Efficient Constituency Tree based Encoding for Natural Language to Bash Translation

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

Bash is a Unix command language used for interacting with the Operating System. Recent works on natural language to Bash translation have made significant advances, but none of the previous methods utilize the problem's inherent structure. We identify this structure and propose a Segmented Invocation Transformer (SIT) that utilizes the information from the constituency parse tree of the natural language text. Our method is motivated by the alignment between segments in the natural language text and Bash command components. Incorporating the structure in the modelling improves the performance of the model. Since such systems must be universally accessible, we benchmark the inference times on a CPU rather than a GPU. We observe a 1.8x improvement in the inference time and a 5x reduction in model parameters. Attribution analysis using Integrated Gradients reveals that the proposed method can capture the problem structure.

1 Introduction

Semantic parsing is one of the central tasks for natural language understanding (NLU). It is defined as the task of generating meaning representations from natural language utterances (Kamath and Das, 2019). Previous works (Yin and Neubig, 2017, Yaghmazadeh et al., 2017, Kim et al., 2020, Agarwal et al., 2021) have used high level languages such as Python, SQL and Bash as meaning representations. This work focuses on generating Bash commands from natural language descriptions of command-line tasks.

Besides being an essential task for NLU, semantic parsing into a high-level language also has real-world applications such as helping developers write programs and making programming universally accessible. The command-line interface has been regarded as an invaluable tool due to its expressiveness, efficiency and extensibility (Agarwal et al., 2021). However, it has a learning curve and requires domain knowledge. An interface with the computer using natural language, on the other hand, remedies these issues. One need not remember the syntax of hundreds of Bash utilities, and instead, one can specify the task in natural language. Such an interface that uses natural language such as English for specifying command-line tasks makes computing accessible to people with little domain knowledge. Therefore, developing a system to generate Bash commands from English is worth one's efforts.

Previous works on this semantic parsing task (Lin et al., 2018, Gros, 2019, Agarwal et al., 2020, Agarwal et al., 2021, Bharadwaj and Shevade, 2021) use various encoder-decoder style architectures. These methods consider the natural language component as a sequence of tokens without utilizing the inherent structure for this problem. The method proposed in this work utilizes the information from the constituency parse tree of the natural language to incorporate this problem structure into the modelling process. Our approach is based on the observation that natural language invocations are complex and can be broken down into simpler segments that align with the Bash command components (utilities, flags and arguments). We incorporate this observation in our method to provide an inductive bias to the Transformer model (Vaswani et al., 2017), making the search space of solutions more aligned with the task at hand. The models are evaluated on the NL2Bash dataset (Lin et al., 2018) obtained from the NeurIPS 2020 Natural Language Context to Command (NLC2CMD) contest (Agarwal et al., 2021). The proposed method outperforms the winning solution from the NLC2CMD contest in terms of generation accuracy while also achieving a speedup of 1.8x and reducing the parameter count by 5x over it. It also performs better than models like T5 (Raffel et al., 2019) and

Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 3159 - 3168

CodeT5 (Wang et al., 2021) which are trained on a large amount of data and then fine-tuned on the dataset under study. Our code is available at https://github.com/Shikhar-S/Segmented-Invocation-Transformer

Our main contributions are the following:

- We identify the structure for natural language to Bash generation task and propose a constituency tree based method for incorporating the structure in the Transformer (Vaswani et al., 2017) framework. The proposed modification improves the performance of the Transformer on this task.
- We benchmark the Transformer against the proposed architecture. Results show a reduction in inference time and the number of parameters.
- We conduct attribution analysis using Integrated Gradients (Sundararajan et al., 2017) to analyze the proposed method's workings.

First, we formally describe the problem statement. Section 2 describes the structure for the problem, and Section 3 describes our approach to model the structure and expected gains in the inference time via a complexity analysis of the decoding phase. Section 4 describes the dataset used and its preprocessing. In Section 5, we describe the experiments conducted for checking the correctness and efficiency of our approach and analyzing the results. Section 6 compares our work with other related works. Finally, in Section 7 we conclude and discuss some directions for future work.

Problem Statement. Let I be the set of all natural language invocations, C be the set of all Bash

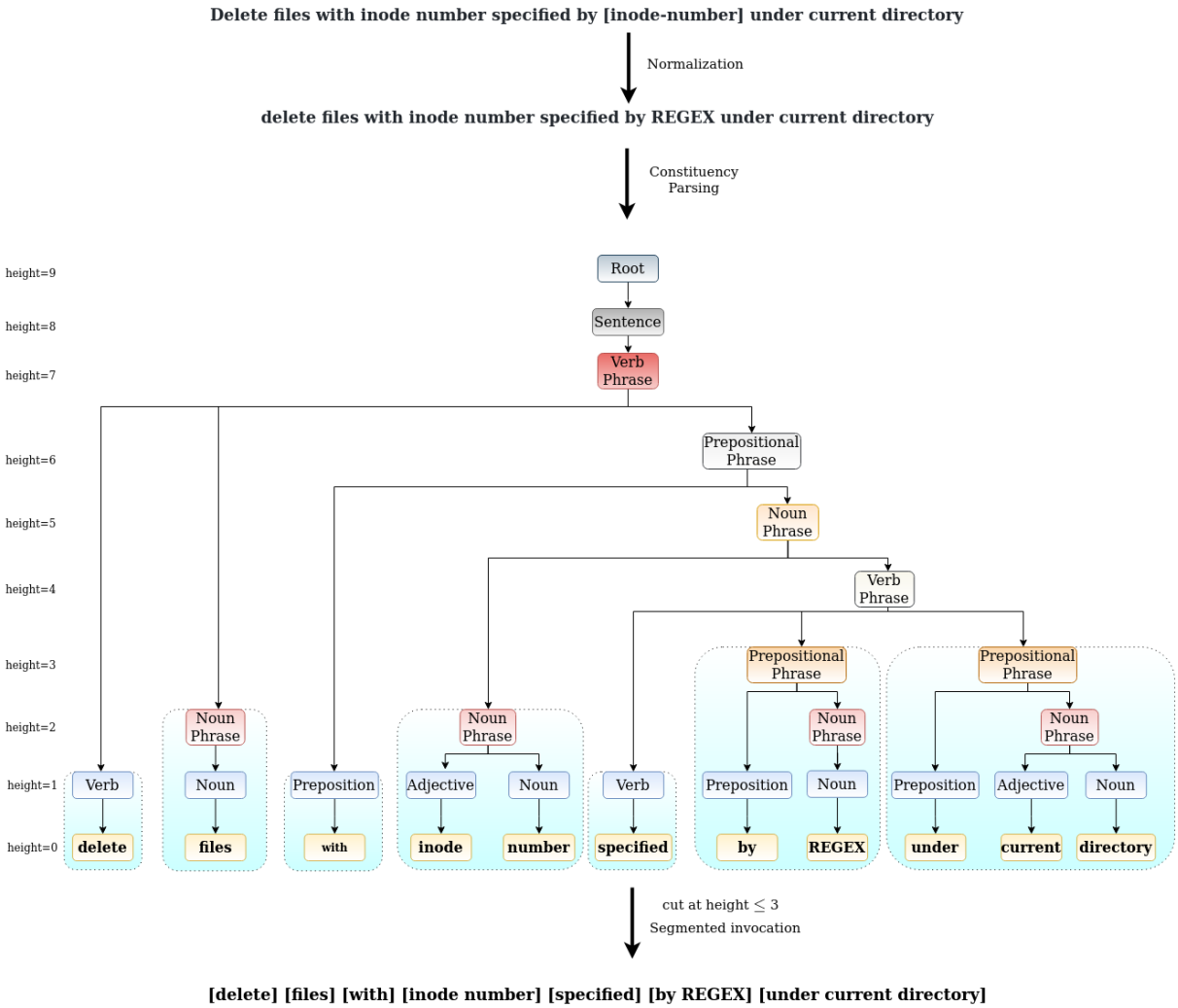

Figure 1: Segmenting Invocation: First, the raw invocation is normalized to remove patterns and file paths. This normalized invocation is then parsed to obtain a constituency tree. Then, the tree is cut at a threshold height to create subtrees. The leaves of each subtree form tokens in the segments.

commands and $\mathcal{D} := \{(nlc, c)\}$ be a parallel natural language invocation-Bash command dataset, where $nlc \in I$ and $c \in C$.

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.

For example,
nlc = *Delete files with inode number specified by REGEX under current directory.*

```
c = find . -inum REGEX -exec rm -i {} \;
```

2 Problem Structure

On conducting a manual analysis of a few examples from the training data, it was observed that natural language invocations are often complex and can be broken down into simpler descriptive segments needed for the task. These segments would often map directly to a Bash command component- a utility, a flag or an argument. For instance, consider

Invocation

| ***Delete*** | *files* | *with* | ***inode number*** | *specified* | ***by REGEX*** | ***under current directory***.|

BashCommand

```
find . -inum REGEX -exec rm -i {} \;
```

Invocation segment	Bash command component
Delete	`rm`
inode number	`-inum`
by REGEX	`REGEX`
under current directory	`.`

Table 1: Segments from natural language invocation that align with Bash command components (utilities, flags and arguments).

Table 1 lists the segments of the invocation that map to the command components. These segments represent meaningful self-contained constituents of a complex invocation. In the above instance, "*under current directory*" and "*by REGEX*" are segments representing single concepts.

3 Method

We frame the problem as a translation task from English to Bash. In Section 3.1 we describe our approach to incorporate the structure in modelling natural language invocations. Section 3.2 describes the proposed architecture and an analysis of its computational complexity at inference time.

3.1 Segmenting Invocation using Constituency Tree

The constituency tree represents the syntactic structure of a sentence based on phrase structure grammar (Chomsky, 1956). We propose a simple method that utilizes the constituency tree for segmenting natural language invocations. Our method is outlined in Figure 1. First, we normalize the invocation to replace patterns and file paths with their types. Next, we parse the normalized English invocation to obtain its constituency parse tree. For all the experiments reported in this work, we use the Stanford CoreNLP parser (Manning et al., 2014). Let the *height* of a node be defined as the number of edges on the longest path from the node to a leaf in the node's subtree (as shown in Figure 1). Then we perform a depth-first traversal on the tree in the left to right order of nodes. While performing the depth-first traversal, we cut the tree at the first node with a height less than a threshold and do not expand the search on this node further. As a result, we obtain various subtrees, where each subtree corresponds to a segment composed of the tokens in the leaves of the subtree. Finally, all segments are collected from the subtrees to obtain the segmented invocation.

3.2 Segmented Invocation Transformer

Let the $nlc = [t_1, t_2, \ldots t_n]$ be composed of n tokens. The invocation segmentation procedure takes the constituency tree for nlc and the threshold height as inputs and returns k segments $[s_1, s_2, \ldots s_k]$, where each segment $s_i = [t_j, t_{j+1}, \ldots t_{j+n_i-1}]$ is composed of n_i tokens such that $\sum_{i=1}^{k} n_i = n$.

We use a Transformer (Vaswani et al., 2017) based architecture and modify the Transformer encoder to capitalize on the segmentation information obtained from the constituency tree. Specifically, an averaging layer is introduced before the Transformer encoder to capture the local structure (Section 2). From the embedded token sequence comprising of n vectors, the averaging layer computes a sequence of k segment embeddings. The input to the averaging layer consists of n vectors, each resulting from the sum of token embedding and the corresponding sinusoidal position embedding. These are grouped into k segments, and the

averaging layer then computes the mean over each segment to produce a sequence of k embedding vectors, one for each segment. On the decoder side, we use the standard Transformer decoder. We name this architecture Segmented Invocation Transformer (SIT), and it is shown in Figure 2. The model is trained by back-propagation on the cross-entropy loss with label smoothing of 0.1.

Complexity Analysis Next, we analyze the computational complexity of the cross-attention of the decoder during inference to point out the improvement over the vanilla Transformer. The decoding occurs in discrete time steps. We shall consider a single time step in this analysis. At each time step, in the cross-attention layer of the decoder, we first construct the keys, query and values and then perform a softmax over the product of keys matrix with the query vector to get the cross-attention scores. Let the dimension of the embedding vectors be d. Considering a single head for simplicity, the construction of values matrix takes $O(kd^2)$ time (from the multiplication of $\mathbb{R}^{k\times d}$ and $\mathbb{R}^{d\times d}$ matrices), where k is the number of segments. Similarly, the construction of query vector takes $O(d^2)$ time (from the multiplication of a $\mathbb{R}^d$ vector with $\mathbb{R}^{d\times d}$ matrix). Multiplying keys matrix ($\mathbb{R}^{k\times d}$) with the query vector ($\mathbb{R}^d$) followed by a softmax (over k attention scores) takes $O(kd+k)$ time. This step is followed by a weighted aggregation of the k values, each being d-dimensional, in $O(kd)$ time. Hence, the overall complexity for cross-attention layer is $O(kd^2 + d^2 + kd)$. Since the dimension d is a constant, this can be simplified to $O(k)$. A vanilla Transformer would incur $O(n)$ time. Therefore, our method provides a constant factor improvement per decoding time step. This advantage adds up due to multiple decoding time steps needed during the inference phase. The time benchmarks (Section 5.2) show these differences in practice.

4 Data and Preprocessing

For evaluating our method, we used the NL2Bash dataset (Lin et al., 2018) provided by the NLC2CMD contest (Agarwal et al., 2021) from NeurIPS 2020. It consists of approximately 10k paired English invocations and Bash commands scraped from Stack Overflow covering over 100 Bash utilities. The dataset was partitioned into five folds. We performed five runs. In each run, we split one fold equally for validation and testing. The remaining four folds were pooled to create the training set for the run. All results mentioned in Table 3 are averaged over these five runs. Invocations and bash commands for all the models to remove file paths and regex. We used the natural language toolkit[1] and the Bash parser[2] shared by the NLC2CMD competition organizers for preprocessing.

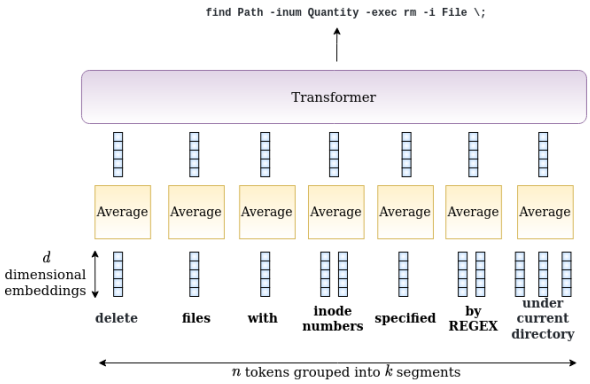

Figure 2: Segmented Invocation Transformer (SIT): We introduce an averaging layer to generate embeddings for each segment. These embeddings are then fed into a standard Transformer that outputs the bash command.

5 Experiments

Section 5.1 describes the experiments conducted to measure our method's accuracy. Section 5.2 describes the time benchmark. Section 5.3 describes the analysis using Integrated Gradients.

5.1 Translation Accuracy

Section 5.1.1 explains the accuracy metric proposed in the NLC2CMD competition. Section 5.1.2 lists the baselines our method is compared with. Section 5.1.3 lists the hyper-parameters and Section 5.1.4 contains a discussion of the results.

5.1.1 NLC2CMD Competition Metric

Agarwal et al. (2021) pointed out the shortcomings of existing evaluation metrics like BLEU score (Papineni et al., 2002), Exact Match accuracy and Template accuracy (Lin et al., 2018) in the context of natural language to Bash generation and proposed a new scoring mechanism for the NLC2CMD competition. This score incentivizes precision and recall of correct utility and flags weighted by the reported

[1] https://github.com/IBM/clai/tree/nlc2cmd/tellina-baseline/src/submission_code/nlp_tools

[2] https://github.com/IBM/clai/tree/nlc2cmd/utils/bashlint

system confidence (δ). It ignores command arguments but considers utilities' order and flags. The translation system is also penalized for producing redundant flags.

Now, we formally describe the competition metric[3] from Agarwal et al. (2021). Let model A output 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 consider $|A(nlc)| \leq 5$ and assume that there is only one ground truth command c corresponding to an invocation nlc. Then, the normalized score of a single prediction is:

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

Here, $\mathbb{I}(\cdot)$ is the indicator function, $U(c)$ 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 $N_i = \max\big(|F(U(c)_i)|, |F(U(\hat{c})_i)|\big)$.

Total score of the prediction is defined as:

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

5.1.2 Baselines

We compare our method with the following baselines:

- **T5** (Raffel et al., 2019): T5 is a Transformer-based model trained on large amount of data. We fine-tuned the T5-small checkpoint by huggingface (Wolf et al., 2020) on our dataset. The input to the model was "translate English to Bash:" followed by the invocation. T5-small and T5-base were tested. T5-small performed better. Results for the same are reported.

- **Code-T5** (Wang et al., 2021): CodeT5 is a T5 derivative proposed to improve the performance on both code understanding and code generation tasks. It is pre-trained on eight programming languages- Java, Ruby, Javascript, Go, PHP, Python, C and Cpp. We fine-tuned the CodeT5-small checkpoint by huggingface (Wolf et al., 2020) on our dataset. The input to the model was "translate English to Bash:" followed by the invocation. CodeT5-small and CodeT5-base were tested. CodeT5-small performed better. Results for the same are reported.

- **Seq2Seq** (Bahdanau et al., 2015): This is an attention enhanced encoder-decoder architecture with a bidirectional LSTM encoder and a unidirectional LSTM decoder.

- **Explainable-NL2BashAST** (Bharadwaj and Shevade, 2021): A natural language to Bash translation model that generates explanations besides Bash commands and uses Abstract Syntax Tree information. It also uses Bash utility description besides the parallel NL2Bash data. We use the code shared by the authors at https://www.github.com/Shikhar-S/Explainable-NL-to-Bash-AST.

- **Magnum** (Agarwal et al., 2021): This is the winner's model from the NLC2CMD contest and the state of the art on this problem. The original system is an ensemble of multiple Transformers (Vaswani et al., 2017) trained with different seeds and batch sizes. We compare with a single model from the ensemble for fair comparison.

5.1.3 Hyper-parameters

SIT uses an embedding dimension of 256 and has 3 encoder layers with 4 attention heads each and 6 decoder layers with 8 attention heads each. It has feed-forward networks with a dimension of 1024 in both encoder and decoder layers. It is trained with a batch size of 499 tokens and gradient accumulation over 150 batches. The height threshold for cutting the subtrees, set to 4, is tuned using the performance on the validation set. Magnum takes in an embedding vector of size 512 and has 6 layers in both encoder and decoder, each with 8 attention heads. Magnum is trained for 2500 steps, with each batch containing 14000 tokens with gradient accumulation over 2 batches and a warm-up scheduler. Seq2Seq has two 256 dimensional bidirectional LSTM layers in the encoder and two 256 dimensional LSTM layers in the decoder with attention between encoder and decoder. T5-small and

[3] https://github.com/IBM/clai/tree/nlc2cmd/utils/metric

Model	CPU Threads	Mean (sec)	Median (sec)	Interquartile Range (sec)
Magnum (Single Model)	1	145.70	144.51	1.98
SIT	1	59.07	59.22	1.87
Magnum (Single Model)	2	105.60	102.92	4.24
SIT	2	51.52	51.80	1.43
Magnum (Single Model)	3	94.57	92.17	2.26
SIT	3	47.63	47.62	1.82
Magnum (Single Model)	4	89.01	87.16	3.94
SIT	4	45.78	45.62	2.48
Magnum (Single Model)	5	86.73	86.01	0.72
SIT	5	46.49	46.36	2.35
Magnum (Single Model)	6	84.80	83.89	3.83
SIT	6	45.68	45.83	1.84

Table 2: Time Benchmarks: Time taken to run inference on 1K examples from the test set. Each entry is computed from 25 repeated runs. System configuration is mentioned in Section 5.2.

CodeT5-small are trained with a batch size of 32 examples, and the number of epochs is tuned based on the validation set performance. The unit of vocabulary for T5 and CodeT5 are subtokens, whereas, for all other models, the vocabulary is built from the whitespace-separated words in the NLC2CMD dataset. For all models, we use beam search with 10 beams and top-5 predictions to generate final Bash commands. We use 1 as the confidence score (δ) for top p predictions and $exp(beam\_score)/2$ for the remaining 5-p. This parameter p is tuned separately on the validation set for each model considered.

5.1.4 Results

Results are reported in Table 3. SIT performs better than all other models. T5 performs the worst, probably due to the large predefined vocabulary of standard T5 models. For every other model, the vocabulary is limited to the NLC2CMD dataset. The winners of NLC2CMD also report that a decrease in vocabulary size increases performance for this task. There is no straightforward way to adapt the vocabulary of T5 and CodeT5 for this dataset. Seq2Seq performs better than T5 and CodeT5. We hypothesize that this is due to the smaller target side vocabulary learned from the data. CodeT5, trained on a large amount of code and natural language data, performs better than T5 but still lags behind SIT. Explainable-NL2BashAST performs better than T5 and Sequence to Sequence but lags behind Magnum because it is developed for commands with a single utility, and it constructs a target sequence that is twice as long as a Bash command. This makes the decoding using beam search less efficient.

Model	Test score
T5 (Raffel et al., 2019)	0.316 ± 0.021
CodeT5 (Wang et al., 2021)	0.355 ± 0.025
Seq2Seq (Bahdanau et al., 2015)	0.362 ± 0.012
Explainable-NL2BashAST (Bharadwaj and Shevade, 2021)	0.390 ± 0.012
Magnum (Single Model) (Agarwal et al., 2021)	0.428 ± 0.010
SIT (Proposed Method)	$\mathbf{0.438 \pm 0.018}$

Table 3: NLC2CMD Competition metric on the test set. Values range from -1 to 1 with higher being better. All entries are averaged over 5 runs and in the form mean $\pm$ standard deviation.

Parameter Efficiency. SIT has 9M parameters, whereas Magnum has 45M parameters. This results from SIT's encoder being much smaller than Magnum's encoder, which is expected since we use the constituency parse tree information to model the natural language sequence. The 5x gain in parameter efficiency is especially important for this task since it will allow the proposed method to be employed in real-world systems with significantly less memory and power consumption.

5.2 Time Benchmark

Configuration. We consider a relatively inexpensive system configuration without access to a Graphics Processing Unit because we expect the users of our system to run it on a standard development machine. We benchmark the inference time for Magnum (Agarwal et al., 2021) and SIT (excluding constituency parsing) on the test set. The benchmarks are run on a 6 core Intel(R) Core(TM) i5-10400 CPU, using `torch.utils.benchmark` available in PyTorch (Paszke et al., 2019). We report results from 25 runs for each setting in Table 2. We benchmark a single model from the Magnum ensemble for

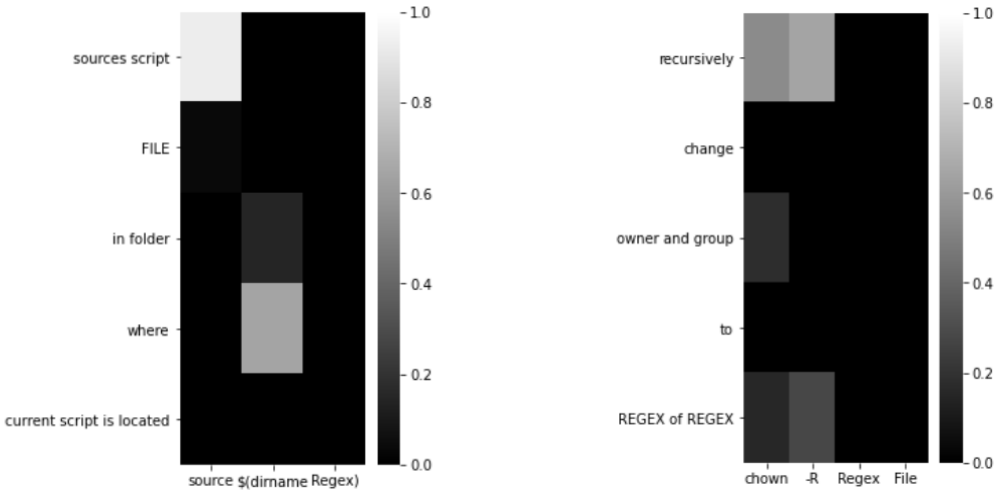

Figure 3: Alignments from Attribution Scores: Normalized invocation is on the left and normalized command is at the bottom.

comparison with a single SIT model.

Results. Results are reported in Table 2. We observe that SIT achieves a median time speedup of almost 1.8x over Magnum. These empirical results are in line with the complexity analysis from Section 3.2. On profiling the code, we find that a significant time spent during inference is to run the decoder, which is expected as the decoder runs in a step-wise fashion. Also, we note that the most time-consuming operations are computing self-attention and cross-attention matrices from keys, queries and values. As described in Section 3.2, SIT has fewer keys and values in the decoder cross-attention layer. Therefore, it leads to an increase in speed during inference.

5.3 Attribution Analysis

To assess if SIT attributes the probability of target tokens to correct invocation segment, we conduct an attribution analysis using Integrated Gradients (IG) (Sundararajan et al., 2017). [4] The IG method computes attribution scores that represent each invocation segment's contribution in predicting a command token. IG takes in the trained model, a baseline invocation and an input invocation as input. The baseline invocation denotes an absence of signal to the model. We use a sequence of `[PAD]` tokens, corresponding to a sequence of zero embedding vectors as the baseline. Integrated Gradients are defined as the path integral of the gradients along the straightline path from the baseline to the input. These are approximated by adding up the gradients along sufficiently small intervals on this straightline path. We used 5K steps for approximating the integral since the network is highly nonlinear.

We clipped the negative attribution scores to zero to draw attention to positive attributions that corresponds to alignments. Some resulting matrices from the test set are plotted as heatmaps and shown in Figure 3. The matrix on the left shows the alignment matrix for the input invocation *'Sources script incl.sh in folder where current script is located'*. The corresponding command is `source $(dirname $0)/incl.sh`. This is a Bash command substitution pattern. The inner command first finds the directory name of the directory containing the currently running script with the `dirname` utility. It then executes the *incl.sh* file in that directory with `source` utility. One can observe that the bash command `source` aligns with the invocation segment *sources script*, and the token `$(dirname` aligns with segments *in folder* and *where* from the invocation.

Similarly, Figure 3 (matrix on the right) shows the alignment matrix for input invocation *'Recursively change owner and group to "$JBOSS_AS_USER" of "$JBOSS_AS_DIR"'*, with the corresponding command `chown -R $JBOSS_AS_USER:$JBOSS_AS_USER $JBOSS_AS_DIR` . The alignment matrix depicts the correspondence between the invocation segment *recursively* and the command flag `-R`. In this instance, we also see that some command components are erroneously attributed. For instance

[4] We use the Integrated Gradient implementation provided by the Captum library - https://www.github.com/pytorch/captum

Command Component	Total Count	Attributed Correctly
Utility	129	94 (72.87%)
Flag	160	110 (68.75%)
Argument	244	84 (34.43%)

Table 4: Results from the evaluation of attributions produced by Integrated Gradients for 100 random examples from the test set.

`-R` is also attributed to *REGEX of REGEX*.

We sample 100 examples[5] from the test set and manually evaluate the attribution matrices produced by the IG method. Attribution for a command component is labelled as correct if one can look at only the positively attributed invocation segments to determine the presence of the command component in the output command. For instance, consider the attribution matrix on the right in Figure 3. Here, the counts of correctly attributed utilities, flags and arguments will be 0,1 and 0 respectively. Only the flag `-R` can be figured out from the positively attributed input segment *recursively*. There is a positive attribution on the invocation segment *owner and group*. However, in the absence of a positive attribution on *change*, one can not conclude that the utility `chown` would be used.

The results of the attribution analysis are shown in Table 4. We observe that utilities and flags have higher attribution accuracy than the arguments. This is due to the preprocessing which normalizes all file paths and regex expressions. It is also observed that sometimes multiple utilities are needed to perform the task in the invocation. There is little alignment between the utilities and invocation segments in such cases. For instance, when utilities like `sed` and `awk` are connected using the pipe operator (`|`) to modify the output of other utilities. Such implicit need for some utilities results in incorrect attribution by SIT. Similarly, it is observed that flags like `-and` and `-or` cannot be explicitly aligned to the invocation segments.

From the attribution analysis, we find that the proposed architecture is indeed able to capture the synchronous structure between natural language segments and Bash command components.

6 Related Work

Early works on semantic parsing explored meaning representations like first-order logic, lambda calculus enhanced first-order logic (Carpenter, 1997), database query languages and operated on handcrafted rules (Johnson, 1984). These were followed by statistical models that were able to learn rules from input-output parallel data (Thompson, 2003, Zettlemoyer and Collins, 2007, Kwiatkowksi et al., 2010).

Recently, there have been many advancements in generating high-level programming languages. Dong and Lapata (2016) and Ling et al. (2016) propose general attention based encoder-decoder style methods for semantic parsing. Rabinovich et al. (2017) propose Abstract Syntax Networks with a dynamically determined modular decoder structure that parallels the structure of the output tree. Yin and Neubig (2017) propose an architecture enhanced by a grammar model that explicitly captures the target language syntax as prior knowledge. Most of the innovations in this area utilize recurrent neural networks (RNN) for modelling natural language input. The method proposed in this work enhances the Transformer encoder with constituency parsing information.

For natural language to Bash, in particular, Lin et al. (2018) created a dataset and proposed an encoder-decoder based architecture. Gros (2019) explore several sequence to sequence models, Abstract Syntax Networks and Nearest Neighbor based models for this task on a custom dataset. Agarwal et al. (2020) proposed a command-line AI assistant for this task. Agarwal et al. (2021) organized a contest in NeurIPS 2020 for natural language to Bash translation and provided a report on the state of the art architectures developed in the contest. Bharadwaj and Shevade (2021) explored the use of Linux manual pages and Abstract Syntax Tree for developing an explainable natural language to Bash translation system. In contrast to these methods, our work explores the synchronous structure of this problem and uses the constituency tree to better model natural language input.

Constituency parse tree of natural language has been used in earlier machine translation and semantic parsing literature. Nguyen et al. (2019) note that the Transformer (Vaswani et al., 2017) struggles to encode hierarchical structures and propose a hierarchical accumulation mechanism that utilizes constituency parse tree to capture this structure for neural machine translation. They achieve this by adding additional parameters that capture the constituency structure of sentences. Our method, in

[5]Attribution matrices for the sampled instances are available at https://github.com/Shikhar-S/Segmented-Invocation-Transformer/blob/main/jup_notebook/attribution_viz.ipynb

contrast, uses constituency tree information in a parameter efficient manner. Xu et al. (2018) construct a syntactic graph from constituency and dependency parse tree and employ a graph to sequence neural network using an RNN decoder. They report improvement over the sequence to sequence model proposed by Dong and Lapata (2016) showing that additional syntactic information helps in semantic parsing. We use a Transformer and show that segmentation structure aids in natural language to Bash translation.

7 Conclusion and Future Work

We propose a method that utilizes information from the constituency tree to better model the structure of natural language to Bash task. Our experiments on the NLC2CMD data show that incorporating the problem structure in the model architecture improves both performance and parameter efficiency. We also run inference time benchmarks and find that the proposed method is faster. Attribution analysis is also conducted to analyze the method.

In this work, we focus on Bash as the meaning representation and identify the structure for natural language to Bash translation. However, we expect a similar structure for other meaning representations like SQL. Applying our method to natural language to SQL task is left for future work. Our method relies on Stanford CoreNLP parser (Manning et al., 2014) for constituency parsing. This is a bottleneck for fully utilizing the efficiency of our approach. It will be interesting to test faster constituency parsers (Zhang et al., 2020a, Zhang et al., 2020b) which can parse around 1K sentences per second.

References

Mayank Agarwal, Jorge J Barroso, Tathagata Chakraborti, Eli M Dow, Kshitij Fadnis, Borja Godoy, Madhavan Pallan, and Kartik Talamadupula. 2020. Project clai: Instrumenting the command line as a new environment for ai agents. *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. In *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, volume 133 of *Proceedings of Machine Learning Research*, pages 302–324. PMLR.

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*.

Shikhar Bharadwaj and Shirish Shevade. 2021. Explainable natural language to bash translation using abstract syntax tree. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 258–267, Online. Association for Computational Linguistics.

Bob Carpenter. 1997. *Type-logical semantics*. MIT press.

Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124.

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)*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

David Gros. 2019. Ainix: An open platform for natural language interfaces to shell commands. *Undergraduate Honors Thesis*.

Tim Johnson. 1984. Natural language computing: the commercial applications. *The Knowledge Engineering Review*, 1(3):11–23.

Aishwarya Kamath and Rajarshi Das. 2019. A survey on semantic parsing. In *Automated Knowledge Base Construction (AKBC)*.

Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to sql: Where are we today? *Proc. VLDB Endow.*, 13(10):1737–1750.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 1223–1233.

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).

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany. Association for Computational Linguistics.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. 2019. Tree-structured attention with hierarchical accumulation. In *International Conference on Learning Representations*.

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, page 311–318, USA. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149.

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.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 3319–3328. JMLR.org.

Cynthia Thompson. 2003. Acquiring word-meaning mappings for natural language interfaces. *Journal of Artificial Intelligence Research*, 18:1–44.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi 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 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.

Kun Xu, Lingfei Wu, Zhiguo Wang, Mo Yu, Liwei Chen, and Vadim Sheinin. 2018. Exploiting rich syntactic information for semantic parsing with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 918–924.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA).

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, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450. Association for Computational Linguistics.

Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 678–687.

Yu Zhang, Zhenghua Li, and Min Zhang. 2020a. Efficient second-order TreeCRF for neural dependency parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3295–3305, Online. Association for Computational Linguistics.

Yu Zhang, Houquan Zhou, and Zhenghua Li. 2020b. Fast and accurate neural CRF constituency parsing. In *Proceedings of IJCAI*, pages 4046–4053.