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\textsuperscript{\small 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).

\textsuperscript{1} Work done at the University of Washington.

\textsuperscript{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.
**Table 1: Example natural language descriptions and the corresponding shell commands from NL2Bash.**

<table>
<thead>
<tr>
<th>Natural Language</th>
<th>Bash Command(s)</th>
</tr>
</thead>
</table>
| 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     | du -a . | sort -rh | head -n5 
find . -type f | sort -nk 5,5 | tail -5 |
| 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 2: In-scope and out-of-scope syntax for the Bash commands in our dataset.

<table>
<thead>
<tr>
<th>In-scope</th>
<th>Out-of-scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Single command</td>
<td>1. I/O redirection &lt;,&lt;&lt;</td>
</tr>
<tr>
<td>2. Logical connectives: &amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>3. Nested commands:</td>
<td>3. Compound statements:</td>
</tr>
<tr>
<td>- pipeline</td>
<td>- command substitution $()</td>
</tr>
<tr>
<td>- process substitution ()</td>
<td>4. Non-bash program strings nested with language interpreters such as awk, sed, python, java</td>
</tr>
</tbody>
</table>

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](https://github.com/TellinaTool/nl2bash/tree/master/data).

We collected 12,609 text–command pairs in total (\(^3\)). After filtering, 9,305 pairs remained (\(^3\)). 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 (\(^3\)).

3.1. Data Collection

We hired 10 Upwork\(^3\) 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 interface 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\(^3\).

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](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](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 (\(^3\)) show that even this task is challenging.

\(^3\)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.
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.

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).

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.

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.

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.

**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 lower-casing, 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
description or a command it has seen in the train set, which allows us to evaluate the model’s ability to generalize.

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
<th>Dev</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td># pairs</td>
<td>8,090</td>
<td>609</td>
<td>606</td>
</tr>
<tr>
<td># unique nls</td>
<td>7,340</td>
<td>549</td>
<td>547</td>
</tr>
</tbody>
</table>

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 (Acc\(_T^k\)) and top-\(k\) command template accuracy (Acc\(_F^k\)). We define Acc\(_F^k\) to be the percentage of test instances for which a correct full command is ranked \(k\) or above in the model output. We define Acc\(_T^k\) 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 (\(\alpha_T\)) and full-command judgement (\(\alpha_F\)).

<table>
<thead>
<tr>
<th>Pair 1</th>
<th>Pair 2</th>
<th>Pair 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha_F)</td>
<td>0.89</td>
<td>0.81</td>
</tr>
<tr>
<td>(\alpha_T)</td>
<td>0.83</td>
<td>0.82</td>
</tr>
</tbody>
</table>

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 automata (DFAs). Others Kwiatakowski 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 `[	exttt{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 programming 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 reformattting 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 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 architecture is the same as the Seq2Seq model. The Tellina model which does template-level translation and argument filling/reformatting in a stage-wise manner yields the second-best full command 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.

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.

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc_T</th>
<th>Acc_F</th>
<th>Acc_T*</th>
<th>Acc_F*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char</td>
<td>0.24</td>
<td>0.27</td>
<td>0.35</td>
<td>0.38</td>
</tr>
<tr>
<td>Token</td>
<td>0.10</td>
<td>0.12</td>
<td>0.53</td>
<td>0.59</td>
</tr>
<tr>
<td>Sub-token</td>
<td>0.19</td>
<td>0.27</td>
<td>0.41</td>
<td>0.53</td>
</tr>
<tr>
<td>CopyNet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char</td>
<td>0.25</td>
<td>0.31</td>
<td>0.34</td>
<td>0.41</td>
</tr>
<tr>
<td>Token</td>
<td>0.21</td>
<td>0.34</td>
<td>0.47</td>
<td>0.61</td>
</tr>
<tr>
<td>Sub-token</td>
<td>0.31</td>
<td>0.40</td>
<td>0.44</td>
<td>0.53</td>
</tr>
<tr>
<td>Tellina</td>
<td>0.29</td>
<td>0.32</td>
<td>0.51</td>
<td>0.58</td>
</tr>
</tbody>
</table>

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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

<table>
<thead>
<tr>
<th>Model</th>
<th>(\text{Acc}_{\text{C}})</th>
<th>(\text{Acc}_{\text{S}})</th>
<th>(\text{Acc}_{\text{T}})</th>
<th>(\text{Acc}_{\text{F}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST-CopyNet</td>
<td>0.36</td>
<td>0.45</td>
<td>0.49</td>
<td>0.61</td>
</tr>
<tr>
<td>Tellina</td>
<td>0.27</td>
<td>0.32</td>
<td>0.53</td>
<td>0.62</td>
</tr>
</tbody>
</table>

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, 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 further 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. 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.

**Constant Enumeration**  
Answer “n” to any prompts in the interactive recursive removal of “dir1”, “dir2”, and “dir3”.

**Complex task**  
Recursively finds all files in a current folder excluding already compressed files and compresses them with level 9.

**Intelligible/Non-grammatical description**  
Find all regular files in the current directory tree and print a command to move them to the current directory.

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.
<table>
<thead>
<tr>
<th>Natural Language</th>
<th>Model Output</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete all empty directories in minimum 2 directories down the root directory</td>
<td>find root -mindepth 2 -type d -empty -delete</td>
<td>Human</td>
</tr>
<tr>
<td>Find all Subscription.java files/directories under current directory and enter into the parent directory of the first one found</td>
<td>find . -name Subscription.java</td>
<td>C-Seq2Seq</td>
</tr>
<tr>
<td>Add executable permission to &quot;pretty-print&quot;</td>
<td>chmod +x pretty-print</td>
<td>C-Seq2Seq</td>
</tr>
<tr>
<td>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</td>
<td>find ... -not -name &quot;*.git&quot;</td>
<td>T-Seq2Seq</td>
</tr>
<tr>
<td>Rename &quot;file.txt&quot; in directories &quot;v1&quot;, &quot;v2&quot;, and &quot;v3&quot;, each to &quot;v1.txt&quot;, &quot;v2.txt&quot;, and &quot;v3.txt&quot; respectively and print the conversion</td>
<td>ls -d v1, v2, v3</td>
<td>ST-Seq2Seq</td>
</tr>
</tbody>
</table>

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 propagated 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. 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-
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.

### Table 12: Comparison of datasets for translation of natural language to (short) code snippets.

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

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.
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 the freelancers who worked with us to make the corpus. We thank Zexuan Zhong for providing us the statistics of the RegexLib dataset. We thank the anonymous reviewers, Kenton Lee, Luheung He, Omer Levy for constructive feedback on the paper draft, and the UW NLP/PLSE groups for helpful conversations.

10. References


Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M.,


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.](image)

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 have 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.

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

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”.

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”.

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”.

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 generally 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 . \n  ( -name skipdir1 -prune , -name skipdir2 -prune -o -name "*.txt" ) \n  -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 permissions 664 and print them.**

```
find . -name "+.o" -perm 664 -print
```

“Non-syntactical command. Should be "+.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/to new/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 $(0.25, 0.25, 0.25, 0.25)$.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU$^1$</th>
<th>BLEU$^3$</th>
<th>TM$^1$</th>
<th>TM$^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td>Char</td>
<td>49.1</td>
<td>56.7</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>Token</td>
<td>36.1</td>
<td>43.9</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>Sub-token</td>
<td>46.0</td>
<td>52.0</td>
<td>0.65</td>
</tr>
<tr>
<td>CopyNet</td>
<td>Char</td>
<td>49.1</td>
<td>56.8</td>
<td>0.54</td>
</tr>
<tr>
<td></td>
<td>Token</td>
<td>44.9</td>
<td>54.2</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>Sub-token</td>
<td><strong>55.3</strong></td>
<td><strong>61.8</strong></td>
<td>0.64</td>
</tr>
<tr>
<td>Tellina</td>
<td>46.0</td>
<td>52.0</td>
<td>0.61</td>
<td>0.70</td>
</tr>
</tbody>
</table>

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.