Transformer with Tree-order Encoding for Neural Program Generation

Klaudia-Doris Thellmann*
TU Dresden

Bernhard Stadler
TU Dresden

Ricardo Usbeck
Fraunhofer IAIS

Jens Lehmann
Fraunhofer IAIS

Abstract

While a considerable amount of semantic parsing approaches have employed RNN architectures for code generation tasks, there have been only few attempts to investigate the applicability of Transformers for this task. Including hierarchical information of the underlying programming language syntax has proven to be effective for code generation. Since the positional encoding of the Transformer can only represent positions in a flat sequence, we have extended the encoding scheme to allow the attention mechanism to also attend over hierarchical positions in the input. Furthermore, we have realized a decoder based on a restrictive grammar graph model to improve the generation accuracy and ensure the well-formedness of the generated code. While we did not surpass the state of the art, our findings suggest that employing a tree-based positional encoding in combination with a shared natural-language subword vocabulary improves generation performance over sequential positional encodings.[1]

1 Introduction

Automatically generating source code from instructions in natural language can reduce the human-machine language barrier. Efforts on overcoming this barrier have led to numerous semantic parsing approaches ranging from statistical semantic parsing with focus on inducing rules (Zelle and Mooney, 1996) or probabilistic grammars (Zettlemoyer and Collins, 2005) to neural semantic parsing approaches based on encoder-decoder architectures, which have proven to be effective in mapping natural language into a formal meaning representation such as logical forms (Jia and Liang, 2016; Suhr et al., 2018) or general-purpose programming languages (GPPL) (Ling et al., 2016; Iyer et al., 2018; Yin and Neubig, 2017, 2018; Iyer et al., 2019).

[1]This paper was authored in late 2020 and early 2021 for the most part.

Recent syntax-driven neural semantic parsing approaches integrated compositional structures of logical forms (Dong and Lapata, 2016) or, in the case of GPPL, the underlying syntax (Rabinovich et al., 2017; Yin and Neubig, 2017; Hayati et al., 2018; Sun et al., 2019; Xu et al., 2020), which substantially improved generation accuracy. However, existing grammar-constrained decoding approaches for a GPPL like Python either use a manually defined list of production rules (Yin and Neubig, 2017) or the ASDL (Wang et al., 1997) abstract grammar definition (Python Software Foundation, 2018) used by the CPython implementation of the Python language (Rabinovich et al., 2017; Yin and Neubig, 2018). The manual approach allows for flexibility in modeling, but requires manually checking and possibly re-modeling the rules with every new release, which is prone to human error. Although the ASDL eliminates these drawbacks, it is not sufficiently restrictive and therefore cannot guarantee the generation of well-formed, i.e., parsable code. In compilers and interpreters, this is not problematic because some component in the pipeline consisting of parser, static checker and dynamic checker catches each of these types of error (Aho et al., 2007). In neural semantic parsing, however, it unnecessarily allows the generation of invalid code.

Building on this finding, we propose a syntax-driven neural semantic parsing approach employing a more restrictive grammar model for the task of generating well-formed code from instructions or descriptions in natural language as depicted in Table 1. Most syntax-driven neural semantic parsing approaches employ encoder-decoder architectures based on RNNs (Dong and Lapata, 2016; Rabinovich et al., 2017; Hayati et al., 2018; Iyer et al., 2019, i.a.). With the Transformer (Vaswani et al., 2017), a more powerful generation of neural architectures was introduced, surpassing RNNs in a variety of NLP tasks, including machine transla-

Dataset	NL Specification	Formal Meaning Representation
Atis	flight from ci0 mn0 dn0	*(lambda $0 e (and (flight $0) (from $0 ci0) (day_number $0 dn0) (month $0 mn0))*
Geo	What is the area of the state with the smallest population density?	*(area:i (argmin \$0 (state:t \$0) (density:i \$0)))*
CoNaLa	How to load a csv file?	*import csv* *with open('file.csv') as csvfile:* *reader = csv.DictReader(csvfile)*
Hearthstone	Treant NAME_END 2 ... Minion TYPE_END Druid PLAYER_CLS_END NIL RACE_END NIL RARITY_END NIL	*class Treant(MinionCard):* *def __init__(self):* *super().__init__("Treant", 1,* *CHARACTER_CLASS.DRUID,* *CARD_RARITY.COMMON)* *def create_minion(self, _):* *return Minion(2, 2)*

Table 1: Examples of natural language specifications or questions and their corresponding formal meaning representation from the datasets used in this work: Atis (Dahl et al., 1994), Geo (Tang and Mooney, 2001), Hearthstone (Ling et al., 2016) and CoNaLa (Yin et al., 2018)

tion (Nguyen et al., 2020; Chen et al., 2018) and constituency parsing (Wang et al., 2019; Harer et al., 2019). Unlike RNNs, Transformers rely solely on self-attention and therefore can capture long-range context dependencies while being easy to parallellize.

Since the Transformer is invariant to sequence ordering, it is of particular importance to explicitly include position information of sequence tokens through learned or fixed positional encoding schemes (Vaswani et al., 2017; Shiv and Quirk, 2019) or biased attention weights (Shaw et al., 2018; Nguyen et al., 2020). While the inclusion of hierarchical information has proven to be effective for code generation, the positional encoding by (Vaswani et al., 2017) can only represent positions in a flat sequence. Taking up this thought, we propose a position encoding scheme for the Transformer which allows the attention mechanism to also attend over hierarchical positions in the input. Similar to (Shiv and Quirk, 2019), we also rely on paths in an abstract tree representation of input code snippets to encode hierarchical code structure. However, (Shiv and Quirk, 2019)'s encoding, cannot be applied to trees of variable width since the depth and width of the encoding is limited.

Our contributions are the following:

1) A restrictive grammar graph model, which can be automatically generated from any tree-like data, not requiring an ASDL with the only condition that ordered typed trees are available as input. The grammar model is used to determine the hierarchical relations for the tree encoding, but can also be used for constrained decoding, ensuring the generation of parsable code.

2) A tree encoding scheme based on the sinusoidal encoding introduced by (Vaswani et al., 2017) to enable the Transformer model to learn hierarchical relations in an ordered tree of arbitrary width.

We evaluated our approach on several benchmark datasets for code generation including Hearthstone (Ling et al., 2016), CoNaLa (Yin et al., 2018), Atis (Dahl et al., 1994) and Geo (Tang and Mooney, 2001). Experimental results show that he tree encoding performs better than the sequential encoding used by the original Transformer architecture on the Hearthstone dataset. On the CoNaLa dataset, we achieve improvements when employing a separate subword vocabulary for the string literals extracted from the code snippets. Our code is publicly available at `https://github.com/SmartDataAnalytics/codeCAI`.

2 Related Work

A large variety of syntax-driven neural semantic parsing approaches generate abstract syntax trees (ASTs) by predicting a sequence of grammar rules. Yin et al. (Yin and Neubig, 2017, 2018), Rabinovich et al. (Rabinovich et al., 2017) and Sun

et al. (Sun et al., 2019) define a grammar model that captures the syntax of the target programming language and generate ASTs based on a series of predicted actions, i.e., apply a production rule to expand a non-terminal node or populate a terminal node. Most of these approaches derive the underlying grammar model from the official Python abstract grammar definition (ASDL). However, abstract grammar definitions are designed for compilation or interpretation (Aho et al., 2007) and not for code generation and therefore, as in the case of the Python grammar, allow syntax trees that do not correspond to parsable code. We counteract this drawback by creating a more restrictive grammar model from the parsable code samples of the training dataset. Based on this grammar model, we generate ASTs by predicting the next AST node on the input tree path in depth-first preorder.

To capture the compositional structure of logical forms, (Dong and Lapata, 2016) propose a tree decoder that recursively generates sub-trees, conveying hierarchical information by feeding the parent node's hidden representation as an additional input. Expanding on this, (Rabinovich et al., 2017) realize a grammar-based modular decoder with a dynamically determined composition of modules that reflect the structure of ASTs. (Sun et al., 2020) propose a grammar dependent Transformer architecture for code generation and use convolutional layers to process natural language input and model hierarchical structure. In contrast, we convey the compositional structure through the attention mechanism aided by a hierarchical positional encoding, not through parent feeding. Since we do not employ a grammar dependent architecture, with constrained decoding disabled, our approach can be applied to any other tree-structured data for which no abstract grammar is available.

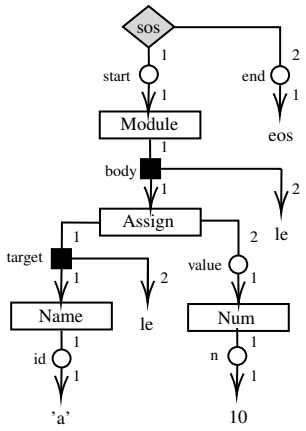
For encoding nodes in a tree, we use the same notion of path as (Shiv and Quirk, 2019), and like theirs, our tree encoding for a path can be described as a composition of affine transformations. However, with (Shiv and Quirk, 2019)'s encoding, there is only one affine transformation per path segment encoding, which might limit the robustness of the encoding. While their tree encoding of a node results from concatenating the one-hot encodings of the node's reversed path segments and weighting the encodings using a learned parameter, we employ the parameter-free sinusoidal encoding by (Vaswani et al., 2017) to encode each path segment. Since the tree depth and width of (Shiv and Quirk, 2019) encoding is fixed, it is not possible to apply it on arbitrarily wide trees, and sibling relationships can only be represented in binary trees. These limitations do not apply to our encoding scheme, which can be applied to ordered trees of variable width.

3 Data Representation

3.1 Input Data

As input for training, we use several Python code corpora (Ling et al., 2016; Yin et al., 2018; Oda et al., 2015) consisting of pairs of natural language instructions and corresponding code snippets. We parse the Python code snippets into abstract syntax trees (ASTs) and generate a sequence of AST tokens in depth-first preorder, as depicted in Figure 1a. The AST consists of object nodes (e.g. *Module*) and attribute nodes visualised by small squares and circles in the figure. An attribute node is either a singleton (e.g. *value*) or an attribute list (e.g. *body*) that can point to several object nodes. We integrated a special node for marking an attribute list end (*le*) and special nodes for marking the start (*sos*) and the end (*eos*) of the code fragment. An object node can be an inner node (e.g. *Module*) or a leaf node (e.g. *'a'*) and can have none, one or more attribute nodes. Each attribute node has a defined order in its object node's list of attribute nodes. An edge from an object node to an attribute indicates that the attribute belongs to that object node, and an edge from an attribute to an object node means that instances of that object node are allowed to appear as the value of the attribute.

We create the sequential representation of an AST by traversing the AST in depth-first preorder and adding the names of the object nodes to the AST sequence. An AST sequence consists of object node tokens and special nodes tokens. If an object node is the last or the only node in an attribute list, we add a list end token after the object node's token in the AST sequence. From the ASTs of the code samples corpora only used for training, we create a grammar model that captures the Python programming language syntax. From the natural language input and the string literals extracted from the ASTs we create a common subword vocabulary using the BPE implementation of (Kudo and Richardson, 2018). From the tokens from the AST sequence, we generate a different word vocabulary without string literals. Using a BPE segmentation



(a) AST

Node	Path
sos	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Module	[1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
Assign	[1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
Name	[1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
'a'	[1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
le	[2, 1, 1, 1, 1, 1, 0, 0, 0, 0]
Num	[1, 2, 1, 1, 1, 1, 0, 0, 0, 0]
10	[1, 1, 1, 2, 1, 1, 1, 1, 0, 0]
le	[2, 1, 1, 1, 0, 0, 0, 0, 0, 0]
eos	[1, 2, 0, 0, 0, 0, 0, 0, 0, 0]

(b) Edge paths

Figure 1: (a) Example of the abstract syntax tree (AST) of the code snippet `a=10`. We create the AST sequence by adding the names of the object nodes in depth-first preorder: `[sos, Module, Assign, Name, 'a', le, Num, 10, le, eos]`. (b) Object node edge paths for the AST of the snippet `a=10`. All edge paths have the same length. Each edge path consists of the path segments of its predecessor nodes.

for string literals has the advantage that we can effectively adjust the vocabulary size and reduce the number of tokens required to encode the input sequences (Kudo, 2018).

3.2 Grammar Model

An ASDL grammar does not necessarily specify the exact set of valid ASTs, but only a context-free, parsing-optimized superset of ASTs. As a result, the grammar is not sufficiently restrictive and therefore allows the generation of ASTs that do not correspond to any parsable program, or ASTs that are parsable but would not withstand simple static checks. For instance, the Python ASDL allows spaces within variable identifiers or constant string or number literals on the left side of an assignment, which is an invalid syntax. CPython's parser (3.7) still allows constructs such as function calls or index operations with constants as targets to be parsed into ASTs, e.g. "1[2] = 3()", resulting in a static check error.

As an improvement, we propose to generate a restricted grammar model that allows only AST compositions (i.e. attribute-child combinations) that occur in a corpus of Python code samples, e.g. the train dataset of a task. The construction of the grammar graph essentially induces a context-free generalization of the ASTs from the training dataset containing only parsable code samples. This way, the likelihood of generating invalid ASTs can be reduced significantly. The grammar graph defines a sub-language of the language defined by the ASDL by construction, so the knowledge contained in the ASDL is still preserved in the grammar graph and not discarded. Indeed, the set of ASTs is restricted to what occurs in the training data, but this will more likely prevent the model from predicting invalid rather than valid AST compositions. The advantage of the restrictive grammar graph is that the model can only learn what it has seen in the data, ensuring the generation of parsable ASTs. In any case, the model does not allow the prediction of AST combinations that it has not seen, i.e., that did not occur in the training data.

We define the grammar model as a bipartite labeled directed graph, whose nodes are the AST object nodes and attribute nodes. The grammar graph, is used to compute edge paths for the tree encoding and can also be used to create constraint masks for limiting possible candidates from the current prediction during inference.

4 Tree Encoding

Since the Transformer model is invariant to the input order, due to its attention-based architecture, required position information needs to be explicitly included in the input sequence. Thus, (Vaswani et al., 2017) encode positions in a sequence with a parameter-free scheme using sinusoidal functions of different frequencies and add these position encodings to the embeddings of the input sequence tokens of the first encoder and decoder layer.

However, flattening ASTs into an input sequence of AST tokens does not preserve the hierarchical order between the tree nodes. In particular, nodes whose tokens are next to each other in the AST

sequence can be far apart in the tree, namely if the preceding token in the AST sequence is the last node of a deep subtree. Conversely, the tokens of two neighboring nodes in the AST are far apart in the sequence when the first of two direct siblings is the root of a large subtree.

To counteract this problem, we propose a tree-based positional encoding built on (Vaswani et al., 2017)'s sinusoidal encoding scheme, which allows the attention mechanism to also attend over hierarchical positions in the input AST sequence.

4.1 Edge Paths

To define the tree encoding, we interpret the AST as an ordered tree $T = (V_T, E_T, r_T)$ consisting of a set of nodes V_T with n object nodes and m attribute nodes, a set of edges E_T and root r_T.

A node $v \in V_T$ is uniquely identified by an edge path p_v of a configurable length L if the depth of T is at most L. An edge path is not a path as in graph theory, i.e., a sequence of neighboring nodes, but rather structured like a path in a file system. For each object and attribute node in the AST, the outgoing edges receive a consecutively numbered edge index idx. For example in Figure 1a, the first edge of the object node *Assign* to the attribute node *target* has index 1 and the second edge to the attribute node *value* has index 2.

The edge path of a node consists of the edge indices of the edges on the path from the node to the root node. Edge paths shorter than L are padded with zeros and the root node's edge path consists only of zeros. For example, the edge path of *Assign* consists of the indices of the four edges on the path to the root node *sos*.

Formally, each edge path $p_v = (idx_{j,l})_{j=1\,l=1}^{n\;\;L}$ consists of edge indices that lead from the target node upwards to the root node. The l-th edge index of the edge path refers to the l-th edge of the (reverse) path from the target node up to the root.

4.2 Encoding Scheme

To encode the position of a node v in a tree T, we apply the sinusoidal encoding by (Vaswani et al., 2017) to each edge index of the node's edge path p_v and compute the positional tree encoding $TE_{j,l} = \left(EE_{idx_{j,l},i\prime}\right)_{i\prime\in\{1,\ldots,d_{idx}\}}$ with $j \in \{1,\ldots,n\}$ and $l \in \{1,\ldots,L\}$ as follows:

$$EE_{idx_{j,l},2i} = sin(\omega_i \cdot idx_{j,l})$$

$$EE_{idx_{j,l},2i+1} = cos(\omega_i \cdot idx_{j,l})$$

with $\omega_i = 1/10000^{\frac{2i}{d_{idx}}}$ for $i \in \{1,\ldots,\frac{d_{idx}}{2}\}$.

The parameter d_{idx} is the dimension of the positional encoding of an edge index $idx_{j,l}$ and determines the number of encodings per edge index, i.e., $\frac{d_{idx}}{2} \in \mathbb{N}$ sine and cosine pairs. This results in an encoding of length $d_{idx} * L$ for an edge path, which also corresponds to d_m, the size of the node's embedding.

The tree encoding TE_{v_j} of a node v_j is the concatenation of the encodings of its edge path indices:

$$TE_{v_j} = TE_{j,1} \| \ldots \| TE_{j,L}$$

with $j \in \{1,\ldots,n\}$.

4.3 Encoding Properties

The tree encoding entails the following properties, which result from the composition of the edge paths and the sine and cosine functions used to encode the edge paths:

Uniqueness property The concatenation of the edge indices encodings of a edge path results in a unique tree encoding for a node v_j. This is because for a node v_j, the combination of its edge path indices $idx_{j,l}$ is unique and the sine (and cosine) function is injective on the products of its edge path indices and each frequency $\omega_{i\prime}$ due to the transcendality of π.

Shifting property The tree encoding of a child node v_j contains the first $(L-1) * d_{idx}$ dimensions of its parent node's tree encoding, shifted to the right by d_{idx} dimensions. Two nodes are siblings if these shifted parent dimensions contain identical values for both nodes. Similar to (Shiv and Quirk, 2019), the shifting property is intended to enable the attention mechanism to identify whether a node is an ancestor or a sibling of a node v_j. However, relative distances between sibling nodes cannot be represented by shifting alone.

Linear combination property Due to the mathematical properties of the sine and cosine functions (Vaswani et al., 2017) hypothesizes that it is possible for the attention mechanism to attend to sequential order relationships, i.e., to learn relative positions. We make use of this property to allow attending to sibling nodes by their relative positions. From the positional encoding of an edge index $idx_{j,l}$, the positional encoding of another

edge index $idx_{j,l} + k$ can be computed for any integer offset k, using the sum rules for trigonometric functions. The positional encoding of a siblings of a node v_j with an offset $k < 0$ and l = 1 can be expressed as a linear combination of the sine and cosine values of the node v_j edge encoding.

5 Evaluation

5.1 Datasets

For the evaluation we use the Python code generation benchmarks CoNaLa (Yin et al., 2018) and Hearthstone (HS) (Ling et al., 2016) and for semantic parsing we use the datasets GEO (Tang and Mooney, 2001) and Atis (Dahl et al., 1994).

Hearthstone is a corpus for the automatic generation of code for cards in the trading card game HearthStone and consists of 665 Python classes, each representing one card. CoNaLa is a Python corpus containing 2,379 curated and 593,891 mined NL-code pairs mined from the developer forum Stack Overflow. CoNaLa contains as NL intent real-world questions to diverse implementation topics instead of pseudo-code annotations. GEO is a corpus of natural language questions about US geography and consists of 600 training and 280 test examples of NL-Prolog queries pairs. Atis is a corpus of natural language queries for a flights database featuring 4473 training and 448 test examples of NL-lamba-calculus pairs.

5.2 Metrics

As evaluation metrics, we use the BLEU score as implemented by (Yin et al., 2018), the exact match accuracy and the token- and sequence-level prefix precision and recall.

The BLEU score measures the similarity between the generated code and the reference code in terms of n-grams. We compute the token-level BLEU score on normalized and tokenized code by determining the precision on n-grams, then take the geometric mean of the precisions, and apply a brevity penalty for predictions shorter than the expected code.

Exact match accuracy measures the ratio of predictions that were predicted without a single error. This ensures the semantic correctness of the code, but it is very conservative in that it doesn‘t allow the slightest syntactic variation, and that no distinction is made between a prediction that is 0% correct and one that is 99% correct.

Token- and sequence-level prefix precision and recall interpolate exact match accuracies based on the idea that if some prefix was predicted correctly, the model didn‘t make a mistake up to that point, so it is alright to give the model some credit for that. On token level, it measures the ratio of tokens in correct prefixes, while on sequence level, it measures the average percentage of the longest correct prefix, ignoring snippet length. The prefix-based metrics also partially measure semantical correctness because the expected snippet is by definition semantically correct.

5.3 Experimental Setup

We performed the experiments on a cluster of IBM AC922 nodes with dual Power9 CPUs (2.80-3.10 GHz, 22 cores each), 256 GB RAM and 6 Nvidia Volta V100 accelerators with 32 GB RAM. And on nodes with AMD EPYC CPUs (2.3 GHz, 24 cores each), 1 TB RAM and 8 A100 GPUs with 40 GB RAM per node.

In the following experiments, unless specified otherwise, we train each task for 500 epochs with batch size 15, learning rate 0.0001, 6 encoder and 6 to 8 decoder layers, 16 attention heads, model dimension 512, maximum tree height 32 and feed-forward dimension 2048. For inference, we set the number of beams to $k = 30$ and use a maximum beam length of 250 tokens.

5.4 Evaluation Results

We performed the evaluation on two tasks, namely semantic parsing on GEO and ATIS and code generation on CoNaLa and Hearthstone. In both cases, the goal is to generate formal meaning representations from natural language input, that is, lambda-calculus expressions or Python code.

We scored about 18% and 70% BLEU score for the CoNaLa and Hearthstone benchmarks, respectively (cf. table 2). State of the art approaches have more sophisticated implementations that include dynamic composite neural architecture, additional embedded information, pre-training, or re-ranking.

Our results on the semantic parsing datasets GEO and ATIS are listed in table 3. On ATIS we achieved an exact match accuracy of 86,1%, which is comparable to all our baselines except for the leading approach. The exact match accuracy on GEO is below the baselines, which we conjecture to be caused by the small amount of training data.

To test whether the tree-encoded Transformer learns to predict the AST structure correctly, we

Authors	Name	BLEU
Yin et al. 2019	tranX	24,4%
Yin et al. 2019	tranX + rerank	30,1%
Xu et al. 2020	tranX + pre-trained	**32,3**%
Our system		18,1%

(a) CoNaLa

Authors	Name	BLEU
Yin et al. 2019	tranX	75,8%
Hayati et al. 2018	ReCode	78,4%
Rabinovich et al. 2017	ASNs	79,2%
Sun et al. 2020	TreeGen-B	**81,8**%
Our system		70,7%

(b) Hearthstone

Table 2: Comparison BLEU score with the state of the art on CoNaLa and Hearthstone.

Authors	Name	GEO	Atis
Rabinovich et al. 2017	ASNs	87,1%	85,9%
Yin & Neubig 2018	tranX	88,2%	86,2%
Dong & Lapata 2018	oracle sketch	**93,9**%	**95,1**%
Shiv et al. 2019	Seq2Tree Tform	84,6%	86,4%
Our system		80,4%	86,1%

Table 3: Comparison EM accuracy with the state of the art on GEO and Atis.

Pos. Enc.	Incl. Str. Lit.	EM Acc.	Seq vs. Tree	Seq. Recall	Seq vs. Tree	Token Recall	Seq vs. Tree	Seq. Prec.	Seq vs. Tree	Token Prec.	Seq vs. Tree
Seq	yes	4,5%		26,9%		23,0%		28,4%		26,4%	
Seq	no	10,6%		48,9%		42,5%		52,2%		48,7%	
Tree	yes	7,6%	**+3,0%**	28,9%	**+2,0%**	23,7%	**+0,7%**	30,5%	**+2,1%**	27,9%	**+1,4%**
Tree	no	12,1%	**+1,5%**	50,0%	**+1,1%**	43,7%	**+1,2%**	53,9%	**+1,7%**	51,3%	**+2,7%**

Table 4: Exact match accuracy and longest common prefix on Hearthstone.

looked at the exact match accuracy and token- and sequence-level precision and recall, as shown in table 4.

We masked all string literals and computed the longest common prefix between the best predicted and the expected AST sequence for each sample from the test dataset. We do this on a model we trained with tree encoding and then on a model we trained with sequential encoding.

With the exclusion of string literals, the prefix precision and recall jump from about 25% to about 50% with both sequential and tree encoding. From the prefix analysis, we can take away that the string literals have a significant impact on the quality of the prediction and that longer sequences are more difficult to predict. What we also found is that tree encoding gives an improvement of up to 3.0% when excluding string literals over sequential encoding.

6 Conclusion

We propose a Transformer-based architecture with a tree-based positional encoding and constrained decoding based on a grammar model derived from training data. We evaluate it on four different datasets. While we do not surpass state-of-the-art methods, we see relative improvement of applying tree encoding over sequential encoding.

In the future, we will work on improved versions of training with constraint masks and grammar models that respect larger contexts than only the immediate containing attribute. Also, we will further investigate attention-based mechanisms for generating tree structures, also in the context of domain-specific languages as generation target.

7 Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF, 01IS18026A-D) by funding the competence center for Big Data and AI "ScaDS.AI Dresden/Leipzig". The authors gratefully acknowledge the GWK support for funding this project by providing computing time through the Center for Information Services and HPC (ZIH) at TU Dresden.

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: principles, techniques, & tools*, 2. ed., pearson internat. ed edition. Pearson, Addison Wesley. pp. 97-99 and 386-387.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 2552–2562.

Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. 1994. Expanding the scope of the atis task: The atis-3 corpus. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43. Association for Computational Linguistics.

Jacob Harer, Christopher P. Reale, and Peter Chin. 2019. Tree-transformer: A transformer-based method for correction of tree-structured data. Computing Research Repository (CoRR).

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930. Association for Computational Linguistics.

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5425–5434. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652. Association for Computational Linguistics.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany. Association for Computational Linguistics.

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

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

Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. 2020. Tree-structured attention with hierarchical accumulation. In *International Conference on Learning Representations*.

Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584.

Python Software Foundation. 2018. CPython 3.7 abstract grammar definition "Parser/Python.asdl".

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

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"*. Association for Computational Linguistics.

Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to map context-dependent sentences to executable formal queries. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2238–2249. Association for Computational Linguistics.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural CNN decoder for code generation. 33:7055–7062.

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):8984–8991.

Lappoon R. Tang and Raymond J. Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Machine Learning: ECML 2001*. Springer Berlin Heidelberg.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc.

Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages October 15-17, 1997*, pages 213–228, Santa Barbara, California, USA.

Yaushian Wang, Hung-Yi Lee, and Yun-Nung Chen. 2019. Tree transformer: Integrating tree structures into self-attention. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1061–1070. Association for Computational Linguistics.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052. Association for Computational Linguistics.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 476–486, New York, NY, USA. Association for Computing Machinery.

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

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, page 1050–1055. AAAI Press.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 658–666. AUAI Press.