Empirical Study of Transformers for Source Code

Nadezhda Chirkova
HSE University
Moscow, Russia
nchirkova@hse.ru

Sergey Troshin
HSE University
Moscow, Russia
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., it 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 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.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**.

KEYWORDS

neural networks, transformer, variable misuse detection, function naming, code completion

1 INTRODUCTION

Transformer [37] is currently a state-of-the-art architecture in a lot of source code processing tasks, including code completion [20], code translation [21, 32], and bug fixing [14]. 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 [14]. This parallelism also speeds up training and prediction.

Transformer is often applied to source code directly, treating code as a sequence of language keywords, punctuation marks, and identifiers. In this case, a neural network mostly relies on identifiers, e. g. variable names, to make predictions [1, 22]. High-quality variable names can be a rich source of information about the semantics of the code; however, this is only an 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, additional mechanisms are usually used, e.g. positional encoding for processing sequential structure. In recent years, a line of research has developed mechanisms for utilizing tree structure of code in Transformer [14, 20, 32]. However, the most effective way of utilizing syntactic information in Transformer is still unclear for three reasons. First, the mechanisms were developed concurrently, so they were not compared to each other by their authors. Moreover, different works test the proposed mechanisms on different code processing tasks, making it hard to align the empirical results reported in the papers. Secondly, the mentioned works used standard Transformer with positional encodings as a baseline, while modern practice uses more advanced modifications of Transformer, e.g., equipping it with relative attention [31]. As a result, it is unclear whether using sophisticated mechanisms for utilizing syntactic information is needed at all. Thirdly, most of the works focus on utilizing tree structure in Transformer and do not investigate the effect of processing other syntax components, e. g. the syntactic units of the programming language.

In this work, we conduct an empirical study of using Transformer for processing source code. Firstly, we would like to answer the question, what is the best way of utilizing syntactic information in Transformer, and to provide practical recommendations for the use of Transformers in software engineering tasks. Secondly, we aim at understanding whether Transformer is generally suitable for capturing code syntax, to ground the future development of Transformers for code. Our contributions are as follows:

- We re-implement several approaches for capturing syntactic structure in Transformer and investigate their effectiveness in three code processing tasks on two datasets. 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 neural code processing.
- We introduce an anonymized setting in which all user-defined identifiers are replaced with placeholders, and show that Transformer is capable of making meaningful predictions based purely on syntactic information, in all three tasks. We

N. Chirkova and S. Troshin

(a) Code: `elem = lst[idx]`

(b) Abstract syntax tree (AST):

(c) AST depth-first traversal:

Assign	NameStore	SubscriptLoad	...
⟨empty⟩	elem	⟨empty⟩	...

...	NameLoad	Index	NameLoad
...	lst	⟨empty⟩	idx

(d) Tree positional encodings:

1	2	3	4	5	6
/	/1	/2	/2/1	/2/2	/2/2/1

stack-like enc.
1: 000 000 000
2: 100 000 000
3: 010 000 000
4: 100 010 000
5: 010 010 000
6: 100 010 010

(e) Tree relative attention:

	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

(f) GGNN Sandwich:

Types of edges:
· Parent (P)   · Left (L)
· Child (C)   · Right (R)
              · Self (S)

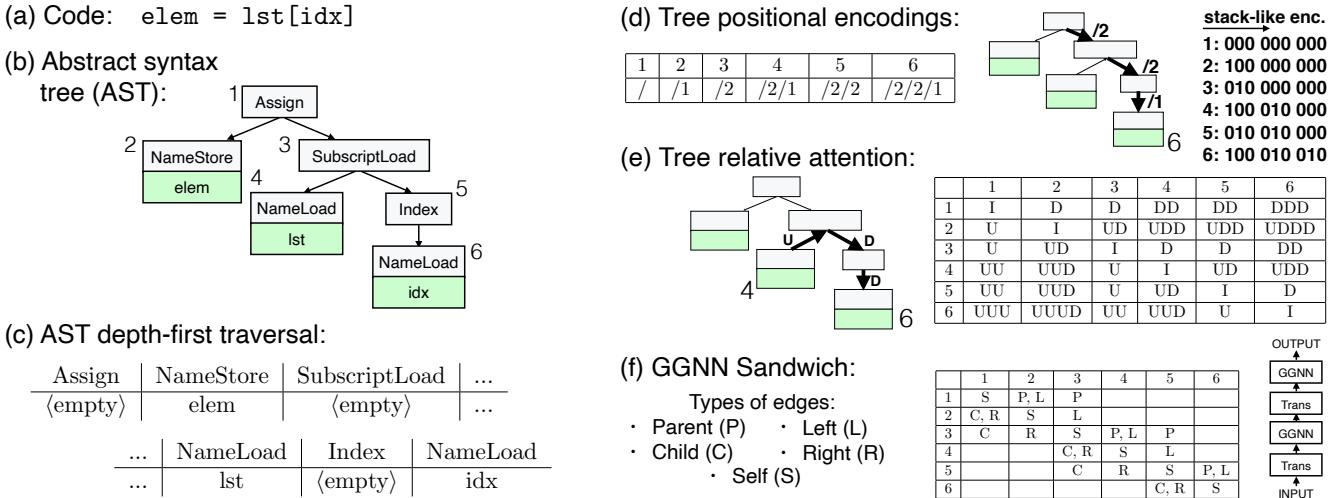|	1	2	3	4	5	6
1	S	P, L	P			
2	C, R	S	L			
3	C	R	S	P, L	P	
4			C, R	S	L	
5			C	R	S	P, L
6					C, R	S

OUTPUT
GGNN
Trans
GGNN
Trans
INPUT

Figure 1: Illustration of mechanisms for processing AST structure in Transformer.

also show that using the proposed anonymization can improve the quality of the model, either a single Transformer or an ensemble of Transformers.

- We conduct an ablation study of different syntax-capturing components in Transformer, underlining which ones are essential for achieving high quality and analysing the results obtained for the anonymized setting.

Our source code is available at https://github.com/bayesgroup/code_transformers.

The rest of the work is organized as follows. In Section 2 we review the existing approaches for utilizing syntactic information in Transformer. In Section 3 we describe our methodology for the empirical evaluation of Transformer capabilities to utilize syntactic information. The following Sections 6–8 describe our empirical findings. In Section 10, we give the review of the literature connected to our research. Finally, Section 11 discusses threats to validity and Section 12 concludes the work.

2 REVIEW OF TRANSFORMERS FOR SOURCE CODE

Abstract syntax tree. A syntactic structure of code is usually represented in the form of an abstract syntax tree (AST). Each node of the tree contains a type, which represents the syntactic unit of the programming language (e.g. "Assign", "Index", "NameLoad"), some nodes also contain a value (e.g. "idx", "elem"). Values store user-defined variable names, reserved language names, integers, strings etc. 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 $c_1, \ldots c_L$, $c_i \in \{1, \ldots, M\}$ (M is a vocabulary size) to a sequence of d_{model}-dimensional representations $y_1, \ldots, y_L$ 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 $x_1, \ldots, x_L$, $x_i \in \mathbb{R}^{d_{model}}$.

A key ingredient of a Transformer block is a self-attention layer that maps input sequence $x_1, \ldots x_L$, $x_i \in \mathbb{R}^{d_{model}}$ to a sequence of the same length: $z_1, \ldots, z_L$, $z_i \in \mathbb{R}^{d_z}$. Self-attention first 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:

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

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, $h = 1, \ldots, H$. The outputs are concatenated and projected to obtain $\hat{x}_i = [z_i^1, \ldots, z_i^H]W^O$, $W^O \in \mathbb{R}^{Hd_z \times d_{model}}$. A 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 the consequent stacking of the described blocks. When applying Transformer to generation tasks, future elements ($i > j$) are masked in self-attention (Transformer *decoder*). Without this masking, the stack of the layers is called Transformer *encoder*. In the sequence-to-sequence task, when both encoder and decoder are used, the attention from decoder to encoder is also incorporated into the model.

2.2 Passing ASTs to Transformers

For our study, we select two commonly used NLP approaches for utilizing sequential structure and three approaches developed specifically for utilizing source code structure in Transformer.

Sequential positional encodings and embeddings. 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.

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 up with positional representations $p_i \in R^{d_{model}}$: $\hat{x}_i = x_i + p_i$. For example, positional embeddings imply learning the embedding vector of each position $i \in 1 \ldots L$: $p_i = e_i$, $e_i \in R^{d_{model}}$. Positional encoding implies computing p_i based on sine and cosine functions. We include positional embeddings in our comparisons and add the prefix "sequential" to the title of this mechanism. This approach was used as a baseline in several of works [14, 32].

Sequential relative attention. Shaw et al. [31] proposed relative attention for capturing the order of the input elements. They augment self-attention with relative embeddings:

$$z_i = \sum_j \tilde{\alpha}_{ij}(x_j^v + e_{i-j}^v), \; \tilde{\alpha}_{ij} = \frac{\exp(a_{ij})}{\sum_j \exp(a_{ij})}, \; a_{ij} = \frac{x_i^q(x_j^k + e_{i-j}^k)^T}{\sqrt{d_z}},$$

(2)

where $e_{i-j}^v, e_{i-j}^k \in \mathbb{R}^{d_z}$ are learned embeddings for each relative position $i - j$, e. g. one token is located two tokens to the left from another token. This mechanism, that we call *sequential relative attention*, was shown to substantially outperform sequential positional embeddings and encodings in sequence-based text processing tasks. Ahmad et al. [1] reach 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. Inspired by previously discussed works, several authors developed mechanisms for processing trees. Shiv and Quirk [32] 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 reverse order and padding short paths with zeros from the right. The authors also introduce the learnable parameters of the encoding, their number equals $d_{model}/(n_w \cdot n_d)$. Paths longer than n_d are clipped (the root node is clipped first). The authors binarize ASTs to achieve $n_w = 2$. To avoid this binarization, we replace all child numbers greater than n_w with n_w, and select the best hyperparameters n_w and n_d using grid search, see details in section 4.

Shiv and Quirk [32] 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 [34].

Tree relative attention. An extension of sequential relative attention for trees was proposed by Kim et al. [20]. 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 into self-attention. Learning multidimensional embeddings for the tree input requires much more memory than for 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 into the attention mechanism as follows (other formulas stay the same):

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

(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 showed that using modified attention improves quality when applying Transformer 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) [4]. 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 a model captures *local* dependencies in data well but lacks a *global* view of the input program, that is the Transformer's forte. Inspired by this reasoning, Hellendoorn et al. [14] propose alternating Transformer and GGNN layers as illustrated in Figure 1(f), to combine the strengths of both models. GGNN layer relies on passing messages through edges for a fixed number of iterations (number of passes). The model is called GGNN Sandwich by the authors, and the details can be found in [14]. GGNN Sandwich was shown to be effective in the variable misuse detection task, i.e., predicting the location of a bug and the location used to fix the bug (copy variable). GGNN Sandwich outperformed standard Transformer with sequential positional encodings.

Our work focuses of processing syntactic information in Transformer, thus we do not use data- and control-flow edges. Data- or control-flow edges are hard to incorporate in other mechanisms except GGNN Sandwich. In our GGNN Sandwich, we use AST edges, edges connecting the neighbouring nodes in the AST depth-first traversal, and edges connecting nodes to themselves, see illustration in fig. 1(f).

Hellendoorn et al. [14] also propose a model called GREAT that is inspired by relative attention and incorporates 1-dimensional edge embeddings into the attention mechanism. This model is conceptually similar to the tree relative attention, thus we do not include GREAT in our comparison.

3 LIMITATIONS OF EXISTING APPROACHES AND METHODOLOGY OF THE WORK

As shown in Section 2 several approaches for processing ASTs in Transformers have been proposed. However, it is still unclear which approaches perform better than others and what mechanisms to use in practice. First, all the works discussed in Section 2 conduct

experiments with different tasks making it hard to align the results. Moreover, almost all the listed works compare their approaches with the vanilla Transformer, i.e., Transformer with sequential positional encodings or embeddings, while modern practices use advanced mechanisms, like sequential relative attention, by default. Even works that propose tree-processing approaches inspired by sequential relative attention do not include this mechanism as a baseline. That is, it is unclear whether using advanced tree-processing mechanisms 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 (VM) detection, function naming (FN), and code completion (CC), and two source code datasets: Python150k [38] and JavaScript150k [30]. We selected tasks that are often used as benchmarks in the literature and on which the compared approaches were tested by their authors. Our selection also covers various Transformer configurations, i.e., encoder only (VM), decoder only (CC) and encoder-decoder (FN). We selected the Python150k dataset because it is often used in the literature, and JavaScript150k because it is distributed by the same authors and has the same format.

We re-implement all mechanisms described in Section 2 in a unified framework and investigate the most effective approach for processing ASTs in Transformer in different tasks. We answer the following research questions:

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

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. [20] and Shiv and Quirk [32] attach values as separate child nodes so that each node stores only one item (type or value), while Hellendoorn et al. [14] propose omitting types. The former approach increases input length and thus makes code processing significantly slower, while the latter approach loses type information. We choose an in-between strategy inspired by the approach of Li et al. [23] used for RNNs: we associate the `<empty>` value 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 preserves the initial AST structure and allows us to easily ablate types or values, leaving the other item in each node present.

Some works show that splitting values based on `snake_case` or `CamelCase`, or using splitting techniques such as byte-pair encoding may improve the quality [6, 19]. We do not use splitting into subtokens for two reasons. Firstly, splitting makes sequences much longer, resulting in a substantial slow down of training procedure because of quadratic Transformer complexity w.r.t. the input length. Secondly, splitting breaks the one-to-one correspondence between AST nodes and values, i.e., several values belong to one AST node. There are different ways of adapting AST-based Transformers to the described problem: one option is to average embeddings over subtokens [14], another option is to assign a chain of subtokens

as a child of a node and then directly apply tree-processing mechanisms. A third option is to modify tree-processing mechanisms, e.g., duplicate paths for all subtokens in tree positional encoding or duplicate tree relations for all pairs of subtokens of two tokens in tree relative attention. As a result, the question of how splitting into subtokens affects syntax-capturing mechanisms requires a separate study which we leave for the future work.

An important part of our methodology is conducting experiments in two settings, namely *anonymized* and *full-data*. The full-data setting corresponds to the conventional training of Transformer on ASTs parsed from code. In this case, Transformer has two sources of information about input code snippets: syntactic information and user-defined identifiers (stored in node values). Identifiers 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 identifiers with placeholders `var1`, `var2`, `var3` etc. will lead to the same result of code execution and will not change the semantics of the algorithm the code implements. Here we mean that all occurrences of an identifier are replaced with the same placeholder, thus, important information about identifier repetition is saved. We call this renaming identifiers 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 the anonymized setting allows a better understanding of the capabilities of Transformer to utilize syntactic information. More details on the anonymization procedure are given in Appendix A.

Another important part of our methodology is a thoughtful splitting of the dataset into training and testing sets, which includes splitting by repository and removing code duplicates. Alon et al. [6], LeClair et al. [22] notice that code files inside one repository usually share variable names and code patterns, thus splitting files from one repository between training and testing sets simplifies predictions for the 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 (this strategy is called splitting by repository). Even using this strategy, duplicate code can still occur in the testing set, since developers often copy code from other projects or fork other repositories. Allamanis [3] underline that in commonly used datasets up to 20% of testing objects can be repeated in the training set, biasing evaluation results. As a result, the deduplication step is needed after data splitting.

4 EXPERIMENTAL SETUP

Data. In all tasks, we use the Python150k (PY) dataset [28] (redistributable version) and JavaScript150k (JS) dataset [30] downloaded from the official repository at https://eth-sri.github.io. Both datasets consist of code files downloaded from Github and are commonly used to evaluate code processing models.

Most research use the 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 [3], so we release a new split of the dataset. We remove duplicate files from both datasets using the list of duplicates provided by Allamanis [3]. We also filter out absolutely identical code files, and when selecting

functions from code files, we additionally filter out absolutely identical functions. We split data into training / validation / testing sets in proportion 60% / 6.7% / 33.3% based on Github usernames (each repository is assigned to one username).

Preprocessing details for each task are given below. We release our data split and our source code including scripts for downloading data, deterministic code for data preprocessing, models, training etc. We largely rely on the implementations of other research, and compare the quality of our baseline models to the results reported in other papers, when possible; see details in Section 9.

Variable misuse task (VM). For the variable misuse task, we use the setup and evaluation strategy of Hellendoorn et al. [14]. Given the code of a function, the task is to identify two positions (using two pointers): one in which position a wrong variable is used, and one in 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 as $\mathrm{argmax}_{1 \leqslant i \leqslant L} \mathrm{softmax}([u^T y_1, \ldots, u^T y_L, b])$, $y_i \in \mathbb{R}^{d_{\mathrm{model}}}$, $u \in \mathbb{R}^{d_{\mathrm{model}}}$, $b \in \mathbb{R}$ (b is a learnable scalar corresponding to the no-bug position), [...] denotes the concatenation of the elements into a vector of scalars. The second pointer is computed in a similar way but without b. The model is trained using the cross-entropy loss.

To process the original dataset for the variable misuse task, we select all top-level functions, including functions inside classes, from all (filtered) 150K files, and filter out functions: longer than 250 nodes (to avoid very long functions); and functions with less than three positions containing user-defined variables or less than three distinct user-defined variables (to avoid trivial bug fixes). We select a function with a root node type FunctionDef for PY, and FunctionDeclaration or Function Expression for JS. The resulting training / validation / testing set consists of 417K / 48K / 231K functions for PY and 202K / 29K / 108K for JS. One function may occur in the dataset up to 6 times, 3 times with a synthetically generated bug and 3 times without a bug. Following [14], we use this strategy to avoid biasing towards long functions with a lot of different variables. The buggy examples are generated synthetically by choosing random bug and fix positions from positions containing user-defined variables.

We use the joint localization and repair accuracy metric of [14] 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.

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 the classic sequence-to-sequence Transformer architecture that outputs the function name word by word. A particular implementation is borrowed from [1] (the paper used another dataset). Firstly, we pass function code to the Transformer encoder to obtain code representations $y_1, \ldots, y_L$. Then, the Transformer decoder generates the method name word by word, and during each word

generation, decoder attends to $y_1, \ldots, y_L$ (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 the Transformer decoder. In the encoder, we consider different structure-capturing mechanisms. We use greedy decoding. We train the whole encoder-decoder model end-to-end, optimizing the cross-entropy loss.

To obtain the processed dataset for the function name task, we select all top-level functions, including functions inside classes, from all (filtered) 150K files, and filter out functions longer than 250 AST nodes (to avoid very long functions), functions for which the name could not be extracted (a lot of FunctionExpressions in JS are anonymous), and functions with names consisting of only underscore characters and names containing rare words (less than 5 / 3 occurrences in the training set for PY / JS). To extract functions, we use the same root node types as in the VM task. The resulting dataset consists of 523K / 56K / 264K training/validation/testing functions for PY and 186K / 23K / 93K for JS. We replace function name in the AST with a special <fun_name> token. To extract target function names, we remove extra underscores and split each function name based on *CamelCase* or *snake_case*, e.g., name _get_feature_names becomes [get, feature, names]. A mean±std length of function name is 2.42±1.46 words for PY and 2.22±1.23 for JS.

We assess the quality of the generated function names using the 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, the 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 functions. We choose the F1 metric following Alon et al. [6] who solved a similar task with another dataset and model.

Code completion task (CC). For the task of code completion, we use the setup, metrics and Transformer implementation of Kim et al. [20]. The task is to predict the next node (t_i, v_i) in the depth-first traversal of AST $[(t_1, v_1), \ldots, (t_{i-1}, v_{i-1})]$. We predict type t_i and value v_i using two fully connected layers with softmax on top of the prefix representation y_i: $P(t_i) = \mathrm{softmax}(W^t y_i)$, $W^t \in \mathbb{R}^{\#\mathrm{types} \times d_{\mathrm{model}}}$, $P(v_i) = \mathrm{softmax}(W^v y_i)$, $W^v \in \mathbb{R}^{\#\mathrm{values} \times d_{\mathrm{model}}}$.

To obtain the dataset for the code completion task, we use full ASTs from (filtered) 150k 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 $\frac{1}{2}n_{ctx}$. The overlap provides a context for 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 overlapping splitting procedure is borrowed from [20] and is needed since processing extremely long sequences is too slow in Transformer because of its quadratic complexity w.r.t. input length. The resulting dataset consists of 186K / 20K / 100K training / validation / testing chunks for PY and 270K / 32K / 220K for JS.

We mostly focus on value prediction, since it is the more complex task, 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.

Table 1: Selected hyperparameters for different structure-capturing mechanisms, tasks and datasets. The details on selecting hyperparameters are given in Appendix A.

Model (hypers.)	Lang.	VM	FN	CC
Seq. rel. attn.	PY	8	250	32
(max. dist.)	JS	8	250	32
Tree pos. enc.	PY	8, 16	16, 8	16, 8
(max width, depth)	JS	4, 8	2, 64	16, 32
Tree rel. attn.	PY	100	600	1000
(rel. vocab. size)	JS	600	100	1000
GGNN Sandwich	PY	12, 3, N	6, 2, Y	N/A
(num. layers,	JS	12, 3, N	6, 2, Y	N/A
num. edge types,				
is GGNN first?)				

We use mean reciprocal rank (MRR) to measure the model quality since it reflects the practical application of code completion: MRR = $\frac{1}{N} \sum_{i=1}^{N} 1/rank_i$, where $rank_i$ is a position of the i-th true token in the model ranking, N is the total number of target tokens in a dataset, excluding <padding> and <empty> tokens. As in [20], we assign zero score if the true token is out of top 10 predicted tokens.

Hyperparameters. We list general hyperparameters for the variable misuse / function naming / code completion tasks using slashes. Our Transformer models have 6 layers, 6 / 6 / 8 heads, $d_{model} = 512$. We limit vocabulary sizes for values up to 50K / 50K / 100K tokens and preserve all types. As discussed in Section 3, we do not split values into subtokens. We train all Transformers using Adam with a starting learning rate of 0.00001 / 0.0001 / 0.0001 and a batch size of 32 for 25 / 15 / 20 epochs for PY and 40 / 25 / 20 epochs for JS (the number of functions in the JS dataset is smaller than in PY dataset, thus more epochs are needed). In the code completion task, we use the cosine learning rate schedule [25] with 2000 warm-up steps and a zero minimal learning rate, and a gradient clipping of 0.2. In the variable misuse task and in the function naming task for JS, we use a constant learning rate. In the function naming task for PY, we decay the learning rate by 0.9 after each epoch. In the function naming task, we also use a gradient clipping of 5. We use residual, embedding and attention dropout with $p = 0.2$ / 0.2 / 0.1. All models were trained three times, to estimate the standard deviation of the quality (except hyperparameter tuning). In all experiments we report the quality on the test set, except hyperparameter tuning where we report the quality on the validation set. We train all models on one GPU (NVIDIA Tesla P40 or V100).

The hyperparameters for different structure-capturing mechanisms were tuned using grid search, based on the quality on the the validation set, for each dataset–task combination individually. For sequential relative attention, we tune the maximum relative distance between the elements of the sequence. For tree positional encoding, we tune the maximum path width and the maximum path depth. For tree relative attention, we tune the size of the relation vocabulary. For the GGNN Sandwich model, we consider 6-layer and 12-layer configurations of alternating Transformer (T) and GGNN (G) layers, we also consider placing both types of layers first i.e., [T, G, T, G, T, G] or [G, T, G, T, G, T] (and similarly for 12 layers).

GGNN layers include 4 message passes. We also consider omitting edges of types Left and Right. Sequential positional embeddings do not have hyperparameters. The number of parameters in all AST-based modifications of Transformer are approximately the same, except GGNN Sandwiches: 12-layer Sandwich incorporates slightly more parameters than vanilla Transformer, while 6-layer incorporates slightly fewer parameters. The details and the Tables on hyperparameter search are given in Appendix A, the resulting hyperparameters are listed in Table 1.

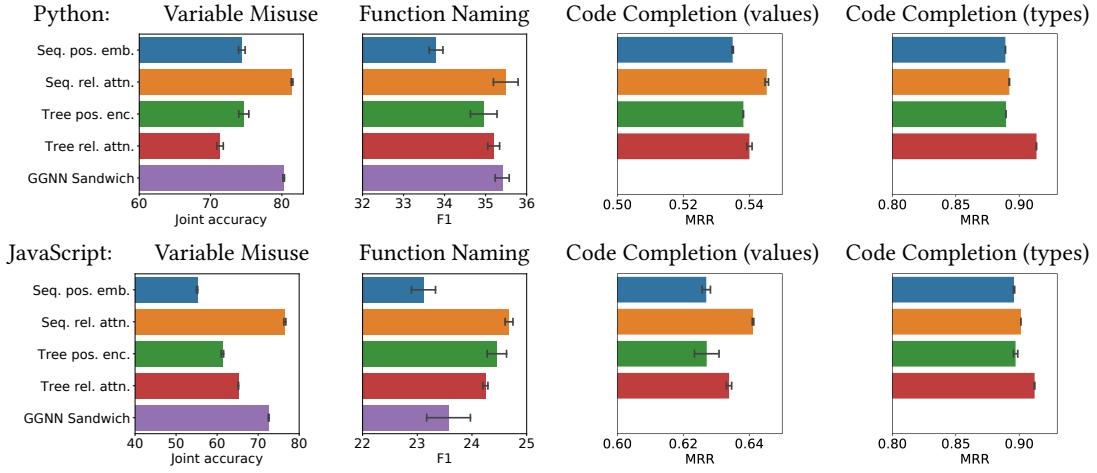# 5 COMPARISON OF APPROACHES FOR UTILIZING SYNTACTIC STRUCTURE IN TRANSFORMER

We begin with investigating which of the mechanisms for utilizing AST structure in Transformer is the most effective one. We obtain the trees storing a (type, value) pair in each node using the approach described in Section 3 and pass these trees to Transformer, equipped with one of the mechanisms described in Section 2. GGNN Sandwich is not applicable to code completion because message-passing involves all nodes and prohibits using masking in the decoder.

The results are presented in Figure 2. In function naming, most structure-capturing mechanisms perform similarly. In Section 7, we show that in this task quality is not affected much even if we completely ablate structure information, i.e., do not use any structure-capturing mechanism and treat input as a *set* of (type, value) pairs. That is, Transformer hardly utilizes syntactic structure when predicting function names. However, in other tasks, this is not the case and there is more variability in different mechanisms performance.

Utilizing structure information in the input embeddings is not effective: sequential positional embeddings and tree positional encodings do not achieve highest score in any task, except function naming where tree positional encodings perform on par with other mechanisms.

Utilizing structure in the 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 by a high margin in type prediction task (this model was developed for code completion task). The last result is interpretable since tree relative attention helps to find relatives in AST tree, e.g., parent and siblings, which is important in type prediction. The advantage of sequential relative attention is that it can use the multidimensional embeddings of relations: the sequential relations are shared across objects, leading to affordable 3-dimensional embedding tensors of shape (length, length, embedding dimension). In contrast, tree relative attention can only afford one-dimensional embeddings of relations, because tree-based relations are not shared between objects in a mini-batch, and extracting them for a mini-batch would already lead to a 3-dimensional tensor: (batch, length, length).

GGNN Sandwich achieves high results in the variable misuse task for which this model was developed. 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. The original

Figure 2: A comparison of different mechanisms for processing AST structure in Transformer, in the full-data setting. The numeric data for barplots is given in Appendix C.

Table 2: Time- and storage-consumption of different structure-capturing mechanisms for the variable misuse task on the Python dataset.

Model	Train time (h/epoch)	Preprocess time (ms/func.)	Add. train data (GB)
Seq. pos. emb.	2.3	0	0
Seq. rel. att.	2.7	0	0
Tree pos. enc.	2.5	0.4	0.3
Tree rel. attn.	3.9	16.7	18
GGNN Sandwich	7.2	0.3	0.35

work on GGNN Sandwiches also uses additional types of edges which would improve the performance of this model further. Using these types of edges is out of scope of this work, since we focus on utilizing *syntactic* information, thus we only use syntax-based edges.

In Appendix B, we visualize the progress of test metrics during training, for different structure-capturing mechanisms in the full-data setting. This Appendix also presents the comparison of structure-capturing mechanisms in the anonymized setting that is described in Section 3 and implies replacing values in ASTs with unique placeholders. The leading mechanisms are the same in all tasks as in the full data setting, considered above. An interested reader may also find examples of attention maps for different mechanisms in Appendix D.

In Table 2, we list training time and the size of auxiliary data needed for different structure-capturing mechanisms. GGNN Sandwich model requires twice the time for training (and prediction) compared to other models, because of the time-consuming message passing mechanism. Tree relative attention requires dozens of gigabytes for storing pairwise relation matrices for all training objects that could be replaced with slow on-the-fly relation matrix generation. Tree positional encodings and GGNN Sandwich models also require additional disk space for storing preprocessed graph

Table 3: Comparison of combinations of sequential relative attention (SRA) with other structure-capturing approaches. All numbers in percent, standard deviations: VM: 0.5%, FN: 0.4%, CC: 0.1%. Bold emphasizes combinations that significantly outperform SRA. *In the VM task, SRA+GGNN Sandwich significantly outperforms SRA during the first half of epochs, but loses superiority at the last epochs, for both datasets. On the Python dataset, SRA+GGNN Sandwich outperforms SRA by one standard deviation at the last epoch.

	Model	VM	FN	CC (val.)
PY	SRA	81.42	35.73	54.53
PY	SRA + Seq. pos. emb.	80.77	33.99	54.37
	SRA + Tree pos. enc.	81.73	34.71	54.63
	SRA + Tree rel. attn.	81.58	35.41	**54.91**
	SRA + GGNN sand.	82.00*	33.39	N/A
JS	SRA	76.52	24.62	64.11
JS	SRA + Seq. pos. emb.	73.17	23.09	63.97
	SRA + Tree pos. enc.	74.73	23.70	**64.49**
	SRA + Tree rel. attn.	76.34	24.71	**64.79**
	SRA + GGNN sand.	75.33*	21.44	N/A

representations, but the sizes of these files are relatively small. 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 structure-capturing mechanisms. In Table 3, we show that *using sophisticated structure-capturing mechanisms may be useful for further improving sequential relative attention if we combine two mechanisms.* We find that tree relative attention (for both datasets) and tree positional encoding (for JS) improve the score in

Table 4: Illustration of different kinds of models used in the experiments. The code snippet and its AST used in the illustration may be found in Figure 1 (a, b).

Model	Input representation
Syntax+Text	`[(Assign, <empty>), (NameStore, elem), …` `…, (Index, <empty>), (NameLoad, idx)]`
Syntax	`[(Assign, <empty>), (NameStore, <var1>), …` `…, (Index, <empty>), (NameLoad, <var3>)]`
Text	`[elem, lst, idx]`
Constant	Predicts the most frequent target for any input

the value prediction task, while GGNN Sandwich may improve the score in the variable misuse task, especially at earlier epochs.

6 CAPABILITY OF TRANSFORMER TO UTILIZE SYNTACTIC INFORMATION

When developing approaches for utilizing syntactic information in Transformer, the majority of works mostly focus on the tree *structure*. We discussed the utilization of structure in the previous section, but we also would like to investigate the influence of other AST components, namely *types* and *values*. In the next section, we conduct an ablation study of the mentioned components, and in this section, we investigate whether Transformer is generally capable of utilizing the syntactic information in source code, i. e. does processing AST components improve performance. This conceptual experiment tests the overall suitability of Transformer for source code processing. We formalize the specified question in full-data and anonymized settings as follows:

- *Syntax+Text* vs. *Text*: First, we test whether using syntactic information in addition to the textual information is beneficial, compared to using pure textual information. To do so, we compare the quality of Transformer trained on full AST data (*Syntax+Text*) with the quality of Transformer trained on a sequence of non-empty values (*Text*), see Table 4 for illustrations. The *Text* model relies only on textual information stored in values and does not have access to any other kind of information.
- *Syntax* vs. *Constant*: Secondly, we test whether Transformer is able to make meaningful predictions given *only* syntactic information, without textual information. To do this, we test whether the quality of Transformer trained on the *anonymized* AST data (*Syntax*) is better than the quality of a simple constant baseline (*Constant*). Anonymization removes identifiers, i.e., textual information, but preserves information about identifiers' repetition, which may be essential for understanding the program. The *Constant* model outputs the most frequent target, e.g., no-bug in VM and name `init` for PY and `exports` for JS in FN. Since anonymized AST data contains only syntactic information and does not contain textual information, the only way the Transformer can outperform the *Constant* baseline on this data is to capture some information from AST.

All the described models are trained with sequential relative attention. Deduplicating the dataset is very important in this experiment to avoid overestimating the *Syntax* baseline.

The results for three tasks are presented in Figure 3. In all cases, *Syntax+Text* outperforms *Text*, and *Syntax* outperforms *Constant*. In Figure 4, we present example predictions for code completion and function naming tasks for Python language with all four models. In code completion, syntax-based models capture frequent coding patterns well, for instance, in example (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 the if-statement checking whether the bucket exists, while *Text* model outputs frequent tokens associated with buckets.

In the function naming task, the *Text* model captures code semantics based on variable names and sometimes selects wrong "anchor" variables, e.g., `split` in example (c), while the *Syntax+Text* model utilizes AST information and outputs correct word `dict`. The *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, the *Syntax* model is capable of distinguishing general code purpose, e.g., model uses word `get` in example (c) and word `read` in example (d). To sum up, *Transformer is indeed capable of utilizing syntactic information.*

Interestingly, in code completion (value prediction, PY) and variable misuse detection tasks (JS), the *Syntax* model, trained on the anonymized data outperforms the *Syntax+Text* model trained on full data, though the latter uses more data than the former. The reason is that the values vocabulary on full data is limited, so approx. 25% of values are replaced with the `UNK` token and cannot be predicted correctly. On the other hand, this is not a complication for the *Syntax* model, which anonymizes both frequent and rare identifiers in the same way. For example, in Figure 4(b), the *Syntax* model correctly predicts the misspelled token `argmnents` while for the *Syntax+Text* model, this token is out-of-vocabulary and so model outputs frequently used strings for printing. One more advantage of the *Syntax* model in the value prediction task is that it is twice faster in training because of the small output softmax dimension. In the function naming task, the *Syntax* model performs substantially worse than *Syntax+Text*, because variable names provide much *natural language* information needed to make natural language predictions. To sum up, *anonymization may lead to a higher quality than using full data.*

7 ABLATION STUDY OF MECHANISMS FOR UTILIZING SYNTACTIC INFORMATION IN TRANSFORMER

In this section, we investigate the effect of ablating different AST components on the performance in three tasks. This ablation study is important for both providing practical recommendations and understanding what mechanisms are essential for making reasonable predictions in the anonymized setting, discussed in the previous section. We consider three AST components (comments in items regard the usual scenario without ablation): (*types*): the types of nodes are passed as one of the Transformer inputs; (*values*): the (anonymized) values are passed as one of the Transformer inputs;
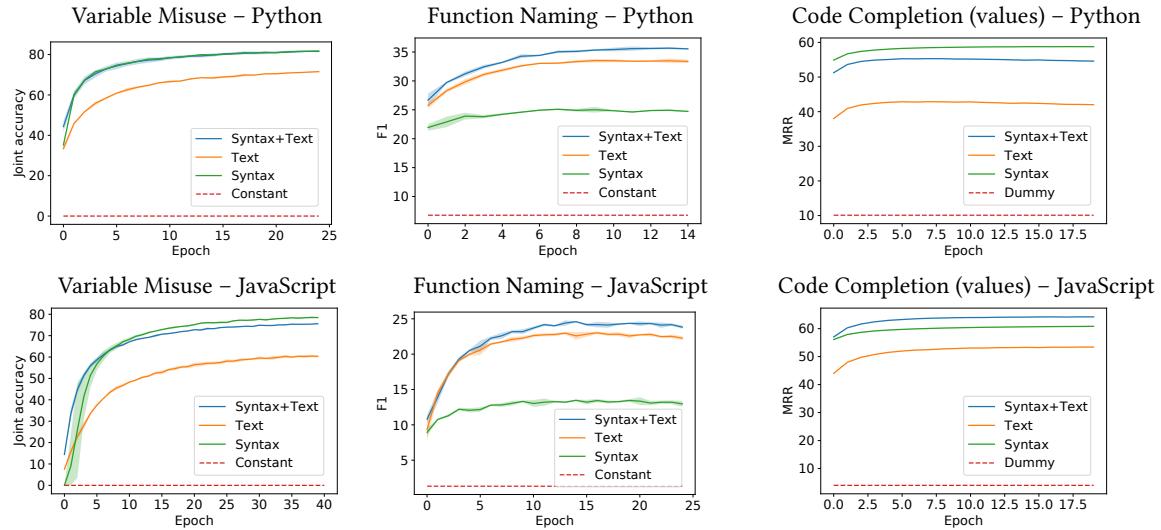
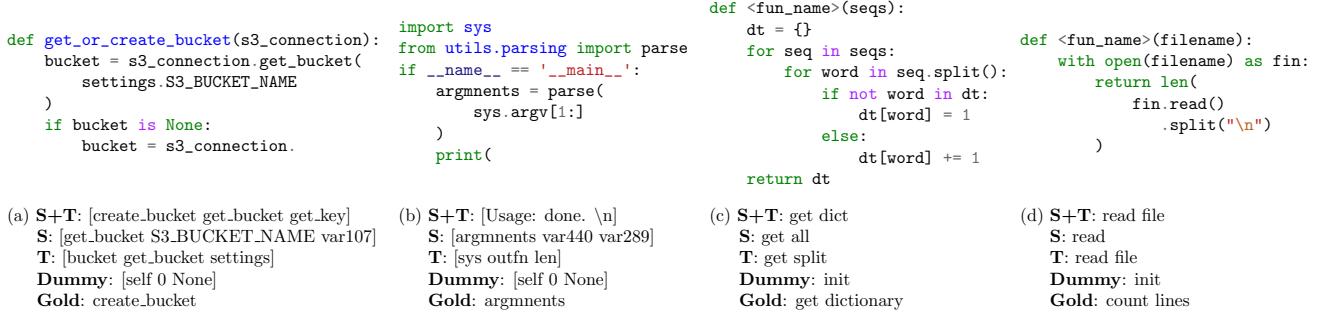**Figure 3: Comparison of syntax-based Transformer models with text-only and constant baselines.**

(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

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

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

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

Figure 4: Example predictions in code completion (a, b; top 3 predictions) and function naming (c, d) tasks. S: *Syntax,* T: *Text.*

(*structure*): AST structure is processed using one of the mechanisms discussed in Section 2.

As in Section 6, we consider both anonymized and full-data settings. We ablate AST components one by one in both models *Syntax* and *Syntax+Text* and check whether the quality drops. Ablating *types* for *Syntax+Text* was in fact performed in Section 6, but we repeat the results in this experiment's table and also report this ablation for the anonymized setting. Ablating *structure* means turning off all syntax-capturing mechanisms so that the input of the *Syntax+Text* / *Syntax* model will be viewed as an *unordered set* of (type, value) / (type, anonymized value) pairs. Ablating *values* means using only types as the input to the model — the numbers are the same for both full-data and anonymized settings. 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 5. *In variable misuse and code completion, all AST components are essential for achieving high quality results.* Particularly, in variable misuse, all ablations result in a large quality drop, in both settings, and in code completion, ablating types results in a large quality drop and ablating structure — in substantial drop. Interestingly, anonymization plays an important role in achieving high quality in the variable misuse task, with absent values from the data.

However, the observations differ for function naming. In this task, (1) ablating *types* results in a substantial quality drop in both settings, (2) ablating *structure* results in a small (but significant) quality drop in both settings, (3) ablating *values* in the full-data setting results in the large quality drop, and (4) ablating *anonymized values* does not affect the performance in the anonymized setting. The first and the third observations underline the importance of using both types and values in practice. The second and the fourth observations show that *Transformer is now far from utilizing all the information stored in AST when predicting function names*. Particularly, in the anonymized setting, Transformer predicts function names mostly based on *types*. It hardly uses syntactic *structure*, and does not use information about value repetition which is stored in *anonymized values* and is essential for understanding the algorithm that the code implements. Overcoming this issue is an interesting direction for future research.

Table 5: Ablation study of processing different AST components 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 types: only values or anonymized values are passed to Transformer; AST w/o an.val.: only types are passed to Transformer. N/A – not applicable.

		Full data			Anonymized data		
		Var. misuse	Fun. naming	Comp. (val.)	Var. misuse	Fun. naming	Comp. (val.)
Python	Full AST	**81.59**±0.50%	**35.73**±0.19%	**54.59**±0.2%	**81.71**±0.41%	**25.26**±0.15%	**58.76**±0.2%
	AST w/o struct.	26.81±0.47%	34.80±0.24%	53.1±0.1%	12.41±0.58%	23.29±0.18%	57.75±0.05%
	AST w/o types	71.55±0.28%	33.60±0.23%	42.01±0.05%	58.55±0.51 %	12.50±1.5%	41.26±0.05%
	AST w/o an.val.	32.44±0.35%	25.25 ±0.06%	N/A	32.44±0.35%	**25.25**±0.06%	N/A
JavaScript	Full AST	**75.60**±0.15%	**24.62**±0.14%	**64.2**±0.05	**78.47**±0.26%	**13.66**±0.30%	**60.82**±0.07%
	AST w/o struct.	17.25±0.83%	23.40±0.12%	61.53±0.15%	5.37±0.97%	11.25±0.08%	58.59±0.1%
	AST w/o types	60.33±0.50%	23.09±0.09%	53.4±0.1 %	43.53±0.92 %	8.10±1.4%	42.91±0.1%
	AST w/o an.val.	42.56±0.24%	13.64±0.07%	N/A	42.56±0.24%	**13.64**±0.07%	N/A

Table 6: Comparison of ensembles. Notation: ST – *Syntax+Text*, S – *Syntax*, & denotes ensembling. All models are trained with sequential relative attention. All numbers in percent, standard deviations: VM: 0.5%, FN: 0.4%, CC: 0.1%.

	Models	VM	FN	CC (types)	CC (values)
PY	ST	81.42	**35.73**	89.22	54.53
	ST & ST	82.80	35.61	**89.39**	56.35
	S	81.83	25.26	88.42	58.6
	S & S	82.57	25.46	88.65	59.29
	ST & S	**86.72**	32.15	**89.49**	**61.84**
JS	ST	76.52	**24.62**	90.14	64.11
	ST & ST	77.25	24.53	**90.56**	65.68
	S	78.53	13.66	88.22	60.71
	S & S	79.65	13.19	88.52	61.42
	ST & S	82.29	19.33	90.32	**68.33**

8 ENSEMBLING OF SYNTAX-BASED MODELS

As was shown in Figure 4, *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 initializations and averaging their predictions after softmax [8]. We use sequential relative attention in this experiment.

In Table 6, we compare an ensemble of *Syntax+Text* and *Syntax* models with ensembles of two *Syntax+Text* and of two *Syntax* 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.

9 VALIDATING OUR IMPLEMENTATIONS AND COMPARING TO OTHER WORKS

We ensure the validity of our results in two ways: by relying on the code of already published works, and by comparing our numbers

achieved for the commonly used data split to the numbers in the corresponding papers. Particularly, we use the model / loss / metrics / overlapping chunks code of Kim et al. [20] as the baseline for the CC task, we rewrite (line by line) the main parts of the model / loss / metrics code of Hellendoorn et al. [14] in PyTorch, as the baseline for the VM task, and we use the model / loss / metrics code of Ahmad et al. [1] as the baseline for the FN task.

For VM, the vanilla Transformer of Hellendoorn et al. [14] achieve 67.7% joint accuracy and we achieve 64.4%: the results are close to each other. Here the performance is given for our model, closest to the model of Hellendoorn et al. [14]: the *Text* model of similar size and with similar number of training updates; using our *Syntax+Text* model achieves higher quality. The performance of GGNN Sandwich is high on VM, as in [14]. For CC, with tree relative attention, we achieve 59.79 / 91.65 MRR (values / types) while Kim et al. [20] achieved 58.8 / 91.9 (their "TravTrans variant"); and for standard Transformer (no structure information), we achieve 59.66 / 89.16 MRR while Kim et al. [20] achieved 58.0 / 87.3 (their "TravTrans") respectively, again the results are close. For FN, we used the code of Ahmad et al. [1] with our custom data and targets, so the results are not comparable, but we checked that their code produces the same numbers on their data as in the paper.

The results given in our paper are for our custom data split and thus are not directly comparable to the numbers in other works. We argue that data resplitting is crucial for achieving correct results, see details in Section 3. At the same time, the remaining experimental setup (e. g. architecture, metrics) is the same as in recent works.

10 RELATED WORK

Variable misuse. The field of automated program repair includes a lot of different tasks, see [27] for a review, we focus on a particular variable misuse detection task. This task was introduced by Allamanis et al. [4] who proposed using GGNN with different types of edges to predict the true variable name for each name placeholder. Vasic et al. [36] enhances the VM task by learning to jointly classify, localize bug and repair the code snippet. They use an RNN equipped with two pointers that locate and fix the bug. Hellendoorn et al. [14] improved the performance on VM task, using Transformers, GREAT model, and GGNN Sandwich model.

Code summarization. The task of code summarization is formalized in literature in different ways: given a code snippet, predict the docstring [16], the function name [5], or the accompanying comment [17]. Allamanis et al. [5] propose using convolutional neural networks for generating human readable function names, while Iyer et al. [17] proposed using LSTM [15] with attention to generate natural language summaries. Alon et al. [7] proposed sampling random AST paths and encoding them with bidirectional LSTM to produce natural method names and summaries of the code. Fernandes et al. [11] proposed combining RNNs/Transformers with GGNN. Ahmad et al. [1] empirically investigate Transformers for code summarization, showing that Transformer with sequential relative attention outperforms Transformer equipped with positional encodings as well as a wide range of other models, e, g.RNNs.

Code completion. Early works on code generation built probabilistic models over the grammar rules. Maddison and Tarlow [26] learned Markov Decision Process over free context grammars, utilizing AST. Raychev et al. [29] learned decision trees predicting AST nodes. Sun et al. [33] generated code by expanding AST nodes, using natural language comments as additional source of information. Li et al. [24] used LSTM with a pointer, this model either generates the next token from vocabulary or copies the token from a previously seen position. Kim et al. [20] proposed using Transformers for code generation, enhanced them with tree relative attention and showed that the resulting model significantly outperforms RNN-based models as well as other models [29].

Recent advances in neural source code processing. The recent line of work is dedicated to learning contextual embeddings for code on the basis of Bidirectional Encoder Representations from Transformers (BERT) [9]. Such models are firstly pretrained on large datasets providing high-quality embeddings of code, and then fine-tuned on small datasets for downstream tasks [10, 12, 18]. All these Transformer-based models treat code as text and can potentially benefit from the further utilization of the syntactic information. Another line of research regards making Transformers more time- and memory-efficient [35]. Investigating the applicability of such methods to syntax-based Transformers is an interesting direction for future research.

Investigating neural networks for code with omitted variable names. A few of previous works considered training neural networks with omitted variable names: Gupta et al. [13], Xu et al. [39] trained RNNs on the data with anonymized variables, Ahmed et al. [2] replaced variables with their types. LeClair et al. [22] investigated the effect of replacing all values in the AST traversal with <unk> value in the code summarization task, and concluded that the quality of an RNN trained on such data is extremely low. Their result aligns with ours, while we consider a more general procedure of value anonymization (that saves information about value repetition) and investigate the effect of using anonymization in a wider set of tasks for a Transformer architecture.

11 THREATS TO VALIDITY

We did our best to make out comparison of different AST processing mechanisms as fair as possible. However, the following factors could potentially affect the validity of our results: using the same training hyperparameters for all models, not using subtokenization, and not using data- and control-flow edges in GGNN Sandwich. The decision not to use subtokenization was explained in Section 3. Moreover, we underline that sequential relative attention, our best performing mechanism, allows for easy combination with any subtokenization technique which will result in further quality improvement. On the contrary, this is not the case for more complex considered AST-processing mechanisms. The decision not use control- and data-flow edges was explained in Section 2. We note that adding data- and control-flow edges to the GGNN Sandwich equipped with sequential relative attention would increase the quality of this combined model even further. As for the training hyperparameters, tuning them for each model individually would be very expensive given our limited computational resources. However, we note that our models differ only in the AST-processing mechanism that is a relatively small change to the architecture. Thus we assume that using the same training hyperparameters for different models is permissible in our work.

12 CONCLUSION

In this work, we investigated the capabilities of Transformer to utilize syntactic information in source code processing. Our study underlined the following practical conclusions:

- sequential relative attention is a simple, fast and not considered as the baseline in previous works mechanism that performs best in 3 out of 4 tasks (in some cases, similarly to other slower mechanisms);
- combining sequential relative attention with GGNN Sandwich in the variable misuse task and with tree relative attention or tree positional encoding in the code completion task may further improve quality;
- omitting types, values or edges in ASTs hurts performance;
- ensembling Transformer trained on the full-data with Transformer trained on the anonymized data outperforms the ensemble of Transformers trained on the same kind of data.

Further, our study highlighted two conceptual insights. On the one hand, Transformers are generally capable of utilizing syntactic information in source code, despite they were initially developed for NLP, i. e. processing sequences. On the other hand, Transformers utilize syntactic information fully not in all tasks: in variable misuse and code completion, Transformer uses all AST components, while in function naming, Transformer mostly relies on a set of types and values used in the program, hardly utilizing syntactic structure.

ACKNOWLEDGMENTS

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[2] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation Error Repair: For the Student Programs, from the Student Programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training* (Gothenburg, Sweden) *(ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 78–87. https://doi.org/10.1145/3183377.3183383

[3] Miltiadis Allamanis. 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* (2019).

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 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. https://openreview.net/forum?id=BJOFETxR-

[5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference on Machine Learning (ICML)*.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 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. https://openreview.net/forum?id=H1gKYo09tX

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gKYo09tX

[8] Arsenii Ashukha, Alexander Lyzhov, Dmitry Molchanov, and Dmitry Vetrov. 2020. Pitfalls of In-Domain Uncertainty Estimation and Ensembling in Deep Learning. In *International Conference on Learning Representations, ICLR 2020*. https://openreview.net/forum?id=BJxI5gHKDr

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[11] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1ersoRqtm

[12] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[14] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *International Conference on Learning Representations, ICLR 2020*. https://openreview.net/forum?id=B1lnbRNtwr

[15] Sepp Hochreiter and JÃ©rgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735

[16] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 200–210. https://doi.org/10.1145/3196321.3196334

[17] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[18] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020 (Proceedings of Machine Learning Research)*. PMLR.

[19] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE 20)*. Association for Computing Machinery, New York, NY, USA, 1073–1085. https://doi.org/10.1145/3377811.3380342

[20] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code Prediction by Feeding Trees to Transformers. arXiv:2003.13848 [cs.SE]

[21] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *arXiv preprint arXiv:2006.03511*. arXiv:2006.03511 [cs.CL]

[22] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 795–806. https://doi.org/10.1109/ICSE.2019.00087

[23] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 4159–25.

[24] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 4159–4165. https://doi.org/10.24963/ijcai.2018/578

[25] I. Loshchilov and F. Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *ICLR*.

[26] Chris Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*, Eric P. Xing and Tony Jebara (Eds.). PMLR, Bejing, China, 649–657. http://proceedings.mlr.press/v32/maddison14.html

[27] Martin Monperrus. 2020. The Living Review on Automated Program Repair. (Dec. 2020). https://hal.archives-ouvertes.fr/hal-01956501 working paper or preprint.

[28] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/2983990.2984041

[29] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (Oct. 2016), 731–747. https://doi.org/10.1145/3022671.2984041

[30] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 761–774. https://doi.org/10.1145/2837614.2837671

[31] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 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)*. Association for Computational Linguistics, New Orleans, Louisiana, 464–468. https://doi.org/10.18653/v1/N18-2074

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

[33] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 05 (Apr. 2020), 8984–8991. https://doi.org/10.1609/aaai.v34i05.6430

[34] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1556–1566. https://doi.org/10.3115/v1/P15-1150

[35] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient Transformers: A Survey. arXiv:2009.06732 [cs.LG]

[36] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ByloJ20qtm

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V.

Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[38] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[39] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit Message Generation for Source Code Changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19.* International Joint Conferences on Artificial Intelligence Organization, 3975–3981. https://doi.org/10.24963/ijcai.2019/552

A ADDITIONAL EXPERIMENTAL DETAILS

A.1 Hyperparameter tuning

To ensure the fairness of comparing different mechanisms for processing AST in Transformer, we tune their hyperparameters individually for each task–dataset combination.

Sequential positional embeddings do not have hyperparameters. For sequential relative attention, we tune the maximum relative distance between the elements of the sequence. We consider options [8, 32, 128, 250]: from small distance to the maximum input length.

For tree positional encoding, we tune the maximum path width n_w and maximum path depth n_d. Since the embedding size is fixed and equal to $d_{\text{model}} = 512$, and the number of learnable parameters in the tree positional encoding equals $d_{\text{model}}/(n_w \cdot n_d)$ (see details in [32]), the greater values of n_w and n_d we use, the less number of the parameters in the encoding layer we have. We selected options to cover both cases when we have wide and deep paths but only one parameter as well as narrow and shallow paths but more parameters.

For tree relative attention, we tune the size of the relations vocabulary. All relations follow the pattern "k_u nodes up and k_d nodes down", $k_u, k_d = 0, 1, 2, \ldots$. We select the top of the most frequent relations in the dataset. We consider options from relatively small vocabulary to the full vocabulary of relations.

For GGNN Sandwich model, we consider 6-layer and 12-layer configurations of alternating Transformer (T) and GGNN (G) layers, we also consider placing both types of layers first i. e. [T, G, T, G, T, G] or [G, T, G, T, G, T] (and similarly for 12 layers). GGNN layers include 4 message passes. We also consider omitting edges of types Left and Right (see Figure 1(f)).

The results for hyperparameter tuning are given in Table 7. Each model was trained once, but since we have several choices for hyperparameters, we could analyse whether there are stable dependencies or the difference in quality is a result of noise. The choice of the maximum relative distance is stable across datasets in all tasks: in the VM task, distance equal to 8 is chosen, which is interpretable since in this task, local dependencies matter much; in contrast, in the FN task, small values of maximum distance perform poorly, and in this task, global dependencies are important. The similar reasoning applies to tree positional encoding: in the "local" VM task, small values of (n_w, n_d) are chosen, while in the "global" FN task, large values outperform (4, 8) combination. For tree relative attention, changing the relation vocabulary size does not affect quality much: rare relations' embeddings are updated only several times during training, so including them does not improve the performance. For the GGNN Sandwich, the choice of the optimal hyperparameters is again stable across datasets. In the VM task, the bigger the model, the higher the quality: deep 12-layer models substantially outperform 6-layers ones, and 3-edge-type models outperform 3-edge-type models. In contrast, in the FN task,

the 12-layer model fails to train properly, while the 6-layer model which achieves good quality.

A.2 Anonymization procedure.

We anonymize values so that inside one code snippet, different values are mapped to different placeholders var1, var2, var3 etc. We use random strategy for assigning placeholders: we fix the vocabulary of 1000 placeholders and for unique each value in each code snippet, choose a placeholder randomly. For example, the code snippet total_cnts[i][w] = title_cnt[i][w] + text_cnt[i][w] may be anonymized as follows: var341[var30][var89] = var785[var30][var89] + var453[var30][var89]. Anonymization of different code snippets is independent: one value may be replaced with different placeholders in different code snippets. For example, value total_cnt may be replaced with var341 in one code snippet and with var110 — in another one.

B COMPARISON OF APPROACHES FOR UTILIZING SYNTACTIC STRUCTURE IN TRANSFORMER IN THE ANONYMIZED SETTING

In this section, we compare mechanisms for utilizing syntactic structure in Transformer in the anonymized setting, i. e. when mechanisms are incorporated in the *Syntax* model described in Section 6. Due to the high computational cost of hyperparameter tuning, in this experiment, we use the same hyperparameters as in the full-data setting. In Section A we discussed that the selected hyperparameters are quite interpretable in different tasks, which allows to hypothesise that in the anonymized setting, the optimal hyperparameters would be the same.
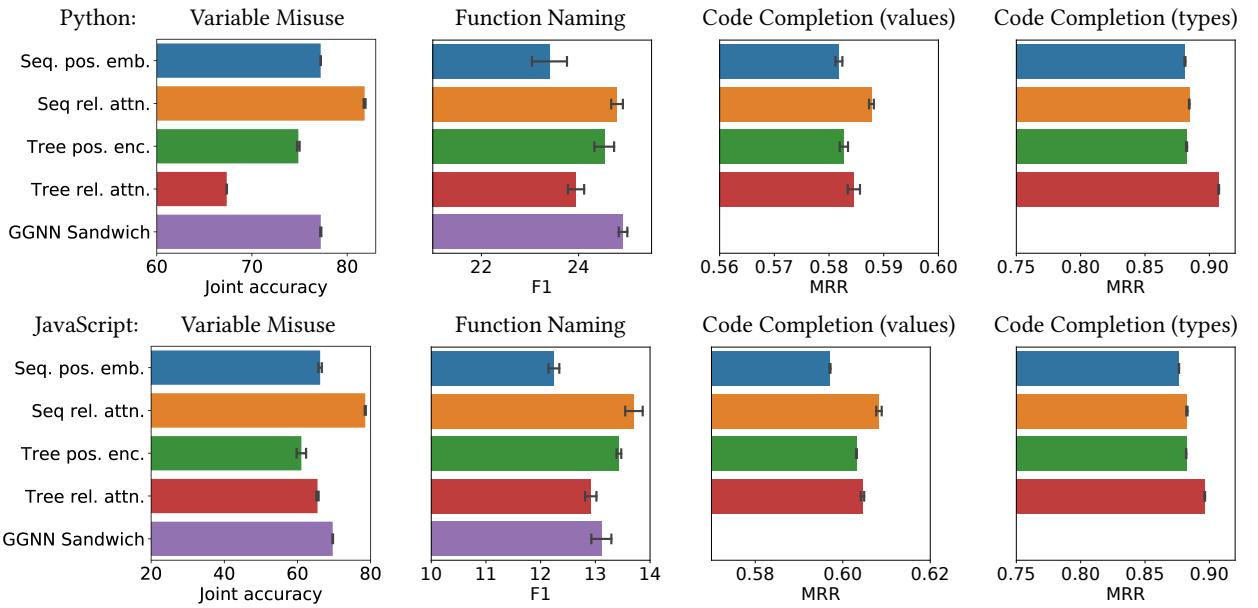
The results are shown in Figure 5. The leading approaches are the same as in the full data setting: sequential relative attention in the VM and CC (value prediction) tasks, tree relative attention in the CC (type prediction) task, and almost similar performance of different mechanisms in the FN task.

C TABLES AND ADDITIONAL PLOTS FOR COMPARING DIFFERENT STRUCTURE-CAPTURING MECHANISMS

Table 8 lists numerical data for barplots presented in Figure 2 in the main part of the paper (comparing AST-processing mechanisms in the full-data setting). Figure 6 visualizes the progress of test metrics during training for different AST-processing mechanisms in the full-data setting. This plots shows that the number of epochs we use is sufficient, i. e. the ordering of methods would not change and all models converged or almost converged. Table 8 lists numerical data for barplots presented in Figure 5 (comparing structure-capturing mechanisms in the anonymized setting).

Table 7: The results of hyperparameter tuning for different structure-capturing mechanisms, tasks and datasets. The quality is measured over the validation dataset (10% of the training data). Variable Misuse: joint localization and repair accuracy set, Function Naming: F1-measure; Code completion (predicting values): MRR, all numbers in percent. Datasets: PY — Python150k, JS – JavaScript150k. Notation: Y – Yes, N – No, all – the full vocabulary of relations is used. Bold font emphasizes the maximum among values in a column.

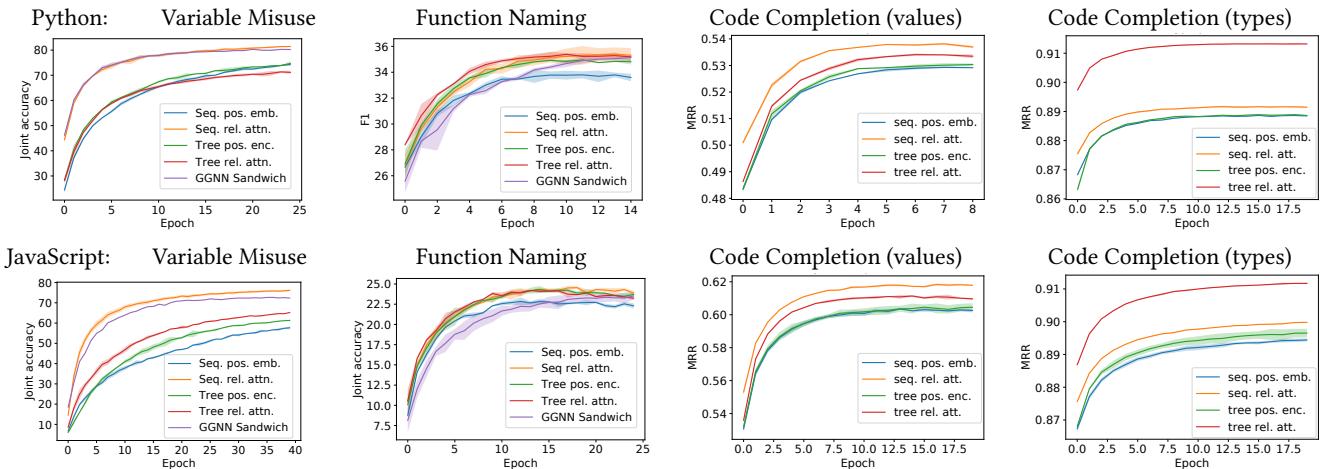| Model (hyperparams.)		Variable misuse			Function naming			Code completion	
		PY	JS		PY	JS		PY	JS
Sequential	8	**81.69**	**76.98**	8	32.63	22.36	8	52.68	61.36
relative attention	32	81.56	75.83	32	33.05	23.43	32	**53.04**	**61.73**
(max. relative distance)	128	80.94	75.52	128	33.43	23.43	128	53.04	61.66
	250	81.05	74.75	250	**33.54**	**23.76**	250	52.95	61.64
Tree positional	2, 64	64.74	55.06	2, 64	32.96	**23.72**	2, 64	50.89	54.49
encoding	4, 8	73.77	**62.14**	4, 8	30.75	21.00	4, 8	52.22	59.3
(max. path width,	8, 16	**74.53**	60.19	8, 16	33.06	22.29	8, 16	52.16	60.23
max. path depth)	16, 8	74.29	59.21	16, 8	**33.48**	22.41	16, 8	**52.38**	60.32
	16, 32	66.78	45.67	16, 32	32.85	23.02	16, 32	52.26	**60.42**
Tree relative	100	**71.69**	65.55	100	33.38	**24.25**	10	51.90	60.32
attention	600	71.31	**66.61**	600	**33.71**	23.78	100	52.54	60.94
(relations vocabulary size)	1500	71.52	64.38	1500	33.45	23.67	1000	**52.66**	**61.19**
	all	71.36	66.20	all	32.18	23.95	all	52.46	61.05
GGNN Sandwich	6, 2, N	70.39	65.25	6, 2, N	33.04	23.36			
(the number of layers,	6, 2, Y	70.25	65.87	6, 2, Y	**33.61**	**24.05**			
the number of edge types,	6, 3, N	78.16	68.74	6, 3, N	32.72	21.40			
is GGNN the first layer?)	6, 3, Y	78.33	71.06	6, 3, Y	31.79	21.49		N/A	
	12, 2, N	71.96	68.99	12, 2, N	5.37	5.11			
	12, 2, Y	72.60	68.30	12, 2, Y	9.85	5.11			
	12, 3, N	**80.71**	**73.88**	12, 3, N	21.26	3.40			
	12, 3, Y	80.61	73.09	12, 3, Y	20.36	6.27			

Figure 5: A comparison of different mechanisms for processing AST structure in Transformer, in the anonymized setting. The numeric data for barplots is given in Table 9. Please mind that the x-axis limits are different from Figure 2.

Table 8: The numerical data for the comparison of different mechanisms for processing AST structure in Transformer, in the full-data setting.

	Variable misuse		Function naming		Code completion (Values)		Code completion (Types)	
Model	PY	JS	PY	JS	PY	JS	PY	JS
Seq. pos. emb.	74.38±0.56	55.12±0.19	33.79±0.20	23.11±0.27	53.50±0.02	62.7±0.18	88.90±0.02	89.59±0.07
Seq. rel. attn.	81.42±0.16	76.52±0.34	35.49±0.35	24.67±0.18	54.53±0.07	64.11±0.03	89.22±0.02	90.14±0.002
Tree pos. enc.	74.65±0.99	61.29±0.41	34.95±0.40	24.45±0.21	53.82±0.007	62.70±0.45	88.96±0.005	89.71±0.21
Tree rel. attn.	71.33±0.54	65.17±0.09	35.19±0.17	24.24±0.11	54.00±0.10	63.39±0.11	91.36±0.01	91.2±0.03
GGNN Sandwich	80.23±0.15	72.58±0.14	35.40±0.21	23.57±0.50	N/A	N/A	N/A	N/A

Table 9: The numerical data for the comparison of different mechanisms for processing AST structure in Transformer, in the anonymized setting.
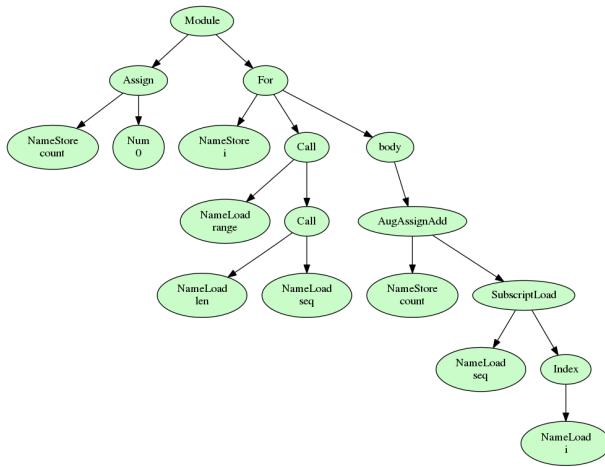
	Variable misuse		Function naming		Code completion (Values)		Code completion (Types)	
Model	PY	JS	PY	JS	PY	JS	PY	JS
Seq. pos. emb.	77.21±0.02	66.19±0.65	23.40±0.51	12.24±0.12	58.18±0.08	59.71±0.01	88.10±0.04	87.64±0.01
Seq. rel. attn.	81.83±0.12	78.53±0.17	24.79±0.15	13.70±0.19	58.78±0.05	60.83±0.08	88.44±0.03	88.25±0.06
Tree pos. enc.	74.86±0.19	61.07±1.82	24.52±0.25	13.43±0.05	58.27±0.09	60.31±0.01	88.23±0.05	88.19±0.01
Tree rel. attn.	67.35±0.03	65.47±0.44	23.94±0.20	12.91±0.12	58.45±0.16	60.45±0.05	90.72±0.04	89.64±0.03
GGNN Sandwich	77.22±0.08	69.64±0.11	24.91±0.11	13.11±0.22	N/A	N/A	N/A	N/A

Figure 6: A comparison of different mechanisms for processing AST structure in Transformer: test metrics by epochs.

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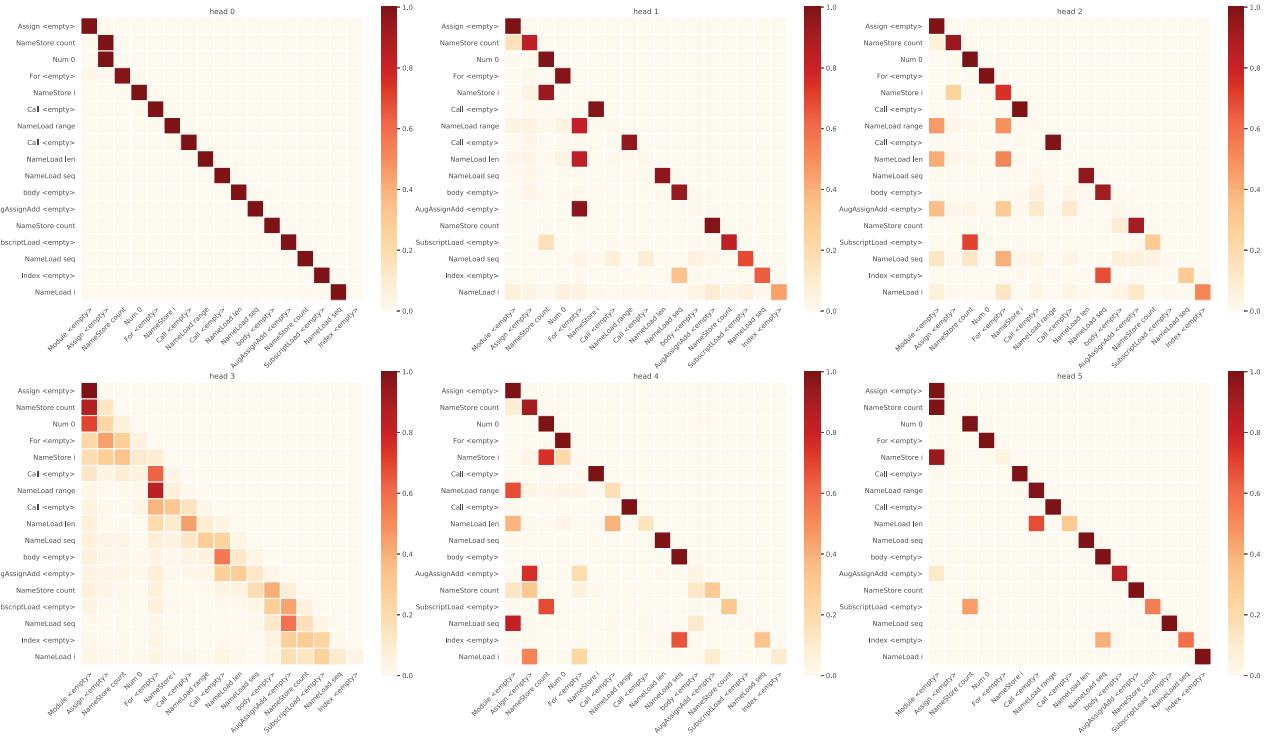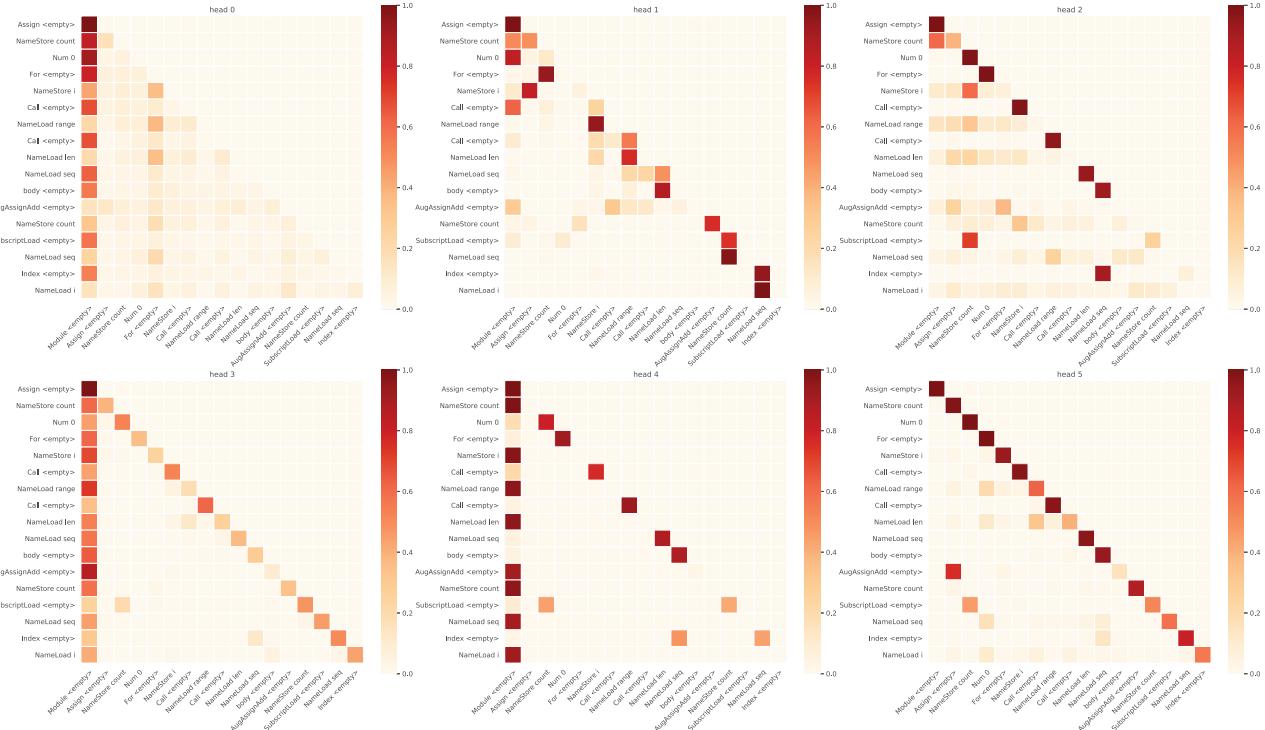

Figure 7: Code snippet (and its AST) used to visualize attention maps.

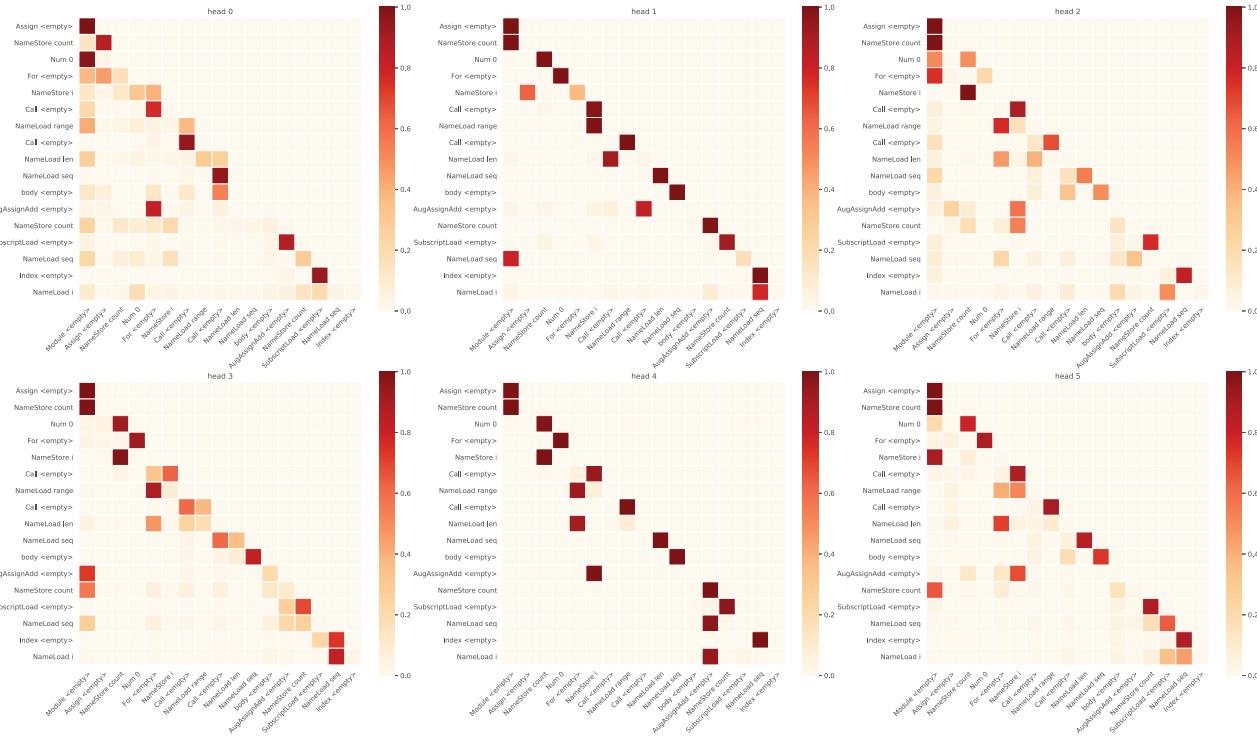## D ATTENTION MAPS FOR DIFFERENT STRUCTURE-CAPTURING MECHANISMS IN TRANSFORMER

In Figures 8, 9, 10, 11 we present attention maps for different structure-capturing mechanisms 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 mechanisms. We observe that all mechanisms except tree relative attention mostly attend to the last predicted tokens. Transformer with tree positional encodings (figure 9) 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 9. 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 11) 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.
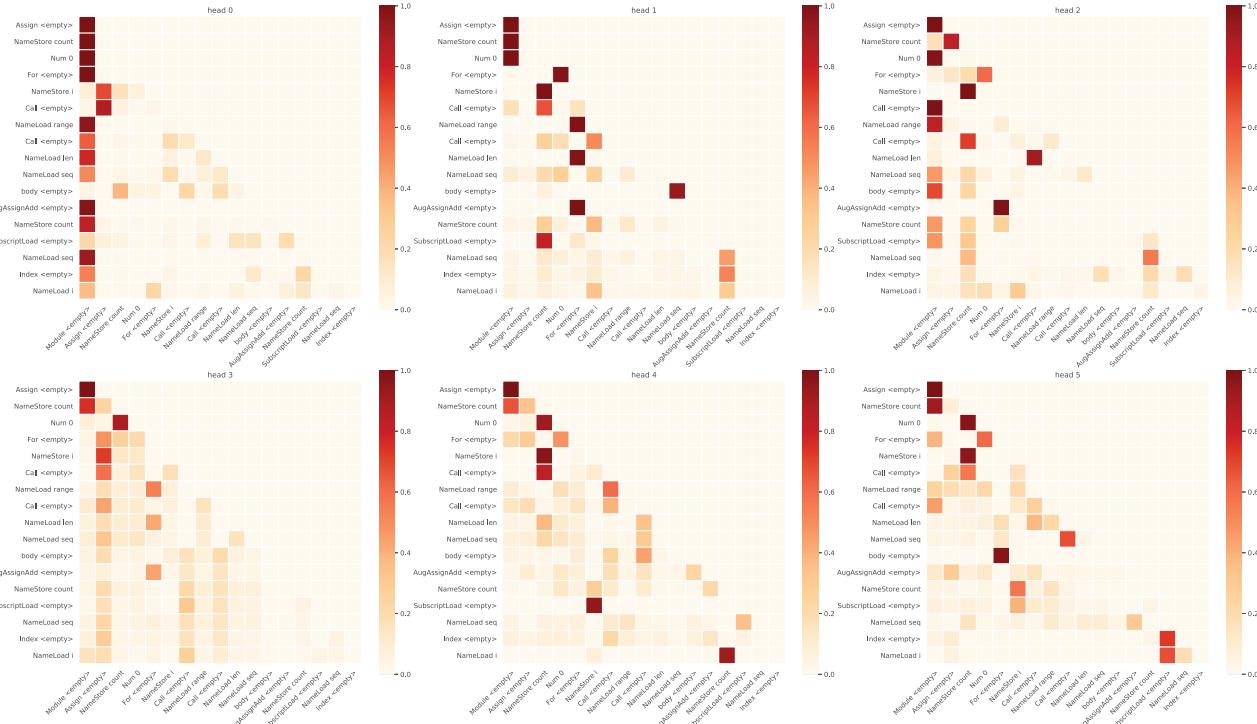
Figure 8: Attention maps for the 1st layer of *Syntax* model, sequential positional embeddings, code completion task. Y axis: what the model predicts, X axis: which token the model attends to.

Figure 9: 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 10: 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 11: 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.