Empirical Study of Transformers for Source Code

Nadezhda Chirkova,[1] **Sergey Troshin,** [1]

[1] Samsung-HSE Laboratory, National Research University Higher School of Economics, Moscow, Russia
{nchirkova, stroshin}@hse.ru

Abstract

Initially developed for natural language processing (NLP), Transformers are now widely used for source code processing, due to the format similarity between source code and text. In contrast to natural language, source code is strictly structured, i. e. follows the syntax of the programming language. Several recent works develop Transformer modifications for capturing syntactic information in source code. The drawback of these works is that they do not compare to each other and all consider different tasks. In this work, we conduct a thorough empirical study of the capabilities of Transformers to utilize syntactic information in different tasks. We consider three tasks (code completion, function naming and bug fixing) and re-implement different syntax-capturing modifications in a unified framework. We show that Transformers are able to make meaningful predictions based purely on syntactic information and underline the best practices of taking the syntactic information into account for improving the performance of the model.

1 Introduction

Transformer (Vaswani et al. 2017) is currently a state-of-the-art architecture in a lot of source code processing tasks, including code completion (Kim et al. 2020), code translation (Lachaux et al. 2020; Shiv and Quirk 2019), and bug fixing (Hellendoorn et al. 2020). Particularly, Transformers were shown to outperform classic deep learning architectures, e. g. recurrent (RNNs), recursive and convolutional neural networks in the mentioned tasks. These architectures focus on *local* connections between input elements while Transformer processes all input elements in parallel and focuses on capturing *global* dependencies in data, producing more meaningful code representations (Hellendoorn et al. 2020). Parallelism also speeds up training and prediction.

Transformer is often applied to source code straightforwardly, treating code as a sequence of language keywords, punctuation and variable names. In this case, a neural network mostly relies on variable names to make predictions (Ahmad et al. 2020). High-quality variable names can be a rich source of information about the semantics of the code; however, this is only the indirect, secondary source of information. The primary source of information of what the code implements is its syntactic structure.

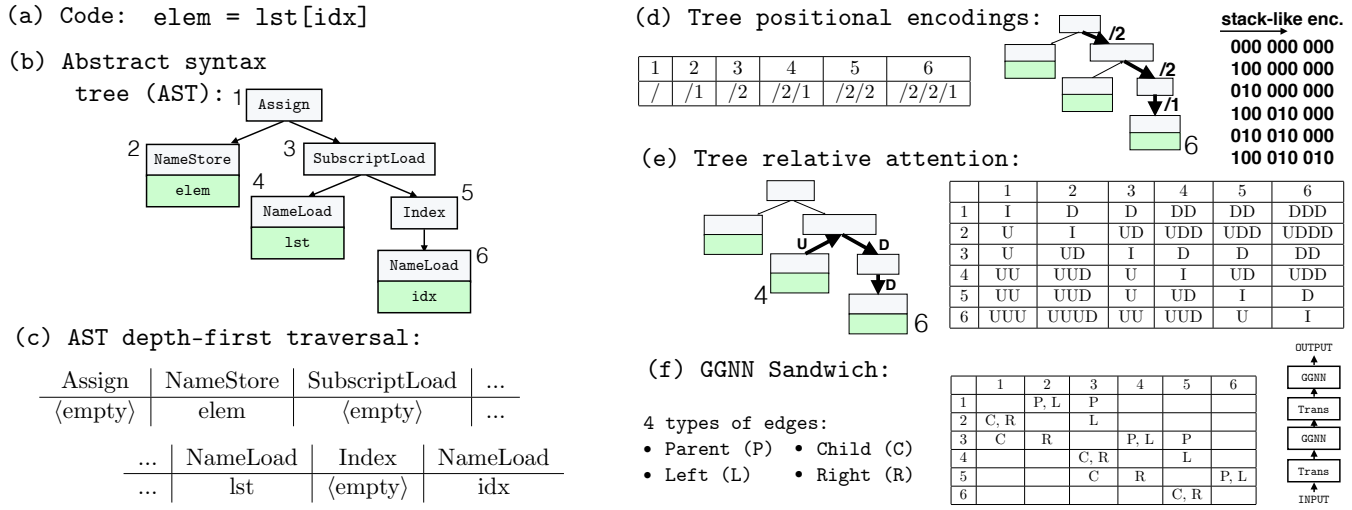
Transformer architecture relies on the self-attention mechanism that is not aware of the order or structure of input elements and treats the input as an unordered *bag* of elements. To account for the particular structure of the input, e. g. sequential structure, additional mechanisms are usually used, e. g .positional encodings. In recent years, a line of works has developed mechanisms for utilizing tree structure of code in Transformer (Shiv and Quirk 2019; Kim et al. 2020; Hellendoorn et al. 2020). However, what is the most effective way of utilizing syntactic information in Transformer is still unclear due to two reasons. Firstly, the mentioned mechanisms were developed concurrently, so they were not compared to each other by their authors. Additionally, all these mechanisms were tested on different code processing tasks, so empirical results in the papers cannot be aligned. Secondly, the mentioned works used standard Transformer with positional encodings as a baseline, while modern practice implies using more advanced modifications of Transformer, e. g. equipping it with relative attention. As a result, it is unclear whether using sophisticated mechanisms for utilizing syntactic information is needed at all.

In this work, we conduct a thorough empirical study of using Transformers for processing source code, pursuing a goal of underlining best practices for utilizing syntactic information in Transformer. Our contributions are as follows:

- We re-implement several approaches for capturing syntactic information in Transformers and investigate their effectiveness in three code processing tasks. We underline the importance of evaluating code processing models on several different tasks and believe that our work will help to establish standard benchmarks in the field of neural code processing.
- We introduce an anonymized setting in which all user-defined variable names are replaced with placeholders, and show that Transformer is capable of making meaningful predictions based purely on syntactic information, in all three tasks. We also show that using the proposed anonymization can improve the quality of the model, either single Transformer or an ensemble of Transformers.

We will release all our source code with the final version of the paper.

The rest of the work is organized as follows. In section 2 we give a literature review of the existing approaches for utilizing syntactic information in Transformers. In section 3 we describe our methodology for the empirical evaluation



	1	2	3	4	5	6
1	I	D	D	DD	DD	DDD
2	U	I	UD	UDD	UDD	UDDD
3	U	UD	I	D	D	DD
4	UU	UUD	U	I	UD	UDD
5	UU	UUD	U	UD	I	D
6	UUU	UUUD	UU	UUD	U	I

	1	2	3	4	5	6
1		P, L	P			
2	C, R		L			
3	C	R		P, L	P	
4			C, R		L	
5			C	R		P, L
6					C, R	

Figure 1: Illustration of approaches for processing AST in Transformer.

of Transformer capabilities to utilize syntactic information. The following sections 5–8 describe our empirical findings.

2 Review of Transformers for Source code

Abstract syntax tree. A syntactic structure of code is usually represented in the form of abstract syntax tree (AST). Each node of the tree contains type, which represents the part of the syntax (e.g. "Assign", "Index", "NameLoad"), some nodes also contain a value (e.g. "idx", "elem"). An example AST for a code snippet is shown in figure 1(b).

2.1 Transformer architecture and self-attention mechanism

We describe Transformer architecture for a task of mapping input sequence $c_1, \dots c_L$, $c_i \in \{1, \dots, M\}$ (M is a vocabulary size) to a sequence of d_{model}-dimensional representations $y_1, \dots, y_L$ that can be used for making task-specific predictions in various tasks. Before being passed to the Transformer blocks, the input sequence is firstly mapped into a sequence of embeddings $x_1, \dots, x_L$, $x_i \in \mathbb{R}^{d_{model}}$.

A key ingredient of Transformer block is a self-attention layer that maps input sequence $x_1, \dots x_L$, $x_i \in \mathbb{R}^{d_{model}}$ to a sequence of the same length: $z_1, \dots, z_L$, $z_i \in \mathbb{R}^{d_z}$. Self-attention firstly computes key, query and value vectors from each input vector: $x_j^k = x_j W^K$, $x_j^q = x_j W^Q$ and $x_j^v = x_j W^V$. Each output z_i is computed as a weighted combination of inputs:

$$z_i = \sum_j \tilde{\alpha}_{ij} x_j^v, \quad \tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}, \quad a_{ij} = \frac{x_i^q {x_j^k}^T}{\sqrt{d_z}} \tag{1}$$

Attention weights $\tilde{\alpha}_{ij} \geqslant 0$, $\sum_{j=1}^{L} \alpha_{ij} = 1$ are computed based on query-key similarities. Several attention layers (heads) are applied in parallel with different projection matrices W_h^V, W_h^Q, W_h^K, $h = 1, \dots, H$. The outputs are concatenated and projected to obtain $\hat{x}_i = [z_i^1, \dots, z_i^H] W^O$, $W^O \in \mathbb{R}^{H d_z \times d_{model}}$. One Transformer block includes the described multi-head attention, a residual connection, a layer normalization and a position-wise fully-connected layer. The overall Transformer architecture is composed by consequent stacking of the described blocks. During training on generation tasks, i. e. in the Transformer *decoder*, future elements ($i > j$) are masked in self-attention.

2.2 Passing ASTs to Transformers

Transformers were initially developed for NLP and therefore were augmented with sequence-capturing mechanisms to account for sequential input structure. As a result, the simplest way of applying Transformers to AST is to traverse AST in some order, e. g. in depth-first order (see figure 1(c)), and use standard sequence-capturing mechanisms.

Sequential positional encodings and embeddings. To account for the sequential nature of the input, standard Transformer is augmented with positional encodings or positional embeddings. Namely, the input embeddings $x_i \in R^{d_{model}}$ are summed with positional representations $p_i \in R^{d_{model}}$: $\hat{x}_i = x_i + p_i$. For example, positional embeddings imply learning an embedding vector of each position $i \in 1 \dots L$: $p_i = e_i$, $e_i \in R^{d_{model}}$. Positional encodings imply computing p_i based on sine and cosine functions. We include positional embeddings in our comparisons, since they performed a bit better than positional encoding in our preliminary experiments, and add prefix "sequential" to the title of this mechanism. The described approach was used as a baseline in a bunch of works (Shiv and Quirk 2019; Hellendoorn et al. 2020).

Sequential relative attention. Shaw, Uszkoreit, and Vaswani (2018) proposed relative attention for capturing the order of the input elements. They augment self-attention with relative embeddings:

$$z_i = \sum_j \tilde{\alpha}_{ij} (x_j^v + e_{ij}^V), \quad a_{ij} = \frac{x_i^q (x_j^k + e_{ij}^K)^T}{\sqrt{d_z}}, \tag{2}$$

where $e^V_{ij}, e^K_{ij} \in \mathbb{R}^{d_z}$ are learned embeddings for each relative position $i - j$. This mechanism that we call *sequential relative attention* was shown by the authors to substantially outperform sequential positional embeddings and encodings in sequence-based text processing tasks. Ahmad et al. (2020) make the same conclusion evaluating sequential relative attention in a task of code summarization, i. e. generating natural language summaries for code snippets.

Tree positional encodings. A bunch of works got inspiration in how Transformers process sequences to develop mechanisms for processing trees. Shiv and Quirk (2019) develop positional encodings for tree-structured data, assuming that the maximum number n_w of node children and the maximum depth n_d of the tree are relatively small. Example encodings are given in figure 1(d). The *position* of each node in a tree is defined by its path from the root, and each child number in the path is encoded using n_w-sized one-hot vector. The overall representation of a node is obtained by concatenating these one-hot vectors in a reverse order and padding short paths with zeros from the right. Paths longer than n_d are clipped (root node is clipped first). The authors binarize ASTs to achieve $n_w = 2$. To avoid it, we replace all child numbers greater than n_w with n_w.

The authors tested the approach on the task of code translation (code-to-code) and semantic parsing (text-to-code). The Transformer with tree positional encodings outperformed standard Transformer with sequential positional encodings and TreeLSTM.

Tree relative attention. An extension of sequential relative attention for trees was proposed by Kim et al. (2020). In a sequence, the distance between two input positions is defined as the number of positions between them. Similarly, in a tree, the distance between two nodes can be defined as the shortest path between nodes, consisting of $n_U \geqslant 0$ steps *up* and $n_D \geqslant 0$ steps *down*, see example in figure 1(e). Now, similarly to sequential relative attention, we can learn embeddings for the described distances and plug them in self-attention. Learning multidimensional embeddings for the tree input requires much more memory than for the sequential input, since distances in the tree are object-specific, while distances in the sequence are the same for all objects in a mini-batch. As a result, the authors use scalar embedding $r_{ij} \in \mathbb{R}$ for the distance between nodes i and j and plug it in attention mechanism as follows:

$$\tilde{\alpha}_{ij} = \frac{\exp(a_{ij} \cdot r_{ij})}{\sum_j \exp(a_{ij} \cdot r_{ij})}. \quad (3)$$

Our preliminary experiments suggested that using summation $\alpha_{ij} + r_{ij}$ instead of multiplication leads to a higher final score. The authors tested the approach on the task of code completion, i. e. predicting the next token, and was shown to outperform standard Transformer applied to AST traversal and to code as text.

GGNN Sandwich. Due to the graph nature of AST, source codes are often processed using graph gated neural networks (GGNN) (Allamanis, Brockschmidt, and Khademi 2018). To add more inductive bias, AST is augmented with edges of several additional types, e. g. reflecting data- and control-flow in the program. Such model captures well *local* dependencies in data but lacks a *global* view on the input program, that is the Transformer's forte. Inspired by this reasoning, Hellendoorn et al. (2020) propose alternating Transformer and GGNN layers as illustrated in figure 1(f), to combine the positive sides of both models. GGNN layer relies on passing messages though edges for a fixed number of iterations (number of passes). The model is called GGNN sandwich by the authors, the details can be found in (Hellendoorn et al. 2020). The approach was tested on the variable misuse detection task, i. e. predicting a location with a bug and a location used to fix the bug (copy variable). GGNN Sandwich outperformed standard Transformer with sequential positional encodings. In our experiments, we focus on syntactic edges and do not use data- or control-flow edges.

3 Limitations of existing approaches and methodology of the work

As shown in section 2, during the last year, community came up with several approaches for processing ASTs in Transformers. However, it is still unclear what approaches perform better than others and what techniques to use in practice. Firstly, all works discussed in section 2 conduct experiments with different tasks making it impossible to align the results. Moreover, almost all listed works compare their approaches with vanilla Transformer, i. e. Transformer with sequential positional encodings or embeddings, while modern practices imply using advanced techniques like sequential relative attention by default. Even works that propose tree-processing approaches inspired by sequential relative attention (Kim et al. 2020; Hellendoorn et al. 2020) do not include this technique as a baseline. That is, it is unclear whether using advanced tree-processing approaches is beneficial at all. Secondly, the existing approaches focus on capturing tree *structure* and do not investigate the influence of other components of AST, i. e. types and values.

In this work, we conduct a thorough empirical study on utilizing AST in Transformers. We consider three code processing tasks, variable misuse detection, function naming and code completion. We re-implement all approaches described in section 2 in a unified framework and investigate what is the most effective approach for processing ASTs in Transformer in different tasks. We aim to answer the following research questions:

- Are Transformers generally capable of utilizing syntactic information represented via AST?
- What components of AST (structure, node types and values) do Transformers use in different tasks?
- What is the most effective approach for utilizing AST *structure* in Transformer?

There is no common practice of preprocessing ASTs, particularly, processing values: each node in AST is associated with a type, but not all nodes have associated values. Kim et al. (2020) and Shiv and Quirk (2019) attach values as separate child nodes so that each node stores only one item (type or value), while Hellendoorn et al. (2020) propose omitting types. The former approach increases input length and so

makes code processing significantly slower while the latter approach loses type information. We choose the in-between strategy inspired by the approach of (Li et al. 2018) used for RNNs: we associate <EMPTY> values with nodes that do note have values, so that each node i in AST has both type t_i and value v_i, see figure 1(c). This setup allows easy ablating types or values, leaving another item in each node present.

An important part of our methodology is conducting experiments in two settings, namely *anonymized* and *non-anonymized*. The non-anonymized setting corresponds to the conventional training of Transformer on ASTs parsed from code, referred to as *full data* below. In this case, Transformer has two sources of information about input code snippet: syntactic information and user-defined variable names (stored in node values). Variable names usually give much *additional* information about the semantics of the code, however, their presence is not necessary for correct code execution: renaming all user-defined variable names to placeholders `var1`, `var2`, `var3`[1] etc. will lead to the same result of code execution and will not change the semantics of what algorithm the code implements. We call the described procedure of renaming variables with placeholders as *anonymization*. In the anonymized setting, the input code is represented purely with syntax structure and the only way Transformer can make meaningful predictions is to capture information from AST. In this way, using anonymized setting allows better understanding of the capabilities of Transformer to utilize syntactic information. The general idea of testing models on variable-free data was introduced in (LeClair, Jiang, and McMillan 2019) for RNNs, however, they did not consider anonymizaton and removed values completely, loosing much information, and considered only code summarization task.

Another important part of our methodology is a thoughtful splitting of the dataset into training and testing sets that includes splitting by repository and removing code duplicates. Splitting by repository implies that all files from one repository should always be put into one set, either training or testing one, to avoid data leak (Alon et al. 2019; LeClair, Jiang, and McMillan 2019). For the deduplication. i. e. removing testing code present in the training set that can bias evaluation results, we use the resources provided by Allamanis (2019).

4 Experimental setup

Data. In all tasks, we use Python150k (Raychev, Bielik, and Vechev 2016b) dataset that is commonly used to evaluate code processing models. The dataset includes 150K files collected from GitHub. Most of the works use train-test split provided by the authors of the dataset, however, this split does not follow best practices described in section 3 and produce biased results (Allamanis 2019), so we release a new split of the dataset. We split *repositories* between training and testing sets in approx. proportion 2:1 (both lists of repositories can be found in Appendix A) and also use deduplication procedure of Allamanis (2019). We describe the details of data preprocessing for different tasks in Appendix A.

Variable misuse task (VM). For the variable misuse task, we borrow the setup and evaluation strategy of (Hellendoorn et al. 2020). Given the code of a function, the task is to output two positions (using two pointers): in what position a wrong variable is used and which position a correct variable can be copied from (any such position is accepted). If a snippet is non-buggy, the first pointer should select a special no-bug position. We obtain two pointers by applying two position-wise fully-connected layers and softmax over positions on top of Transformer outputs. We use the joint localization and repair accuracy metric of (Hellendoorn et al. 2020) to assess the quality of the model. This metric estimates the portion of buggy samples for which the model correctly localizes and repairs the bug. We also measured localization accuracy and repair accuracy independently and found that all three metrics correlate well with each other. The dataset for the task is obtained by injecting synthetic variable misuse bugs in functions.

Function naming task (FN). In this task, given the code of a function, the task is to predict the name of the function. To solve this task, we use classic sequence-to-sequence Transformer architecture (Ahmad et al. 2020) that outputs function name word by word. We assess the quality of generated function names using F1-metric. If gtn is a set of words in ground-truth function name and pn is a set of words in predicted function name, F1-metric is computed as $2PR/(P+R) \in [0,1]$, where $P = |gtn \cap pn|/|pn|$, $R = |gtn \cap pn|/|gtn|$, $|\cdot|$ denotes the number of elements. F1 is averaged over objects. We choose F1 metric following (Alon et al. 2019) who solved the similar task with another dataset and model. A mean length of function name in the preprocessed dataset is 2.43 words.

Code completion task (CC). For the task of code completion, we borrow the setup, metrics and Transformer implementation of (Kim et al. 2020). The task is to predict the next node (t_i, v_i) in the depth-first traversal of AST $[(t_1, v_1), \ldots, (t_{i-1}, v_{i-1})]$. Due to space limitation, we mostly focus on value prediction and present results for type prediction where they significantly differ from other tasks. We optimize the sum of cross-entropy losses for types and values. We use mean reciprocal rank (MRR) to measure the quality of the model since it reflects the practical application of code completion: $\mathrm{MRR}(rank) = n^{-1} \sum_{i=1}^{L} 1/rank_i$, where $rank_i$ is a position of the true token in the model ranking. As in (Kim et al. 2020), we assign zero score if the true token is not in the top 10 predicted tokens. During data preprocessing, we split AST into overlaping chunks and do not calculate loss or metrics over the intersection twice.

Hyperparameters. The training details and hyperparameters are listed in the Appendix A. All models are trained via cross entropy optimization using Adam. All models were trained three times. For sequential relative attention we use the maximum distance between elements of 32. For tree relative attention, we limit the number of relations to 600 most frequent ones. For tree encodings we use $n_w = 4$, $n_d = 32$ / 8 depending on the task. GGNN sandwich model consists of 3 Transformer and 3 GGNN layers (each with

[1] All occurrences of a variable are replaced with the same placeholder.

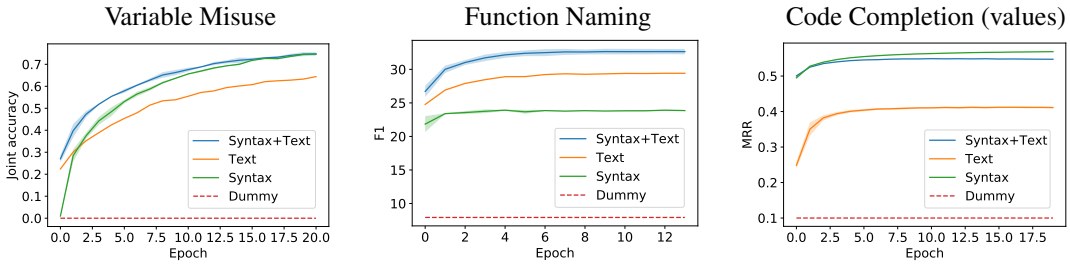


Figure 2: Comparison on syntax-based Transformer models with text-only and constant (dummy) baselines.

```
def get_or_create_bucket(s3_connection):
    bucket = s3_connection.get_bucket(
        settings.S3_BUCKET_NAME
    )
    if bucket is None:
        bucket = s3_connection.
```

(a) **S+T**: [create_bucket get_bucket get_key]
S: [get_bucket S3_BUCKET_NAME var107]
T: [bucket get_bucket settings]
Dummy: [self 0 None]
Gold: create_bucket

```
import sys
from utils.parsing import parse
if __name__ == '__main__':
    argmnents = parse(
        sys.argv[1:]
    )
    print(
```

(b) **S+T**: [Usage: done. \n]
S: [argmnents var440 var289]
T: [sys outfn len]
Dummy: [self 0 None]
Gold: argmnents

```
def <fun_name>(seqs):
    dt = {}
    for seq in seqs:
        for word in seq.split():
            if not word in dt:
                dt[word] = 1
            else:
                dt[word] += 1
    return dt
```

(c) **S+T**: get dict
S: get all
T: get split
Dummy: init
Gold: get dictionary

```
def <fun_name>(filename):
    with open(filename) as fin:
        return len(
            fin.read()
                .split("\n")
        )
```

(d) **S+T**: read file
S: read
T: read file
Dummy: init
Gold: count lines

Figure 3: Example predictions in code completion task (a, b) and function naming task (c, d). **S**: Syntax, **T**: Text.

4 message passes), alternating with each other so that the total number of layers is the same to that of other models. We use edges of 3 types: AST edges, sequential edges connecting neighboring nodes in the AST depth-first traversal, and self-loops. The number of parameters in all AST-based modifications of Transformer are approximately the same.

5 Capability of Transformers to utilize syntactic information

We begin with investigating whether Transformers are generally capable of capturing the syntactic information in source code. We formalize this question in anonymized and non-anonymized settings as follows:

- *Syntax+Text* vs. *Text*: Firstly, we test whether using syntactic information in addition to the text representation of a program is beneficial compared to using the text representation of the program alone. In other words, we compare the quality of Transformer trained on full AST data with the quality of Transformer trained on a sequence of non-empty values.
- *Syntax* vs. *Dummy*: Secondly, we test whether Transformer is able to make meaningful predictions given *only* syntactic information. In other words, we test whether the quality of Transformer trained on *anonymized* AST data is higher than the quality of a simple constant baseline (most frequent target). Since anonymized AST data contains only syntactic information and does not contain text information, the only way the Transformer can outperform dummy baseline on this data is to capture some information from AST.

All described models are trained with sequential positional embeddings. Using more sophisticated AST-capturing techniques would strengthen models *Syntax* and *Syntax+Text* and make it even easier for them to outperform *Text* and *Dummy* models. Deduplicating dataset is highly important in this experiment to avoid overestimating *Syntax* baseline.

The results for three tasks are presented in figure 2. In all cases, *Syntax+Text* outperforms *Text*, and *Syntax* outperforms *Dummy*. In figure 3, we present example predictions for code completion and function naming tasks with all four models. In code completion, syntax-based models well capture frequent coding patterns, for example, in (b) *Syntax* model correctly chooses (anonymized) value `argmnents` and not `argv` or `parse` because `argmnents` goes before assignment. In example (a), *Syntax+Text* correctly predicts `create_bucket` because it goes inside if-statement checking whether the bucket exists, while *Text* model outputs frequent tokens associated with buckets.

In function naming task, *Text* model captures code semantics based on variable names and sometimes selects wrong "anchor" variables, e. g. `split` in example (c), while *Syntax+Text* model utilizes AST information and outputs correct word `dict`. *Syntax* model does not have access to variable names as it processes data with placeholders, and outputs overly general predictions, e. g. `get all` in example (c). Nevertheless, *Syntax* model is capable of distinguishing general code purpose, e. g. model uses word `get` in example (c) and word `read` in example (d).

Interestingly, in code completion (value prediction) and variable misuse detection tasks, *Syntax* model trained on the anonymized data performs on par with or even marginally outperforms *Syntax+Text* model trained on full data, though the latter uses more data than the former. The reason is that

	Full data			Anonymized data		
	Var. misuse	Fun. naming	Comp. (val.)	Var. misuse	Fun. naming	Comp. (val.)
Full AST	**74.59** %	**32.83**$\pm$ 0.35%	**54.89** $\pm$ 0.07%	**74.71**%	**23.92** $\pm$ 0.004 %	**56.87** $\pm$ 0.09%
AST w/o struct.	26.1%	**33.0** $\pm$ 0.08 %	54.2 $\pm$ 0.1%	12.12%	23.28 $\pm$ 0.02%	55.07 $\pm$ 0.05%
AST w/o an.val.	N/A	N/A	N/A	32.1%	**24.00** $\pm$ 0.15%	N/A

Table 1: Ablation study of syntax-capturing mechanisms in Transformer. Bold emphasises best models and ablations that do not hurt the performance. AST w/o struct.: Transformer treats input as a bag without structure; AST w/o an.val.: only types are passed to Transformer. N/A – not applicable.

the values vocabulary on full data is limited, so approx. 25% values are replaced with `UNK` token and cannot be predicted correctly, which is not a complication for *Syntax* model. For example, in figure 3(b), *Syntax* model correctly predicts misspelled token `argmnents` while for *Syntax+Text* model, this token is out-of-vocabulary and so model outputs frequently used strings for printing. One more advantage of *Syntax* model in value prediction task is 2.5 times speed-up in training because of small output softmax dimension. In function naming task, *Syntax* model performs significantly worse than *Syntax+Text*, because variable names provide much *natural language* information needed to make natural language predictions.

6 Ablation study of mechanisms for utilizing syntactic information in Transformer

In the previous experiment, we found that taking syntactic information (AST) into account in Transformer is beneficial compared to the same model without syntactic information. As discussed in section 3, we use three syntax-capturing mechanisms in Transformer: (*types*) passing types of nodes as one of the Transformer inputs; (*structure*) processing AST structure using one of the mechanisms discussed in section 2; (*an. values*) anonymizing values – in case we do not use values from the data. Now we investigate, what mechanisms are essential for achieving the benefit in quality and what can be ablated without quality drop.

As in section 5, we consider both anonymized and non-anonymized setting. We ablate syntax-capturing techniques one by one in both models *Syntax* and *Syntax+Text* and check, whether the quality drops. Ablating *types* was in fact performed in section 5. Ablating *structure* means turning off sequential positional embeddings so that the input of the *Syntax+Text* / *Syntax* model will be viewed as a *bag* of (type, value) / (type, anonymized value) pairs. Ablating *an. values* means using only types as the input to the model; we consider this ablation only in the anonymized setting. We skip this ablation in the code completion task, since in this case, anonymized values are the target of the model.

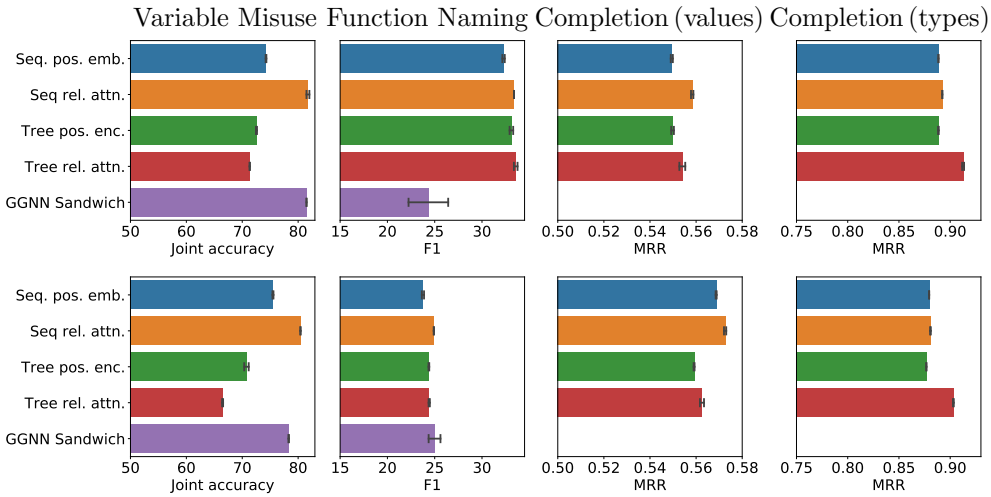
The results are presented in table 1. Ablating *structure* leads to a worse quality in all cases except function naming task in the non-anonymized setting. Summarizing this result with the results of section 5, we conclude that both *types* and *structure* contribute to the superiority of *Syntax+Text* over *Text* model in variable misuse and code completion tasks, while in function naming task, only types contribute.

Using value anonymization (instead of removing values) in *Syntax* model is important not in all tasks. In variable misuse task, ablating *an. values* leads to a substantial drop in quality. This is an interpretable result: knowing whether to nodes store similar value is important for predicting a node with a wrong value. In contrast, in function naming task, ablating *an. values* does not change quality. This means that *Syntax* model sees code as such in which all variables are replaced with `<var>` — such code would be hard to understand for human. As we saw in section 5, such predictions may be to some extent meaningful and *reflect the general semantics* of code. However, the described result means that Transformers are now far from understanding the semantics of the code based on the rules of the programming language.

7 Comparison of approaches for utilizing syntactic structure in Transformer

Our next step is to investigate what of the approaches for utilizing *structure* information in Transformer described in section 2 is the most effective one. All models considered in this section are variants of either *Syntax+Text* of *Syntax* model. GGNN Sandwich is not applicable to the code completion task because message-passing involves all nodes and prohibits using masking in decoder.

The results are presented in figure 4 (left). Utilizing information about structure in the input embeddings is not effective: sequential positional embeddings and tree positional encodings perform worse than other models in all tasks, except function naming on full data where tree positional encodings perform on par with other methods. Utilizing structure in self-attention mechanism is much more effective: in all tasks at least one of sequential relative attention and tree relative attention is the best performing model. Sequential relative attention achieves the highest score in variable misuse and value prediction tasks, while tree relative attention outperforms others with a high margin in type prediction task. It is an interpretable result since tree relative attention helps to find relatives in AST tree, e. g. parent and siblings, that is important in type prediction. GGNN Sandwich achieves high results in variable misuse task but perform poorly in function naming task. The reason is that in variable misuse detection, the goal is to choose two variables, and *local* message passing informs each variable of its role in a program and makes variable representations more meaningful. We notice that the high quality of GGNN Sandwich model is achieved only when using two types of edges (AST edges and sequential edges), using only AST edges lead to a slightly worse quality. In function naming task, the goal



	Train time (h/ep)	Add. tr. data size
Seq. pos.	2.3	0
Tree. pos.	2.5	0.3
Seq. rel.	2.7	0
Tree rel.	3.9	18
GGNN sand.	7.2	0.35

Figure 4: A comparison of different AST-based Transformer modifications in the non-anonymized- (top left) and anonymized setting (bottom left). Time- and storage-consumption of different AST-capturing techniques for variable misuse task (right).

is to capture *global* program semantics and so inserting *local* GGNN layers reduces quality. An interested reader may find examples of attention maps for different AST-capturing techniques in Appendix B.

In figure 4 (right), we list training time and the size of additional training data needed for AST-capturing mechanism for different models. GGNN Sandwich model requires twice time for training (and prediction) compared to other models, because of time-consuming message passing. Tree relative attention requires dozens of gigabytes for storing pairwise relation matrices for all training objects. Tree positional encodings and GGNN sandwich models also require additional disk space for storing preprocessed graph representations, but the size of these files is not so large. Sequential positional embeddings and relative attention are the most efficient models, in both time- and disk-consumption aspects.

To sum up, we emphasise sequential relative attention as the most effective and efficient approach for capturing AST structure in Transformer.

Combining AST-capturing techniques. In table 2, we show that although sophisticated AST-capturing techniques game away in solo comparison, they may be useful for further improving sequential relative attention in *combination* with it. We find that tree relative attention improves the score in value prediction task, while GGNN Sandwich improves the score in variable misuse task.

8 Ensembling of syntax-based models

As was shown in figure 3, *Syntax+Text* and *Syntax* models capture dependencies of different nature and are orthogonal in a sense of handling missing values and first time seen tokens. This allows hypothesizing that *ensembling* two mentioned models can boost the performance of the Transformer. We use the standard ensembling approach that implies training networks from different random initialization and averaging their predictions after softmax (Ashukha et al. 2020), and ensemble Transformers trained with sequential relative attention. In table 3 we compare an ensemble of *Syntax+Text* and *Syntax* models with ensembles of two *Syntax+Text* models. We observe that in variable misuse and value prediction tasks, ensembling models that view input data in two completely different formats is much more effective than ensembling two similar models. This is the way how using anonymized data may boost the Transformer's performance.

Model	VM	FN	CC (val.)
SRA	81.54 %	33.38 %	55.79
SRA + Seq. pos. emb.	80.31 %	31.78 %	55.73
SRA + Tree rel. attn.	80.59 %	33.01 %	**56.26**
SRA + Tree pos. enc.	81.74 %	32.52 %	55.98
SRA + GGNN sand.	**84.14** %	27.67 %	N/A

Table 2: Comparison of combinations of sequential relative attention (SRA) with other AST-capturing approaches, non-anonymized setting. Standard deviations: VM: 0.5%, FN: 0.4%, CC: 0.1%.

Models	VM	FN	CC (typ.)	CC (val.)
ST	81.54 %	**33.38**%	89.1 %	55.79 %
ST & ST	83.15 %	32.57 %	**89.4** %	56.86 %
ST & S	**85.73**%	29.42%	88.51 %	**59.87** %

Table 3: Comparison of ensembles. Notation: ST – *Syntax+Text*, S – *Syntax*, & denotes ensembling.

9 Conclusion

In this work, we showed that (a) Transformer is capable of making meaningful predictions in code processing tasks based purely on AST, with anonymized variable names; (b) training Transformer purely on AST can be useful for achieving high quality either of a single model or an ensemble; (c) a widely used in NLP sequential relative atten-

tion is the most promising technique for utilizing AST in source code processing that outperforms approaches developed specifically for AST.

Acknowledgments

This research has been supported by Samsung Research, Samsung Electronics and through computational resources of HPC facilities at NRU HSE.

References

Ahmad, W. U.; Chakraborty, S.; Ray, B.; and Chang, K.-W. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Allamanis, M. 2019. The adverse effects of code duplication in machine learning models of code. *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* .

Allamanis, M.; Brockschmidt, M.; and Khademi, M. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. URL https://openreview.net/forum?id=BJOFETxR-.

Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL https://openreview.net/forum?id=H1gKYo09tX.

Ashukha, A.; Lyzhov, A.; Molchanov, D.; and Vetrov, D. 2020. Pitfalls of In-Domain Uncertainty Estimation and Ensembling in Deep Learning. In *International Conference on Learning Representations, ICLR 2020*. URL https://openreview.net/forum?id=BJxI5gHKDr.

Hellendoorn, V. J.; Sutton, C.; Singh, R.; Maniatis, P.; and Bieber, D. 2020. Global Relational Models of Source Code. In *International Conference on Learning Representations, ICLR 2020*. URL https://openreview.net/forum?id=B1lnbRNtwr.

Kim, S.; Zhao, J.; Tian, Y.; and Chandra, S. 2020. Code Prediction by Feeding Trees to Transformers.

Lachaux, M.-A.; Roziere, B.; Chanussot, L.; and Lample, G. 2020. Unsupervised Translation of Programming Languages. In *arXiv preprint arXiv:2006.03511*.

LeClair, A.; Jiang, S.; and McMillan, C. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 795–806. IEEE Press. doi:10.1109/ICSE.2019.00087. URL https://doi.org/10.1109/ICSE.2019.00087.

Li, J.; Wang, Y.; Lyu, M. R.; and King, I. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, 4159–25. AAAI Press. ISBN 9780999241127.

Loshchilov, I.; and Hutter, F. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *ICLR*.

Raychev, V.; Bielik, P.; and Vechev, M. 2016a. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, 731–747. New York, NY, USA: Association for Computing Machinery. ISBN 9781450344449. doi:10.1145/2983990.2984041. URL https://doi.org/10.1145/2983990.2984041.

Raychev, V.; Bielik, P.; and Vechev, M. T. 2016b. Probabilistic model for code with decision trees. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* .

Shaw, P.; Uszkoreit, J.; and Vaswani, A. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, 464–468. New Orleans, Louisiana: Association for Computational Linguistics. doi:10.18653/v1/N18-2074. URL https://www.aclweb.org/anthology/N18-2074.

Shiv, V.; and Quirk, C. 2019. Novel positional encodings to enable tree-based transformers. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32*, 12081–12091. Curran Associates, Inc. URL http://papers.nips.cc/paper/9376-novel-positional-encodings-to-enable-tree-based-transformers.pdf.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; and Polosukhin, I. 2017. Attention is All you Need. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 30*, 5998–6008. Curran Associates, Inc. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

A Experimental details

Dataset. We use Python150k dataset (Raychev, Bielik, and Vechev 2016a) downloaded from the official repository at https://eth-sri.github.io/py150 (version 1). The dataset consists of code files downloaded from Github. Proprocessing details for each task are given below. One of the important parts of our methodology is a thoughtful splitting of the dataset into training and testing sets that includes splitting by repository and removing code duplicates. Alon et al. (2019); LeClair, Jiang, and McMillan (2019) notice that code files inside one repository usually share variable names and code patterns, so splitting files from one repository between training and testing sets simplifies prediction for testing set and leads to a data leak. To avoid this, one should put all files from one repository into one set, either training or testing one (this strategy is called splitting by repository). Even with this strategy, duplicate code can still occur in testing set, since developers often copy code from other projects or fork other repositories. Allamanis (2019) underline that in commonly used datasets, up to 20% of testing objects can be repeated in the training set, biasing evaluation results. They also release a tool for deduplicating code datasets.

Taking the described facts into account, we remove duplicate files from Python150k dataset using the list of duplicates provided by (Allamanis 2019), and split data into training/testing set based on Github usernames (each repository is assigned to one username). We list usernames selected for training and testing sets at https://github.com/python150k-reposplit/repo_lists. Avoiding data leaks is highly important in our study: inaccurate dataset splitting can lead to the overestimation of the quality of the model trained in the anonymized setting.

Variable misuse task (VM). For the variable misuse task, we borrow the setup and evaluation strategy of (Hellendoorn et al. 2020). Given the code of a function, the task is to output two positions (using two pointers): in what position a wrong variable is used and which position a correct variable can be copied from (any such position is accepted). If a snippet is non-buggy, the first pointer should select a special no-bug position. We obtain two pointers by applying two position-wise fully-connected layers and softmax over positions on top of Transformer outputs. For example, the first pointer selects position $\text{argmax}_{1 \leqslant i \leqslant L} \text{softmax}([u^T y_1, \dots, u^T y_L, b])$, $y_i \in \mathbb{R}^D$, $u \in \mathbb{R}^D$, $b \in \mathbb{R}$ (b is a learnable scalar corresponding to the no-bug position), $[\cdot]$ denotes composing vector of scalars. The second pointer is computed is a similar way but without b. The model is trained using cross-entropy loss.

To obtain a dataset, we select all top-level functions, including functions inside classes, from all 150K files, and filter out functions longer than 250 nodes, and functions with less than three positions containing user-defined variables or less than three distinct user-defined variables. The resulting training/testing set consists of 476K / 238K functions. One function may occur in the dataset up to 6 times, 3 times with synthetically generated bug and 3 times without bug. The buggy examples are generated synthetically by choosing random buggy and fix position from positions containing user-defined variables.

Function naming task (FN). In this task, given the code of a function, the task is to predict the name of the function. To solve this task, we use classic sequence-to-sequence Transformer architecture (Ahmad et al. 2020) that outputs function name word by word. Firstly, we pass function code to Transformer encoder obtaining code representations $y_1, \dots, y_L$. Then, Transformer decoder generates method name word by word, and during each word generation, decoder attends to $y_1, \dots, y_L$ (using encoder-decoder attention) and to previously generated tokens (using masked decoder attention). To account for the sequential order of the function name, we use sequential positional embeddings in Transformer decoder. We use greedy decoding. The whole encoder-decoder model is trained end-to-end with cross-entropy loss being optimized.

To obtain a dataset, we select all top-level functions, including functions inside classes, from all 150K files (filtered during deduplication), and filter out functions longer than 250 AST nodes, functions with name consisting of only underscore characters and names containing rare words (less than 5 occurrences in the training set). The resulting dataset consists of 592K / 295K training/testing functions. We replace function name in the AST with special `fun_name` token. To obtain target function names, we split each function name based on *CamelCase* or *snake_case* and remove underscore characters. A mean length of function name is 2.43 words.

Code completion task (CC). For the task of code completion, we borrow the setup, metrics and Transformer implementation of (Kim et al. 2020). The task is to predict the next node (t_i, v_i) in the depth-first traversal of AST $[(t_1, v_1), \dots, (t_{i-1}, v_{i-1})]$. We predict type t_i and value v_i using two fully connected layers with softmax on top of prefix representation y_i: $P(t_i) = \text{softmax}(W^t y_i)$, $W^t \in \mathbb{R}^{\#\text{types} \times D}$, $P(v_i) = \text{softmax}(W^v y_i)$, $W^v \in \mathbb{R}^{\#\text{values} \times D}$.

To obtain a dataset, we use full code files, removing sequences with length less than 2. If the number of AST nodes is larger than $n_{ctx} = 500$, we split AST into overlapping chunks of length n_{ctx} with a shift $\frac{1}{2} n_{ctx}$. Overlap provides a context to the model. For example, if the length of AST is 800, we select the following samples: $AST[:500], AST[250:750], AST[300:800]$. We do not calculate loss or metrics over the intersection twice. For the previous example, the quality of predictions is measured only on $AST[:500], AST[500:750], AST[750:800]$. The resulting dataset consists of 241K / 114K training/testing chunks.

Hyperparameters. We list hyperparameters for variable misuse / function naming / code completion tasks using slashes. Our Transformer model has 6 layers, 8 / 8 / 6 heads, d_{model} equals 512 / 512 / 384. We limit vocabulary size for values up to 50K / 50K / 100K tokens and preserve all types. We train all Transformers using Adam with a starting learning rate of 0.00001 / 0.0001 / 0.0001 and batch size of 32 for 20 / 15 / 20 epochs. In code completion task, we use cosine learning rate schedule (Loshchilov and Hutter 2017) with warmup_step of 2000 and zero minimal learning rate, and gradient clipping of 0.2. In variable misuse task, we use

constant learning rate. In function naming task, we decay learning rate by 0.6 after each epoch and use gradient clipping of 5. We use residual, embedding and attention dropout with $p = 0.2 \ / \ 0.2 \ / \ 0.1$.

Hardware and software. We train all models on one GPU (NVIDIA Tesla P40 or V100). We use Python of version 3.7.2 and PyTorch of version 1.2.0 for variable misuse and function naming tasks, and Python of version 3.7.5 and PyTorch of version 1.5.1 for code completion task.

Anonymization. As discussed in section 3, we flatten an AST using the depth-first traversal and pass the sequence of (type, value) pairs to the Transformer. In the anonymized setting, we rename all values using placeholders `var1`, `var2`, `var3` etc. so that all occurrences of one value are replaced with the same placeholder, but different values are replaced with different placeholders. We limit the vocabulary of anonymized values to 500 elements (all examples incorporate no greater than 500 different values). To anonymize one code example, we gather the set of all different values in this example and map it to a random subset of `var1`, ..., `var500`.

Ensembling. To ensemble Transformers, we use the standard approach that implies training networks from different random initializations and averaging their predictions after softmax (Ashukha et al. 2020). The only exception is code completion task in which we cannot average the predictions of *Syntax+Text* and *Syntax* models because of different vocabularies. In this case only, we stack the predictions of both models and rank the obtained stacked predictions. We then estimate the positions of top tokens for each model $\text{rank}_{i1}, \text{rank}_{i2}$. The resulting MRR is $\frac{1}{n}\sum_{i=1}^{L} \max(\text{rank}_{i1}^{-1}, \text{rank}_{i2}^{-1})$. If $\min(\text{rank}_{i1}, \text{rank}_{i2}) > 10$, we assign zero score to the token i.

B Attention maps for different AST-capturing modifications of Transformer

In figures 6, 7, 8, 9 we present attention maps for different Transformer modifications on the code completion task in the anonymized setting. We visualize attention maps for the first layer since low level interactions should reflect the differences in considered modifications. We observe that all modifications except tree relative attention mostly attend to the last predicted tokens. Transformer with tree positional encodings (figure 7) always pays significant attention to the root node, that is easy to find because of zero tree encoding, or to some "anchor" nodes, e. g. `For` node in the first map of figure 7. In other words, tree positional encodings allow emphasizing important nodes in the tree. Also this model is able to distinguish children numbers. Transformer with tree relative attention (figure 9) often watches at siblings of the node to be predicted, but this model cannot differentiate child numbers, by construction. The attention maps for this model are smoother than for other models, because this model introduces multiplicative coefficients to the weights after softmax in self attention.

```
count = 0
for i in range(len(seq)):
    count += seq[i]
```

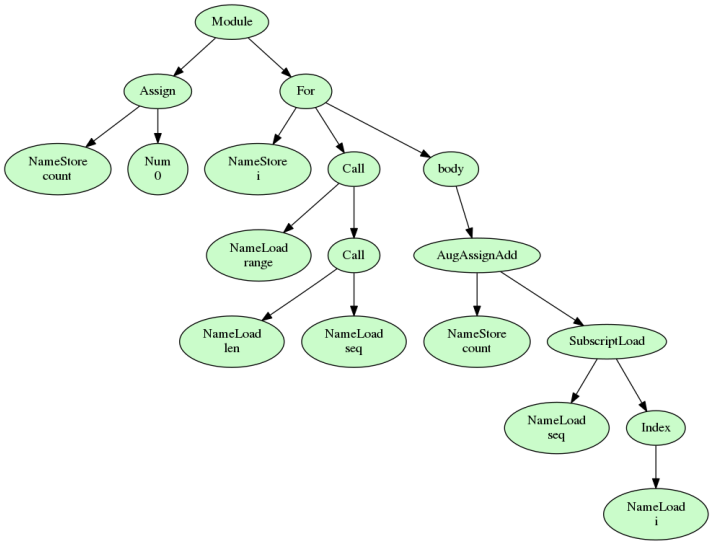


Figure 5: Code snippet (and its AST) used to visualize attention maps.

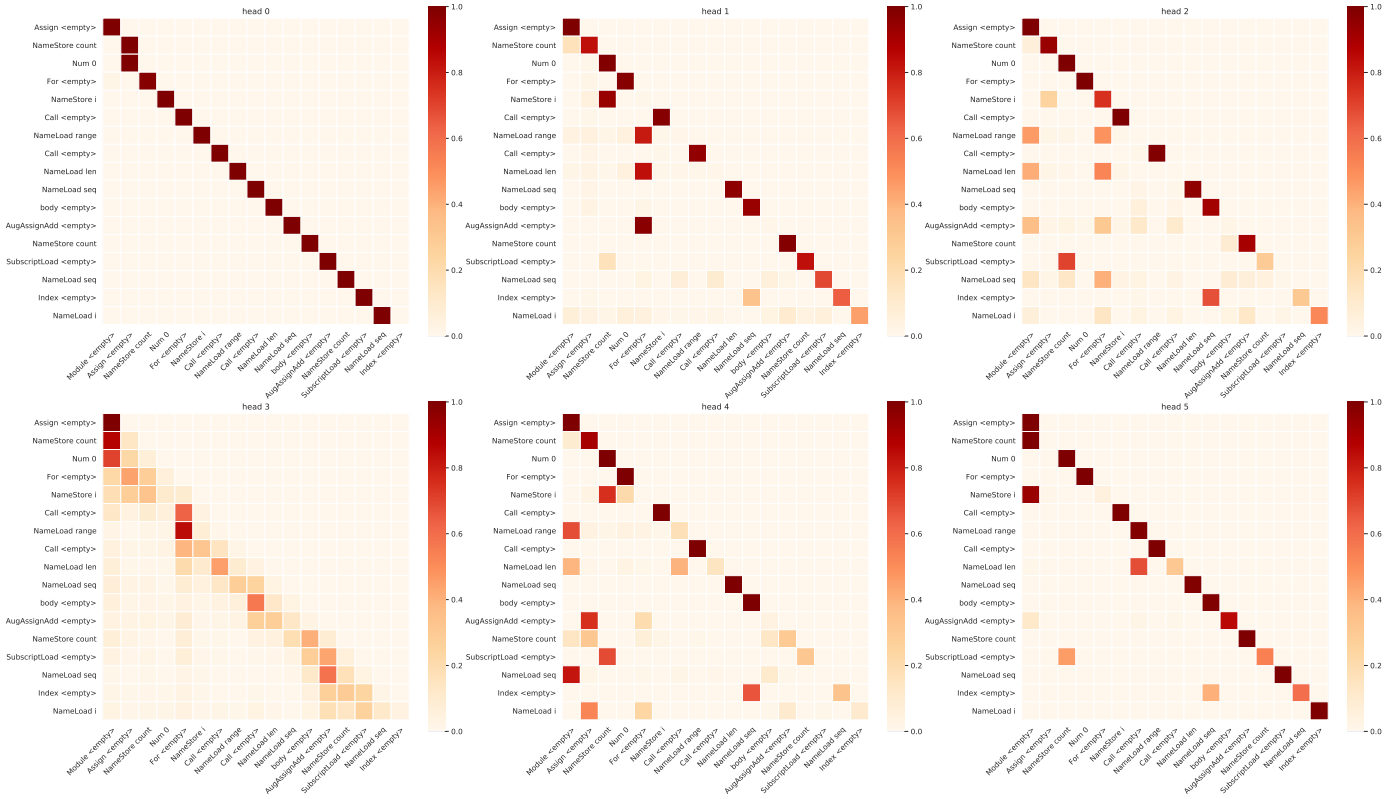


Figure 6: Attention maps for the 1st layer of *Syntax* model, sequential positional embeddings, code completion task. Y axis: what the model predicts, X axis: which token the model attends to.

Figure 7: Attention maps for the 1st layer of *Syntax* model, tree positional encodings, code completion task. Y axis: what the model predicts, X axis: which token the model attends to.

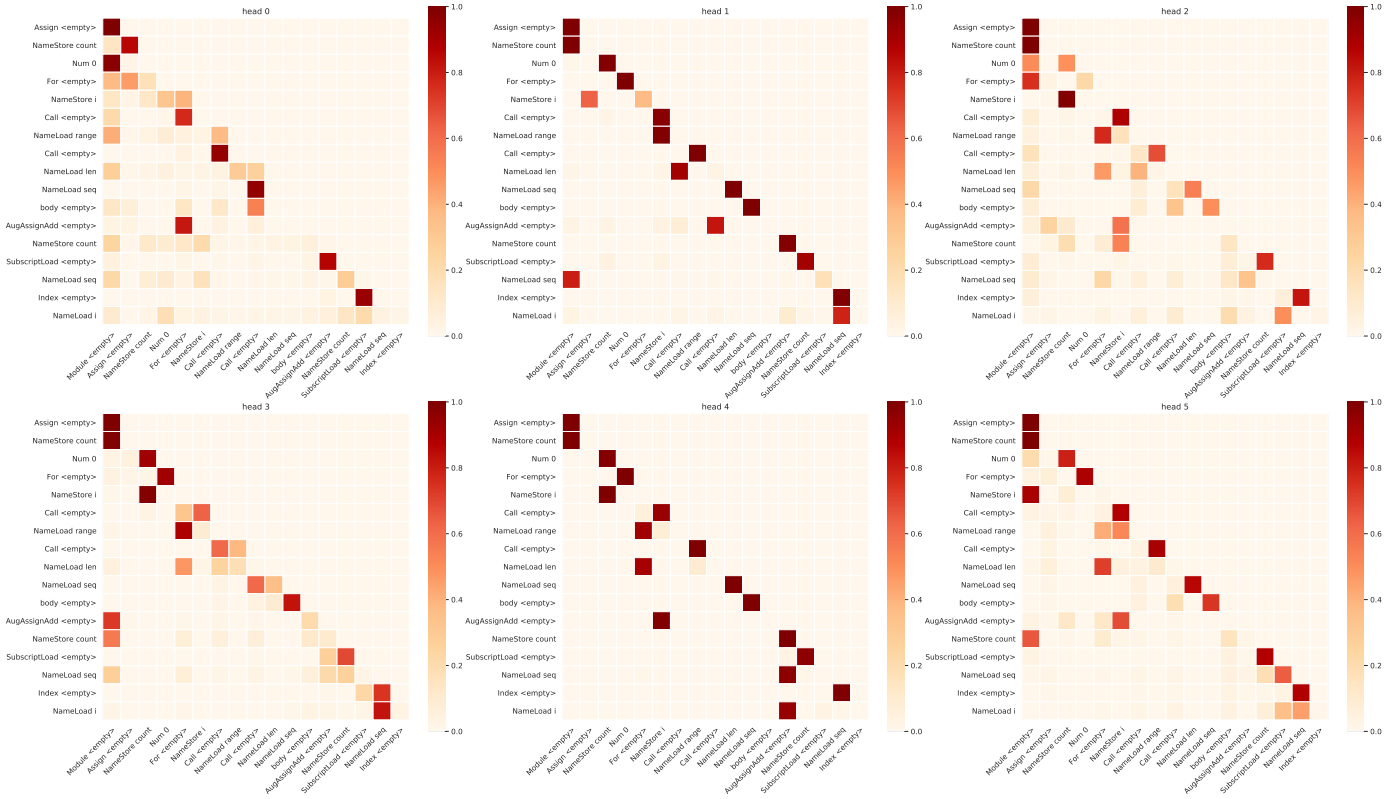


Figure 8: Attention maps for the 1st layer of *Syntax* model, sequential relative attention, code completion task. Y axis: what the model predicts, X axis: which token the model attends to.

Figure 9: Attention maps for the 1st layer of *Syntax* model, tree relative attention, code completion task. Y axis: what the model predicts, X axis: which token the model attends to.