NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System

Xi Victoria Lin*, Chenglong Wang, Luke Zettlemoyer, Michael D. Ernst
Salesforce Research, University of Washington, University of Washington, University of Washington
xilin@salesforce.com, {clwang,lsz,mernst}@cs.washington.edu

Abstract

We present new data and semantic parsing methods for the problem of mapping english sentences to Bash commands (NL2Bash). Our long-term goal is to enable any user to easily solve otherwise repetitive tasks (such as file manipulation, search, and application-specific scripting) by simply stating their intents in English. We take a first step in this domain, by providing a large new dataset of challenging but commonly used commands, along with the baseline methods to establish performance levels on this task.

Keywords: Natural Language Programming, Natural Language Interface, Semantic Parsing

1. Introduction

The dream of using English or any other natural language to program computers has been around almost as long as the task of programming itself (Sammet, 1966). Although significantly less precise than a formal language (Dijkstra, 1978), natural language programming would be universally accessible and even relatively modest implementations would support the automation of highly repetitive tasks such as file manipulation, search, and other application-specific scripting (Dahl et al., 1994; Quirk et al., 2015; Desai et al., 2016).

In this work, we present new data and semantic parsing methods on a novel and ambitious domain – natural language operating system control. Our long-term goal is to enable any user to perform otherwise repetitive tasks on their computers by simply stating their goals in English. We take a first step in this direction, by providing a large new dataset (NL2Bash) of challenging but commonly used commands, along with the baseline methods to establish performance levels on this task[1].

The NL2Bash problem can be seen as a type of semantic parsing, where the goal is to map sentences to formal representations of their underlying meaning (Mooney, 2014). The dataset we introduce provides a new type of target meaning representations (shell commands), and is significantly larger (from two to ten times) than most existing semantic parsing benchmarks (Dahl et al., 1994; Popescu et al., 2003). Other recent work in semantic parsing has also focused on programming languages, including regular expressions (Locascio et al., 2016), IFTTT scripts (Quirk et al., 2015), and SQL queries (Kwiatkowski et al., 2013; Iyer et al., 2017; Zhong et al., 2017). However, the shell command we consider presents unique challenges, due to irregular syntax (no syntax tree representation for the command options), wide domain coverage ($\geq$ 100 commands), and a relatively large percentage of unseen words (e.g. commands can manipulate arbitrary files).

We constructed the NL2Bash corpus with frequently used commands scraped from websites such as question-answering forums, tutorials, tech blogs, and course materials. We gathered a set of high-quality descriptions of the command functionality from bash programmers. Table 1 shows several examples. After careful quality control, we were able to gather over 9,000 English-command pairs, covering over 100 unique bash commands.

We also present a set of experiments to demonstrate that NL2Bash is a challenging task which is worthy of future study. We build on recent work in neural semantic parsing (Dong and Lapata, 2016; Ling et al., 2016), by evaluating a relatively standard seq2seq model (Sutskever et al., 2014) and a CopyNet (Gu et al., 2016) model. We also include a recently proposed hybrid neural model, Tellina, that includes manually designed heuristics for better detect and translate rare command arguments (Lin et al., 2017). The best performing system obtains top-1 command structure accuracy of 55%, and top-1 full command accuracy of 32%. These performance levels, although far from perfect, are high enough to be directly useful in a well designed interface (Lin et al., 2017), and also suggest these is significant room for future modeling innovations with the data.

2. Domain: Linux Shell Commands

As shown in Table 1, there are three basic components in a shell command: utility (e.g. `find`, `grep`, `tar`), command option (e.g. `-l`, `-name`) and argument (e.g. `*.java`, `5`). These inputs specify command-specific runtime behavior and can have idiomatic syntax (e.g. see the `{}` and `\;` arguments to `grep` in Table 1).

There are over 250 Bash commands, and new ones are regularly introduced by third party developers. To prevent sparsity in the dataset, we focus on only the 100 most commonly used Linux commands[2] We also restrict the target commands to be one-liners (those that can be specified in a single line) which perform atomic functions and leave synthesizing blocks of shell scripts to future work. Table 2 summarizes the in-scope and out-scope syntactic structures of the shell commands we considered.

* Work done at the University of Washington.

[1] The Bourne-again shell (Bash) is a popular Unix shell and command language. Our data collection and modeling approaches can be trivially generalized to other Unix shell languages.

[2] http://www.oliverelliott.org/article/computing/ref_unix/

<table>
<tr><th>Natural Language</th><th>Bash Command</th></tr>
<tr><td>find .java files in the current directory tree that contain the pattern 'TODO' and print their names</td><td><code>grep -l "TODO" *.java</code>
<code>find . -name "*.java" -exec grep -il "TODO" {} \;</code>
<code>find . -name "*.java" | xargs -I {} grep -l "TODO" {}</code></td></tr>
<tr><td>display the 5 largest files in the current directory and its sub-directories</td><td><code>ls -l | sort -nk 5,5 | tail -5</code>
<code>du -a . | sort -rh | head -n5</code>
<code>find . -type f -printf '%s %p\n' | sort -rn | head -n5</code></td></tr>
<tr><td>search for all jpg images on the system and archive them to tar ball "images.tar"</td><td><code>tar -cvf images.tar *.jpg</code>
<code>tar -rvf images.tar *.jpg</code>
<code>find / -type f -name "*.jpg" -exec tar -cvf images.tar {} \;</code></td></tr>
</table>

Table 1: Sample natural language descriptions and corresponding shell commands. In our data, the mapping between natural language and shell commands is many-to-many.

<table>
<tr><td>In-scope</td><td>1. Single command
2. Logical connectives: <code>&&</code>, <code>||</code>, parentheses <code>()</code>
3. Nested commands:
- pipeline <code>|</code>
- command substitution <code>$()</code>
- process substitution <code><()</code></td></tr>
<tr><td>Out-of-scope</td><td>1. I/O redirection <code><</code>, <code><<</code>
2. Variable assignment <code>=</code>
3. Compound statements:
- code blocks such as <code>if</code>, <code>for</code>, <code>while</code>
- function definition
4. Non-bash program strings nested with language interpreters such as <code>awk</code>, <code>sed</code>, <code>python</code>, <code>java</code></td></tr>
</table>

Table 2: Syntax scope of the target shell commands in our dataset.

3. Corpus Construction

We collected 12,609 text–command pairs from the web, each consisting of a natural language description and a shell command. After filtering, 9,301 pairs remained (§3.1.). We split this data into train, dev, and test sets, subject to the constraint that no natural language description appears in more than one split and no shell command appears in more than one split (§3.3.). Our dataset is publicly available for use by other researchers.

3.1. Data Collection

We hired 10 Upwork[3] freelancers who were familiar with shell scripting. They recorded text–command pairs from web pages such as question-answering forums, tutorials, tech blogs, and course materials. We provided a web application to assist with searching, page browsing, data entry, and deduplication.

The freelancers copied the shell command from the webpage, and either copied the text from the webpage or generated the text based on their background knowledge and the web page context.[4] We restricted the natural language description to be a single sentence, as we found that in most cases one sentence is enough to accurately describe the command. The freelancer generated one natural language description for each command on a webpage. A freelancer might annotate the same command multiple times in different webpages, and multiple freelancers might annotate the same command (on the same or different webpages). Collecting multiple different descriptions increases language diversity in the dataset. On average, each freelancer collected 50 pairs per hour.

We used an automated process to filter and clean the dataset, as described below. [5]

Filtering We discard all commands which met the following criteria.

- Non-syntactic commands that violate the syntax specification in the Linux manual pages.[6]
- Commands that contain out-of-scope shell operators, as shown in Table 2.
- Commands that refer to language interpreters (e.g. `python` and `c++`) or third-party software (e.g. `brew` and `emacs`), as they contain strings in other programming languages.
- Commands that are mostly used as keywords in longer bash scripts (e.g. `alias` and `set`).

Cleaning We corrected spelling errors in the natural language descriptions using a probabilistic spell checker.[7] We also manually corrected a subset of the spelling errors that bypassed the spell checker in the natural language and in the shell commands.

For commands, we removed `sudo` and shell input prompt characters such as "$" and "#" from the beginning of each command. We replaced absolute command pathnames by their base names (e.g., we changed "/bin/find" to `find`).

3.2. Corpus Statistics

After filtering and cleaning, our dataset contains 9,301 pairs. The commands invoke 100 shell utilities (fig. 1) using 537 distinct flags.

[3] http://www.upwork.com/

[4] We told the freelancers that "a natural language description needs to be precise enough such that another programmer without seeing the original shell command can write a semantically equivalent one after reading your description." In manual inspection of 50 cases we found the data quality to be high.

[5] The code is distributed along with the dataset.

[6] https://linux.die.net/man/

[7] http://norvig.com/spell-correct.html

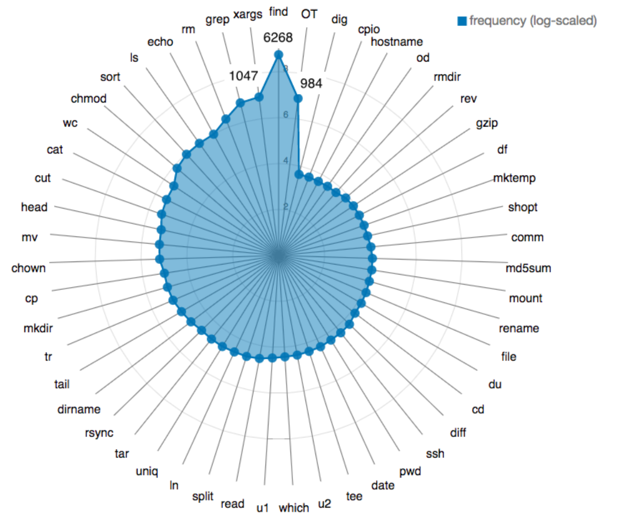

Figure 1: Top 50 most frequent bash utilities in the dataset and their frequencies plotted in log scaled (`u1` denotes `basename` and `u2` denotes `readlink`). The top most frequent utility `find` appeared in 6,268 commands and the second most frequent utility `xargs` appeared in 1,047 commands. "OTHER" corresponds to the rest of the 50 utilities combined and they appeared in 984 commands in total.

There are 8,555 unique natural language descriptions (6,237 unique tokens) and 7,583 unique bash commands (5,640 unique tokens) in the dataset. On average, each natural language description has 1.1 bash command solutions and each bash command has 1.2 natural language descriptions.

3.3. Data Split

We split the filtered data into train, development (dev) and test sets. We first clustered the pairs by the natural language description after lemmatization and stop-words filtering — a cluster contains all pairs with the identical natural language description. The clusters were randomly split into train and test at a ratio of 5:1. We then put all test pairs whose target command has appeared in train into the train set. This prevents a model from achieving high performance on the dataset by trivially memorizing training data, which allows us to evaluate the model's ability to generalize. For hyperparameter tuning, we randomly selected 100 examples from the train set as dev, given that they do not overlap with the rest of the training examples in terms of natural language description or command[8]. The resulting training, dev, and test split has 8,146, 100, and 1,055 pairs respectively.

4. Evaluation Methodology

In our dataset, one natural language description may have multiple correct bash command translations, and one bash command may be phrased in multiple different natural language descriptions.[9] This presents challenges for evaluation since our dataset does not cover all correct command solutions.

Traditionally, BLEU is wildly adopted to measure the translation quality (Doddington, 2002). human judgments. However, the n-gram overlap captured by BLEU is not effective in measuring either syntactic correctness or semantic similarity for formal languages.

Ideally, since bash commands are executable, one can expect to formally verify if two bash commands are identical or different. Unfortunately, no formal semantics for bash exists. Alternately, the commands can be tested: they can be executed in a virtual environment and the execution outcome can be compared to the reference outcome. However, it is challenging to test in all possible environments using all possible file systems, usernames, dates, etc. Moreover, passing the same test cases gives no guarantee that the two commands are semantically equivalent (Guu et al., 2017), nor that the text is an accurate description of the command.

Manual Evaluation Due to the challenges mentioned above, we adopted only manual evaluation metrics for this task. We hired three Upwork freelancers who are familiar with shell scripting. To evaluate a particular system, the freelancers independently evaluated the correctness of its top-3 translations for all test examples. For each command translation, we use the majority vote of the three freelancers as the final evaluation.

We report two types of accuracy: top-k full-command accuracy ($\mathrm{Acc}_\mathrm{F}^k$) and top-$k$ command-template accuracy ($\mathrm{Acc}_\mathrm{T}^k$). We define $\mathrm{Acc}_\mathrm{F}^k$ to be the percentage of test examples[10] for which a correct full command is ranked k or above in the model output. We define $\mathrm{Acc}_\mathrm{T}^k$ to be the percentage of test examples for which a correct command template is ranked k or above in the model output (i.e., ignoring errors in the constants).

5. Challenges and Comparison to Existing Datasets

We observed the following characteristics of shell commands which present unique challenges for sematic parsing.

- Wide application domain: the application domain of shell commands ranges from file system management, text processing, to advanced operating system control functions such as process management and networking. Solving semantic parsing for Bash is equivalent to solving it for each of the applications.

- Flexibility: many bash commands have a large set of functional flags (§ 3.2.) and multiple commands can be combined to solve more complex tasks. This often results in multiple correct solutions for one task (table 1), and posts challenges for both training and evaluation.

- Irregular syntax: the Bash interpreter uses a shallow syntactic grammar to parse pipelines, code blocks and other high-level syntax structures. The command options are parsed largely through pattern matching and

[8] We used a small dev set since it makes quick computation of manual evaluation metrics possible. And we found that the metrics over 100 examples accurately reflects the relative performance of the different choices

[9] This many-to-many mapping is common in machine translation datasets (Papineni et al., 2002), but rare for semantic parsing ones.

[10] We treat the ⟨bash, NL⟩ pairs with the same NL description as a single test example.

each command can have customized rules for parsing its options. Many commands have arguments subjected to specific format (e.g. to specify an `ssh` remote, the format needs to be `[USER@]HOST:SRC`). Grammar-driven semantic parsing approaches (Yin and Neubig, 2017; Guu et al., 2017) hence become difficult to apply.

- Domain-specific entities: Bash commands have large amount of open-vocabulary arguments (e.g. file names, date/time expression etc.), which causes the dataset vocabulary to contain a long tail of low-frequency tokens.

6. Baseline Systems Performance

We evaluated two neural machine translation models which have demonstrated strong performance in both NL-to-NL translation and NL-to-code translation tasks, namely, Seq2Seq (Sutskever et al., 2014; Dong and Lapata, 2016) and CopyNet (Gu et al., 2016).

6.1. Baseline Systems

Seq2Seq The Seq2Seq (sequence-to-sequence) model parameterizes conditional probability of an output sequence given the input sequence with a recurrent neural network (RNN) encoder-decoder architecture (Jain and Medsker, 1999; Sutskever et al., 2014). When applied to NL-to-code translation the input natural language and output commands are both treated as a sequence of tokens, and the command sequences with the highest conditional probabilities according to the model were selected as candidate translations.
We used the same attention mechanism and cross-entropy training objective as (Sutskever et al., 2014). We used the gated recurrent unit (GRU) (Chung et al., 2014) formulation for the RNN cells, and a bidirectional RNN (Schuster and Paliwal, 1997) encoder.

CopyNet CopyNet (Gu et al., 2016) is an extension of the Seq2Seq model which is trained to select sub-sequences in the input sequence and output them at proper places in the output sequence. The copy action is integrated with regular token generation in the RNN decoder and the entire model is trained end-to-end. We used the same copying formulation as (Gu et al., 2016). Other parts of the architecture were kept the same as our Seq2Seq implementation.
We evaluated both Seq2Seq and CopyNet at three levels of token granularities: token, character and sub-token. We expect that the character and sub-token based models are more tolerant to the sparsity caused by the domain-specific entities (§ 5.). We used a regular expression based natural language tokenizer and a parser augmented from Bashlex[11] to parse and tokenize the bash commands. [12]

[11] `https://github.com/idank/bashlex`

[12] For the sub-token based models, we broke down every token into consecutive sequences of alphabet letters and consecutive sequences of numbers; all other special characters are treated as an individual token. A sequence of sub-tokens obtained from the same token are padded with the special `SUB_START` and `SUB_END` symbols. For example, the file path "/home/dir03/*.txt" is converted to the sub-token sequence `SUB_START`, "/", "home", "/", "dir", "03", "/", "*", ".", "txt", `SUB_END`.

Model		Acc^1_F	Acc^3_F	Acc^1_T	Acc^3_T
Seq2Seq	Token	0.10	0.17	0.47	**0.62**
	Char	0.12	0.20	0.23	0.30
	Sub-token	0.20	0.24	0.43	0.48
CopyNet	Char	0.15	0.23	0.23	0.30
	Token	0.25	0.34	**0.52**	0.60
	Sub-token	0.28	**0.36**	0.43	0.53
Tellina		**0.31**	**0.36**	0.50	0.59

Table 3: Development set translation accuracies of the baseline system.

Tellina In addition, we evaluated a stage-wise synthesis model, Tellina (Lin et al., 2017), which first abstracts the entity constants in both the input natural language and output commands into semantic types (e.g. `File` and `Size`), then performs translation at the template level and finally fills the command template slots using the extracted entities via a learned alignment model and reformatting heuristics.

Hyperparameters We set up the decoder RNN to be 400-dimensional, and the two RNNs in the bi-directional encoder to be 200-dimensional. We optimize the learning objective with mini-batched Adam (Kingma and Ba, 2014), using a initial learning rate of 0.0001 and the default momentum hyperparameters of Adam. Our mini-batch size is 128. We used variational dropout (Gal and Ghahramani, 2016) with 0.4 dropout rate. For decoding we set the beam size to 100. The hyperparameters were set based on the model's performance on a development dataset (§3.3.). We kept all characters appeared in train for the character models and all tokens appeared four or more times in train for the token and sub-token based models.
We will release our trained models and source code along with the dataset.

6.2. Evaluation Results

Table 3 shows the performance of the baselines systems on our development set. Notice that CopyNet consistently outperform the vanilla Seq2Seq model at all token granularities. In general, using tokens gives high command structure accuracy, while using sub-tokens gives high full command accuracy. The Tellina model which does token-level template translation and argument filling/reformatting in a stage-wise manner gives the highest full command accuracy. The CopyNet model using sub-tokens perform on par with it. The performance of character-based models is inferior compared to token and sub-token based models, especially in terms of command structure accuracy.
The test set accuracies of the best performing approach, Tellina, are Acc^1_F =32% and Acc^1_T =55%, which is slightly higher than its dev set scores.

7. Conclusions and Future Work

We studied the problem of mapping English sentences to Bash commands (NL2Bash), by introduce a large new dataset and baseline methods. Experiments demonstrated competitive performance of existing models but there is significant room for future work on this challenging semantic parsing problem.

8. Acknowledgements

The research was supported in part by DARPA under the DEFT program (FA8750-13-2-0019), the ARO (W911NF-16-1-0121), the NSF (IIS1252835, IIS-1562364), gifts from Google and Tencent, and an Allen Distinguished Investigator Award. The authors thank Kenton Lee, Luheng He and Omer Levy for their constructive feedback on the paper draft, and the UW NLP/PLSE groups for helpful conversations on the work.

9. Bibliographical References

Regina Barzilay et al., editors. (2017). *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Association for Computational Linguistics.

Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling.

Dahl, D. A., Bates, M., Brown, M., Fisher, W., Hunicke-Smith, K., Pallett, D., Pao, C., Rudnicky, A., and Shriberg, E. (1994). Expanding the scope of the atis task: The atis-3 corpus. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 43–48, Stroudsburg, PA, USA. Association for Computational Linguistics.

(2016). *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., and Roy, S. (2016). Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, number 1 in ICSE '16, pages 345–356, New York, NY, USA. ACM.

Dijkstra, E. W. (1978). On the foolishness of "natural language programming". In Friedrich L. Bauer et al., editors, *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, germany*, volume 69 of *Lecture Notes in Computer Science*, pages 51–53. Springer.

Doddington, G. (2002). Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the Second International Conference on Human Language Technology Research*, HLT '02, pages 138–145, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Dong, L. and Lapata, M. (2016). Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, August. Association for Computational Linguistics.

Gal, Y. and Ghahramani, Z. (2016). A theoretically grounded application of dropout in recurrent neural networks. In Daniel D. Lee, et al., editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1019–1027.

Gu, J., Lu, Z., Li, H., and Li, V. O. K. (2016). Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers* (DBL, 2016).

Guu, K., Pasupat, P., Liu, E. Z., and Liang, P. (2017). From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In Barzilay and Kan (Barzilay and Kan, 2017), pages 1051–1062.

Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., and Zettlemoyer, L. (2017). Learning a neural semantic parser from user feedback. *CoRR*, abs/1704.08760.

Jain, L. C. and Medsker, L. R. (1999). *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

Kwiatkowski, T., Choi, E., Artzi, Y., and Zettlemoyer, L. (2013). Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1545–1556, Seattle, Washington, USA, October. Association for Computational Linguistics.

Lin, X. V., Wang, C., Pang, D., Vu, K., Zettlemoyer, L., and Ernst, M. D. (2017). Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kociský, T., Wang, F., and Senior, A. (2016). Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers* (DBL, 2016).

Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., and Barzilay, R. (2016). Neural generation of regular expressions from natural language with minimal domain knowledge. In Jian Su, et al., editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP, November 1-4, 2016*, pages 1918–1923, Austin, Texas, USA. The Association for Computational Linguistics.

Mooney, R. J. (2014). Semantic parsing: Past, present, and future.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA. Association for Computational Linguistics.

Popescu, A.-M., Etzioni, O., and Kautz, H. (2003). Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, pages 149–157, New York, NY, USA. ACM.

Quirk, C., Mooney, R. J., and Galley, M. (2015). Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Volume 1: Long Papers*, pages 878–888, Beijing, China. The Association for Computer Linguistics.

Sammet, J. E. (1966). The use of english as a programming language. *Communications of the ACM*, 9(3):228–230.

Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11):2673–2681, November.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA. MIT Press.

Yin, P. and Neubig, G. (2017). A syntactic neural model for general-purpose code generation. In Barzilay and Kan (Barzilay and Kan, 2017), pages 440–450.

Zhong, V., Xiong, C., and Socher, R. (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.