Explainable Natural Language to Bash Translation using Abstract Syntax Tree

Shikhar Bharadwaj and **Shirish Shevade**
Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, KA 560012, India
`{shikharb,shirish}@iisc.ac.in`

Abstract

Natural language processing for program synthesis has been widely researched. In this work, we focus on generating Bash commands from natural language invocations with explanations. We propose a novel transformer based solution by utilizing Bash Abstract Syntax Trees and manual pages. Our method incorporates tree structure information in the transformer architecture and provides explanations for its predictions via alignment matrices between user invocation and manual page text. Our method performs on par with the state of the art performance on Natural Language Context to Command task and performs better than fine-tuned T5 and Seq2Seq models.

1 Introduction

Natural Language Processing (NLP) of programming language source code is a well-established field with a recent resurgence of using NLP techniques to assist programming. Advances have been made in code generation (Svyatkovskiy et al., 2020), code summarization (Ahmad et al., 2020), bug detection (Gupta et al., 2019), code translation (Lachaux et al., 2020) etc. However, there is a lack of explainable methods for assisting software developers.

Bash is a Unix command language with numerous utilities (like `cut`, `tr`, `sed` etc) for interacting with the Operating System. It is difficult for a novice developer to remember the purpose and syntax of these Bash utilities. Usually, to understand the intended usage of a utility, one looks up relevant manual page using `man`, `apropos` and `info`. However, these pages have been described as "frequently hard to read", byzantine and formal (Cozzie et al., 2011) and finding relevant information from them itself becomes a challenging task. Hence it gets difficult for a new user to learn Bash. Previous works in this area (Agarwal et al., 2021, 2020; Gros, 2019) have explored Transformer (Vaswani et al., 2017), Sequence to Sequence with attention (Bahdanau et al., 2015) (Seq2Seq), grammar guided methods and retrieval based methods for translating Natural Language to Bash commands. However, these methods are predictive and do not provide explanations or reasoning for their predictions. These methods propose a black-box solution to the translation problem relying on only parallel data. There is a lack of user trust in such predictions.

Motivated by these challenges, we propose an explainable neural machine translation model. To the best of our knowledge, ours is the first method to explore the use of manual page data for assisting translation. Besides translating natural language invocations to Bash code, our model also provides alignment matrices between user invocations and manual text, which explain the predictions by the model. Such alignments could be helpful to gain developer confidence in using the system. At the same time, it can help improve the learning curve for Bash while providing a more user-friendly interface with the Operating System.

We apply our method to NLC2CMD challenge (Agarwal et al., 2021) dataset (Lin et al., 2018) and compare the results with baselines like T5 (Raffel et al., 2019), Seq2Seq and the winning solution to NLC2CMD challenge (Agarwal et al., 2021). Our method performs better than T5 and Seq2Seq while providing explanations for the predictions. Code for our model is available on github [1]. We also release parsed and cleaned utility descriptions from linux manuals and *tldr*[2] (a community-contributed collection of utility descriptions) which were used in our experiments.

Problem Statement Assume that I is the set of all natural language invocations, C is the set of all Bash commands without pipes, process-

[1] `https://github.com/Shikhar-S/Explainable-NL-to-Bash-AST`

[2] `https://github.com/tldr-pages/tldr`

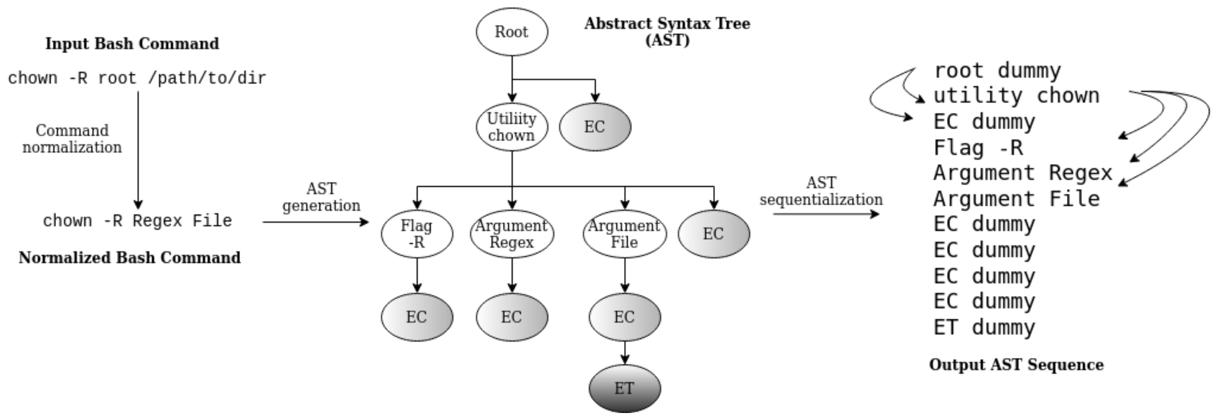


Figure 1: Target side processing for command `chown -R Regex File`: Nodes in grey are End of Children (EC) and End of Tree (ET) nodes. These are artificially introduced. The token sequence on the right is used as the target sequence for our translation model. Curved arrows in the AST sequence denote edges from parent tokens. The figure has been labelled with only a few edges for clarity.

substitution and command-substitution, $\mathcal{U}$ is the set of all Bash utilities and S is the set of all sentences from manual pages of Bash commands.

Given a dataset $\mathcal{D} := \{P, M\}$, where

- P is parallel natural language invocation-Bash command data of form (nlc, c), and
- M is the paired utility-utility description documentation data of form (u, d)

where $nlc \in I$, $c \in C$, $u \in \mathcal{U}$, and $d \subset S$, the task is to design an algorithm that, given an invocation $nlc \in I$ and dataset $\mathcal{D}$, outputs

- a set of Bash command-confidence pairs $(\hat{c}, \delta)$ such that $\hat{c} \in C$ is the predicted Bash command that performs the task specified in nlc and $\delta \in [0, 1]$ is the associated confidence score, and for each pair in this set also outputs,
- $\mathbf{A}_u^{nlc}$ an $\#nlc\times$ $\#d$ alignment matrix $\forall u \in U(\hat{c})$, where $\#nlc$ denotes the number of tokens in nlc, $\#d$ denotes the total number of tokens in all sentences in d, $U(\hat{c})$ denotes the set of all utilities in Bash command $\hat{c}$ and $(u, d) \in M$.

For example, nlc = *Recursively change ownership of all files in /path/to/dir to root.*
$c =$ `chown -R root /path/to/dir`
$U(c) = \{$`chown`$\}$

Notation We use lower case bold characters, **x**, to represent row vectors, upper case bold characters, **W**, to represent two dimensional matrices, italic characters, s, to represent scalars and upper case bold-italic characters, ***X***, to represent sequence of row vectors. Subscripts, as in $\mathbf{x}_i$, represent i^{th} coordinate of the vector **x** while superscript, as in $\mathbf{x}^i$, are used to index vectors in a sequence. The set of positive even numbers is denoted by N_e and the set of positive odd numbers is denoted by N_o. Element-wise product of two vectors is denoted by $\odot$. The inner product of two vectors is denoted by $\langle\cdot,\cdot\rangle$ and the Frobenius inner product of two matrices is denoted by $\langle\cdot,\cdot\rangle_F$.

2 Method

We model the problem as a natural language to Bash Abstract Syntax Tree (AST) translation task by transforming a Bash command to an AST. Our method is motivated by human cognition while programming. A developer usually looks up information from documentation while performing a programming task. Similarly, our model uses a convolutional neural network to match invocation against manual page data. It then generates a Bash command guided by manual page information. To incorporate the information at the correct time-step during the command generation process, we utilize Abstract Syntax Tree (AST) for Bash.

Abstract Syntax Tree An AST for a Bash command consists of nodes with two attributes: **structure** and **value**. The structure attribute denotes the broad type of node and the value attribute denotes the specific value of the node. For instance, for one of the children of the `root` node in Figure 1, `utility` is a structure attribute and `chown` is a value attribute.

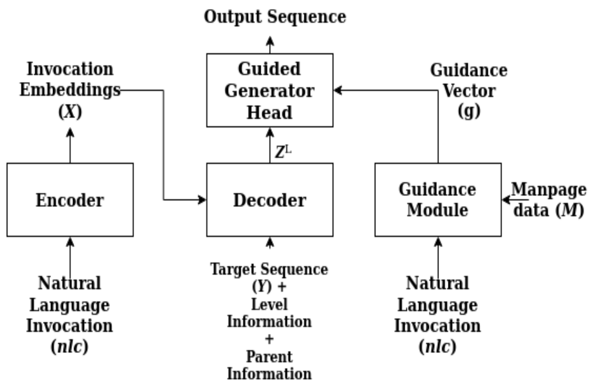

Output Sequence
Invocation Embeddings (X)
Guided Generator Head
Guidance Vector (g)
Z^L
Encoder
Decoder
Guidance Module
Manpage data (M)
Natural Language Invocation (nlc)
Target Sequence (Y) + Level Information + Parent Information
Natural Language Invocation (nlc)

Figure 2: Broad Architecture: Encoder is the standard transformer encoder. Decoder is modified to handle AST sequence. Guidance module guides the generator head via a guidance vector. Output sequence is $\hat{\boldsymbol{Y}}$.

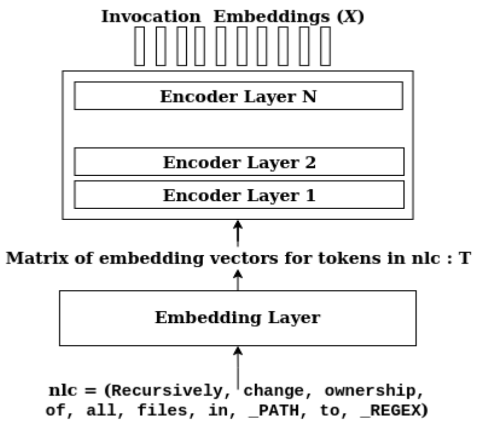

Invocation Embeddings (X)
Encoder Layer N
Encoder Layer 2
Encoder Layer 1
Matrix of embedding vectors for tokens in nlc : T
Embedding Layer
nlc = (Recursively, change, ownership, of, all, files, in, _PATH, to, _REGEX)

Figure 3: Encoder Stack: Standard transformer encoder.

Preprocessing Figure 1 illustrates the target side processing. The Bash command c is first normalized to replace all arguments with their types. Then we use the Bash parser by Agarwal et al. (2020) for parsing the normalized commands. The AST thus generated is sequentialized by traversing the tree in level order and enumerating node attributes. Hence, each node in the AST generates two tokens in the target sequence - one for structure and another for value of the node. From these tokens we construct **structure** and **value** vocabularies, V_1 and V_2 respectively. For nodes that do not have value attribute we introduce a *dummy* value. For instance, in Figure 1 the first 4 elements of the AST sequence are `(root, dummy, utility, chown)`. Some tokens in structure vocabulary are *root*, *utility*, *flag*, *argument* etc. Some tokens in value vocabulary are `chown`, `-R`, `ls` etc. The AST sequence is represented as a sequence of one-hot vectors $\boldsymbol{Y} = (\mathbf{y}^1, \ldots, \mathbf{y}^n)$ where $\forall i \in N_o, \mathbf{y}^i$ is a $|V_1|$ dimensional one hot row vector representing a token in structure vocabulary and $\forall i \in N_e, \mathbf{y}^i$ is a $|V_2|$ dimensional one hot row vector representing a token in value vocabulary.

Source side parallel data is also normalized and tokenized. Let $nlc = (t_1, t_2, \ldots t_{\#nlc})$ be the resulting sequence of tokens in nlc. For instance, $nlc =$ (*Recursively, change, ownership, of, all, files, in, _PATH, to, _REGEX*).

Broad Architecture A transformer based model is used for translating from source to target sequence and convolution filters are used to obtain information from manual pages. We use the standard transformer encoder for mapping the tokenized input invocation nlc to invocation embedding sequence $\boldsymbol{X}$. A guidance vector, $\mathbf{g}$, is also obtained from the whole manual page data (M) and nlc. Given $\boldsymbol{X}$ and the guidance vector $\mathbf{g}$, our enhanced decoder then generates an output sequence of probability distributions $\hat{\boldsymbol{Y}}$ over vocabularies V_1 and V_2. We train the model in an end-to-end fashion by backpropagating on a combined loss function.

The transformer decoder is extended with Tree Coordinates (Section 2.1) and Parent Attention (Section 2.2) to handle AST sequence. Section 2.3 describes a convolution net based approach to obtain relevant information from manual page data. Section 2.3.1 describes a method to analyze the filters for obtaining explainability information in the form of $\mathbf{A}_u^{nlc}$ matrices.

Transformer Encoder Tokens from nlc are embedded in $\mathbb{R}^k$ dimensional space (See Figure 3). The resulting matrix $\mathbf{T} \in \mathbb{R}^{\#nlc \times k}$ is then processed via the standard transformer encoder. As a result we obtain $\boldsymbol{X} = (\mathbf{x}^1, \ldots, \mathbf{x}^{\#nlc})$, a sequence of k dimensional embedding vectors.

Transformer Decoder Since the sequence $\boldsymbol{Y}$ is a sequentialization of an AST, it does not have explicit AST structure information for each token. To help the model in understanding AST structure, we embed this information. Two new blocks - Tree Coordinates and Parent Attention - are introduced for this purpose.

First, we embed target sequence $\boldsymbol{Y}$ to obtain an embedded target sequence $\mathbf{Z}^1 = (\mathbf{z}^{1,1}, \ldots, \mathbf{z}^{n,1})$. We then process $\mathbf{Z}^1$ through L decoder layers in an enhanced decoder stack to obtain $\mathbf{Z}^L$. A guided generator head then produces the output sequence $\hat{\boldsymbol{Y}} = (\hat{\mathbf{y}}^1, \ldots, \hat{\mathbf{y}}^n)$.

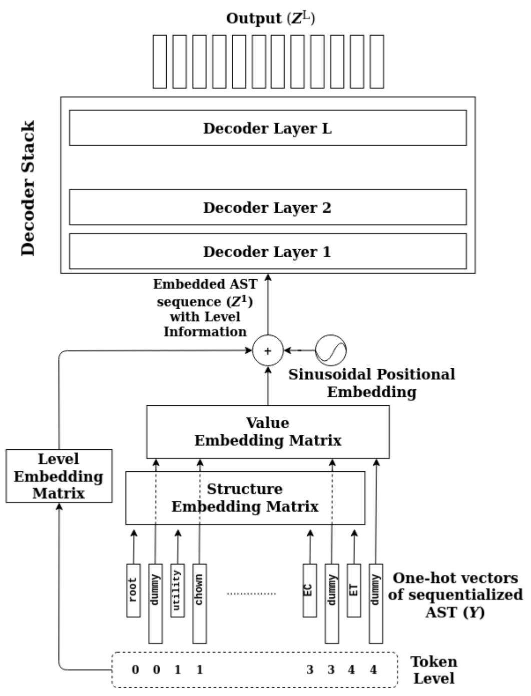


Figure 4: Embedding with Tree Coordinates: The one-hot vectors from two different vocabularies are embedded by the corresponding embedding matrices.To these, along with the usual sinusoidal positional embedding, we also embed and add the level of the node that generated the token.

2.1 Tree Coordinates

Once the AST is converted into a sequence, it is difficult to extract level information from sequence tokens. Level information is useful to know because certain nodes usually occur only at certain levels in the AST. For instance, `utility` only occurs at the first level. Therefore, we propose Tree Coordinate embeddings to incorporate level information about the nodes. Figure 4 explains the procedure graphically.

2.2 Parent Attention

It is also difficult to extract parent information for each token from the AST sequence. We hypothesize that adding attention only between the token and its children tokens could help in generalization. This parent attention block aids in the flow of information from parent to children.

After obtaining initial embeddings for target sequence ($\mathbf{Z}^1$ in Figure 4) the embedding sequence is transformed via multiple stacked decoder layers.

For the l^{th} layer, given an embedding sequence of length n, $\mathbf{Z}^l = (\mathbf{z}^{1,l}, ..., \mathbf{z}^{n,l})$, the decoder layer generates $\mathbf{z}^{i,l+1}$ as

$$\mathbf{z}^{i,l+1} = \|_{h=1}^{H} \Sigma_{j=1}^{i} \alpha_{ij}^{l,h} (\mathbf{z}^{j,l} \mathbf{W}_{l,h}^{V})$$

Here $\|_{h=1}^{H}$ denotes concatenation over the outputs of H attention heads. The attention value $\alpha_{ij}^{l,h}$ for head h, between tokens at positions i and j is computed as

$$\alpha_{ij}^{l,h} = \frac{exp(e_{ij}^{l,h})}{\Sigma_{j'=1}^{i} exp(e_{ij'}^{l,h})}$$

where $e_{ij}^{l,h}$ are computed as

$$e_{ij}^{l,h} = \frac{\mathbf{z}^{i,l} \mathbf{W}_{l,h}^{Q} (\mathbf{z}^{j,l} \mathbf{W}_{l,h}^{K} + \mathbf{a}^{ij,l})^{\top}}{\sqrt{k}}$$

Here, $\mathbf{W}_{l,h}^{K}$, $\mathbf{W}_{l,h}^{Q}$ and $\mathbf{W}_{l,h}^{V} \in \mathbb{R}^{k \times k_h}$ transform embeddings into keys, queries and values respectively, $\mathbf{a}^{ij,l} \in \mathbb{R}^{k_h}$, k_h is the dimension of attention head and $k_h = k/H$ where H is the number of heads.

In the above equation if one ignores $\mathbf{a}^{ij,l}$ term, the attention function is the standard multi-head attention. The trainable parameters, $\mathbf{a}^{ij,l}$, are introduced to capture information flow from parent at position j to child at position i and are non-zero only for parent tokens.

2.3 Guidance Module

We propose an explainable guidance module that guides the decoding process. We hypothesize that user invocation and utility descriptions will share similar n-grams, which can be easily captured by convolution filters. These n-grams may help in the translation process. Information from input invocation and utility description of a utility u is aggregated into the vectors $\mathbf{v}$ and $\mathbf{w}^u$ respectively. For doing so, r text convolution filters are applied, followed by sigmoid activation and max-pooling over time (Kim, 2014). Figure 5 explains this procedure for generating $\mathbf{v}$. Then, the cosine similarity between $\mathbf{v}$ and $\mathbf{w}^u$ determines if the utility u is useful for completing the task specified in invocation. Using cosine similarity allows for an easy analysis of the filters.

Obtaining invocation representation using CNN Using zero padding wherever necessary and defining a sliding window of size p over the matrix of invocation embeddings, $\mathbf{T} \in \mathbb{R}^{\#nlc \times k}$ as,

$$\mathbf{T}_i = [\mathbf{t}_{i-p+1}, \ldots \mathbf{t}_{i-1}, \mathbf{t}_i]; 1 \leq i \leq \#nlc$$

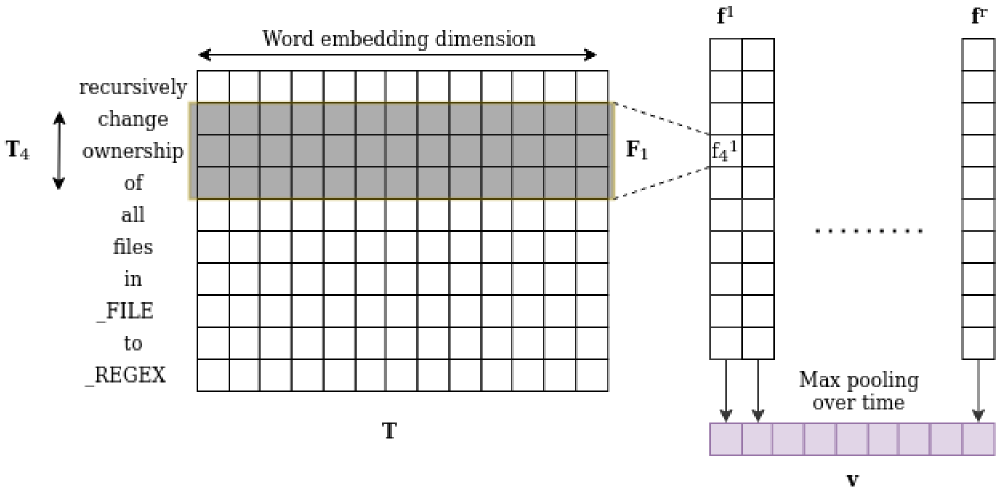


Figure 5: Obtaining invocation representation in Guidance Module: The convolution filter $\mathbf{F}_1$ passes over the matrix of token embeddings $\mathbf{T}$ to output a feature map $\mathbf{f}^1$. This is followed by max pooling which results in the invocation representation vector - $\mathbf{v}$. In this figure $\mathbf{T}_4$ denotes a window over token embeddings ending on the fourth token in the sequence.

we have $\mathbf{T}_i \in \mathbb{R}^{p \times k}$. Here $\mathbf{t}_i$ denotes the embedding vector for token t_i from nlc and $[\ldots]$ denotes row-wise concatenation. From each filter $\mathbf{F}_j \in \mathbb{R}^{p \times k}$, we calculate

$$f_i^j = sigmoid(\langle \mathbf{T}_i, \mathbf{F}_j \rangle_F + b_{F_j})$$

Here b_{F_j} is the bias term for filter $\mathbf{F}_j$. Concatenating these scalars, we obtain a feature map from filter $\mathbf{F}_j$

$$\mathbf{f}^j = [f_1^j, f_2^j, ...f_{\#nlc}^j]$$

Then a max pooling layer is applied to this vector, to get $v_j = \max_i \mathbf{f}_i^j$. Finally from r such filters we obtain the embedding vector for input invocation:

$$\mathbf{v} = [v_1, v_2, ...v_r]$$

Obtaining utility representation using CNN For a given utility u, let $\mathbf{D}_u \in \mathbb{R}^{\#d \times k}$ be the matrix obtained by embedding its utility description d. Here $\#d$ is the total number of tokens in all sentences in d. In a manner similar as before, applying r convolution filters on $\mathbf{D}_u$ followed by sigmoid activation and max pooling gives us a representation vector $\mathbf{w}^u$ corresponding to utility u.

Obtaining Guidance Vector We assume that the similarity between user invocation and utility description determines if the utility is necessary for performing the task specified in the invocation. For the invocation representation, $\mathbf{v}$, the similarity is computed for all utility-utility description, (u_i, d_i), pairs in man-page data, and aggregated in the form of a *guidance vector* as follows:

$$\mathbf{g} = [g_{u_1}, g_{u_1}, \ldots, g_{u_{|M|}}]$$

Here $g_{u_i} = \frac{\langle \mathbf{v}, \mathbf{w}^{u_i} \rangle}{||\mathbf{v}||.||\mathbf{w}^{u_i}||}$. Each coordinate of the guidance vector is the cosine similarity between invocation representation and utility representation. This *guidance vector* is used to guide the command generation process.

2.3.1 Analyzing Guidance Filters for Explanation

From Figure 5 observe that each coordinate in the representation vector $\mathbf{v}$ corresponds to an n-gram in the input invocation. This is true because each coordinate in a feature map $\mathbf{f}^i$ corresponds to a window over certain n-gram in the invocation text and during the max-pooling step we copy the maximum of these coordinates in the corresponding coordinate of $\mathbf{v}$. Same is true for $\mathbf{w}^u$ and n-grams in utility description. Since cosine similarity is proportional to the dot product of the embedding vectors, we can find out the embedding vector coordinates that increase the cosine similarity score. Then, the n-grams in the corresponding texts can be obtained. These n-grams are important for predicting whether the utility is needed for a given task. Figure 6 explains the procedure to construct alignment matrix from these n-grams.

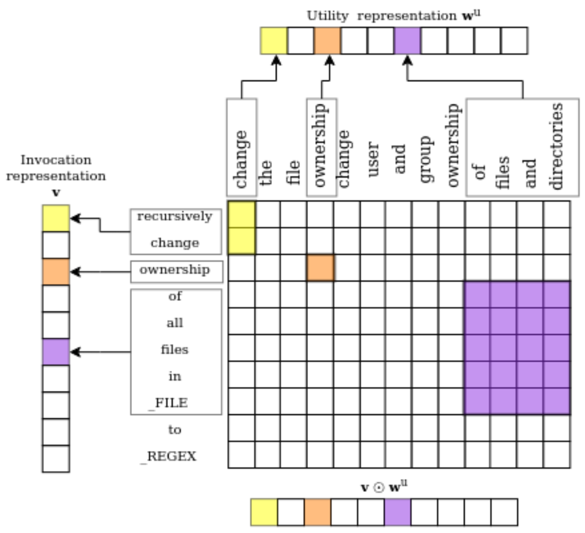


Figure 6: Generating an alignment matrix: Top 3 (in descending order) coordinates of $\mathbf{v} \odot \mathbf{w}^u$ are shown in color. Correponding n-grams are shown in boxes. Initialising from a zero matrix, we increment the values in sub-matrices corresponding to these 3 n-grams.

2.4 Guided Decoding

Guidance information is incorporated during the generation step. For each time step in the generation process, a probability distribution over either the *structure* vocabulary or the *value* vocabulary is obtained. Beam search is used for decoding, resulting in a sequentialized AST representation. This representation is converted back to an AST, from which the final prediction command is obtained.

Let $\mathbf{Z} = (\mathbf{z}^1, \dots, \mathbf{z}^n)$ be the sequence of vectors produced by final decoder layer. For $i \in N_e$, on $\mathbf{z}^i$ we apply a linear transformation followed by softmax to obtain the distribution $\hat{\mathbf{y}}^i$ over *structure* tokens. This is the usual transformer decoding process. For $i \in N_o$, we selectively add the guidance at the correct time-step during generation. The selective addition of guidance information is possible because of the manner in which the AST sequence is generated. We know the structure attribute of the node prior to generating its value attribute. So for $i \in N_e$, we first apply a linear transformation on $\mathbf{z}^i$ to obtain a *logit* vector $\mathbf{p} \in \mathbb{R}^{|V_2|}$. We introduce a trainable row vector $\mathbf{s} \in \mathbb{R}^{|V_2|}$. Then if $\underset{i}{\operatorname{argmax}}\ \hat{\mathbf{y}}^{i-1}$ is the index for `utility` token, we modify $\mathbf{p}^i \leftarrow \mathbf{p}^i + \mathbf{s} \odot R(\mathbf{g})$ where $R : \mathbb{R}^{|M|} \mapsto \mathbb{R}^{|V_2|}$ maps the guidance vector from the space of utilities to *value* vocabulary. Intuitively, R aligns the guidance vector to match the indexing of *value* vocabulary. For non-utility tokens it fills in the output vector with zeros. Finally, we apply softmax over $\mathbf{p}^i$ to obtain the probability distribution $\hat{\mathbf{y}}^i$ over *value* tokens.

2.5 Loss

Let $\hat{\boldsymbol{Y}} = (\hat{\mathbf{y}}^1, \dots, \hat{\mathbf{y}}^n)$ be the output obtained after guided decoding (Section 2.4).

For $i \in N_e$, $\hat{\mathbf{y}}^i \in \mathbb{R}^{|V_1|}$ are probability distributions over *structure* tokens. For $i \in N_o$, $\hat{\mathbf{y}}^i \in \mathbb{R}^{|V_2|}$ are probability distributions over *value* tokens.

The model is trained by back-propagation on loss L:

$$L = L_{structure} + \lambda_1 L_{value} + \lambda_2 L_{guide}$$

where $L_{structure}$ and L_{value} are standard cross-entropy functions over corresponding index in target and output and L_{guide} is the cosine embedding loss.

$$L_{structure} = \frac{1}{n/2} \sum_{i \in N_e} \Sigma_{j=1}^{|V_1|} (\mathbf{y}_j^{i+1} \ln \hat{\mathbf{y}}_j^i)$$
$$L_{value} = \frac{1}{n/2} \sum_{i \in N_o} \Sigma_{j=1}^{|V_2|} (\mathbf{y}_j^{i+1} \ln \hat{\mathbf{y}}_j^i)$$
$$L_{guide} = \frac{1}{|M|} \Sigma_{(u,d) \in M} l_u$$

where $l_u = \mathbb{I}(u \in U(c)) * (1 - cos(\mathbf{v}, \mathbf{w}^u)) + \mathbb{I}(u \notin U(c)) * max(0, cos(\mathbf{v}, \mathbf{w}^u))$. Here $U(c)$ is the set of utilities in command c and $\mathbb{I}(\cdot)$ is the indicator function.

3 Experiments

3.1 Dataset

We tested our method on the data from NLC2CMD competition (Agarwal et al., 2021). This data was filtered to remove all commands that had `pipe`, `command substitution` and `process substitution`. Filtering resulted in 3203 invocation-command pairs. Then the data was split into ten folds to perform multiple runs. In a single run, one fold was kept hidden for testing, and the remaining nine folds were pooled and split for training and validation purposes in the ratio of 9:1. All the results mentioned in Section 4 were obtained by averaging over these ten runs. For generating documentation data, we parsed the linux manual pages to get utility - utility description pairs. To these, we also appended the parsed and cleaned utility descriptions from *tldr* pages, which are community collected manuals for Bash utilities. The natural language parallel data was normalized to remove file paths and regex. We performed similar normalization on target side data as well. This normalization is performed for all models mentioned in Table 1.

3.2 Metric

To evaluate the model, we use the metric defined in Agarwal et al. (2021). This was the metric for NLC2CMD competition. This metric ignores command arguments but considers the order of predicted utilities and flags for each utility.

A model A outputs top-5 translations as follows: $A : nlc \mapsto \{q | q = (\hat{c}, \delta)\}$. Here the tuple $(\hat{c}, \delta)$ represents the predicted command $\hat{c}$ with associated confidence score δ. We assume that $|A(nlc)| \leq 5$ and there is only one ground truth command c corresponding to an invocation nlc.

The normalized score of a single prediction is calculated as follows:

$$S(q) = \sum_{i \in [1,T]} \frac{1}{T} \times \Big(\mathbb{I}(U(\hat{c})_i = U(c)_i) \times \frac{1}{2}\Big(1 + \frac{1}{N_i}(2 \times |F(U(\hat{c})_i) \cap F(U(c)_i)| - |F(U(\hat{c})_i) \cup F(U(c)_i)|)\Big) - \mathbb{I}(U(\hat{c})_i \neq U(c)_i) \Big)$$

Here, $U(c)$ is the sequence of Bash utilities in the command c, $F(u)$ is the set of flags for utility u in respective command, $T = \max\big(|U(c)|, |U(\hat{c})|\big)$ and $N_i = \max\big(|F(U(c)_i)|, |F(U(\hat{c})_i)|\big)$. $\mathbb{I}(\cdot)$ is the indicator function.

Overall score of the prediction is given by:

$$Score = \begin{cases} \max_{q \in A(nlc)} S(q), & \text{if } S(q) > 0 \text{ for some } q \in A(nlc); \\ \frac{1}{|A(nlc)|} \sum_{q \in A(nlc)} S(q), & \text{otherwise.} \end{cases}$$

3.3 Baselines

Following baselines are considered for comparison:

- **Magnum**: This is the winner's solution to NLC2CMD challenge (Teng and Fu, 2020) and the state of the art on this dataset. The model is an ensemble of multiple transformer models trained with multiple seeds and batch sizes. They also use a pre-processing routine to normalize file paths and regex patterns in both English text and Bash commands. We compare with a single model from the ensemble for a fair comparison.

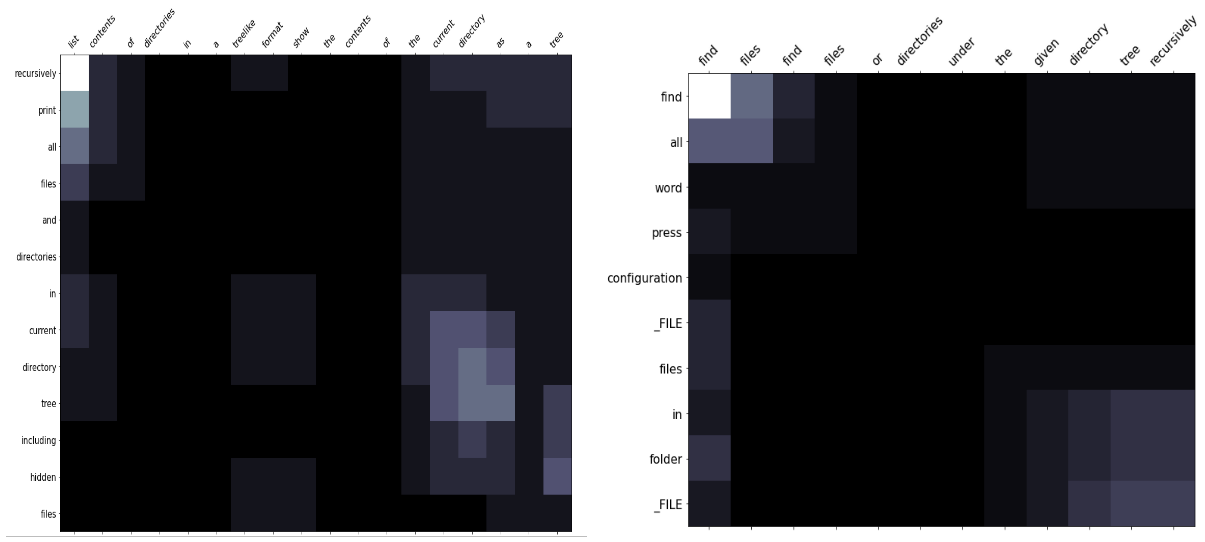
- **T5**: T5 is a transformer based model proposed by Raffel et al. (2019). It is pre-trained on vast amounts of data. We fine tune T5 on our dataset. The T5-small model by huggingface (Wolf et al., 2020) performed the best among T5-small, T5-large and T5-base. We report the results for T5-small. The input to this model was "translate English to Bash:" followed by the input invocation, and the target was the normalized Bash command.

- **Seq2Seq**: Sequence to sequence with attention was proposed by Bahdanau et al. (2015). It is an LSTM based encoder-decoder model that uses the attention mechanism to dynamically generate context for decoding.

We use beam search for decoding all models with a beam size of 10. Predictions are sorted by beam search score. The metric score is calculated with top-5 predictions, with prediction confidence (δ) equal to 1 for top-t predictions and equal to $exp(beam\_score)/2$ for bottom $5 - t$. We tune all the models on the validation set to find the best setting of t for each model.

3.4 Hyper-parameters

We train our method with a batch size of 10 examples and gradient accumulation over 50 steps with default settings for Adam optimizer. Both encoder and decoder process 512 dimensional vectors. We have 6 encoder layers, 6 decoder layers and 8 attention heads in both layers. For the guidance module, 200 filters with window sizes 1,2,3,4,5 and 6 were used. In the loss function we set λ_1 equal to 1 and λ_2 to 10. Magnum is a 512 dimensional transformer model with 6 encoder layers, 6 decoder layers and 8 attention heads in both encoder and decoder layers. It was trained for 2500 steps with



(a) $\mathbf{A}_u^{nlc}$ matrix, $u =$`tree` (b) $\mathbf{A}_u^{nlc}$ matrix, $u =$`find`

Figure 7: Alignment Matrices: Invocation is on the left, and utility description is at the top in both sub-figures. Lighter sub-matrices denote that more filters picked up corresponding n-grams.

each batch consisting of 14000 tokens and gradient accumulation over 2 steps and a warm-up scheduler. For Seq2Seq, we used two 256 dimensional bidirectional LSTM layers for the encoder and two 256 dimensional LSTM layers for the decoder with attention in between encoder and decoder. For T5, we fine tuned the T5-small model on our dataset.

4 Results and Analysis

Results are mentioned in Table 1. Our method performs comparable to the state of the art solution (Magnum) and better than other baselines like T5 and Seq2Seq.

We hypothesize that since the AST sequence is at least twice the length of the original command, it is, therefore, more challenging to generate an AST sequence. However, this sequentialization helps by making the model explainable. Even when all baselines directly generate the Bash command, our method performs better than two of them. Moreover, T5 is trained on massive amounts of parallel data, and our model trained on only 2,600 examples and 116 utility descriptions surpasses T5. Therefore, domain-specific data, like manual pages, is helpful for Bash command generation.

Model	Test score
Seq2Seq (Bahdanau et al., 2015)	0.594 ± 0.032
T5 (Raffel et al., 2019)	0.639 ± 0.027
Magnum (Agarwal et al., 2021)	0.685 ± 0.027
Proposed Method - Parent Attention	0.503 ± 0.154
Proposed Method - Tree Coordinates	0.597 ± 0.052
Proposed Method - Guidance Module	0.630 ± 0.072
Proposed Method	0.651 ± 0.017

Table 1: NLC2CMD Competition metric on test set. Values range from -1 to 1. Higher is better. All entries are averaged over 10 runs and in the form mean $\pm$ standard deviation.

5 Qualitative Analysis

Analyzing the $\mathbf{A}_u^{nlc}$ matrix in Figure 7, it is seen that the convolution filters align *"recursively print all files"* from user invocation to *"list"* from utility description for `tree`. Here both of these n-grams do not share any common token but are semantically similar. This relationship is captured by the guidance module which then predicts that the utility `tree` is needed for completing the task specified in user invocation. It is also observed that the n-grams *"current directory tree"* from invocation and *"the current directory"* from manual text are captured by convolution filters. Thus, n-grams sharing common tokens are also matched accurately. These are semantically similar and hence picked up by convolution filters. Such alignment matrices provide a level of confidence in the system's predictions.

Summing up $\mathbf{A}_u^{nlc} \in \mathbb{R}^{\#nlc \times \#d}$ along the nlc dimension, we obtain a $\#d$-dimensional vector. This vector represents important n-grams in the utility description d. We calculate this vector for every invocation, average them and apply softmax to get

Utility	**Utility Description (from manual pages and $tldr$)**
`wget`	the noninteractive network downloader download files from the web supports http https and ftp
`md5sum`	compute and check md5 message digest calculate md5 cryptographic checksums
`sleep`	suspend execution for an interval delay for a specified amount of time
`rev`	reverse lines characterwise reverse a line of text
`bzip2`	a blocksorting file compressor

Table 2: Averaged alignment distributions: The text is concatenated descriptions from manual pages and $tldr$. Darker tokens have more weight.

an importance distribution over all tokens in the utility description. This distribution captures the distinguishing features of a utility. For example, in Table 2 the filters correctly capture *"network downloader"* for `wget`, which is a two-word summary of what `wget` does. Such auto-generated summaries are useful pedagogical tools for learning Bash.

6 Ablation Study

We propose three components, namely Tree Coordinates (Section 2.1), Parent Attention (Section 2.2) and Guidance Module (Section 2.3). In Table 1 we show the results obtained when each of these components was removed from the system and resulting architecture trained till convergence. It is observed that removing parent attention affects the model the most with a huge drop in the score. Removing tree coordinates also leads to a drop in performance. Finally, without guiding our model performs worse than the final model. It is also observed that the variance of performance score increases as we remove any component from the model with the most increase obtained by removing parent attention.

7 Related Work

Agarwal et al. (2021) provide a comprehensive report on the NLC2CMD contest. Agarwal et al. (2020) propose a recurrent neural net based architecture for Natural Language to Bash translation task that sets the baseline for NLC2CMD contest.

Shaw et al. (2018) propose relative attention as an alternative to sinusoidal positional embeddings (Vaswani et al., 2017) for Transformers. Our parent attention is inspired by their relative attention.

Yin and Neubig (2017) propose a novel RNN based architecture for code generation. Their method uses a grammar model to explicitly capture the target syntax as prior knowledge. Unlike their method, our method works without access to grammar rules and is based on the Transformer architecture.

Shiv and Quirk (2019) design sinusoidal embeddings for incorporating tree structure. Their method works only for binary trees. Zügner et al. (2021) propose a method to jointly learn code context and structure of source code for code summarization task. Kim et al. (2021) propose several ways of communicating code structure to Transformer for code completion task.

Jacovi et al. (2018) present an analysis of convolutional neural networks for text. They show that filters in such networks capture different semantic classes of n-grams.

8 Future Work

In future work, we would experiment with different sequential representations of the AST. Linux manpage is a rich source of information for this task. Exploring it further might be useful. An interesting idea is to split the invocations into constituent tasks. Planned prediction, proposed in Hong et al. (2020), is also an exciting idea to explore.

9 Conclusion

This work explores the use of manual page data for natural language to Bash translation and proposes a cognitively inspired transformer based architecture for generating Abstract Syntax Trees for Bash. Our method also explains its predictions by alignment matrices between user invocation and manual page text. Additionally, we obtain insights about the distinguishing features of each utility from these alignment matrices. We test our method on NLC2CMD data.

References

Mayank Agarwal, Jorge J Barroso, Tathagata Chakraborti, Eli M Dow, Kshitij Fadnis, Borja Godoy, and Kartik Talamadupula. 2020. Clai: A platform for ai skills on the command line. *arXiv preprint arXiv:2002.00762*.

Mayank Agarwal, Tathagata Chakraborti, Quchen Fu, David Gros, Xi Victoria Lin, Jaron Maene, Kartik Talamadupula, Zhongwei Teng, and Jules White. 2021. Neurips 2020 nlc2cmd competition: Translating natural language to bash commands.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Anthony Cozzie, Murph Finnicum, and Samuel T. King. 2011. Macho: Programming with man pages. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA. USENIX Association.

David Gros. 2019. *AInix: An open platform for natural language interfaces to shell commands*. Ph.D. thesis.

Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. 2019. Deep learning for bug-localization in student programs. *CoRR*, abs/1905.12454.

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. 2020. Latent programmer: Discrete latent codes for program synthesis. *CoRR*, abs/2012.00377.

Alon Jacovi, Oren Sar Shalom, and Yoav Goldberg. 2018. Understanding convolutional neural networks for text classification. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 56–65, Brussels, Belgium. Association for Computational Linguistics.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar. Association for Computational Linguistics.

Marie-Anne Lachaux, Baptiste Rozière, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *CoRR*, abs/2006.03511.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations.

Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *NeurIPS 2019*.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *CoRR*, abs/2005.08025.

Zhongwei Teng and Quchen Fu. 2020. Magnum-nlc2cmd.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

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

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context.