:λxy.(x.*y) $\rangle$
$\langle$ at least, R/I/R:$\lambda xy.$((x) {y,}) $\rangle$

Note that the lambda expressions contain type information indicating the number of arguments and the type of those arguments as described in §3.2. However, this information is augmented with a (/) or a (\) for each argument indicating whether that argument comes from the left or the right, in sentence order. Thus R\R/R can be read as a function which first takes an argument of type R on the right then takes another argument of type R on the left, and returns an expression of type R.

Combinators Parses are built by combining lexical entries through the use of a set of combinators. Our parser uses only the two most basic combinators, forward function application and backward function application.[5] These combinators work as follows:

R/R:f R:g → R:f(g) (forward)
R:f R\R:g → R:g(f) (backward)

The forward combinator applies a function to an argument on its right when the type of the argument matches the type of the function's first argument. The backward combinator works analogously. Figure 4 shows an example parse.

4 Parsing Model

For a given lexicon, Λ, and sentence, $\vec{w}$, there will in general be many valid parse trees, $t \in T(\vec{w}; \Lambda)$. We assign probabilities to these parses using a standard log-linear parsing model with parameters θ:

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

Our training data, however, includes only the correct regular expression, r, and not the correct parse, t. The training objective used by the past work in such circumstances, is to maximize the probability of the correct regular expression by marginalizing over all parses which generate that exact regular expression. Such an objective is limited, however, because it does not allow parses that generate semantically correct regexps which are not syntactically equivalent to r, such as those in Figure 2. The main departure of our work is to use an objective which allows such parses through the use of the DFA-EQUAL procedure. DFA-EQUAL uses the process described in §3.1 to determine whether parse t evaluates to a regexp which is semantically equivalent to r, leading to the following objective:

$$O = \sum_i \log \sum_{t|\texttt{DFA-EQUAL}(t, r_i)} p(t|\vec{w}_i; \theta, \Lambda) \quad (1)$$

At testing time, for efficiency reasons, we calculate only the top parse. Specifically, if $r = eval(t)$ is the regexp which results from evaluating parse t, then we generate $t^* = \arg\max_{t \in T(\vec{w})} p(t|\vec{w}; \theta, \Lambda)$, and return $r^* = eval(t^*)$.

5 Learning

Our learning algorithm starts by generating a single lexical entry for each training sample which pairs the full sentence, $\vec{w}_i$, with the associated regular expression, r_i. Formally, we initialize the lexicon as $\Lambda = \{\langle \vec{w}_i, R : r_i \rangle | i = 1 \ldots n\}$. We then run an iterative process where in each iteration we update both Λ and θ for each training sample. Our initial Λ will perfectly parse the training data. However it won't generalize at all to the test data since the lexical entries contain only full sentences. Hence, in each iteration we refine the lexicon by splitting existing lexical entries to generate more granular lexical entries which will generalize better. The candidates for splitting are all lexical entries used by parses which generate the correct regular expression, r_i, for the current training sample. We consider all possible ways to factorize each lexical entry, and we add to Λ a new lexical entry for each possible factorization, as discussed in §5.2. Finally, we update θ by performing a single stochastic gradient ascent update step for each training sample, as discussed in §5.1. See Algorithm 1 for details.

This learning approach follows the structure of the previous work on CCG based semantic parsers (Zettlemoyer and Collins, 2005;

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

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

Functions:

- $\texttt{N-BEST}(\vec{w}; \theta, \Lambda)$ n-best parse trees for $\vec{w}$ using the algorithm from §5.1
- $\texttt{DFA-EQUAL}(t, r)$ calculates the equality of the regexp from parse t and regexp r using the algorithm from §3.1
- $\texttt{SPLIT-LEX}(T)$ splits all lexical entries used by any parse tree in set T, using the process described in §5.2

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

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

Update Lexicon: Λ

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

Update Parameters: θ

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

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

Algorithm 1: The full learning algorithm.

Kwiatkowski et al., 2010). However, our domain has distinct properties that led to three important departures from this past work.

First, we use the DFA based semantic unification process described in §3.1 to determine the set of correct parses when performing parameter updates. This is in contrast to the *syntactic* unification technique, used by Kwiatkowski et al. (2010), and the example based unification used by other semantic parsers, e.g. Artzi and Zettlemoyer (2011). Using semantic unification allows us to handle training data which does not admit a fragment-by-fragment mapping between the natural language and the regular expression, such as the example in Figure 2.

Second, our parser is based on the efficient n-best parsing algorithm of Jimenez and Marzal (2000) instead of the pruned chart parsing algorithm used by the past work (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010). As we show in §8.2, this results in a parser which more effectively represents the most likely parses. This allows our parser to better handle the large number of potential parses that exist in our domain due to the weak typing.

Third, we consider splitting lexical entries used in *any* correct parse, while the past work (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010) considers splitting only those used in the *best* parse. We must utilize a less constrictive splitting policy since our domain does not admit the feature weight initialization technique used in the domains of the past work. We discuss this in §5.2.1. In the remainder of this section we discuss the process for learning θ and for generating the lexicon, Λ.

5.1 Estimating Theta

To estimate θ we will use stochastic gradient ascent, updating the parameters based on one training example at a time. Hence, we can differentiate the objective from equation 1 to get the gradient of parameter θ_j for training example i, as follows:

$$\frac{\partial O_i}{\partial \theta_j} = E_{p(t|\texttt{DFA-EQUAL}(t, r_i), \cdot)}\left[\phi_j(t, \vec{w}_i)\right] - E_{p(t|\cdot)}\left[\phi_j(t, \vec{w}_i)\right] \quad (2)$$

This gives us the standard log-linear gradient, which requires calculating expected feature counts. We define the features in our model over individual parse productions, admitting the use of dynamic programming to efficiently calculate the unconditioned expected counts. However, when we condition on generating the correct regular expression, as in the first term in (2), the calculation no longer factorizes, rendering exact algorithms computationally infeasible.

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

We use the n-best parses to calculate an approximate version of the gradient. Specifically, T_i is the

set of n-best parses for training sample i, and C_i includes all parses t in T_i such that $\texttt{DFA-EQUAL}(t, r_i)$. We calculate the approximate gradient as:

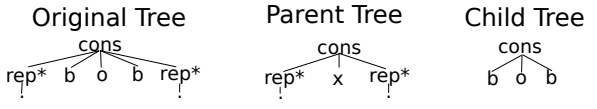
$$\Delta = E_{p(t|t\in C_i;\theta,\Lambda)}[\phi(t,\vec{w}_i)] - E_{p(t|t\in T_i;\theta,\Lambda)}[\phi(t,\vec{w}_i)] \quad (3)$$

In contrast to our n-best technique, the past work has calculated equation (2) using a beam search approximation of the full inside-outside algorithm (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010; Liang et al., 2011). Specifically, since the conditional probability of t given r does not factorize, a standard chart parser would need to maintain the full logical form (i.e. regular expression) for each subparse, and there may be an exponential number of such subparses at each chart cell. Thus, they approximate this full computation using beam search, maintaining only the m-best logical forms at each chart cell.

Qualitatively, our n-best approximation always represents the most likely parses in the approximation, but the number of represented parses scales only linearly with n. In contrast, the number of parses represented by the beam search algorithm of the past work can potentially scale exponentially with the beam size, m, due to its use of dynamic programming. However, since the beam search prunes myopically at each chart cell, it often prunes out the highest probability parses. In fact, we find that the single most likely parse is pruned out almost 20% of the time. Furthermore, our results in §8 show that the beam search's inability to represent the likely parses significantly impacts the overall performance. It is also important to note that the runtime of the n-best algorithm scales much better. Specifically, as n increases, the n-best runtime increases as $O(n|\vec{w}|\log(|\vec{w}||P| + n)$, where P is the set of possible parse productions. In contrast, as m is increased, the beam search runtime scales as $O(|\vec{w}|^5 m^2)$, where the $|\vec{w}|^5$ factor comes from our use of headwords, as discussed in §6. In practice, we find that even with n set to $10,000$ and m set to 200, our algorithm still runs almost 20 times faster.

5.2 Lexical Entry Splitting

Each lexical entry consists of a sequence of n words aligned to a typed regular expression function, $\langle w_{0:l}, T : r\rangle$. Our splitting algorithm considers all possible ways to split a lexical entry into two new lexical entries such that they can be recombined via function application to obtain the original lexical entry. This process is analogous to the syntactic unification process done by Kwiatkowski et al. (2010).



(a) (b) (c)

Figure 5: The tree in (a) represents the lambda expression from the lexical entry $\langle$with bob, R:$.*\texttt{bob}.*\rangle$. One possible split of this lexical entry generates the parent lexical entry $\langle$with, R/R:$\lambda x.(.*\texttt{x}.*)\rangle$ and the child lexical entry, $\langle$bob, R:$\texttt{bob}\rangle$, whose lambda expressions are represented by (b) and (c), respectively.

We first consider all possible ways to split the lambda expression r. The splitting process is most easily explained using a tree representation for r, as shown in Figure 5(a). This tree format is simply a convenient visual representation of a lambda calculus function, with each node representing one of the function type constants from Figure 3. Each split, $s \in S(r)$, generates a child expression s_c and a parent expression s_p such that $r = s_p(s_c)$. For each node, n, in r besides the root node, we generate a split where s_c is the subtree rooted at node n. For such splits, s_p is the lambda expression r with the sub-expression s_c replaced with a bound variable, say x. In addition to these simple splits, we also consider a set of more complicated splits at each node whose associated function type constant can take any number of arguments, i.e. `or`, `and`, or `cons`. If $C(n)$ are the children of node n, then we generate a split for each possible subset, $\{V|V \subset C(n)\}$. Note that for `cons` nodes V must be contiguous. In §6 we discuss additional restrictions placed on the splitting process to avoid generating an exponential number of splits. For the split with subset V, the child tree, s_c, is a version of the tree rooted at node n pruned to contain only the children in V. Additionally, the parent tree, s_p, is generated from r by replacing all the children in V with a single bound variable, say x. Figure 5 shows an example of such a split. We only consider splits in which s_c does not have any bound variables, so its type, T_c, is always either R or I. The type of s_p is then type of the original expression, T augmented by an additional argument of the child type, i.e. either T/T_c or $T\backslash T_c$.

Each split s generates two pairs of lexical entries,

one for forward application, and one for backward application. The set of such pairs of pairs is:

$$\{(\langle w_{0:j}, T/T_c : s_p\rangle, \langle w_{j:l}, T_c : s_c\rangle),$$
$$(\langle w_{0:j}, T_c : s_c\rangle, \langle w_{j:l}, T\backslash T_c : s_p\rangle)|$$
$$(0 \leq j \leq l) \wedge (s \in S(r))\}$$

5.2.1 Adding New Lexical Entries

Our model splits all lexical entries used in parses which generate correct regular expressions, i.e. those in C_i, and adds *all* of the generated lexical entries to Λ. In contrast, the previous work (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010) has a very conservative process for adding new lexical entries. This process relies on a good initialization of the feature weights associated with a new lexical entry. They perform this initialization using a Giza++ alignment of the words in the training sentences with the names of functions in the associated lambda calculus expression. Such an initialization is ineffective in our domain since it has very few primitive functions and most of the training examples use more than half of these functions. Instead, we add new lexical entries more aggressively, and rely on the n-best parser to effectively ignore any lexicon entries which do not generate high probability parses.

6 Applying the Model

Features To allow inclusion of head words in our features, our chart cells are indexed by start word, end word, and head word. Thus for each parse production we have a set of features that combine the head word and CCG type, of the two children and the newly generated parent. Additionally, for each lexical entry $\langle \vec{w}_i, R : r_i\rangle \in \Lambda$, we have four types of features: (1) a feature for $\langle \vec{w}_i, R : r_i\rangle$, (2) a feature for $\vec{w}_i$, (3) a feature for $R : r_i$, and (4) a set of features indicating whether $\vec{w}_i$ contains a string literal and whether the leaves of r_i contain any exact character matches (rather than character range matches).

Initialization In addition to the sentence level initialization discussed in §5 we also initialize the lexicon, Λ, with two other sets of lexical entries. The first set is all of the quoted string literals in the natural language phrases from the training set. Thus for the phrase, "lines with 'bob' twice" we would add the lexical entry $\langle$ 'bob', R:`bob` $\rangle$. We also add lexical entries for both numeric and word representations of numbers, such as $\langle$ 1, R:`1`$\rangle$ and $\langle$ one, R:`1`$\rangle$. We add these last two types of lexical entries because learning them from the data is almost impossible due to data sparsity. Lastly, for every individual word in our training set vocabulary, we add an identity lexical entry whose lambda expression is just a function which takes one argument and returns that argument. This allows our parser to learn to skip semantically unimportant words in the natural language description, and ensures that it generates at least one parse for every example in the dataset. At test time we also add both identity lexical entries for every word in the test set vocabulary as well as lexical entries for every quoted string literal seen in the test queries. Note that the addition of these lexical entries requires only access to the test queries and does not make use of the regular expressions (i.e. labels) in the test data in any way.

Parameters We initialize the weight of all lexical entry features except the identity features to a default value of 1 and initialize all other features to a default weight of 0. We regularize our log-linear model using the L^2-norm and a λ value of 0.001. We use a learning rate of $\alpha = 1.0$, set $n = 10,000$ in our n-best parser, and run each experiment with 5 random restarts and $K = 50$ iterations. We report results using the pocket algorithm technique originated by Gallant (1990).

Constraints on Lexical Entry Splitting To prevent the generation of an exponential number of splits, we constrain the lexical entry splitting process as follows:

- We only consider splits at nodes which are at most a depth of 2 from the root of the original tree.
- We limit lambda expressions to 2 arguments.
- In unordered node splits (`and` and `or`) the resulting child can contain at most 4 of the arguments.

These restrictions ensure the number of splits is at most an M-degree polynomial of the regexp size. The unification process used by Kwiatowski et al. (2010) bounded the number of splits similarly.

7 Experimental Setup

Dataset Our dataset consists of 824 natural language and regular expression pairs gathered using Amazon Mechanical Turk (Turk, 2013) and oDesk (oDesk, 2013).[6] On Mechanical Turk we asked workers to

[6] This is similar to the size of the datasets used by past work.

generate their own original natural language queries to capture a subset of the lines in a file (similar to UNIX `grep`). In order to compare to example based techniques we also ask the Mechanical Turk workers to generate 5 positive and 5 negative examples for each query. On oDesk we hired a set of programmers to generate regular expressions for each of these natural language queries. We split our data into 3 sets of 275 queries each and tested using 3-fold cross validation. We tuned our parameters separately on each development set but ended up with the same values in each case.

Evaluation Metrics We evaluate by comparing the generated regular expression for each sentence with the correct regular expression using our DFA equivalence technique. As discussed in §3.1 this metric is exact, indicating whether the generated regular expression is semantically equivalent to the correct regular expression. Additionally, as discussed in §6, our identity lexical entries ensure we generate a valid parse for every sentence, so we report only accuracy instead of precision and recall.

Baselines We compared against six different baselines. The *UBL* baseline uses the published code from Kwiatkowski et al. (2010) after configuring it to handle the lambda calculus format of our regular expressions.[7] The other baselines are ablated and/or modified versions of our model. The *BeamParse* baselines replace the `N-BEST` procedure from Algorithm 1 with the beam search algorithm used for parsing by past CCG parsers (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2010).[8] The *StringUnify* baseline replaces the `DFA-EQUAL` procedure from Algorithm 1 with exact regular expression string equality. The *HeuristicUnify* baselines strengthen this by replacing `DFA-EQUAL` with a smart heuristic form of semantic unification. Our heuristic unification procedure first flattens the regexp trees by merging all children into the parent node if they are both of the same type and of type `or`, `and`, or `cons`. It then sorts all children of the `and` and `or` operators. Finally, it converts both regexps back to a flat string and compares these strings for equivalence. This process should more effective than any form of syntactic unification and any simpler heuristics. The *ExampleUnify* baseline represents the performance of the example based semantic unification techniques. It replaces `DFA-EQUAL` with a procedure that evaluates the regexp on all the positive and negative examples associated with the given query and returns true if all 10 are correctly classified. Finally, *BeamParse-HeuristicUnify-TopParse* uses the same algorithm as that for *BeamParse-HeuristicUnify* except that it only generates lexical entries from the top parse instead of all parses. This more closely resembles the conservative lexical entry splitting algorithm used by Kwiatkowski et al.

[7] This was done in consultation with the original authors.

[8] we set the beam size to 200, which is equivalent to the past work. With this setting, the slow runtime of this algorithm allowed us to run only two random restarts.

Model	Percent Correct
UBL	36.5%
BeamParse-HeuristicUnify	9.4%
BeamParse-HeuristicUnify-TopParse	22.1%
NBestParse-StringUnify	31.1%
NBestParse-ExampleUnify	52.3%
NBestParse-HeuristicUnify	56.8%
Our Full Model	**65.5%**

Table 1: Accuracy of our model and the baselines.

8 Results

Our model outperforms all of the baselines, as shown in Table 1. The first three baselines – *UBL*, *BeamParse-HeuristicUnify*, and *BeamParse-HeuristicUnify-TopParse*– represent the algorithm used by Kwiatkowski et al. Our model outperforms the best of these by over 30% in absolute terms and 180% in relative terms.

The improvement in performance of our model over the *NBestParse-StringUnify*, *NBestParse-ExampleUnify* and *NBestParse-HeuristicUnify* baselines highlights the importance of our DFA based semantic unification technique. Specifically, our model outperforms exact string based unification by over 30%, example based semantic unification by over 13% and our smart heuristic unification procedure by 9%. These improvements confirm that leveraging exact semantic unification during the learning process helps to disambiguate language meanings.

8.1 Effect of Additional Training Data

Table 2 shows the change in performance as we increase the amount of training data. We see that our model provides particularly large gains when there

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

Table 2: Results for varying amounts of training data.

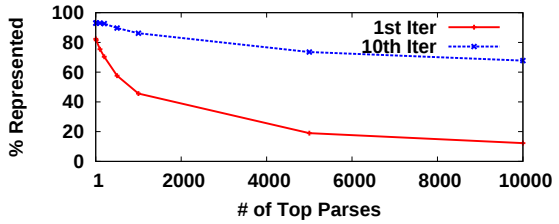


Figure 6: This graph compares the set of parses represented by the n-best algorithm used in our model to the set of parses represented by the beam search algorithm used by the past work. Note that our n-best algorithm represents 100% of the top 10000 parses.

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

8.2 Beam Search vs. N-Best

A critical step in the training process is calculating the expected feature counts over all parses that generate the correct regular expression. In §4 we discussed the trade-off between approximating this calculation using the n-best parses, as our model does, verses the beam search model used by the past work. The effect of this trade-off can be seen clearly in Figure 6. The n-best parser always represents the n-best parses, which is set to 10,000 in our experiments. In contrast, on the first iteration, the beam search algorithm fails to represent the top parse almost 20% of the time and represents less than 15% of the 10,000 most likely parses. Even after 10 iterations it still only represents 70% of the top parses and fails to represent the top parse almost 10% of the time. This difference in representation ability is what provides the more than 30% difference in accuracy between the *BeamParse-HeuristicUnify* version of our model and the *NBestParse-HeuristicUnify* version of our model.

9 Conclusions and Future Work

In this paper, we present a technique for learning a probabilistic CCG which can parse a natural language text search into the regular expression that performs that search. The key idea behind our approach is to use a DFA based form of semantic unification to disambiguate the meaning of the natural language descriptions. Experiments on a dataset of natural language regular expression pairs show that our model significantly outperforms baselines based on a state-of-the-art model.

We performed our work on the domain of regular expressions, for which semantic unification is tractable. In more general domains, semantic unification is undecidable. Nevertheless, we believe our work motivates the use of semantic inference techniques for language grounding in more general domains, potentially through the use of some form of approximation or by restricting those domains in some way. For example, SAT and SMT solvers have seen significant success in performing semantic inference for program induction and hardware verification despite the computational intractability of these problems in the general case.

10 Acknowledgments

The authors acknowledge the support of Battelle Memorial Institute (PO#300662) and NSF (grant IIS-0835652). We thank Luke Zettlemoyer, Tom Kwiatkowski, Yoav Artzi, Mirella Lapata, 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

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

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

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

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

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

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

Jeffrey Friedl. 2006. *Mastering Regular Expressions*. OReilly.

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

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

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

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

R.J. Kate and R.J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, volume 44, page 913.

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

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

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

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

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

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

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

N. Moreira and R. Reis. 2012. Implementation and application of automata.

oDesk. 2013. `http://odesk.com/`.

H. Poon and P. Domingos. 2009. Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 1–10. Association for Computational Linguistics.

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

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

M. Steedman. 2001. *The syntactic process*. MIT press.

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

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

Mechanical Turk. 2013. `http://mturk.com/`.

Y.W. Wong and R.J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In

Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, pages 439–446. Association for Computational Linguistics.

Y.W. Wong and R. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, volume 45, page 960.

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

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

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

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