Program Synthesis from Natural Language Using Recurrent Neural Networks

Xi Victoria Lin
UW CSE
Seattle, WA, USA
xilin@cs.washington.edu

Chenglong Wang
UW CSE
Seattle, WA, USA
clwang@cs.washington.edu

Deric Pang
UW CSE
Seattle, WA, USA
dericp@cs.washington.edu

Kevin Vu
UW CSE
Seattle, WA, USA
kevin.m.vu@gmail.com

Luke Zettlemoyer
UW CSE
Seattle, WA, USA
lsz@cs.washington.edu

Michael D. Ernst
UW CSE
Seattle, WA, USA
mernst@cs.washington.edu

ABSTRACT

Oftentimes, a programmer may have difficulty implementing a desired operation. Even when the programmer can describe the goal in English, it can be difficult to translate into code. Existing resources, such as question-and-answer websites, tabulate specific operations that someone has wanted to perform in the past, but they are not effective in generalizing to new tasks, to compound tasks that require combining previous questions, or sometimes even to variations of listed tasks.

Our goal is to make programming easier and more productive by letting programmers use their own words and concepts to express the intended operation, rather than forcing them to accommodate the machine by memorizing its grammar. We have built a system that lets a programmer describe a desired operation in natural language, then automatically translates it to a programming language for review and approval by the programmer. Our system, Tellina, does the translation using recurrent neural networks (RNNs), a state-of-the-art natural language processing technique that we augmented with slot (argument) filling and other enhancements.

We evaluated Tellina in the context of shell scripting. We trained Tellina's RNNs on textual descriptions of file system operations and bash one-liners, scraped from the web. Although recovering completely correct commands is challenging, Tellina achieves top-3 accuracy of 80% for producing the correct command structure. In a controlled study, programmers who had access to Tellina outperformed those who did not, even when Tellina's predictions were not completely correct, to a statistically significant degree.

1 INTRODUCTION

Even if a competent programmer knows what she wants to do and can describe it in English, it can still be difficult to write code to achieve the goal. Programmers increasingly work across libraries and programming languages, creating more complex systems than ever before, and cannot memorize every detail of all the systems that must be used.

An increasingly common practice is to seek help from websites such as Stack Overflow. Tutorial and question-answering websites are powerful resources with reams of specific examples of code snippets and explanations of their behavior. When a programmer's exact question has been asked before, the community-vetted answer is invariably useful. However, finding the correct answer may require the use of keywords the programmer does not know. Furthermore, sometimes outdated or incorrect answers persist even if the correct answer also appears. Finally, a programmer who wishes to do something new may waste time searching for it, and then must synthesize it on her own. Even a variation of task on the website may be difficult to find because of different constants and keywords.

Question 1. I have a bunch of ".zip" files in several directories "dir1/dir2", "dir3", "dir4/dir5". How would I move them all to a common base folder? (http://unix.stackexchange.com/questions/67503)

Solution: `find dir*/ -type f -name "*.zip" -exec mv {} "basedir" \;`

Question 2. I have one folder for log with 7 sub-folders. I want to delete all the files older than 15 days in all folders including sub-folders without touching folder structure. (http://unix.stackexchange.com/questions/155184)

Solution: `find . -type f -mtime +15 | xargs rm -f`

Figure 1: Linux command-line questions posted on the Unix Stack Exchange forum. The answer to each is a bash one-liner: a command that can be typed at the bash command line.

Despite their limitations, tutorial and question-answering websites are valuable sources of information that tools should exploit in order to help developers solve programming tasks. We used these websites to gather training data for a novel natural language (NL) to code translation tool that allows the user to express their intent in English and automatically translates it into executable programs. Such synthesis methods have many advantages. The learned models can often generalize to new NL descriptions or synthesize novel code, and the programmer does not need to search through large websites to complete the task. However, they are also inherently error-prone. For the near future, all NL-driven synthesis methods will make mistakes and care must be taken to present their output to users in a way they can easily use, modify, or take inspiration from, without requiring that they blindly accept a single system output.

This paper presents a complete machine learning approach for natural language (NL) to code translation, along with a detailed user study that demonstrates the effectiveness of the overall approach

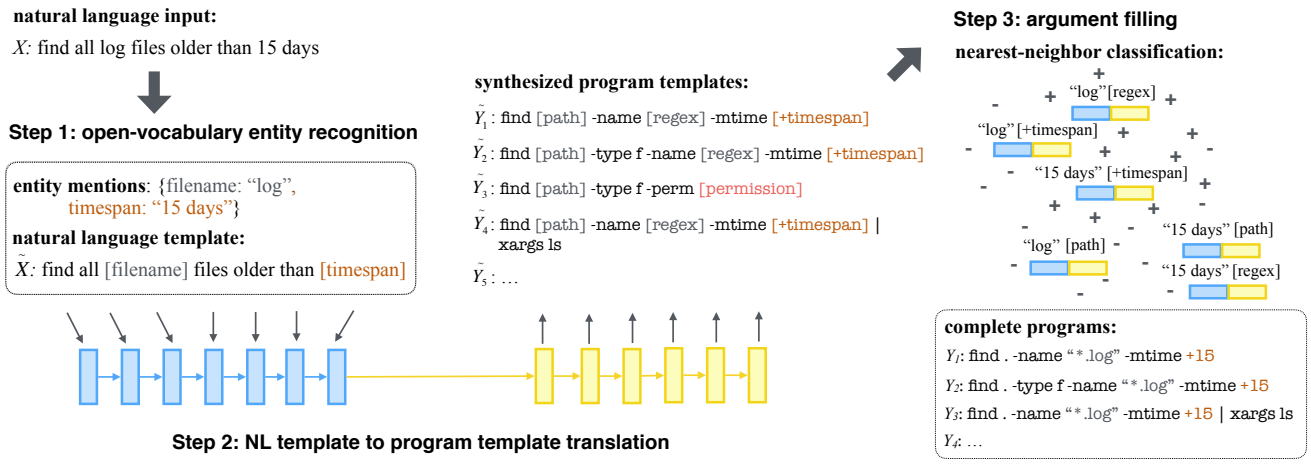

Figure 2: Tellina's three-step architecture for program synthesis from natural language. Step 2 shows the RNN encoder-decoder model used for translating natural language templates to program templates. The blue rectangles and the yellow rectangles represent the RNN cells for the encoder and decoder respectively. Step 3 shows the nearest neighbor classification method used for evaluating the compatibility of an entity mention and a program slot. Each data point is an ⟨entity, slot⟩ pair represented as the concatenation of the hidden state vectors of their corresponding RNN cells. The compatible entity-slot pairs, ⟨"log", [regex]⟩ and ⟨"15 days", "[+timespan]"⟩, are closer to the positive training examples ("+"s), while the non-compatible pairs are closer to the negative training examples ("-"s).

even when the predicted code is not completely correct. Our system, Tellina, is instantiated for the problem of bash scripting, where a user must perform complex file system operations. Figure 1 shows two example tasks that users posed in this domain. Such problems are worthy of study because programmers must perform them often but the individual commands are complex enough to make it difficult to remember the exact details. For example, the Linux `find` utility, which searches the file system, supports more than 70 flags[1] to control the search criteria and perform operations on the returned files; it is also commonly combined with other commands like `mv` or `grep` for more complex tasks. Man pages describe these commands and flags, but can be hard to discover and understand. Community sites exist[2] and cover a wide variety of commands, but can be hard to search and will not cover all users' intents. None of these problems are specific to the command line or shell scripting; similar problems arise in other libraries and languages.

Our learning approach incorporates recent advances in neural machine translation [3, 36, 39] within a novel learning architecture that reduces data sparsity by abstracting over constants in the input specifications (e.g., names of files, dates, etc.). A key challenge is to learn to align the correct input strings with parameter values for the output commands, which we show is possible with a k-nearest neighbor classifier and greedy matching algorithms. We trained the model on a newly gathered corpus of over 5000 natural language and bash command pairs. Our experiments show that the proposed model is competitive for this task: evaluated on unseen natural language descriptions, it achieves 80% top-3 accuracy for determining command structures and 36% top-3 accuracy for full commands.

We also present a user study that measures the performance of programmers using this learned model instantiated in an assistant tool, Tellina. Compared to the current state of the art (man pages and online resources such as question-answering forums and web search tools), programmers benefitted from using Tellina to a statistically significant degree. For example, despite the fact that Tellina does not always produce fully correct command suggestions, it shortened the working time by 21.7% for programmers on bash file system tasks.

In summary, this paper makes the following contributions:

- We propose a novel deep-learning approach for synthesizing programs from natural language. Our approach combines state-of-the-art recurrent neural networks with a learning approach for inserting constants into the generated programs.
- We instantiated the approach for a challenging domain: bash commands containing 17 file system utilities, more than 200 flags, 9 types of open-vocabulary constants, and nested command structures such as pipelines, command substitution, and process substitution.
- In order to provide data for training and evaluation, we collected over 5,000 ⟨NL, command⟩ pairs.
- We evaluated the accuracy of our approach. Our model achieves top-3 accuracy of 80.0% for determining program structure — that is, ignoring constant values. Our model achieves top-3 accuracy of 36.0% for full commands.
- We conducted a controlled user study to determine whether a good, but not perfectly accurate, model aids end users' programming efficiency. Compared to existing programming resources such as man pages, Stack Overflow, and Google, Tellina shortened the working time by 21.7% for end-user programmers on bash file system tasks. This improvement was statistically significant (p-value < 0.01).

[1] Flags are also called command-line options. Some flags also take arguments.
[2] Including Unix Stack Exchange (http://unix.stackexchange.com/), CommandLineFu (http://www.commandlinefu.com/), and Bash One-Liners (http://www.bashoneliners.com/).

In-scope syntax structures:

- Single command
- Logical connectives: `&&`, `||`, parentheses `()`
- Nested commands: pipeline `|`, command substitution `$()`, process substitution `<()`

Out-of-scope syntax structures:

- I/O redirection <, <<
- Variable assignment =
- Parameters `$1`
- Compound statements: `if`, `for`, `while`, `until`, blocks, function definition
- Regex structure (every string is a single opaque token)
- Non-bash program strings triggered by command interpreters such as `awk`, `sed`, `python`, `java`

Figure 3: The subset of bash commands used as Tellina's domain.

- Our source code and dataset will be released upon publication to support reproducibility in software engineering research.

2 PROBLEM DEFINITION AND FORMAL OVERVIEW

Problem Definition. Following Desai et al. [6], we define programming by natural language (PBNL) as synthesizing a ranked list of programs $[Y_1, Y_2, \ldots, Y_k]$ based on the specification expressed as a single natural language sentence (denoted as X). A natural language sentence is defined as a sequence of words $X = (x_1, \ldots, x_m)$ and a program is defined a sequence of tokens $Y = (y_1, \ldots, y_n)$. This definition differs from prior work [6], which relies on a context-free grammar definition of the programming language.

Our Approach. We present a machine learning approach for PBNL, which trains the synthesizer using pairs of natural language and programs (denoted as $\langle X, Y \rangle$). Figure 1 shows two such training examples.

We train this model on a new corpus of English paired with Bash commands that we collected (§5.1), which focuses on a complex subset of the language (see fig. 3). The complete system, Tellina, permits users to input an example queries. Tellina outputs a list of proposed bash commands that may solve the user queries.

Our user studies demonstrate that Tellina significantly outperforms existing alternatives, even when there are mistakes in the set of proposed commands (§7).

3 NEURAL MACHINE TRANSLATION BACKGROUND

3.1 RNN Encoder-Decoder Model

3.1.1 Recurrent Neural Network (RNN). A recurrent neural network [14] encodes an input sequence of vectors into a single vector or expands an input vector into an output sequence of vectors. In the most general case, it serves both purposes at the same time. In our context, the sequences of vectors represent sequences of words or tokens (English sentences or bash commands).

An RNN consists of a set of cells, each consisting of three layers: input, hidden and output (fig. 4). Each token/vector in a sequence is processed by a different cell in turn. The input layer maps a token in the input sequence to an input state vector:

$$\mathbf{x}_t = I(x_t). \tag{1}$$

The hidden layer takes the input state and the previous hidden state as input, and generates the current hidden state as output:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t). \tag{2}$$

The output layer takes the current hidden state as input, and generates an output state vector:

$$\mathbf{o}_t = g(\mathbf{h}_t). \tag{3}$$

Discrete output symbols can be decoded from the output state. The standard practice is to project $\mathbf{o}_t$ into a $|\mathcal{V}|$-dimensional space, where $|\mathcal{V}|$ is the size of the output vocabulary, and apply the softmax function:

$$\mathbf{y}_t = \text{softmax}(W_o \mathbf{o}_t). \tag{4}$$

The softmax function outputs a unit vector. Therefore $\mathbf{y}_t$ can be interpreted as a probability distribution of the output vocabulary conditioned on the partial input history $x_1, \ldots, x_t$:

$$p(y_t|\mathbf{h}_t) = p(y_t|x_1, \ldots, x_t) = \mathbf{v}_i^\top \mathbf{y}_t \tag{5}$$

where $\mathbf{v}_i$ is the one-hot indicator vector for $y_t \in \mathcal{V}$, $y_t = v_i$.

In general, I is a look-up table and f, g are non-linear functions. The Tellina model defines f and g to be gated recurrent units (GRUs) [5].

3.1.2 Encoder-decoder models. As described in §3.1.1, an RNN can be used to encode an input sequence or to generate an output sequence. For most translation problems, the input sequence and output sequence are of different lengths; hence, it is difficult to use a single RNN to model both. Such problems can be solved using the combination of an encoder RNN and a decoder RNN, which is commonly referred to as encoder-decoder modeling (Seq2Seq, fig. 4).[3]

As shown in fig. 4, the encoder RNN and decoder RNN are connected in the hidden layer. The source sequence is fed into the encoder RNN. The output sequence is decoded from the decoder RNN, and the generation is biased by the final hidden state of the encoder RNN. The input symbol at step t is its output symbol at step $t-1$. Using eq. (5):

$$p(y'_t|\mathbf{h}'_t) = p(y'_t|y'_1, \ldots, y'_{t-1}, \mathbf{h}_{final}) = \mathbf{v}_i^\top \mathbf{y}'_t \tag{6}$$

where $y'_t = v_i$. Applying the chain rule to eq. (6), the decoder RNN defines a conditional probability distribution of the target sequences given the (encoded) source sequence:

$$p(y'_1, \ldots, y'_{T'}|\mathbf{h}_{final}) = \prod_{t=1}^{T'} p(y'_t|y'_1, \ldots, y'_{t-1}, \mathbf{h}_{final}). \tag{7}$$

[3] Some recent work used tree-structured RNNs in the encoder-decoder framework [7, 30]. We tried both. A Seq2Tree network did not yield significant performance improvement compared to Seq2Seq, but was dependent on a specific grammar definition. Future work can investigate more sophisticated encoder-decoder architectures.

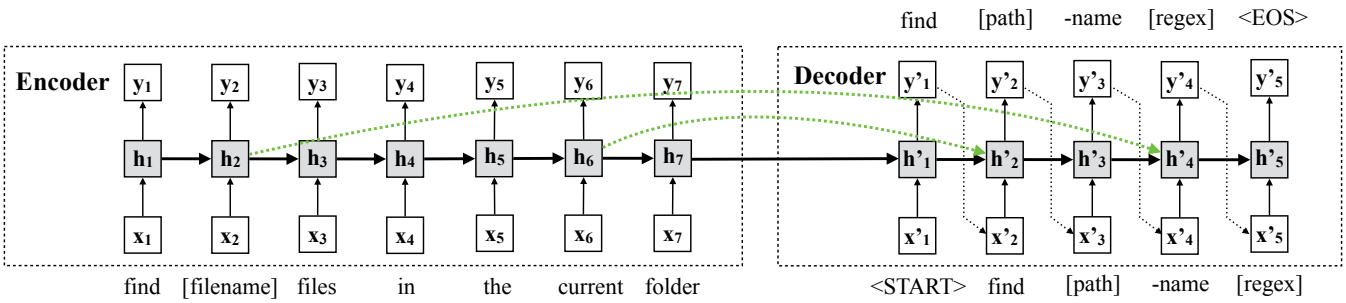

Figure 4: Configuration of the Seq2Seq neural network translation model. Each layer (input, hidden, and output from bottom to top) is labeled with the state vector it produces. The encoder reads the natural language description and passes its final hidden state to the decoder. The decoder consumes the encoder's final hidden state and generates the program starting from the special symbol <START>. For the decoder, each input symbol is the output symbol from the previous step (denoted by the dotted lines connecting the input layer at each step with the previous output layer). The green dotted lines mark the word alignments learned via the attention mechanism. While the attention mechanism computes an alignment score for each pair of encoder hidden state and decoder hidden state, the figure illustrates only the alignments with high scores.

The translation problem is hence reduced to decoding the target sequence with the maximum conditional likelihood:

$$\bar{Y'} = \underset{y'_i,\ldots,y'_{T'}}{\operatorname{argmax}} p(y'_1,\ldots,y'_{T'}|\mathbf{h}_{final}). \tag{8}$$

The optimization in eq. (8) cannot be computed efficiently since $p(y'_1,\ldots,y'_{T'}|\mathbf{h}_{final})$ does not factorize step-wise. Effective approximations include beam search [36] (which Tellina uses) or greedily picking the token with the maximum local score at each step.

3.1.3 Attention. Recent work in neural machine translation has shown that the model can benefit from making the prediction of a target output token depend directly on the encodings of the input tokens [3, 37]. For example, in fig. 4 the choice of the output token "[regex]" might be triggered by the tokens "[filename]" and "files" in the source sequence.

An *attention mechanism* [37] can be used to find and encode the relevant inputs. Specifically, it computes an alignment model α between the encoder and decoder hidden states, and a context vector $\mathbf{c}_t$ which is a sum of all encoder hidden states weighted by α:

$$\alpha_t^i = a(\mathbf{h}'_t, \mathbf{h}_i) \tag{9}$$

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_t^i \mathbf{h}_i. \tag{10}$$

The context vector $\mathbf{c}_t$ is then used as the input to the decoder output layer together with the current hidden state $\mathbf{h}'_t$

$$\mathbf{o}'_t = g(\mathbf{h}'_t, \mathbf{c}_t). \tag{11}$$

In general, α_t^i is defined to be the probability that the output at step t of the decoder is translated from the input at step i of the encoder. Following Vinyals et al. [37], we define α_t as a m-dimensional probability vector output by a one-layer feed-forward network which takes the decoder hidden state $\mathbf{h}'_t$ and the encoder hidden states $\mathbf{h}_i$ as input.

$$u_t^i = w^\top \tanh(W_1\mathbf{h}_i + W_2\mathbf{h}'_t) \tag{12}$$

$$a_t^i = \operatorname{softmax}(u_t^i) \tag{13}$$

where w, W_1 and W_2 are learnable parameters of the feed-forward network.

3.2 RNN Encoder-Decoder with Copying

A vanilla RNN encoder-decoder model only outputs tokens in the target language seen in the training set. On the other hand, it frequently happens in translation that certain segments in the input sequence are replicated in the output sequence, despite of not appearing in the training set. For example, in natural language translation, the names of people and organizations may be direct copies from the input sequence; for natural language to program translation, frequently the system needs to copy file names, string expressions, etc.

To address this problem, latest neural machine translation models have introduced mechanisms which is able to "copy" certain segments of the input sequence to its output sequence [9, 11, 23, 35]. While "copying" appropriate segments from the input seems to be a discrete operation, such models are fully differentiable and are trained in an end-to-end fashion. We explain the CopyNet model [9] which was adopted in our full model in this section.

3.2.1 Prediction with Copying and Generation. Let $\mathcal{V}$ be the output vocabulary of the decoder,$\mathcal{X}$ be the set of words appeared in the input sequence X. Denote all tokens which $\notin \mathcal{V} \cup \mathcal{X}$ with the <UNK> symbol.

CopyNet is essentially an RNN encoder-decoder augmented with copying probability in its output distribution. The encoder is the same as one in a vanilla RNN encoder-decoder. The decoder RNN defines at each time step t the probability of outputting $y'_t \in \mathcal{V} \cup \mathcal{X}$ as the sum of two probabilities

$$p(y'_t|\mathbf{h}'_t) = p_g(y'_t|\mathbf{h}'_t) + p_c(y'_t|\mathbf{h}'_t). \tag{14}$$

where p_g stands for the probability of generating y'_t from $\mathcal{V}$ and p_c stands for the probability of copying y'_t from X. The two terms are defined as follows.

$$p_g(y'_t|\mathbf{h}'_t) = \begin{cases} e^{\psi_g(y_t)}/Z, & y_t \in \mathcal{V} \\ 0, & y_t \in \mathcal{X} \cap \bar{\mathcal{V}} \\ e^{\psi_g(\texttt{<UNK>})}/Z, & y_t \notin \mathcal{V} \cup \mathcal{X} \end{cases} \quad (15)$$

$$p_c(y'_t|\mathbf{h}'_t) = \begin{cases} \sum_{j:x_j=y_t} e^{\psi_c(x_j)}/Z, & y_t \in \mathcal{X} \\ 0, & y_t \notin \mathcal{X} \end{cases} \quad (16)$$

ψ_g and ψ_c are score functions for generation-mode and copy-mode, which share the normalization function

$$Z = \sum_{v \in \mathcal{V} \cup \{\texttt{<UNK>}\}} e^{\psi_g(v)} + \sum_{j:x_j \in X} e^{\psi_c(x_j)}$$

.

Generate-mode The scoring function ψ_g is the same as one in the generic RNN encoder-decoder (§3.1.2), i.e.

$$\psi_g(y_t = v_i) = \mathbf{v}_i^\top W_o \mathbf{o}_t, \quad y_t \in \mathcal{V} \cup \{\texttt{<UNK>}\}. \quad (17)$$

Copy-mode The scoring function ψ_c is defined as[4]

$$\psi_c(y_t = x_i) = \tanh(\mathbf{h}_i^\top W_3)\mathbf{h}'_t, \quad y_t \in \mathcal{X}. \quad (18)$$

3.2.2 State Update with Copying.

4 PARTIAL-TOKEN BASED RNN ENCODER-DECODER WITH COPYING

4.1 Training and Hyperparameter Settings

The Tellina model uses a bi-directional RNN [34] encoder, which consists of a forward RNN that reads the input sequence from left to right and a backward RNN that reads the input sequence in the reversed order. The hidden states of the two RNNs are concatenated to generate the output state. The rest of the construction is the same as the base model presented above.

We trained the encoder-decoder using pairs of natural language and program templates. We use the standard sequence-to-sequence training objective [36] which maximizes the likelihood of the ground-truth program template given the natural language template. We trained the neural network (consists of all token embeddings and layer parameters) end-to-end using mini-batch gradient descent with Adam [17].

We set up the decoder RNN to be 400-dimensional, and the two RNNs in the bi-directional encoder to be 200-dimensional. We set the mini-batch size to 16, the initial learning rate to 0.0001, and the momentum hyperparameters of Adam to their default values [17]. We set the beam size to 100 for beam-search decoding. The hyperparameters were set based on the model's performance on a development dataset (§5.3). We will release our trained model and source code to support reproducible research.

[4]Notice the similarity between the copy scores and the attention scores defined in equation 12. Both scores quantify the direct correlation between an output token and tokens in the input sequence. Some recent work use a single set of scores for both purposes [11, 35]. In our experiments we found the CopyNet [9] formulation to be good enough and left the investigation of other copying variations as future work.

5 DATA

We collected 8,000 pairs of NL description and bash command from the web (§5.1), from which 5,413 pairs remained after filtering (§5.1). We split this data into train, dev, and test sets, subject to the constraint that no NL description appears in more than one dataset (§5.3). Our dataset is publicly available for use by other researchers.

5.1 Data Collection

We hired freelancers who were familiar with shell scripting through Upwork[5] to collect data. They searched for web pages that contain bash commands and recorded command-text pairs from them. Each text in a pair is a sentence that describes the command, either extracted from the webpage or described by the freelancer based on their background knowledge and web page contexts. We restricted the bash commands to be one-liners and the natural language description to be a single sentence. The source web pages include tutorials, tech blogs, question-answering forums, and course materials. The freelancers collected data through a web interface developed by us, which helps them with page searching, pair recording, and duplicate data elimation. On average, each freelancer collected 50 pairs per hour.

Filtering. After obtaining text-command pairs colecled by the freelancer, we first filter the dataset with the following rules. First, we discarded all commands that cannot be parsed by the bash parser Bashlex[6]. Second, we discarded all commands that contain out-of-scope bash operators, as shown in Figure 3. Finally, we discarded commands that contain an operator appeared less than 20 times in our dataset (e.g. `cut`, `du`, `bzip2`, etc.), since the model is unlikely to learn them from the data due to sparsity.

Cleaning. After filtering, we clean both texts and commands in the data set. For texts, we use a probabilistic spell checker[7] to correct spelling errors in the English descriptions. We also manually corrected a subset of the spelling errors that bypassed the spell checker in the English and in the bash commands.

For commands, we first remove `sudo` and shell input prompt characters such as "$" and "#" from the beginning of each command and replaced absolute command pathnames by their base names (e.g., we changed "/bin/find" to `find`). Then, we used a parser augmented from Bashlex to parse the command into an AST. The Bashlex AST handles nested command structures such as pipelines, command substitution, and process substitution. We augmented it to map each argument to the utility or utility flag it attaches to, using the command syntax defined in the Linux man pages. The augmented syntax is used to generate the bash command templates, as described in **??**.

5.2 Data Statistics

After filtering and cleaning, our dataset contains 5,413 ⟨NL, bash⟩ pairs. These commands contain 17 different bash utilities and more than 200 flags. In descending order of frequency, the utilities are `find`, `xargs`, `grep`, `egrep`, `fgrep`, `ls`, `rm`, `cp`, `mv`, `wc`, `chmod`, `chown`, `chgrp`, `sort`, `head`, `tail`, `tar`.

[5]http://www.upwork.com/
[6]https://github.com/idank/bashlex
[7]http://norvig.com/spell-correct.html

	# pairs	cmd/nl	percentage	nl/cmd	percentage
Train	4330	1.21	13%	2.13	27%
Dev	559	1.13	9%	1.39	19%
Test	524	1.19	12%	1.40	15%

Table 1: Data statistics. "cmd/nl" is the average number of bash commands per NL description, and the following column is the percentage of NL with more than one bash command translations. Similarly, "nl/cmd" is the average number of NL descriptions per bash command, and the following column is the percentage of bash command with more than one NL descriptions.

Similar to other machine translation datasets [29], in our ⟨NL, bash⟩ dataset, one natural language description may have multiple corresponding correct bash command solutions, and one bash command may be phrased in multiple different NL descriptions. In general, higher number of such multiple-to-multiple correspondences between NL descriptions and bash commands implies bigger challenges for learning and evaluation. First, we are unlikely to collect all possible translations for an NL description, and the model could wrongly penalize correct predictions which it does not recognize during training. Second, at test time, the model may predict correct translations that are different from the ground-truth, and manual evaluation has to be done to compute the correct evaluation metrics (§6.1.2). We present the statistics of such correspondences in Table 1.

5.3 Data split

We split the filtered data into train, development (dev) and test sets. We first clustered the pairs by their NL templates — a cluster contains all pairs with the identical NL template. Then, we randomly split the clusters into 80% training, 10% dev, and 10% test. This prevents the model from testing a NL template that was included in the training set, which allows us to evaluate the model's ability to generalize to new NL inputs. Table 1 shows the statistics of our data split.

Following the machine learning practice, we first trained our model on the training set and use the dev set to tune the hyperparameters (§4.1). Then, we trained our final model on the combination of both training and dev sets, with the hyperparameters achieving translation accuracy on the dev set (§6.1) obtained during parameter tuning phase.

6 MODEL EVALUATION

We report the end-to-end translation and argument filling accuracy (§6.1) for our new bash dataset, and a short discussion of qualitative results (§6.2).

6.1 Translation Accuracy

We report the end-to-end translation accuracy, both automatic and manually computed, of the Tellina model and a code retrieval baseline.

6.1.1 Baseline Model. We implement a code retrieval (CR) baseline using the *tf-idf* information retrieval (IR) technique [24]. The IR model encodes every NL description in the dataset into a bag-of-words feature vector using the *tf-idf* statistics. Given a test example, it computes the cosine-similarity between the test NL feature vector and the training NL features vectors, and returns the top-k most similar commands. We improved the IR model by first retrieving the command template using the NL template similarity, and then perform argument filling for the NL template using type-matching heuristics.[8]

Model	Acc_F^1	Acc_F^3	Acc_T^1	Acc_T^3
CR Baseline	13.0%	20.6%	54.7%	67.9%
Tellina Model	30.0%	36.0%	69.4%	80.0%

Table 2: Translation accuracies of the Tellina model and the code retrieval baseline.

6.1.2 Evaluation Methodology. We report two types of accuracy: top-k full-command accuracy (Acc_F^k) and top-k command-template accuracy (Acc_T^k). We define Acc_F^k to be the percentage of test examples[9] for which a correct full command is ranked k or above in the model output, and Acc_T^k to be the percentage of test examples for which a correct command template is ranked k or above in the model output (i.e. ignoring errors in the constants).

As described in §5.2, many test examples have more than one correct commands and our collected data may not cover them all. Therefore, we asked three freelancers from Upwork who are familiar with shell scripting to evaluate the model output manually. The freelancers independently examined the top-3 translations of both the CR baseline and the Tellina model for all test examples, and evaluate correctness of the commands at both the full-command level and the command-template level. For each command translation, we use the majority vote of the three freelancers as the final evaluation.

6.1.3 Results. Table 2 shows the translation accuracies of both the Tellina model and the CR baseline. The Tellina model beats the CR baseline by a large margin on all accuracy metrics. It achieves strong template-based accuracy, up to 80% on the top-3 metric, indicating the effectiveness of the neural encoder-decoder model. On the other hand, both models struggle to generate full commands, often making mistakes with one or more of the command arguments. Nonetheless, as we will see in §7.6, users still found these predictions useful, even if the final output needs some corrections before it can be executed.

We also evaluate the argument filling component individually by using **??** to fill in the arguments for ground truth templates. Table 3 shows the precision, recall, F1 measurement on the development set with varying k parameters. The model presents high accuracy in filling arguments to the templates (~86.7% F1 with optimal k). However, due to cascading errors from entity detection and template selection, the overall command correctness ratio is much lower than template correctness ratio, as we will discuss below.

[8] We found the retrieval results based on full NL description to be significantly worse, since the constants are likely to receive high idf weights while not being representative of the command semantics.

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

k	Precision	Recall	F1
1	82.9	87.0	84.9
5	84.6	89.0	86.7
10	82.1	86.2	84.1
100	79.8	84.0	81.9
200	77.2	81.2	79.1

Table 3: Development set performance of the argument filling component for differing k nearest neighbor values.

6.2 Error Analysis

While the template correctness and argument filling accuracies are high (Table 2), we observed that complete command accuracy is much lower overall. To better understand this phenomena, we sampled 50 synthesized commands whose template is correct but full command is incorrect.

Among these 50 incorrect commands, 41 of them are caused by Tellina's failure to recognize the argument from the NL description, 5 are caused by wrong command alignment, 2 caused by the fact that the NL description does not provide a concrete argument, and the last 2 are caused by predicting incorrect templates (which the freelancers failed to catch). The vast majority of the NL entity recognition failures involve missing idioms in the task description. For example, our algorithm is unable to extract the directory argument '/' from the idioms "root directory" or "full file system", and so is the case for extracting permission code "100" from idiom "read permission".

However, while this type of error harms the full-command translation accuracy of Tellina model, it does not significantly harm the tool usability: we observed in our user study (§7) that many users can easily formulate arguments that our algorithm failed to recognize based on their knowledge of the file system; as a result, these errors can be easily fixed given correct template and alignment.

7 USER STUDY

We conducted a user study to determine whether Tellina helps programmers complete file system tasks using bash.

7.1 Experiment Design

We measured each participant's performance under the following two treatment conditions.

- *Control treatment:* The participant may use any local resource (such as man pages and experimentation on the command line) and any Internet resource (such as tutorials, question-and-answer websites, and web search). This emulates how a programmer would normally solve a file system task.
- *Experimental treatment:* The participant may use any of the above resources, and also Tellina.

We adopted a counterbalanced factorial design in which each participant performs two sets of tasks, one with each treatment. This design prevents confounding due to order effects by randomly assigning the participants into four groups, which correspond to all four taskset and treatment combinations.

We recruited 39 students in the computer science major to participate in the experiment (24 graduate students, 15 undergraduates). None were familiar with Tellina. All of them were familiar with bash. We accepted only graduate students who self-reported to be bash users, and we accepted only undergraduates who had completed or were enrolled in our department's Linux tools course.

We excluded data from 4 of the participants, because 3 of them forgot to switch treatment conditions between the tasksets and 1 of them did not complete the study.

7.2 Tasks

Each taskset is made up of 8 tasks. Each task consists of an English description of a file system operation, and a file system in a pre-defined initial state. The desired outcome is either a list of files (possibly with additional attributes such as modification time or number of lines) or a change in the file system, such as deleted/added/modified files. The user's goal is to write a bash command that performs the desired operation without causing extra file system changes. All tasks have outcomes that are automatically verifiable.

The participant may attempt a task multiple times. The participant may reset the file system to its original state in order to start over from a fresh slate. Each task has a 10-minute timeout. If the participant gives up on a task, we count the participant as having spent 10 minutes. In addition, each taskset has a time limit of 40 minutes.[10]

7.2.1 Selection and filtering. The experiment uses real tasks selected from four websites offering programming help: Stack Overflow (http://stackoverflow.com/), Super User (http://superuser.com/), commandlinefu.com (http://www.commandlinefu.com/), and Bash One-Liners (http://www.bashoneliners.com/).

To obtain candidate tasks from the file system domain, we first extracted all questions from these websites tagged with "bash" and "find", and obtained 401 questions. We retained the 146 of them that can be answered using the 17 bash utilities that appear in Tellina's training set. We did not filter tasks by the flags, and a task may require a flag that is not in Tellina's training set. Finally, we randomly sampled 16 out of the 146 tasks. The answers to those 16 tasks use `find`, `xargs`, `mv`, `cp`, `rm`, `grep`, `wc`, `ls`, and `tar`.

7.2.2 Rewriting. We asked researchers who are not involved in this project to paraphrase the descriptions of the selected tasks. This prevents the original task from being trivially found on the web in case the user copy-and-paste the task description into a search engine. Eight researchers in total contributed to the rewriting, and each of them rewrote 1 to 3 tasks. This avoids biasing the participants' phrasing of their natural language queries to be similar to a specific writer.

7.2.3 File system. We selected a code repository from GitHub[11] and used it as the file system for our tasks. The file hierarchy is 4 levels deep and consists of 29 folders and 62 files. We changed all constant values in the task descriptions to match the contents of this file system.

[10] Eleven participants hit this time limit in the experiment (often only for the first taskset).

[11] https://github.com/icecreammatt/class-website-template. This repository is randomly selected and is not related to the authors of this paper.

	Variable	Domain
Independent	Subject	$\{1, \ldots, 35\}$
	Treatment	{Control, Tellina}
	Taskset	$\{TS_1, TS_2\}$
	Order	{1st, 2nd}
Dependent	Time spent (sec.)	[0, 2400]
	Success rate	[0, 1]

Table 4: Experimental variables. Order indicates whether the taskset was the user's first or second taskset. Success rate is the fraction of the 8 tasks in the taskset that the user completed successfully.

Question	Response
Do you want to use Tellina in the future?	5.8 ± 1.2
How often did partially correct suggestions help you?	5.2 ± 1.5
How often were you slowed down by the incorrect suggestions made by Tellina?	3.2 ± 1.4
How easy was it for you to correct the incorrect suggestions made by Tellina?	4.6 ± 1.2

Table 5: Mean and standard deviation of the participant responses to the Likert-scale questions. All questions have scale 1–7.

What features of Tellina are the most helpful to you?
What mistakes made by Tellina affected you most?
Please list the features that you think we should add to Tellina.
Please give us any additional comments you have about Tellina.

Table 6: Open-ended questions in the post-study questionnaire.

7.3 Tool Interface

Tellina is a web application which can be accessed through a URL. It has an interface similar to the Google search engine. A user types a natural language sentence describing a task, then the website displays the RNN model's top 20 bash command translations of the sentence.

To help users understand the output commands, a user can hover over a token, such as a program name or flag, and see the man page description of the token. To provide a level playing field and avoid conflating this explanation feature with use of Tellina, we gave the participants a third-party tool, explainshell (http://explainshell.com/), which provides exactly the same man-page explanation functionality in the control treatment.

To help users in all treatments understand the effects of their commands, we provided a visual file system comparison tool (similar to Araxis Merge, KDiff3, or Meld) that indicates which files and directories differ and permits the user to interactively navigate the file system. This enables a user to understand what is wrong with the command without interpreting voluminous, possibly confusing diff output.

7.4 Training

Before starting a taskset, each participant completes a training task with the appropriate treatment condition. The Tellina website includes a tutorial to explain the usage of the interface (e.g., the input should be a complete imperative sentence) and the output presentation. We assumed all participants were familiar with web search and man pages and did not provide training for them.

7.5 Quantitative results

Table 4 lists the independent and dependent variables. We performed a four-way analysis of variance (ANOVA) for each dependent variable. The statistically significant results ($p < 0.01$) are that all four independent variables predict the time spent, and subject and taskset also predict the success rate.

The effects of subject on time are expected, because the participants are at different bash proficiency levels. There was no statistically significant effect of subject on success rate because the generous time limits (10 minutes per task, 40 minutes per taskset) enabled most users to complete most tasks. The overall success rate was 88%, and we were more interested in how to help programmers become more efficient, because we know that programmers can manage to solve tasks if given enough time.

The effects of order are also expected. On average, the participants spent 20% more time on the first taskset they encountered than on the second (1767 seconds vs. 1414 seconds), but were less successful (84% vs. 92% success rate). This learning effect reflects increasing participant familiarity with the example file system, tools such as file system diff, and bash tricks they learned earlier in the experiment.

The effects of taskset indicate that we failed to create two tasksets of equal difficulty. Users spent 24% more time, but were 10% less successful, with taskset TS_2.

Our counterbalanced factorial design enables the most interesting effects, those of treatment, to be accurately determined despite the effects of subject, order, and taskset. Participants in the Tellina treatment spent on average 22% less time (1397 seconds vs. 1784 seconds). This indicates that Tellina helps programmers to write bash commands in less time. The effect of treatment on success rate is significant only at the $p < 0.1$ level ($\mu_{\text{Tellina}} = 90\%$, $\mu_{\text{Control}} = 85\%$), for the reasons noted above when discussing the effect of subject on success rate.

7.6 Qualitative Results

Each participant filled out a questionnaire about their experience during the study.

The first part of the questionnaire consists of four Likert-scale questions (table 5). On average the participants wanted to use Tellina in the future (5.8/7). Tellina's partially correct suggestions were helpful (5.2/7) and did not slow down the users (3.2/7), but were difficult to correct (4.6/7). These results confirm our hypothesis that programmers are resilient to noise in the Tellina output and can use it as inspiration when formulating the correct command themselves. Anecdotally, while using Tellina ourselves we learned about command-line flags that we had not known about.

The second part of the questionnaire was four open-ended questions asking the participants to comment on specific features of Tellina and suggest future improvement (table 6). Below summarizes our findings.

Useful features. Many participants noted the usefulness of partially correct suggestions. Even when the full command was not correct, the web interface gave documentation and prompted the users to look up command options that they otherwise would not have remembered. Participants also liked the fact that Tellina suggests multiple solutions, which allows users to compare them and decide which one to try. Some participants liked Tellina's argument-filling feature. Tellina returned an answer with constants appropriate to the user's query, enabling the user to try out the command immediately without having to adjust the constants manually.

Limitations. Some participants noted that when Tellina suggested a wrong command that was close to a correct suggestion, it was difficult to trouble-shoot exactly what went wrong. Participants were also frustrated by subtle syntactic errors that prevented the commands from being run as is. (Tellina gives no guarantee that its output is a legal bash command.) Some participants were slowed down by incorrect argument formatting in Tellina's output, such as a missing "+" sign in the argument to `find`'s `-mtime` flag. Sometimes the RNN model suggested unusually complex commands (e.g., long pipelines), and the participants found them distracting even if some of the commands in them were correct.

Suggestions. Many participants requested better explanations of the output commands, so that they can better decide which one to try. For example, Tellina could incorporate a bash→English translator (either hand-coded or a learned RNN model) to explain its bash command output. Some participants suggested flagging which commands are syntactically valid, or providing sample output. Some participants also suggest interactive features that would enable the user to correct the output or give hints such as "the task needs to be solved using the `tar` utility".

8 RELATED WORK

Programming by natural language. There has been extensive research on synthesizing programs from natural language descriptions. Rule-based approaches have been developed to synthesize SQL queries [22], SmartPhone scripts [20], Java method specifications [28], and Spreadsheet programs [12]. These methods can often produce complex programs, but may require non-trivial manual effort to build and maintain. Recently, machine learning techniques have been developed to build probabilistic models to represent the joint distribution of text and programs [6, 13, 16] or the joint distribution of text and some formal specifications [21].

We extend this line of work by introducing an RNN encoder-decoder approach which can be applied to many different synthesis problems with relatively little domain-specific effort.

Our approach is also related to other deep-learning based PBNL approaches. DeepAPI [10] addresses the problem of retrieving API call sequences based on the user's natural language queries, using the RNN encoder-decoder model. CodeMend [33] proposed encoder-decoder models which complete partial programs by jointly modeling user's natural language input and the contextual programs. In comparison, Tellina directly synthesizes executable programs from NL descriptions and is able to handle open world arguments using our novel argument filling algorithm. Neural Programmer [26, 27] is a recently proposed deep learning architecture that allows direct encoding of discrete operators to improve learning efficiency. Since command languages cannot be succinctly represented using a few core operators, it is difficult to apply this approach to our domain. In addition, we present the first controlled user study demonstrating significant effectiveness of such techniques, despite their imperfect accuracy.

Semantic parsing. The problem of mapping natural language to programs or other formal representations has been extensively studied in the natural language processing community [2, 4, 7, 32, 40]. Earlier machine learning research in this area focus on learning formal grammars for mapping language to meaning representations [4, 40].

However, it has recently been shown that deep-learning based approaches work equally well, for example to produce regular expressions [19] and database queries [7, 26]. We expand the domain by studying deep learning for producing command languages, and evaluate the effect of such systems on programmers.

Deep learning in software engineering. Finally, deep-learning based approaches have also been applied to other software engineering problems, such as defect prediction [38], program feature extraction [25, 31], summarizing code using natural language [15], and program induction [8, 18]. In general, these applications leverage big data and design custom neural architectures for each application. In comparison, our focus is on studying the usefulness RNN encoder-decoder models, which have been shown to work well for a wide variety of program synthesis problems.

9 CONCLUSION

This paper presents an approach for program synthesis from natural language that leverages state-of-the-art neural machine translation techniques, augmented with slot (argument) filling and other techniques. We studied the complex domain of bash file system operations and conducted a controlled user study which shows that our tool, Tellina, significantly improves programmers' efficiency despite being imperfect in its program predictions.

As future work, it would be interesting to extend our neural architecture so that the entire framework for entity recognition, template translation and argument filling can be learned end-to-end. It should also be possible to extend the approach to cover more programming languages.

ACKNOWLEDGMENT

The research was supported in part by DARPA under the DEFT program (FA8750-13-2-0019), the ARO (W911NF-16-1-0121), the NSF (IIS1252835, IIS-1562364), gifts from Google and Tencent, and an Allen Distinguished Investigator Award. The authors thank Calvin Loncaric and Huan Sun for their contributions on initial problem exploration and constructive suggestions, and the UW NLP/PLSE group for helpful conversations on the work.

REFERENCES

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

[2] Yoav Artzi and Luke Zettlemoyer. 2011. Bootstrapping Semantic Parsers from Conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '11)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 421–432. http://dl.acm.org/citation.cfm?id=2145432.2145481

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. (2014). http://arxiv.org/abs/1409.0473

[4] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, Seattle, USA, 1533–1544. http://aclweb.org/anthology/D/D13/D13-1160.pdf

[5] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. (2014). http://arxiv.org/abs/1412.3555

[6] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program Synthesis Using Natural Language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 345–356. DOI: http://dx.doi.org/10.1145/2884781.2884786

[7] Li Dong and Mirella Lapata. 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)*. Association for Computational Linguistics, Berlin, Germany, 33–43. http://www.aclweb.org/anthology/P16-1004

[8] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. (2014). http://arxiv.org/abs/1410.5401

[9] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning, See [1]. http://aclweb.org/anthology/P/P16/P16-1154.pdf

[10] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 631–642. DOI: http://dx.doi.org/10.1145/2950290.2950334

[11] Çaglar Gülçehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. 2016. Pointing the Unknown Words, See [1]. http://aclweb.org/anthology/P/P16/P16-1014.pdf

[12] Sumit Gulwani and Mark Marron. 2014. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data, SIGMOD 2014, June 22-27, 2014*. ACM, Snowbird, UT, USA, 803–814. DOI: http://dx.doi.org/10.1145/2588555.2612177

[13] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 416–432. DOI: http://dx.doi.org/10.1145/2814270.2814295

[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. DOI: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[15] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model, See [1], 2073–2083. http://aclweb.org/anthology/P/P16/P16-1195.pdf

[16] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 173–184. DOI: http://dx.doi.org/10.1145/2661136.2661148

[17] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (2014). http://arxiv.org/abs/1412.6980

[18] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. 2016. Neural Random Access Machines. (2016). http://ercim-news.ercim.eu/en107/special/neural-random-access-machines

[19] Nate Kushman and Regina Barzilay. 2013. Using Semantic Unification to Generate Regular Expressions from Natural Language. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013*, Lucy Vanderwende, Hal Daumé III, and Katrin Kirchhoff (Eds.). The Association for Computational Linguistics, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA, 826–836. http://aclweb.org/anthology/N/N13/N13-1103.pdf

[20] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. ACM, New York, NY, USA, 193–206. DOI: http://dx.doi.org/10.1145/2462456.2464443

[21] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. 2013. From Natural Language Specifications to Program Input Parsers. 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*. The Association for Computer Linguistics, 1294–1303. http://aclweb.org/anthology/P/P13/P13-1127.pdf

[22] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB* 8, 1 (2014), 73–84. http://www.vldb.org/pvldb/vol8/p73-li.pdf

[23] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation, See [1]. http://aclweb.org/anthology/P/P16/P16-1057.pdf

[24] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

[25] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, Phoenix, Arizona, USA, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[26] Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei. 2016. Learning a Natural Language Interface with Neural Programmer. (2016). http://arxiv.org/abs/1611.08945

[27] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. 2015. Neural Programmer: Inducing Latent Programs with Gradient Descent. (2015). http://arxiv.org/abs/1511.04834

[28] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 815–825. http://dl.acm.org/citation.cfm?id=2337223.2337319

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 311–318. DOI: http://dx.doi.org/10.3115/1073083.1073135

[30] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. (2016). http://arxiv.org/abs/1611.01855

[31] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building Program Vector Representations for Deep Learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403 (KSEM 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 547–553. DOI: http://dx.doi.org/10.1007/978-3-319-25159-2_49

[32] Chris Quirk, Raymond J. Mooney, and Michel Galley. 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*. The Association for Computer Linguistics, Beijing, China, 878–888. http://aclweb.org/anthology/P/P15/P15-1085.pdf

[33] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 247–258. DOI: http://dx.doi.org/10.1145/2984511.2984544

[34] M. Schuster and K.K. Paliwal. 1997. Bidirectional Recurrent Neural Networks. *Trans. Sig. Proc.* 45, 11 (Nov. 1997), 2673–2681. DOI: http://dx.doi.org/10.1109/78.650093

[35] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. 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*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1073–1083. DOI: http://dx.doi.org/10.18653/v1/P17-1099

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112. http://dl.acm.org/citation.cfm?id=2969033.2969173

[37] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a Foreign Language. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., Montreal, Montreal, Canada, 2773–2781. http://papers.nips.cc/paper/5635-grammar-as-a-foreign-language.pdf

[38] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 297–308. DOI: http://dx.doi.org/10.1145/2884781.2884804

[39] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan

Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. (2016). http://arxiv.org/abs/1609.08144

[40] Luke S. Zettlemoyer and Michael Collins. 2012. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. (2012). http://arxiv.org/abs/1207.1420