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 perform operations such as file manipulation, search, and application-specific scripting by simply stating their goals in English. We take a first step in this domain, by providing a new dataset of challenging but commonly used Bash commands and expert-written English descriptions, along with 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 existed for almost as long as the task of programming itself (Sammet, 1966). Although significantly less precise than a formal language (Dijkstra, 1978), natural language as a programming medium would be universally accessible and would support the automation of highly repetitive tasks such as file manipulation, search, and application-specific scripting (Wilensky et al., 1984; Wilensky et al., 1988; Dahl et al., 1994; Quirk et al., 2015; Desai et al., 2016).

This work presents new data and semantic parsing methods on a novel and ambitious domain — natural language control of the operating system. Our long-term goal is to enable any user to perform tasks on their computers by simply stating their goals in natural language (NL). We take a first step in this direction, by providing a large new dataset (NL2Bash) of challenging but commonly used commands and expert-written descriptions, along with baseline methods to establish performance levels on this task.

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 (Bash[1] 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 data we consider raises unique challenges, due to its irregular syntax (no syntax tree representation for the command options), wide domain coverage (> 100 Bash utilities), and a large percentage of unseen words (e.g. commands can manipulate arbitrary files).

We constructed the NL2Bash corpus with frequently used Bash 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 commands 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 utilities.

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 the standard Seq2seq model (Sutskever et al., 2014) and the CopyNet model (Gu et al., 2016). We also include a recently proposed stage-wise neural semantic parsing model, Tellina, which uses manually defined heuristics for better detecting and translating the command arguments (Lin et al., 2017). We found that when applied at the right sequence granularity (sub-tokens), CopyNet significantly outperforms the stage-wise model, with significantly less pre-processing and post-processing. Our best performing system obtains top-1 command structure accuracy of 49%, and top-1 full command accuracy of 36%. These performance levels, although far from perfect, are high enough to be practically useful in a well-designed interface (Lin et al., 2017), and also suggest ample room for future modeling innovations.

2. Domain: Linux Shell Commands

A shell command consists of three basic components, as shown in Table 1: utility (e.g. `find`, `grep`), option flags (e.g. `-name`, `-i`), and arguments (e.g. `"*.java"`, `"TODO"`). A utility can have idiomatic syntax for flags (see the `-exec ... {} \;` option of the `find` command).

There are over 250 Bash utilities, and new ones are regularly added by third party developers. We focus on 135 of the most useful utilities identified by the Linux user group (http://www.oliverelliott.org/article/computing/ref_unix/), that is, our domain of target commands contain only those 135 utilities.[2] We only considered the target commands that can

* Work done at the University of Washington.

[1]The Bourne-again shell (Bash) is the most popular Unix shell and command language: https://www.gnu.org/software/bash/. Our data collection approach and baseline models can be trivially generalized to other command languages.

[2]We were able to gather fewer examples for the less common ones. Providing the descriptions for them also requires a higher level of Bash expertise of the corpus annotators.

Natural Language	Bash Command(s)

find .java files in the current directory tree that contain the pattern 'TODO' and print their names	`grep -l "TODO" *.java`
	`find . -name "*.java" -exec grep -il "TODO" {} \;`
	`find . -name "*.java" | xargs -I {} grep -l "TODO" {}`

display the 5 largest files in the current directory and its sub-directories	`find .  -type f | sort -nk 5,5 | tail -5`
	`du -a . | sort -rh | head -n5`
	`find . -type f -printf '%s %p\n' | sort -rn | head -n5`

search for all jpg images on the system and archive them to tar ball "images.tar"	`tar -cvf images.tar $(find / -type f -name *.jpg)`
	`tar -rvf images.tar $(find / -type f -name *.jpg)`
	`find / -type f -name "*.jpg" -exec tar -cvf images.tar {} \;`

Table 1: Example natural language descriptions and the corresponding shell commands from NL2Bash.

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

Table 2: In-scope and out-of scope syntax for the Bash commands in our dataset.

be specified in a single line (one-liners).[3] Among them, we omitted commands that contain syntax structures such as I/O redirection, variable assignment, and compound statements because those commands need to be interpreted in context. Table 2 summarizes the in-scope and out-of-scope syntactic structures of the shell commands we considered.

3. Corpus Construction

The corpus consists of text–command pairs, where each pair consists of a Bash command scraped from the web and an expert-generated natural language description. Our dataset is publicly available for use by other researchers: https://github.com/TellinaTool/nl2bash/tree/master/data.
We collected 12,609 text–command pairs in total (§3.1.). After filtering, 9,305 pairs remained (§3.2.). We split this data into train, development (dev), and test sets, subject to the constraint that neither a natural language description nor a Bash command appears in more than one split (§3.4.).

3.1. Data Collection

We hired 10 Upwork[4] freelancers who are familiar with shell scripting. They collected text–command pairs from web pages such as question-answering forums, tutorials, tech blogs, and course materials. We provided them a web inferface to assist with searching, page browsing, and data entry.
The freelancers copied the Bash command from the webpage, and either copied the text from the webpage or wrote the text based on their background knowledge and the webpage context. We restricted the natural language description to be a single sentence and the Bash command to be a one-liner. We found that oftentimes one sentence is enough to accurately describe the function of the command.[5]
The freelancers provided 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/hour.

3.2. Data Cleaning

We used an automated process to filter and clean the dataset, as described below. Our released corpus includes the filtered data, the full data, and the cleaning scripts.

Filtering The cleaning scripts removed the following commands.

- Non-grammatical commands that violate the syntax specification in the Linux man pages (https://linux.die.net/man/).
- Commands that contain out-of-scope syntactic structures shown in Table 2.
- Commands that are mostly used in multi-statement shell scripts (e.g. `alias` and `set`).
- Commands that contain non-bash language interpreters (e.g. `python`, `c++`, `brew`, `emacs`). These commands contain strings in other programming languages.

Cleaning We corrected spelling errors in the natural language descriptions using a probabilistic spell checker (http://norvig.com/spell-correct.html). We also manually corrected a subset of the spelling errors that bypassed the spell checker in both the natural language and the shell commands. For Bash commands, we removed `sudo` and the shell input

[3] We decided to investigate this simpler case prior to synthesizing longer shell scripts because one-liner Bash commands are practically useful and have simpler structure. Our baseline results and analysis (§6.) show that even this task is challenging.
[4] http://www.upwork.com/

[5] As discussed in §6.3., in 4 out of 100 examples, a one-sentence description is difficult to interpret. Future work should investigate interactive natural language programming approaches in these scenarios.

# sent.	# word	# words per sent.		# sent. per word	
		avg.	median	avg.	median
8,559	7,790	11.7	11	14.0	1

Table 3: Natural Language Statistics: # unique sentences, # unique words, # words per sentence and # sentences that a word appears in.

# cmd	# temp	# token	# tokens / cmd		# cmds / token	
			avg.	median	avg.	median
7,587	4,602	6,234	7.7	7	11.5	1

# utility	# flag	# reserv. token	# cmds / util.		# cmds / flag	
			avg.	median	avg.	median
102	206	15	155.0	38	101.7	7.5

Table 4: Bash Command Statistics. The top table contains # unique commands, # unique command templates, # unique tokens, # tokens per command and # commands that a token appears in. The bottom table contains # unique utilities, # unique flags, # unique reserved tokens, # commands a utility appears in and # commands a flag appears in.

.

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

3.3. Corpus Statistics

After filtering and cleaning, our dataset contains 9,305 pairs. The Bash commands cover 102 unique utilities using 206 flags — a rich functional domain.

Monolingual Statistics Tables 3 and 4 show the statistics of natural language (NL) and Bash commands in our corpus. The average length of the NL sentences and Bash commands are relatively short, being 11.7 words and 7.7 tokens respectively. The median word frequency and command token frequency are both 1, which is caused by the large number of open-vocabulary constants (file names, date/time expressions, etc.) that appeared only once in the corpus.[6]

We define a command template as a command with its arguments replaced by their semantic types. For example, the template of `grep -l "TODO" *.java` is `grep -l [regex] [file]`.

Mapping Statistics Table 5 shows the statistics of natural language to Bash command mappings in our dataset. While most of the NL sentences and Bash commands form one-to-one mappings, the problem is naturally a many-to-many mapping problem — there exist many semantically equivalent commands, and one Bash command may be phrased in different NL descriptions. This many-to-many mapping is common in machine translation datasets (Papineni et al., 2002), but rare for traditional semantic parsing ones (Dahl et al., 1994; Zettlemoyer and Collins, 2005).

# cmd per nl			# nl per cmd		
avg.	median	max	avg.	median	max
1.09	1	9	1.23	1	22

Table 5: Natural Language to Bash Mapping Statistics

As discussed in §4. and §6.2., the many-to-many mapping affects both evaluation and modeling choices.

Utility Distribution Figure 1 shows the top 50 most common Bash utilities in our dataset and their frequencies in log-scale. The distribution is long-tailed: the top most frequent utility `find` appeared 6,268 times and the second most frequent utility `xargs` appeared 1,047 times. The 52 least common bash utilities, in total, appeared only 984 times.[7]

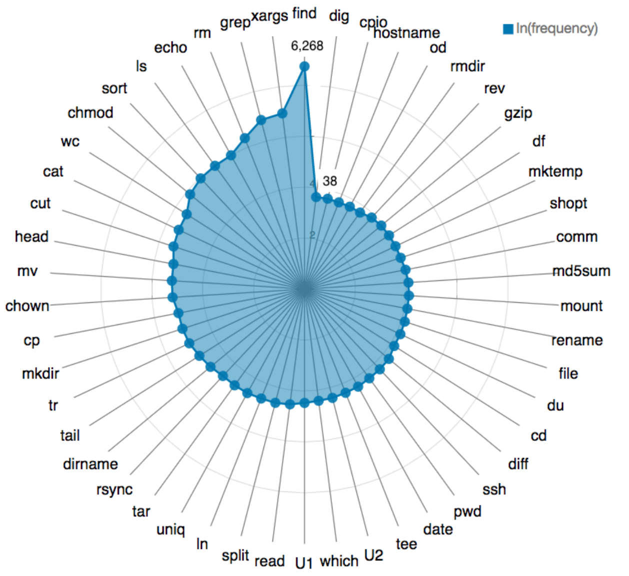


Figure 1: Top 50 most frequent bash utilities in the dataset with their frequencies in log scale. `U1` and `U2` at the bottom of the circle denote the utilities `basename` and `readlink`.

The appendix (§10.) gives more corpus statistics.

3.4. Data Split

We split the filtered data into train, dev, and test sets (Table 6). We first clustered the pairs by NL descriptions — a cluster contains all pairs with the identical normalized NL description. We normalized an NL description by lowercasing, stemming, and stop-word filtering, as described in §6.1.

We randomly split the clusters into train, dev, and test at a ratio of 10:1:1. After splitting, we moved all development and test pairs whose command appeared in the train set into the train set. This prevents a model from obtaining high accuracy by trivially memorizing a natural language

[6] As shown in figure 1, the most frequent bash utilities appeared over 6,000 times in the corpus. Similarly, natural language words such as "files", "in" appeared in 5,871 and 5,430 sentences, respectively. These extremely high frequency tokens are the reason for the significant difference between the averages and medians in Tables 3 and 4.

[7] The utility `find` is disproportionately common in our corpus. This is because we collected the data in two separated stages. As a proof of concept, we initially collected 5,413 commands that contain the utility `find` (and may also contain other utilities). After that, we allow the freelancers to collect all commands that contain any of the 135 utilities described in §2..

description or a command it has seen in the train set, which allows us to evaluate the model's ability to generalize.

	Train	Dev	Test
# pairs	8,090	609	606
# unique nls	7,340	549	547

Table 6: Data Split Statistics

4. Evaluation Methodology

In our dataset, one natural language description may have multiple correct Bash command translations. This presents challenges for evaluation since not all correct commands are present in our dataset.

Manual Evaluation 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 grouped the test pairs that have the same normalized NL descriptions as a single test instance (Table 6). 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 instances 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 instances for which a correct command template is ranked k or above in the model output (i.e., ignoring incorrect arguments).

Table 7 shows the inter-annotator agreement between the three pairs of our freelancers on both the template judgement (α_T) and full-command judgement (α_F).

Pair 1		Pair 2		Pair 3	
α_F	α_T	α_F	α_T	α_F	α_T
0.89	0.81	0.83	0.82	0.90	0.89

Table 7: Inter-annotator agreement.

Previous approaches Previous NL-to-code translation work also noticed similar problems.

(Kushman and Barzilay, 2013; Locascio et al., 2016) formally verify the equivalence of different regular expressions by converting them to minimal deterministic finite automaton (DFAs).

Others (Kwiatkowski et al., 2013; Long et al., 2016; Guu et al., 2017; Iyer et al., 2017; Zhong et al., 2017) evaluate the generated code through execution. As Bash is a Turing-complete language, verifying the equivalence of two Bash commands is undecidable. Alternatively, one can check command equivalence using test examples: two commands can be executed in a virtual environment and their execution outcome can be compared. We leave this evaluation approach to future work.

Some other works (Oda et al., 2015) have adopted fuzzy evaluation metrics, such as BLEU, which is widely used to measure the translation quality between natural languages (Doddington, 2002). Appendix C shows that the n-gram overlap captured by BLEU is not effective in measuring the semantic similarity for formal languages.

5. System Design Challenges

This section lists challenges for semantic parsing in the Bash domain.

Rich Domain The application domain of Bash ranges from file system management, text processing, network control to advanced operating system functionality such as process management. Semantic parsing in Bash is equivalent to semantic parsing for each of the applications. In comparison, many previous works focus on only one domain (§7.).

Out-of-Vocabulary Constants Bash commands contain many open-vocabulary constants such as file/path names, file properties, time expressions, etc. These form the unseen tokens for the trained model. Nevertheless, a semantic parser on this domain should be able to generate those constants in its output. This problem exists in nearly all NL-to-code translation problems but is particularly severe for Bash (§3.3.). What makes the problem worse is that oftentimes, the constants corresponding to the command arguments need to be properly reformatted following idiomatic syntax rules.

Language Flexibility Many bash commands have a large set of option flags, and multiple commands can be combined to solve more complex tasks. This often results in multiple correct solutions for one task (§3.3.), and poses challenges for both training and evaluation.

Idiomatic Syntax The Bash interpreter uses a shallow syntactic grammar to parse pipelines, code blocks, and other high-level syntax structures. It parses command options using pattern matching and each command can have idiomatic syntax rules (e.g. to specify an `ssh` remote, the format needs to be `[USER@]HOST:SRC`). Syntax-tree-based parsing approaches (Yin and Neubig, 2017; Guu et al., 2017) are hence difficult to apply.

6. Baseline System Performance

To establish performance levels for future work, we evaluated two neural machine translation models that 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). We also evaluated a stage-wise natural language programing model, Tellina (Lin et al., 2017), which includes manually-designed heuristics for argument translation.

Seq2Seq The Seq2Seq (sequence-to-sequence) model defines the conditional probability of an output sequence given the input sequence using an RNN (recurrent neural network) encoder-decoder (Jain and Medsker, 1999; Sutskever et al., 2014). When applied to the NL-to-code translation problem, the input natural language and output commands are treated as sequences of tokens. At test time, the command sequences with the highest conditional probabilities were output as candidate translations.

CopyNet CopyNet (Gu et al., 2016) is an extension of Seq2Seq which is able to select sub-sequences of the input sequence and emit them at proper places while generating the output sequence. The copy action is mixed with the regular token generation of the Seq2Seq decoder and the whole model is still trained end-to-end.

Tellina The stage-wise natural language programing model, Tellina (Lin et al., 2017), first abstracts the constants in an NL to their corresponding semantic types (e.g. `File` and `Size`) and performs template-level NL-to-code translation. It then fills the argument slots in the code template with the extracted constants using a learned alignment model and reformatting heuristics.

6.1. Implementation Details

We used the Seq2Seq formulation as specified in (Sutskever et al., 2014). We used the gated recurrent unit (GRU) (Chung et al., 2014) RNN cells and a bidirectional RNN (Schuster and Paliwal, 1997) encoder. We used the copying mechanism proposed by (Gu et al., 2016). The rest of the model architecture is the same as the Seq2Seq model.
We evaluated both Seq2Seq and CopyNet at three levels of token granularities: token, character and sub-token.

Pre-processing We used a simple regular-expression based natural language tokenizer and the Snowball stemmer to tokenize and stem the natural language. We converted all closed-vocabulary words in the natural language to lowercase and removed words in a stop-word list. We removed all NL tokens that appeared less than four times from the vocabulary for the token- and sub-token-based models. We used a Bash parser augmented from Bashlex (https://github.com/idank/bashlex) to parse and tokenize the bash commands.
To compute the sub-tokens[8], we split every constant in both the natural language and Bash commands into consecutive sequences of alphabetical letters and digits; all other characters are treated as an individual sub-token. (All Bash utilities and flags are treated as atomic tokens as they are not constants.) A sequence of sub-tokens as the result of a token split is padded with the special symbols `SUB_START` and `SUB_END` at the beginning and the end. For example, the file path "/home/dir03/*.txt" is converted to the sub-token sequence: `SUB_START`, "/", "home", "/", "dir", "03", "/", "*", ".", "txt", `SUB_END`.

Hyperparameters The dimension of our decoder RNN is 400. The dimension of the two RNNs in the bi-directional encoder is 200. We optimized the learning objective with mini-batched Adam (Kingma and Ba, 2014), using the default momentum hyperparameters. Our initial learning rate is 0.0001 and the mini-batch size is 128. We used variational RNN 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 (§3.4.).
Our baseline system implementation is released on Github: https://github.com/TellinaTool/nl2bash.

[8] As discussed in §6.2., the simple sub-token based approach is surprisingly effective for this problem. It avoids modeling very long sequences, as the character-based models do, by preserving trivial compositionality in consecutive alphabetical letters and digits. On the other hand, the separation between letters, digits, and special tokens explicitly represented most of the idiomatic syntax of Bash we observed in the data: the sub-token based models effectively learn basic string manipulations (addition, deletion and replacement of substrings) and the semantics of Bash reserved tokens such as `$`, `"`, `*`, etc.

Model		$\mathrm{Acc}_\mathrm{F}^1$	$\mathrm{Acc}_\mathrm{F}^3$	$\mathrm{Acc}_\mathrm{T}^1$	$\mathrm{Acc}_\mathrm{T}^3$
Seq2Seq	Char	0.24	0.27	0.35	0.38
	Token	0.10	0.12	**0.53**	0.59
	Sub-token	0.19	0.27	0.41	0.53
CopyNet	Char	0.25	0.31	0.34	0.41
	Token	0.21	0.34	0.47	**0.61**
	Sub-token	**0.31**	**0.40**	0.44	0.53
Tellina		0.29	0.32	0.51	0.58

Table 8: Translation accuracies of the baseline systems on 100 instances sampled from the dev set.

6.2. Results

Table 8 shows the performance of the baseline systems on 100 examples sampled from our dev set. Since manually evaluating all 7 baselines on the complete dev set is expensive, we report the manual evaluation results on a sampled subset in Table 8 and the automatic evaluation results on the full dev set in Appendix C.
Table 11 shows a few dev set examples and the baseline system translations. We now summarize the comparison between the different systems.

Token Granularity In general, token-level modeling yields higher command structure accuracy compared to using characters and sub-tokens. Modeling at the other two granularities gives higher full command accuracy. This is expected since the character and sub-token models need to learn token-level compositions. They also operate over longer sequences which presents challenges for the neural networks. It is somewhat surprising that Seq2Seq at the character level achieves competitive full command accuracy. However, the structure accuracy of these models is significantly lower than the other two counterparts.[9]

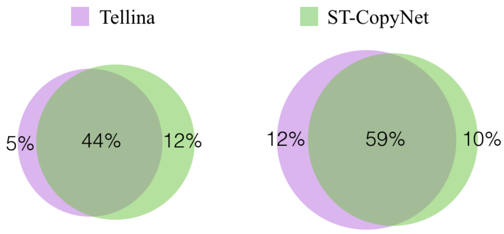
Copying Adding copying slightly improves the character-level models. This is expected as out-of-vocabulary characters are rare. Using token-level copying improves full command accuracy significantly from vanilla Seq2Seq. However, the command template accuracy drops slightly, possibly due to the mismatch between the source constants and the command arguments, as a result of argument reformatting. We observe a similarly significant full command accuracy improvement by adding copying at the sub-token level. The resulting ST-CopyNet model has the highest full command accuracy and competitive command template accuracy.

End-To-End vs. Pipline The Tellina model which does template-level translation and argument filling/reformatting in a stage-wise manner yields the second-best full command accuracy and second-best structure accuracy. Nevertheless, the higher full command accuracy of ST-CopyNet (especially on the $\mathrm{Acc}_\mathrm{T}^3$ metrics) shows that learned string-level transformations out-perform manually written heuristics

[9] (Lin et al., 2017) reported that incorrect commands can help human subjects, even when their arguments contain errors. This is because in many cases the human subjects were able to change or replace the wrong arguments based on their prior knowledge. Given this finding, we expect pure character-based models to be less useful in practice compared to the other two groups if we cannot find ways to improve their command structure accuracy.

Model	Acc^1_F	Acc^3_F	Acc^1_T	Acc^3_T
ST-CopyNet	**0.36**	**0.45**	0.49	0.61
Tellina	0.27	0.32	0.53	0.62

Table 9: Translation accuracies of ST-CopyNet and Tellina on the full test set.



(a) Template Error (b) Full Command Error

Figure 2: Error overlap of ST-CopyNet and Tellina. The number denotes the percentage out of the 100 dev samples.

when enough data is provided. This shows the promise of applying end-to-end learning on such problems in future work.

Table 9 shows the test set accuracies of the top-two performing approaches, ST-CopyNet and Tellina, evaluated on the entire test set. The accuracies of both models are higher than those on the dev set[10], but the relative performance gap holds: ST-CopyNet performs significantly better than Tellina on the full command accuracy, with only a mild decrease in structure accuracy.

Section 6.3. furthur discusses the comparison between these two systems through error analysis.

6.3. Error Analysis

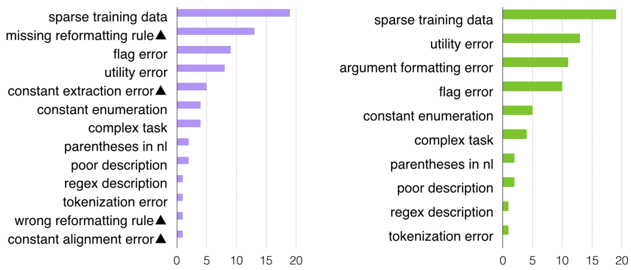
We manually examined the top-1 system outputs of ST-CopyNet and Tellina on the 100 dev set examples and compared their error cases.

Figure 2 shows the error case overlap of the two systems. For a significant proportion of the examples both systems made mistakes in their translation (44% by command structure error and 59% by full command error). This is because the base model of the two systems are similar — they are both RNN based models that perform sequential translation. Many such errors were caused by the NL describing a function that rarely appeared in the train set, or the GRUs failing to capture certain portions of the NL descriptions. For cases where only one of the models makes mistakes, Tellina makes fewer template errors and ST-CopyNet makes fewer full command errors.

We categorized the error causes of each system (Figure 3), and discuss the major error classes below.

Sparsity in Training Data For both models, the top-one error cause is when the NL description maps to utilities or flags that rarely appeared in the train set (Table 10). As mentioned in section 2., the bash domain consists of a large number of utilities and flags and it is expensive to gather enough training data for all of them.

[10]One possible reason is that two different sets of programmers evaluated the results on dev and test.



(a) Tellina Error Classes (b) ST-CopyNet Error Classes

Figure 3: Number of error instances in each error classes of ST-CopyNet and Tellina. The classes marked with ▲ are unique to the pipeline system.

Sparsity in training data	
	find all the text files in the file system and *<u>search only in the disk partition of the root</u>*.
Constant enumeration	
	Answer "n" to any prompts in the interactive recursive removal of <u>"dir1", "dir2", and "dir3"</u>.
Complex task	
	<u>Recursively finds all files in a current folder excluding already compressed files and compresses them with level 9</u>.
Intelligible/Non-grammatical description	
	<u>Find all regular files in the current directory tree and print a command to move them to the current directory</u>.

Table 10: Samples of natural language descriptions for the major error causes.

Common Errors of RNN Translation Models The second major error class is commonly-known errors for RNN-based translation models (utility error, flag error and argument formatting error in Figure 3). When the RNN misinterprets or overlooks certain chunks of NL descriptions, the decoder can generate a wrong utility/flag or omit a utility/flag from the output sequence. Since the ST-CopyNet model also relies on the RNNs to generate sub-token contents, it suffers more from such problems — the sub-token based models in general have more command structure errors and they frequently generated arguments that are a few edit distance away from the correct ones. Interestingly, we noticed that few command template errors are syntax errors. The output commands often remain executable despite the semantic errors in different Bash components.

Constant Enumeration In some cases, the NL descriptions contain sequences of constant values as an enumeration of system objects or string patterns (Table 10). We observed that both models struggled to extract all the constants correctly from this type of descriptions and usually failed to set the extracted constants into the correct command slots. Moreover, long sequences of OOVs also down-qualify the RNN encodings and both models made more command structure errors in such cases.

Natural Language	Model Output	Model
Delete all empty directories in minimum 2 levels down the root directory	`find root -mindepth 2 -type d -empty -delete`	Human
	`find root -mindepth 2 -type d -empty -delete`	C-Seq2Seq
	`find root -mindepth 2 -type d -empty -delete`	C-CopyNet
	`find root -mindepth 2 -type d -empty -delete`	T-Seq2Seq
	`find root -mindepth 2 -type d -empty -delete`	T-CopyNet
	`find root -mindepth 2 -type d -empty -delete`	ST-Seq2Seq
	`find root -mindepth 2 -type d -empty -delete`	ST-CopyNet
	`find . -mindepth 2 -type d -empty -delete`	Tellina
Find all Subscription.java files/directories under current directory and enter into the parent directory of the first one found	`cd $(find . -name Subscription.java -printf '%h\n')`	Human
	`find . -name Subscription.java ...` *incomplete*	C-Seq2Seq
	`find . -name Subscription.java | xargs grep Subscription.java`	C-CopyNet
	`cd $(find . -name Subscription.java | xargs -I {} dirname {})`	T-Seq2Seq
	`cd $(find . -name Subscription.java | xargs -I {} dirname {})`	T-CopyNet
	`cd $(find . -name Subscription.java | xargs -I {} dirname {})`	ST-Seq2Seq
	`cd $(find . -name Subscription.java | xargs -I {} dirname {})`	ST-CopyNet
	`cd $(find . -name "Subscription.java" | xargs -I {} dirname {})`	Tellina
Add executable permission to "pretty-print"	`chmod +x pretty-print`	Human
	`chmod +x prety-printf`	C-Seq2Seq
	`chmod pretty-print pretty-print`	C-CopyNet
	`chmod +x UNK`	T-Seq2Seq
	`chmod +x UNK`	T-CopyNet
	`chmod +x UNK-UNK-UNK`	ST-Seq2Seq
	`chmod +x pretty-print`	ST-CopyNet
	`chmod [permission] "pretty-print"`	Tellina
Search for the string 'git' in all the files under current directory tree without traversing into '.git' folder and excluding files that have 'git' in their names	`find . -path ./.git -prune -o -not -name '*git*' -print |xargs grep git`	Human
	`find . -name git -prune -or -print ...` *incomplete*	C-Seq2Seq
	`find . -name .git -prune -or -name "*.git" -print ...` *incomplete*	C-CopyNet
	`find . -not -name "*.*" -not -path UNK -not -path UNK ...` *incomplete*	T-Seq2Seq
	`find . -not \( -path UNK -prune \) -or \( -type f -and ! -name UNK \) -print0 | xargs -0 -I {} grep ...` *incomplete*	T-CopyNet
	`find . -name '*.git' -not -path '.git' | xargs -I {} grep 'git' {}`	ST-Seq2Seq
	`find . -not -name ".git" -not -path "*.git*" -not -name "*git*" | xargs -I {} grep git {}`	ST-CopyNet
	`find "git" -not -path ".git" -not -name "*" | grep "git"`	Tellina
Rename "file.txt" in directories "v_1", "v_2", and "v_3" each to "v_1.txt", "v_2.txt", and "v_3.txt" respectively and print the conversion	`ls -d v_1,2,3 | xargs -i mv -v {}/file.txt {}/{}.txt`	Human
	`mv file.txt v_1.txt`	C-Seq2Seq
	`mv file.txt v_3.txt`	C-CopyNet
	`mv UNK UNK`	T-Seq2Seq
	`mv UNK UNK`	T-CopyNet
	`diff current_1 {}`	ST-Seq2Seq
	`ssh -i v_1.txt v_3.txt`	ST-CopyNet
	`no output`	Tellina

Table 11: Example predictions of the baseline approaches. The prediction errors are underlined.

Complex Task We found several cases where the NL description specifies a complex task and would be better broken into separate sentences (Table 10). When the task gets complicated, the NL description gets verbose. As noted in previous work (Bahdanau et al., 2014), the performance of RNNs decreases for longer sequences. Giving high-quality NL description for complex tasks are also more difficult for the users in practice — multi-turn interaction is probably necessary for these cases.

Other Classes For the rest of the error cases, we observed that the model failed to translate the specifications in `()`, long descriptions of regular expressions and intelligible/non-grammatical NL descriptions (Table 10). There are also errors propogated from the pre-processing tools such as the NL tokenizer. In addition, the stage-wise system Tellina made a significant number of mistakes specific to its non-end-to-end modeling approach, e.g. the limited coverage of its set of manually defined heuristic rules.

Based on the error analysis, we recommend future work to build shallow command structures in the decoder instead of synthesizing the entire output in sequential manner, e.g. using separate RNNs for template translation and argument filling. The training data sparsity can possibly be alleviated by semi-supervised learning using unlabeled Bash commands or external resources such as the Linux man pages.

7. Comparison to Existing Datasets

This section compares NL2Bash to other commonly-used semantic parsing and NL-to-code datasets.[11] We compare the

[11] We focus on generating utility commands/scripts from natural language and omitted the datasets in the domain of programming challenges (Polosukhin and Skidanov, 2018) and code base model-

Dataset	PL	# pairs	# words	# tokens	Avg. # w. in nl	Avg. # t. in code	NL collection	Code collection	Semantic alignment	Introduced by
IFTTT	DSL	86,960	–	–	7.0	21.8	scraped	scraped	Noisy	(Quirk et al., 2015)
C#2NL*	C#	66,015	24,857	91,156	12	38				(Iyer et al., 2016)
SQL2NL*	SQL	32,337	10,086	1,287	9	46				
RegexLib	Regex	3,619	13,491	179✱	36.4	58.8✱			Good✠	(Zhong et al., 2018)
HeartStone	Python	665	–	–	7	352✱	game card description	game card source code		(Ling et al., 2016)
MTG	Java	13,297	–	–	21	1,080✱				
StaQC	Python	147,546	17,635	137,123	9	86	extracted using ML	extracted using ML	Noisy	(Yao et al., 2018)
	SQL	119,519	9,920	21,413	9	60				
NL2RX	Regex	10,000	560	45✱†	10.6	26✱	synthesized & paraphrased	synthesized	Very Good	(Locascio et al., 2016)
WikiSQL	SQL	80,654	–	–	–	–				(Zhong et al., 2017)
NLMAPS	DSL	2,380	1,014	–	10.9	16.0	synthesized given code	expert written	Very Good	(Haas and Riezler, 2016)
Jobs640★	DSL	640	391	58†	9.8	22.9	user written	expert written given NL		(Tang and Mooney, 2001)
GEO880	DSL	880	284	60†	7.6	19.1				(Zelle and Mooney, 1996)
Freebase917	DSL	917	–	–	–	–				(Cai and Yates, 2013)
ATIS★	DSL	5,410	936	176†	11.1	28.1				(Dahl et al., 1994)
WebQSP	DSL	4,737	–	–	–	–	search log			(Yih et al., 2016)
NL2RX-KB13	Regex	824	715	85✱†	7.1	19.0✱	turker written			(Kushman and Barzilay, 2013)
Django☆	Python	18,805	–	–	14.3	–	expert written given code	scraped		(Oda et al., 2015)
NL2Bash	Bash	9,305	7,790	6,234	11.7	7.7				Ours

Table 12: Comparison of datasets for translation of natural language to (short) code snippets. *: Both C#2NL and SQL2NL were originally collected to train systems that explain code in natural language. ✱: The code length is counted by characters instead of by tokens. †: When calculating # tokens for these datasets, the open-vocabulary constants were replaced with positional placeholders. ✠: For these datasets, the NL-code pairs in their original data sources were not compiled for the purpose of semantic parsing. ★: Both Jobs640 and ATIS consist of mixed manually-generated and automatically-generated NL-code pairs. ☆ The Django dataset consists of pseudo-code/code pairs.

datasets with respect to: (1) the programming language used, (2) size, (3) shallow quantifiers of difficulty (i.e. # unique NL words, # unique program tokens, average length of text and average length of code) and (4) collection methodology. Table 12 summarizes the comparison. We directly quoted the published dataset statistics we have found, and computed the statistics of other released datasets to our best effort.

Programming Languages Most of the datasets were constructed for domain-specific languages (DSLs). Some of the recently proposed datasets use Java, Python, C#, and Bash, which are Turing-complete programming languages. This shows the beginning of an effort to apply natural language based code synthesis to more general PLs.

Collection Methodology Table 12 sorts the datasets by increasing amount of manual effort spent on the data collection. NL2Bash is by far the largest dataset constructed using practical code snippets and expert-written natural language. In addition, it is significantly more diverse (7,790 unique words and 6,234 unique command tokens) compared to other manually constructed datasets.

The approaches of automatically scraping/extracting parallel natural language and code have been adopted more recently. A major resource of such parallel data are question-answering forums (StackOverflow: https://stackoverflow.com/) and cheatsheet websites (IFTTT: https://ifttt.com/ and RegexLib: http://www.regexlib.com/). Users post code snippets together with natural language questions or descriptions in these venues. The problem with these data is that they are loosely aligned and cannot be directly used for training. Extracting good alignments from them is very challenging (Quirk et al., 2015; Iyer et al., 2016; Yao et al., 2018). That being said, these datasets significantly surpasses the manually gathered ones in terms of size and diversity, hence demonstrating significant potential for future work.

Alternatively, Locascio et al. (2016) and Zhong et al. (2017a) proposed synthesizing parallel natural language and code using a synchronous grammar. They also hired Amazon Mechanical Turkers to paraphrase the synthesized natural language sentences in order to increase their naturalness and diversity. While the synthesized domain may be less diverse compared to naturally existed ones, they served as an excellent resource for data augmentation or zero-shot learning. The downside is that developing synchronous grammars for domains other than simple DSLs is challenging, and other data collection methods are still necessary for them.

The different data collection methods are complimentary and we expect to see more future work mixing different strategies.

8. Conclusions

We studied the problem of mapping English sentences to Bash commands (NL2Bash), by introducing a large new dataset and baseline methods. NL2Bash is by far the largest NL-to-code dataset constructed using practical code snippets and expert-written natural language. Experiments demonstrated competitive performance of existing models as well as significant room for future work on this challenging semantic parsing problem.

ing (Nie et al., 2018).

9. 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. We thank Zexuan Zhong for providing us the statistics of the RegexLib dataset. We thank Kenton Lee, Luheng He, Omer Levy, and the anonymous reviewers for their constructive feedbacks on the paper draft. We thank the UW NLP/PLSE groups for helpful conversations on the work.

10. Bibliographical References

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Cai, Q. and Yates, A. (2013). Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 423–433. The Association for Computer 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.

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*.

Guu, K., Pasupat, P., Liu, E. Z., and Liang, P. (2017). From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1051–1062.

Haas, C. and Riezler, S. (2016). A corpus and semantic parser for multilingual natural language querying of openstreetmap. In Kevin Knight, et al., editors, *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 740–750. The Association for Computational Linguistics.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Volume 1: Long Papers*, pages 2073–2083, Berlin, Germany.

Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., and Zettlemoyer, L. (2017). Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 963–973.

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.

Kushman, N. and Barzilay, R. (2013). Using semantic unification to generate regular expressions from natural language. In Lucy Vanderwende, et al., editors, *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013*, pages 826–836, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA. The Association for Computational Linguistics.

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*.

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 *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP, November 1-4, 2016*, pages 1918–1923, Austin, Texas, USA.

Long, R., Pasupat, P., and Liang, P. (2016). Simpler context-dependent logical forms via model projections. 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*.

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

Nie, P., Li, J. J., Khurshid, S., Mooney, R., and Gligoric, M. (2018). Natural language processing and program analysis for supporting todo comments as software evolves. In *Proceedings of the AAAI Workshop of Statistical Modeling of Natural Software Corpora*.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. (2015). Learning to generate pseudo-code from source code using statistical machine translation (T). In Myra B. Cohen, et al., editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 574–584. IEEE Computer Society.

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.

Polosukhin, I. and Skidanov, A. (2018). Neural Program Search: Solving Programming Tasks from Description and Examples. *ArXiv e-prints*, February.

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.

Tang, L. R. and Mooney, R. J. (2001). Using multiple clause constructors in inductive logic programming for semantic parsing. In Luc De Raedt et al., editors, *Machine Learning: EMCL 2001, 12th European Conference on Machine Learning, Freiburg, Germany, September 5-7, 2001, Proceedings*, volume 2167 of *Lecture Notes in Computer Science*, pages 466–477. Springer.

Wilensky, R., Arens, Y., and Chin, D. (1984). Talking to unix in english: An overview of uc. *Commun. ACM*, 27(6):574–593, June.

Wilensky, R., Chin, D. N., Luria, M., Martin, J., Mayfield, J., and Wu, D. (1988). The berkeley unix consultant project. *Comput. Linguist.*, 14(4):35–84, December.

Yao, Z., Weld, D., Chen, W.-P., and Sun, H. (2018). Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 27th International Conference on World Wide Web, WWW 2018, Lyon, France, April 23 - 27, 2018*.

Yih, W., Richardson, M., Meek, C., Chang, M., and Suh, J. (2016). The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 2: Short Papers*.

Yin, P. and Neubig, G. (2017). A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450.

Zelle, J. M. and Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 1050–1055. AAAI Press.

Zettlemoyer, L. S. and Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, pages 658–666, Arlington, Virginia, United States. AUAI Press.

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

Zhong, Z., Guo, J., Yang, W., Xie, T., Lou, J.-G., Liu, T., and Zhang, D. (2018). Generating regular expressions from natural language specifications: Are we there yet? In *Proceedings of the AAAI Workshop of Statistical Modeling of Natural Software Corpora*.

Appendices

A Additional Data Statistics

A1. Distribution of Less Frequent Utilities

Figure 4 illustrates the frequencies of the 52 least frequent bash utilities in our dataset. Among them, the most frequent utility `dig` appeared only 38 times in the dataset. 7 utilities appeared 5 times or less. We discuss in the next session that many of such low frequent utilities cannot be properly learned at this stage, since the limited number of training examples we have cannot cover all of their usages, or even a reasonably representative subset.

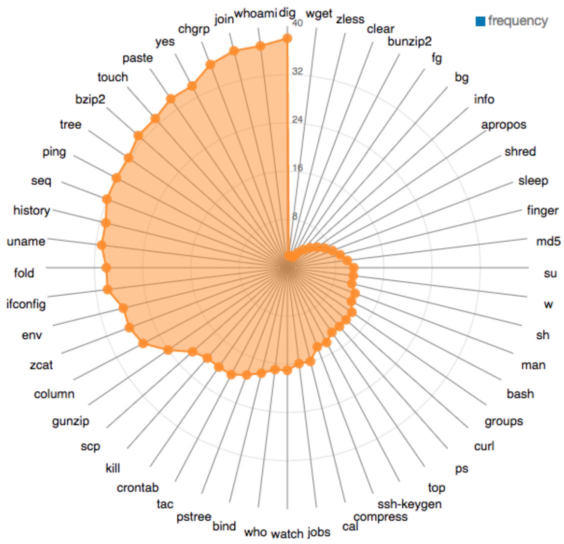


Figure 4: Frequency radar chart of the 52 least frequent bash utilities in the datasets.

A2. Flag Coverage

Table 13 shows the total number of flags (both long and short) a utility has and the number of flags of the utility that appeared in the training set. We show the statistics for the 10 most and least frequent utilities in the corpus. We estimate the total number of flags a utility has by the number of flags we manually extracted from its GNU man page. The estimation is a lower bound as we might miss certain flags due to man page version mismatch and human errors.

Noticed that for most of the utilities, only less than half of their flags appear in the train set. One reason contributed to the small coverage is that most command flags has a full-word replacement for readability (e.g. the readable replacement for `-t` of `cp` is `--target-directory`), yet most Bash commands written in practice uses the short flags. We could solve this type of coverage problem by normalizing the commands to contain only the short flags. (Later we can use deterministic rules to show the readable version to the user.) Nevertheless, for many utilities a subset of their flags are still missing from the corpus. Conducting zero-shot learning for those missing flags is an interesting future work.

Utility	# flags	# flags in train set
find	103	68
xargs	32	15
grep	82	42
rm	17	7
echo	5	2
sort	50	19
chmod	14	4
wc	13	6
cat	19	4
sleep	2	0
shred	17	4
apropos	30	0
info	34	2
bg	0	0
fg	0	0
wget	171	2
zless	0	0
bunzip2	14	0
clear	0	0

Table 13: Training set flag coverage. The upper-half of the table shows the 10 most frequent utilities in the corpus. The lower-half of the table shows the 10 least frequent utilities in the corpus.

B Data Quality

We asked two freelancers to evaluate 100 text-command pairs sampled from our train set. The freelancers did not author the sampled set of pairs themselves. We asked the freelancers to judge the correctness of each pair. We also asked the freelancers to judge if the natural language description is clear enough for them to understand the descriptor's goal. We then manually examined the judgments made by the two freelancers and summarize the findings below.

The freelancers identified errors in 15 of the sampled training pairs, which results in approximately 85% annotation accuracy of the training data. 3 of the errors are caused by the fact that some utilities (e.g. `rm`, `cp`, `gunzip`) handle directories differently from regular files, but the natural language description failed to clearly specify if the target objects include directories or not. 4 cases were typos made by our annotators when copying the constant values in a command to their descriptions. Being able to automatically detect constant mismatch may reduce the number of such errors. (Automatic mismatch detection can be directly added to the annotation interface.) The rest of the 8 cases were caused by the annotators mis-interpreted/omitted the function of certain flags/reserved tokens or failed to spot syntactic errors in the command (listed in Table 14). For many of these cases, the Bash commands are only of medium length — this shows that accurately describing all the information in a Bash command is still an error-prone task for Bash programmers. Moreover, some annotation mistakes are more thought-provoking as the operations in those examples might be difficult/unnatural for the users to describe at test time. In these cases we should solicit the necessary information from the users through alternative ways, e.g. asking multi-choice questions for specific options or asking the user for examples.

Only 1 description was marked as "unclear" by one of the freelancers. The other freelancer still judged it as "clear".

Find all executables under /path directory

```
find /path -perm /ugo+x
```

"Executables generaly means executable files, thus needs `-type f`. Also, `/ugo+x` should be `-ugo+x`. The current command lists all the directories too as directories generally have execute permission at least for the owner (`/ugo+x` allows that, while `-ugo+x` would require execute permission for all)."

*Search the current directory tree for all regular non-hidden files except *.o*

```
find ./ -type f -name "*" -not -name "*.o"
```

"Criteria not met: non-hidden, requires something like -not -name `'.*'`."

Display all the text files from the current folder and skip searching in skipdir1 and skipdir2 folders

```
find . \( -name skipdir1 -prune , -name skipdir2 -prune -o -name "*.txt" \) -print
```

"Result includes `skipdir2` (this directory name only), the `-o` can be replaced with comma , to solve this."

Find all the files that have been modified in the last 2 days ... missing `-daystart` description

```
find . -type f -daystart -mtime -2
```

"`daystart` is not specified in description."

Find all the files that have been modified since the last time we checked

```
find /etc -newer /var/log/backup.timestamp -print
```

"'Since the last time we checked', the backup file needs to be updated after the command completes to make this possible."

Search for all the .o files in the current directory which have permisssions 664 and print them.

```
find . -name *.o -perm 664 -print
```

"Non-syntactical command. Should be `.o` or `"*.o"`."

Search for text files in the directory "/home/user1" and copy them to the directory /home/backup

```
find /home/user1 -name '*.txt' | xargs cp -av --target-directory=/home/backup/ --parents
```

"`--parents` not specified in description, it creates all the parent dirs of the files inside target dir, e.g, a file named `a.txt` would be copied to `/home/backup/home/user1/a.txt`."

Search for the regulars file starting with HSTD ... missing case insensitive description which have been modified yesterday from day start and copy them to /path/tonew/dir

```
find . -type f -iname 'HSTD*' -daystart -mtime 1 -exec cp {}/path/to new/dir/ \;
```

"Case insensitive not specified but `-iname` used, extra spaces in `/path/to new/dir/`."

Table 14: Training examples whose NL description has errors (underlined). The error explanation is written by the freelancer.

Similar trend were observed during the manual evaluation — the freelancers have little problem understanding each other's descriptions.
It is worth noting that while we found 15 wrong pairs out of 100, for 13 of them the annotator only misinterpreted one of the command tokens. Hence the overall performance of the annotators is high, especially given the large domain size.

C Automatic Evaluation Results

We report two types of fuzzy evaluation metrics automatically computed over full dev set in table 15. We define TM as the maximum percentage of close-vocabulary token (utilities, flags and reserved tokens) overlap between a predicted command and the reference commands. (TM is a command structure accuracy measurement.) TM^k is the maximum TM score achieved by the top-k candidates generated by a system. We use BLEU as an approximate measurement for full command accuracy. BLEU^k is the maximum BLEU score achieved by the top-k candidates generated by a system. We set the BLEU score weights to be (0.25, 0.25, 0.25, 0.25).

Model		BLEU^1	BLEU^3	TM^1	TM^3
Seq2Seq	Char	49.1	56.7	0.57	0.64
	Token	36.1	43.9	0.65	**0.75**
	Sub-token	46	52	0.65	0.71
CopyNet	Char	49.1	56.8	0.54	0.61
	Token	44.9	54.2	0.65	0.74
	Sub-token	**55.3**	**61.8**	0.64	0.71
Tellina		46	52	0.61	0.70

Table 15: Automatically measured performance of the baseline systems on the full dev set.

First, we observed from table 15 that while the automatic evaluation metrics agrees with the manual ones (Table 8) on the system with the highest full command accuracy and the system with the highest command structure accuracy, they do not agree with the manual evaluation in all cases (e.g. character-based models have the second-best BLEU score). Second, the TM score is not discriminative enough – several systems scored similarly on this metrics.