Application of Neural Networks in the Semantic Parsing Re-Ranking Problem

Reginald Long
reglong@stanford.edu

Colin Wei
colinwei@stanford.edu

Abstract

Automatic program generation allows end-users to benefit from greatly from increased productivity. However, general Natural Language Programming tools fail to provide the benefits of the ambiguity and expressivity of English. We reduce program generation into a semantic parsing problem. Given a command and input, we procedurally generate a large set of candidate programs, and then align the command to a program using deep learning. We then use neural network models, specifically recurrent neural networks, to predict the correct program. By restricting the domain to string operations, we show how we can perform reasonably well..

1 Introduction

The ability to write programs greatly increases the productivity of the end user; however, for the average person, learning how to program is out of reach. Automatic program generation would solve that problem by obviating the need for users to learn how to program.

In this paper, we explore program generation. Given an utterance in English, we wish to build a simple Java program, compile, execute, and return the result to the user. We provide evidence that, in simple domains, we can reasonably tackle this problem. At a high level, we approach program generation as a semantic parsing problem. We interpret each "parse" as a program, and attempt to align user commands to corresponding programs.

2 Background/Related Work

Whereas the structure of a program is hierarchical and exact, English is ambiguous. This presents immediate difficulties, and potentially makes it impossible to build large systems using natural language. As Edsger Dijkstra discusses in "On the Foolishness of 'Natural Language Programming'"[11], the formal symbolism of programming makes it easier to reason about programs, and that any interface from English to machine code would necessarily induce many complexities. He concludes by saying, "From one gut feeling I derive much consolation: I suspect that machines to be programmed in our native tongues...are as damned difficult to make as they would be to use."

Early iterations of Natural Language Programming certainly confirmed that suspicion. Below is an example of Bubble Sort in Pegasus, one of the first Natural Language Programming languages[9].

Bubble Sort

```
Take the array [3, 5, 7, 4, 6, 2, 1].
Print "Before: " and print the array.
Count from one up to the size of the array:
Go over the array from the beginning to the end minus the counter:
    If the current element is bigger than the following element then exchange
    the current element with the following element.
Print "After: " and print the array.
```

Although Pegasus certainly uses natural language, it is no less formal than C code, and is probably longer. We don't gain any productivity from writing the program in English. Over the last few years, people have tried to incorporate NLP techniques to allow more ambiguity in the commands, and have restricted the domain to some success. In 2012, Cozzie et. al[10] built Macho, a system that maps user commands to simple unix commands like cp, grep, sort, etc., but requires the user to provide a unit test to constrain the problem.

Most of the recent work on domain restricted Natural Language Programming is in Regina Barzilay's Lab at MIT. Branavan et. al[12][13] built a system that would read Windows troubleshooting articles and execute the instructions using a search procedure. More recently, in 2014, Kushman et. al[14] built a system that would solve algebra word problems by aligning the world problem to a template, and then solving the linear system of equations given by the template.

3 Problem Statement

Let I denote the input, and let u denote a user command. For a given u, we generate a set of candidate programs χ. Each program $p \in \chi$ performs one or more operations on I. We score each $p \in \chi$, and choose the highest scoring program.

3.1 Domain

Our universe of programs consists of simple string operations on an ArrayList of strings. Each string in the ArrayList is referred to as a "group" (For example, the first group in (string abc) (string dbd) (string ldk) would refer to (string abc)). There are 3 groups in each example. The allowed operations are as follows:

1. Prepending characters to a string (E.g. "add 'afg' to the front of the first group".)
2. Appending characters to a string (E.g. "add 'klj' to the end of the middle group")
3. Removing characters from the front of a string (E.g. "remove 'b' from 'bcfg')
4. Removing characters from the end of a string (E.g. "remove 'g' from 'bcfg')
5. Reversing a string (E.g. "reverse the second group).
6. Swapping the location of two groups (E.g. "switch groups 1 and groups 2")
7. Doubling a string (E.g. "double group 3")
8. No Action (E.g. "don't do anything in this step")

3.2 Dataset

To obtain our dataset, we first randomly generate length 3 ArrayLists of strings. We treat these generated ArrayLists as a sequence, and use the allowed programs above to iteratively obtain the next ArrayList in the sequence from the previous. We use Amazon Mechanical Turk to annotate examples in the dataset with English descriptions of which program operations used to modify the groups.

Table 1: Sample Task and Response

Step	Input	User Command
1	AA BB CC	
2	AAAA BB CC	Double the first group
3	AAAA BB CCCC	Double the last group
4	AAAA BB SUCCCCC	Add "SUC" to the start of the last group
5	AAAA FOBB SUCCCCC	Add "FO" to the start of the middle group
6	SUCCCCC FOBB AAAA	Swap the first and last groups
7	AAAA FOBB SUCCCCC	Repeat the previous step
8	SUCCCCC FOBB AAAA	Repeat the previous step
9	SUCCCCC FOBB AAA	Delete the final letter of the last group
10	SUCCCCC FOBB FOAAA	Add "FO" at the start of the last group

From the responses, we generate an example by taking the Input of the previous step, and the User Command of the current step. Our examples are of the form: "AA BB CC ||| Double the first group." where everything before the ||| is the input, and everything after is the command. The target value in this case would be "AAAA BB CC". We can construct an arbitrarily long number of commands for each task. In this paper, we construct all length 1 (1 command) examples, and all length 2 (2 command) examples. In the one command case, we obtain 6127 training examples, 681 dev examples, and 700 test examples. In the two command case, we obtain 2300 training examples, 700 dev, and 700 test examples.

3.3 Evaluation

We compare our results relative to oracle accuracy. An oracle is said to be "correct" if a program that generates the desired output is contained in χ. We say our model correctly predicts the program if it chooses the same program that the oracle chooses. If x represents the number of correct oracle programs, and y represents the number of correct model programs, then our accuracy is x/y.

4 Approach

We construct a grammar that is capable of performing all of the operations described in the Problem Statement. However, due to the varied and expressive nature of natural language, it would be impossible to build a robust system relying solely on pre-defined rules/templates. Therefore, we instead overgenerate parses. Our approach follows the general guidelines of [4]: we "anchor" on certain words in the utterance, and then use those "anchors" to recursively enumerate every possible parse using those words. We then prune our parses using a scoring function. Whereas traditional scoring functions in this method would use a linear combination of features derived from the parses, our scoring function in this case relies on vector space embeddings of the words. We experiment with the following embedding models: a simple linear embedding, a siamese neural network embedding, and a recurrent neural network. Since we have the parse trees of our training examples with respect to our grammar, we can apply our model on the parse trees to rank the order of parses. Let $s(x, y)$ be the score of the parse with respect to the training utterance y. Let x^* be the correct parse, and m be our margin. Let $W(y)$ be the set of incorrect parses with respect to utterance y. Then we optimize the following max-margin objective:

$$\sum_{i, x_i \sim W(y_i)} \max\left(0, m - (s(x_i^*, y_i) - s(x_i, y_i))\right) \tag{1}$$

where x_i is sampled uniformly and randomly from $W(y_i)$.

4.1 Linear Embedding

Inspired by the method in Bordes et. al [15], we use a simple linear embedding as a baseline. As in [15], we set a parameter $U \in \mathbb{R}^{d \times |V|}$ to define our word vectors. Then given inputs $v_y, v_x \in \mathbb{R}^{|V|}$ as the one-hot sentence vectors indictating the presence of the words in our parse/utterance, we simply set

$$s(x, y) = (Uv_x)^T (Uv_y) \tag{2}$$

4.2 Siamese Network

We introduce nonlinearities on top of the basic idea of the linear embedding model - allow word vectors for similar-denotation parse and utterance words to align, and separate word vectors for different-denotation parse and utterance words. Our model is defined as follows:

$$\begin{aligned} w_x &= Uv_x & w_y &= Uv_y \\ h_x &= \tanh(Hw_x + b) & h_y &= \tanh(Hw_y + b) \\ & s(x, y) = -||h_x - h_y||_2 \end{aligned} \tag{3}$$

4.3 Recurrent Neural Network

Motivated by the Siamese network, we adopt a recurrent neural network in a Siamese structure. The main advantage of the RNN is that it allows us to better combine our word vectors rather than summing over the one-hot sentence vector.

$$\begin{aligned} w_x^{(t)} &= Uv_x^{(t)} & w_y^{(t)} &= Uv_y^{(t)} \\ h_x^{(t)} &= \tanh(Hh_x^{(t-1)} + w_x^{(t)}) & h_y^{(t)} &= \tanh(Hh_y^{(t-1)} + w_y^{(t)}) \\ & s(x, y) = -||h_x^{(l_x)} - h_y^{(l_y)}||_2 \end{aligned} \tag{4}$$

where $v_x^{(t)}, v_y^{(t)}$ are one-hot vectors for the parse and utterance words at time t, respectively, and l_x, l_y are the number of words in the parse/utterance, respectively.

4.4 Baseline

To compare deep learning methods against conventional feature scoring, we perform a baseline evaluation using the general guidelines in [4]. We first construct a grammar that is expressive enough to capture most of the correct parses in our training examples, achieving an oracle accuracy of 91.1% in the 1 command case, and 76% in the 2 command case.

We then train feature weights in order to correctly score our parses. For any possible parse of our input, we use unigram features, bigram features, function argument features, and a feature that indicates whether the parse compiles as a program. More specifically, our unigram features are defined as follows: for every word w in the utterance, and every function call f for the parse of that utterance, we add an indicator feature corresponding to the tuple (w, f). Likewise, we define the bigram features for every pair of words in the utterance. For example, for the utterance "add abc to group 1", with a parse corresponding to the function call for "append", the tuple $(\text{"add"}, \text{append})$ would be a unigram feature with value 1. We compute the parse score as a dot product of feature weights and our feature vector and then use a log-linear model over candidate parse scores to assign a probability to each possible parse.

5 Experiments

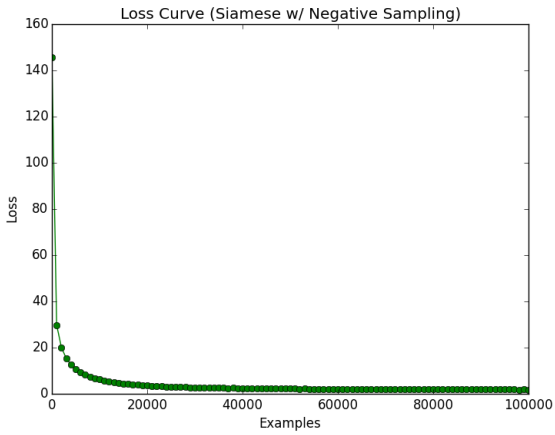
5.1 Results

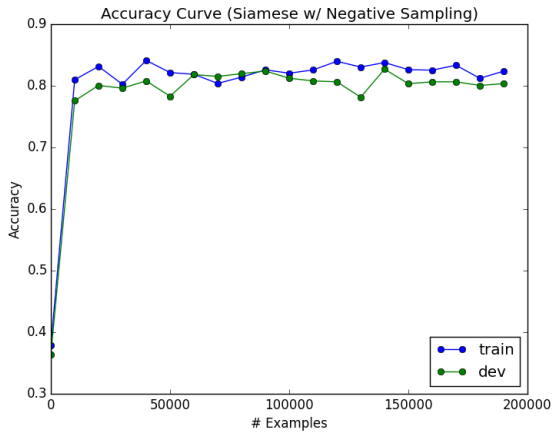
Table 2: Results for 1 Command Dataset

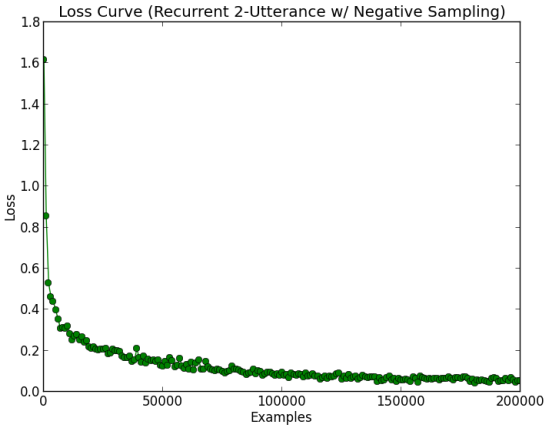
Type	Train	Dev	Test
Baseline	0.6371	0.6272	0.6221
Word Embeddings	0.6279	0.5425	0.5257
Siamese Network 2	0.8278	0.8105	0.8029
RNN	0.9711	0.9721	0.9685

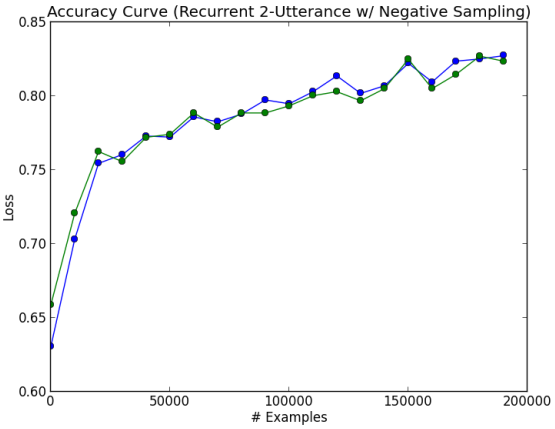
Table 3: Results for 2 Command Dataset

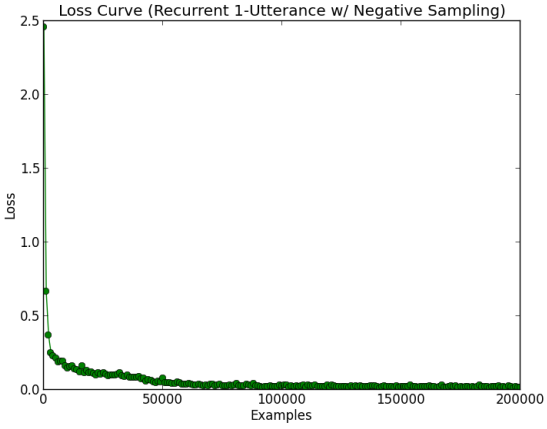
Type	Train	Dev	Test
Baseline	0.5631	0.5539	0.5578
Word Embeddings	0.2857	0.2409	0.2314
Siamese Network	0.5066	0.4152	0.4114
RNN	0.8273	0.8236	0.8171

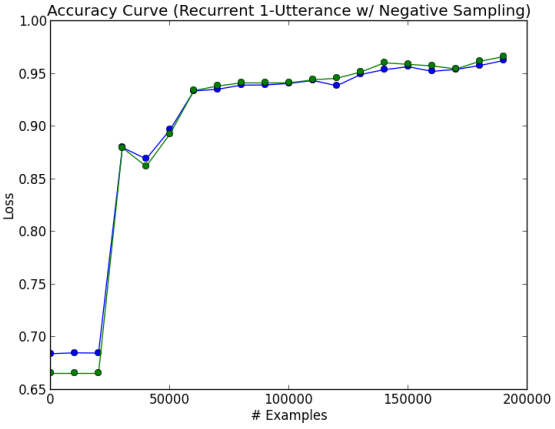












5.2 Sample Generated Programs (Correctly Generated)

Our programs are automatically generated using a variant of Java with Lisp-like syntax, which makes it easier to sequentially apply functions and to procedurally generate/execute our set of programs.

Example 1 (One Command):
Input:ww pp eeee
Utterance: remove the last letter in the third group
Program: (derivation (formula (call edu.stanford.nlp.sempre.SFun.remove (call edu.stanford.nlp.sempre.SFun.parseStartState (string ”gggg nnnn xxiad”)) (number 3) (boolean

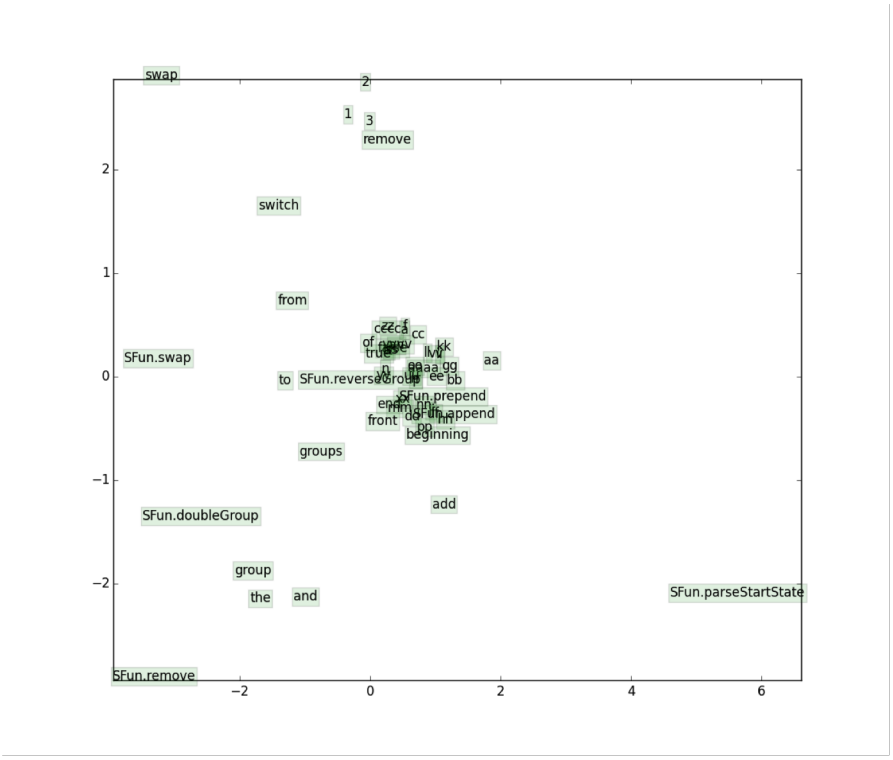


Figure 1: Subset of Word Vectors trained on RNN

false)))
Output: gggg nnnn xxia

Example 2 (One Command):
Input:ww pp eeee
Utterance: at the end of group 3 add a
Program: (derivation (formula (call edu.stanford.nlp.sempre.SFun.append (call edu.stanford.nlp.sempre.SFun.parseStartState (string "ww pp eeee")) (string a) (number 3)))
Output: ww pp eeeea

Example 3 (Two Commands):
Input: kk qqf wwt
Utterance: add kk to first group. remove k from first group.
Program: (derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.remove (var g) (number 1) (boolean true))) ((lambda g (call edu.stanford.nlp.sempre.SFun.doubleGroup (var g) (number 1))) (call edu.stanford.nlp.sempre.SFun.parseStartState (string "kk qqf wwt")))))
Output: kkk qqf wwt

Example 4 (Two Commands):
Input: mm qq yy
Utterance: swap the first and third groups. cut the second group in half.
Program: (derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.remove (var g) (number 2) (boolean true))) ((lambda g (call edu.stanford.nlp.sempre.SFun.swap (var g) (number 1) (number 3))) (call edu.stanford.nlp.sempre.SFun.parseStartState (string "mm qq yy")))))
Output: yy q mm

5.3 RNN Error Analysis

Table 4: Two Utterance Errors

Type of Error	Proportion of Errors
Wrong Arguments	0.8657
Wrong Functions	0.8544
Correct Function, Wrong Arguments	0.1455
Correct Arguments, Wrong Function	0.1343

Our percentage of errors due to wrong arguments is higher than the percentage of errors due to wrong functions. This is to be expected: If we have a wrong function, then more likely than not we would have also supplied the incorrect arguments.

The most common error (occurred 6% of the time) is when fail to rank the identity function the highest.
Example
Input: nn ww zz
Utterance: swap the 1 and 3 groups. undo the last operation
Correct Program: (derivation (formula (call edu.stanford.nlp.sempre.SFun.parseStartState (string ”nn ww zz”)))
Predicted Program: (derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.reverseGroup (var g) (string 3))) (call edu.stanford.nlp.sempre.SFun.parseStartState (string nn ww zz”))))
Analysis: The RNN predicts a program that reverses group 3, whereas the oracle parse does nothing (applies identity function). However, the two outcomes are functionally the same, since reversing ”zz” does nothing. In the future, it would be better to explicitly model the ”undo” operation, which would probably make it easier for the neural network models to learn.

The second most common error (occurred 6% of the time) is using prepend instead of doubleGroup.
Example
Input: aa cc xx
Utterance: add aa to the first group. remove c from the second group. **Correct Program**:(derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.remove (var g) (number 2) (boolean true))) ((lambda g (call edu.stanford.nlp.sempre.SFun.append (var g) (string aa) (number 1))) (call edu.stanford.nlp.sempre.S Fun.parseStartState (string ”aa cc xx”)))))
Predicted Program: (derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.remove (var g) (number 2) (boolean false))) ((lambda g (call edu.stanford.nlp.sempre.SFun.prepend (var g) (string aa) (number 1))) (call edu.stanford.nlp.sempre. SFun.parseStartState (string ”aa cc xx”)))))
Analysis: The worker response is slightly incorrect; the intention was for the worker to say ”double group”, which is why the oracle parse involves a doubleGroup call. However, our neural network model does predict a program that is functionally the same as the oracle program, since adding ”aa” to the group containing ”aa” is the same as doubling ”aa”.

The third most common error (occurred 5% of the time) is parsing a program with doubleGroup and remove in reversed order.
Example
Input: bb j n
Utterance: double group 1. halve group 3
Correct Program:(derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.remove (var g) (number 3) (boolean true))) ((lambda g (call edu.stanford.nlp.sempre.SFun.doubleGroup (var g) (number 1))) (call edu.stanford.nlp.sempre.SFun.parseStartState (string ”bb j nn”)))))
Predicted Program: (derivation (formula ((lambda g (call edu.stanford.nlp.sempre.SFun.doubleGroup (var g) (number 1))) ((lambda g (call edu.stanford.nlp.sempre.SFun.remove (var g (number 3) (boolean true))) (call edu.stanford.nlp.sempre.SFun.parseStartSta e (string ”bb j nn”)))))
Analysis: The correct program would first double the first group, and then remove a ’n’ from the third group. Our predicted program does that in the opposite order. It could be the case that our

RNN does not currently keep track of order.

6 Conclusions

As expected, the RNN model performed the best. The key reason for this is that the RNN allows us to take into account factors such as word order in the original utterance. For example, the Siamese network consistently erred on "swap" functions, as it cannot mathematically distinguish correct input order since word vectors are added without taking order to account. Meanwhile, the RNN was able to get almost all of these examples correct.

Because of this blindness to word order in particular, the Siamese and linear embedding models performed much worse on the 2 utterance dataset compared to the one utterance dataset. The performance of the RNN dropped between these datasets too, though not as dramatically. This decrease can be explained by the fact that the number of candidate parses increases by a substantial amount in between the 1-utterance commands and the 2 utterance commands because SEMPRE is able to find more working parses for the 2 utterance commands. As a result, the problem of ranking these parses becomes more difficult.

Our work suggests that applying deep learning to the general problem of semantic parsing can be a very promising avenue of research. Our hybrid model combines using hand-specified grammars to construct our candidate parses and then using deep learning to rank these parses. As seen by the effectiveness of our RNN compared to the baseline, this approach allows us to avoid the problem of hand-crafting features to rank the parses, which is that these features could be cumbersome to exhaustively implement. Vector space embedding models allow greater flexibility. Likewise, it would also be difficult to implement a deep learning model that generates a program from scratch given the natural language utterance, as this would require much more data and would almost be a machine translation task for programming languages. Applying semantic parsing first lets us avoid this issue by narrowing the search space for potential function translations.

7 Future Work

There are still different network architectures and scoring functions we can explore - for example, instead of L2 distance, there are other popular distance measures we can try for our scoring function. Cosine distance, for example, seems to be better for the purpose of comparing word vectors. In terms of network architecture, we could expand upon the RNN model by implementing more complicated networks such as bi-directional RNNs, for example.

There are also many ways to extend the scope of our problem. One possible direction is to extend the number of utterances to parse 3 or more examples. However, this does not seem to be that technically interesting.

We are able to perform well because the workers generally gave enough information to completely determine each function call in the program. This is fairly cumbersome, however, and unlikely to be used in practice. We would like to incorporate context dependencies into parsing programs, that is, allow the end user to omit arguments and refer to previous commands. In our dataset, there are some commands that say "undo", "repeat the last step", etc. that we are currently unable to parse correctly. This is partially due to the current limitations of SEMPRE in generating logical forms. Instead of relying on traditional ML to perform the semantic parsing, we could build an end-to-end neural network system that would handle everything from program generation to scoring.

Lastly, another direction would be to broaden/choose a different domain. Simple string processing has a relatively limited domain; a more complex domain could induce more interesting phenomena that would require new techniques to solve.

8 References

[1] Alexander, J.A. & Mozer, M.C. (1995) Template-based algorithms for connectionist rule extraction. In G. Tesauro, D. S. Touretzky and T.K. Leen (eds.), *Advances in Neural Information Processing Systems 7*, pp. 609-616. Cambridge, MA: MIT Press.

[2] Bower, J.M. & Beeman, D. (1995) *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* New York: TELOS/Springer-Verlag.

[3] Hasselmo, M.E., Schnell, E. & Barkai, E. (1995) Dynamics of learning and recall at excitatory recurrent synapses and cholinergic modulation in rat hippocampal region CA3. *Journal of Neuroscience* **15**(7):5249-5262.

[4] Liang, Percy, and Christopher Potts. (2014) Bringing machine learning and compositional semantics together. Annual Reviews of Linguistics (submitted), 0.

[5] Zettlemoyer, Luke S. and Collins, Michael. (2005) Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. *arxiv.org published*

[6] Liang, Percy, Michael I. Jordan, and Dan Klein. "Learning dependency-based compositional semantics." Computational Linguistics 39.2 (2013): 389-446.

[7] Kwiatkowski, Tom, et al. "Scaling semantic parsers with on-the-fly ontology matching." (2013).

[8] Socher, Richard, et al. "Parsing natural scenes and natural language with recursive neural networks." Proceedings of the 28th international conference on machine learning (ICML-11). 2011.

[9] R. Knoll and M. Mezini, Pegasus: first steps toward a naturalistic programming language, in OOPSLA 06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. New York, NY, USA: ACM, 2006, pp. 542559.

[10] A. Cozzie, S. King. "Macho: Writing Programs with Natural Language and Examples" in Univeresity of Illinois Technical Report, 2012.

[11]E. W. Dijkstra, On the foolishness of natural language programming, http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/ EWD667.html.

[12]S.R.K. Branavan, H. Chen, L. Zettlemoyer and R. Barzilay "Reinforcement Learning for Mapping Instructions to Actions", Proceedings of ACL, 2009.

[13]S.R.K. Branavan, N. Kushman, T. Lei, and R. Barzilay, "Learning High-Level Planning from Text",Proceedings of ACL, 2012.

[14]N. Kushman, Y. Artzi, L. Zettlemoyer, and R. Barzilay"Learning to Automatically Solve Algebra Word Problems", Proceedings of ACL, 2014.

[15]Bordes, Antoine, Sumit Chopra, and Jason Weston. "Question answering with subgraph embeddings." arXiv preprint arXiv:1406.3676 (2014).