StructCoder: Structure-Aware Transformer for Code Generation

SINDHU TIPIRNENI, MING ZHU, and CHANDAN K. REDDY, Department of Computer Science, Virginia Tech, Arlington, VA, USA

There has been a recent surge of interest in automating software engineering tasks using deep learning. This article addresses the problem of code generation, in which the goal is to generate target code given source code in a different language or a natural language description. Most state-of-the-art deep learning models for code generation use training strategies primarily designed for natural language. However, understanding and generating code requires a more rigorous comprehension of the code syntax and semantics. With this motivation, we develop an encoder-decoder Transformer model in which both the encoder and decoder are explicitly trained to recognize the syntax and dataflow in the source and target codes, respectively. We not only make the encoder structure aware by leveraging the source code's syntax tree and dataflow graph, but we also support the decoder in preserving the syntax and dataflow of the target code by introducing two novel auxiliary tasks: Abstract Syntax Tree (AST) path prediction and dataflow prediction. To the best of our knowledge, this is the first work to introduce a structure-aware Transformer decoder that models both syntax and dataflow to enhance the quality of generated code. The proposed StructCoder model achieves state-of-the-art performance on code translation and text-to-code generation tasks in the CodeXGLUE benchmark and improves over baselines of similar size on the APPS code generation benchmark. Our code is publicly available at https://github.com/reddy-lab-code-research/StructCoder/.

CCS Concepts: • **Computing methodologies** → **Neural networks**; *Natural language processing*; • **Software and its engineering** → *Automatic programming;*

Additional Key Words and Phrases: Deep learning, language models, code generation, Transformer

ACM Reference format:
Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. StructCoder: Structure-Aware Transformer for Code Generation. *ACM Trans. Knowl. Discov. Data.* 18, 3, Article 70 (January 2024), 20 pages.
https://doi.org/10.1145/3636430

1 INTRODUCTION

Code generation is the problem of generating code in a specified target language given source code that is either imperfect or in a different language, or generating code from a natural language description. In this article, we consider the problem of generating target code given source code in a different language (code translation) or a natural language description (text-to-code generation). Code translation has applications in migrating legacy codebases to contemporary programming languages and porting existing software to various other platforms [1, 27, 35]. Text-to-code

Authors' address: S. Tipirneni, Ming Zhu, and C. K. Reddy, Dept. of Computer Science, Virginia Tech, 900 N Glebe Rd, Arlington, VA 22203; e-mails: tsaisindhura@vt.edu, mingzhu@vt.edu, reddy@cs.vt.edu.

1556-4681/2024/01-ART70
https://doi.org/10.1145/3636430

generation models can potentially increase programmers' productivity by simplifying and speeding up the software development process, as developers often write code to solve a problem or implement logic that is stated in natural language[1]. Transformer-based deep learning methods have recently gathered significant attention in this domain. However, these existing models do not effectively utilize the code structure, especially during the decoding of target code. To address this limitation, we propose StructCoder, which models the syntax and dataflow in both source and target codes with a structure-aware encoder and decoder.

Traditional code translation tools have been designed using hand-crafted rules based on the **Abstract Syntax Tree (AST)** [27]. One such popular tool is Babel[1], which converts modern JavaScript code to older versions for backward compatibility. Other notable source-to-source translators include c2go[2], grumpy[3], TypeScript[4], etc. However, the design of such tools demands a lot of time and effort, as it requires proficiency in both source and target languages [35]. Moreover, such tools are specific to the particular programming language pairs they are designed for. Since the task of generating code from natural language text is more difficult than translation due to the inherent ambiguity in natural language, almost all text-to-code generation tools are based in **artificial intelligence (AI)**.

Code generation bears a strong resemblance to natural language generation as both involve the creation of a sequence of words or tokens. Since natural language generation using deep learning has achieved great success in recent years, it is natural to exploit similar deep learning–based approaches for code generation as well. However, the code domain faces a unique set of challenges. Since the generated code is to be understood by a machine as opposed to a human, it is even more important for the generated code (compared with natural language) to adhere to a specific syntax. Moreover, since a minor change in code could alter its function, it is also critical to preserve the semantic information from the source code during translation. To generate syntactically correct code, some of the existing approaches for code generation leveraged the AST structure by learning to generate inorder traversal of AST [15], learning to generate production rules for AST based on a grammar, encoding AST paths using **recurrent neural networks (RNNs)** [2], and using AST-based attention [13, 15] in sequence models. Guo et al. [7] hypothesize that **a Data Flow Graph (DFG)**, which contains more semantic information and is less complex than AST, is a more useful structure to learn code representations. They incorporate DFG into the Transformer encoder by appropriately masking the attention matrix. Our model, *StructCoder*, consists of a Transformer encoder that incorporates both syntax and dataflow of source code by embedding root-leaf paths in the AST and using a modified self-attention framework, called *structure-aware self-attention*.

Code generation heavily relies on the decoder to generate code that is syntactically correct while simultaneously preserving the semantics present in the input. Structcoder advances the state-of-the-art by incorporating a structure-aware Transformer decoder that is designed to preserve the syntax and semantics of the generated code. None of the existing pretrained Transformer models constrains the generated code structure. In our work, we not only incorporate source AST and DFG into the encoder but also drive the decoder to learn the target syntax and dataflow by introducing novel AST- and DFG-related tasks. Particularly, we train the decoder to predict all the root-leaf paths in the target AST and to predict the DFG edges.

Similar to pretrained language models [4, 16, 24, 33], pretrained code models using Transformer [1, 5, 14, 36] have resulted in significant performance gains on code-related tasks. While some

[1]https://github.com/babel/babel
[2]https://github.com/elliotchance/c2go
[3]https://github.com/google/grumpy
[4]https://github.com/microsoft/TypeScript

pretext tasks such as **Masked Language Modeling (MLM)** and **Replaced Token Detection (RTD)** only pretrain the encoder, other pretext tasks such as **Denoising Autoencoding (DAE)** and **Back Translation (BT)** jointly pretrain both the encoder and decoder. StructCoder falls in the latter category and is pretrained using a structure-based DAE task. Moreover, since the structure-based components introduced in this work can be added to any existing Transformer model, we may initialize most of the StructCoder weights using one of the pretrained code models to avoid pretraining from scratch, which can be quite expensive. The main contributions of this work are listed below:

(1) We develop a Transformer-based encoder-decoder model called StructCoder for code generation in which both encoder and decoder are structure aware. (a) The encoder incorporates AST's root-leaf path embeddings and a structure-aware self-attention framework to model source code structure. (b) The decoder is trained to recognize target syntax and dataflow via two novel auxiliary tasks: AST paths prediction and dataflow prediction.
(2) We pretrain StructCoder using a structure-based DAE objective in which the input code as well as its AST and DFG are partially corrupted and the model is trained to generate the original input code and also perform the auxiliary tasks.
(3) Our experiments demonstrate that the proposed model achieves state-of-the-art performance on the code translation and text-to-code generation tasks in the CodeXGLUE [18] benchmark, and outperforms similarly sized baselines on the APPS code generation benchmark.

The subsequent sections of this article are organized as follows. Section 2 discusses existing methods for modeling code structure and developing pretrained Transformers for code. Section 3 provides a detailed description of our proposed methodology. In Section 4, we present experimental results, comparing our model against the baselines on code translation and text-to-code generation datasets. We also conduct an ablation study and discuss more aspects of StructCoder's performance. Section 5 concludes the article.

2 RELATED WORK

2.1 Leveraging Structure to Generate Code

To leverage code structure in deep models, many approaches have utilized ASTs. Some approaches modeled code completion as a language modeling task by ordering the code tokens using a depth-first traversal of AST. Li et al. [15] used **long short-term memory (LSTM)** appended with parent–child attention whereas Alon et al. [2] encoded each root-to-leaf path with an LSTM. Kim et al. [13] used the Transformer to encode the sequenced AST by encoding AST paths into self-attention. For text-to-code generation, Rabinovich et al. [23] proposed a modular decoder to recursively generate target AST. Brockschmidt et al. [3], Sun et al. [28], Yin and Neubig [34] construct ASTs by generating production rules based on a grammar. Jiang et al. [11] proposed an LSTM decoder equipped with AST-enhanced attention to generate a sequence of production rules by attending to previously generated rules and one future rule. To go beyond the standard preorder traversal for AST node generation, Jiang et al. [12] used a Reinforcement Learning framework for dynamically selecting the branch to expand at an intermediate AST node, and Xie et al. [32] used two separate models for preorder and breadth-first traversals that are jointly trained via mutual distillation. Unlike these methods, we keep the conventional Transformer decoder architecture intact and introduce auxiliary structure-related components on top of the decoder's final hidden representations so that StructCoder is trained to preserve target code structure while not requiring the generation of such structures (AST/DFG) during inference. Building on top of the conventional Transformer architectures not only allows us to utilize existing pretrained models for better

Table 1. A Summary of the Recent Pretrained Models for Code Generation

Model	Encoder-only pretraining	Encoder-Decoder pretraining	Encoder structure-awareness	Decoder structure-awareness
CodeBERT [5]	MLM, RTD	–	–	–
GraphCodeBERT [7]	MLM, EP, NA	–	DFG	–
Transcoder [27]	MLM	DAE, BT	–	–
PLBART [1]	–	DAE	–	–
DOBF [14]	–	DOBF	–	–
CodeT5 [30]	IT	MSP, MIP, NL-PL dual generation	Identifiers	Identifiers
MuST[35]	–	DAE, MuST	–	–
StructCoder (ours)		structure-based DAE, NL-PL dual generation	AST, DFG	AST, DFG

Abbreviations: DFG: Data Flow Graph, MLM: Masked Language Modeling, DAE: Denoising Autoencoding, RTD: Replaced Token Detection, BT: Back Translation, EP: DFG Edge Prediction, NA: Alignment prediction between code tokens and DFG nodes, DOBF: Deobfuscation, IT: Identifier Tagging, MSP: Masked Span Prediction, MIP: Masked Identifier Prediction, MuST: Multilingual Snippet Translation.

initialization but also makes the advances in the area of Transformers more easily applicable to our model.

2.2 Pretrained Transformers for Code

The recent state-of-the-art results on most natural language generation tasks are obtained by pretraining huge deep learning models on large datasets with carefully designed pretext tasks. Since code generation is very similar to text generation and there is abundant unsupervised code data available through open source code repositories, pretraining code generation models using similar pretext tasks has been successful. Most recent state-of-the-art pretrained models for code utilize the Transformer [29] architecture and are discussed below.

CodeBERT [5] performs encoder-only pretraining using Masked Language Modeling and Replaced Token Detection as pretext tasks on the CodeSearchNet dataset. Transcoder [27] is an unsupervised translation model that pretrains both encoder and decoder using Denoising Autoencoding and Back-Translation with only monolingual datasets. PLBART [1] is pretrained with DAE objective using 680M Java and Python functions. DOBF [14] attempts to understand code structure with a deobfuscation pretext task in which every occurrence of a sampled identifier is replaced by an uninformative token. Code Transformer [36] modifies the attention computations in the encoder according to AST-based distances. CodeT5 [30] pretrains a T5 model [25] with code data in 8 programming languages. In contrast to PLBART, which treats code data as plain sequences, CodeT5 includes identifier-aware objectives in the training, which helps maintain the correctness of the code. However, CodeT5 does not include any structural information of the code in training. Zhu et al. [35] improve code translation performance by introducing a fine-grained snippet-level translation task during pretraining. GraphCodeBERT [7] utilizes code structure in the form of a DFG that contains semantic information as opposed to the syntactic information in AST. However, the decoder is completely unaware of the code structure in all of the above methods. *Our model advances the domain of code generation by being the first to train a structure-aware Transformer encoder and decoder by modeling both syntax and dataflow.* A summary of the pretext tasks and code structures used by the above Transformer-based methods along with our approach is provided in Table 1.

Table 2. Notations Used in This Article

Notation	Definition
$S = (s_1, \ldots, s_{\lvert S \rvert})$	Input/source token sequence
$T = (t_1, \ldots, t_{\lvert T \rvert})$	Output/target token sequence
$\mathcal{T} = (N, N_{leaf}, r, p(.), L^{ast})$	AST (Abstract Syntax Tree) with root node r
N	Set of AST nodes
$N_{leaf} = \{l_1, \ldots, l_{\lvert N_{leaf} \rvert}\}$	Set of AST leaf nodes
$\lvert l_i \rvert$	No. of nodes on the root-l_i path in AST
$p : N \backslash \{r\} \longrightarrow N$	Parent node mapping in AST
$L^{ast} \in \{0,1\}^{(\text{\# code tokens}) \times \lvert N_{leaf} \rvert}$	Token-leaf linking matrix
$\mathcal{Y}$	Set of node types
$n.type \in \mathcal{Y}$	Type of a node n in AST
$E_{type}(.)$	Node type embedding
$E_{height}(.)$	Node height embedding
$\mathcal{G} = (V, D, L^{dfg})$	DFG (Data Flow Graph)
$V = \{v_1, v_2, \ldots, v_{\lvert V \rvert}\}$	Set of variables in DFG
$D \in \{0,1\}^{\lvert V \rvert \times \lvert V \rvert}$	DFG adjacency matrix
$L^{dfg} \in \{0,1\}^{(\text{\# code tokens}) \times \lvert V \rvert}$	Token-variable linking matrix
$E_x \in \mathbb{R}^d$	Embedding of x
$A(.,.)$	Attention score before softmax
W_q, W_k	Query and Key projection matrices
$\phi : \mathbb{Z}_{\geq 0} \longrightarrow \mathbb{R}$	Relative position embedding
$h_i \in \mathbb{R}^d$	Hidden state from the last decoder layer at i^{th} position
l_{t_i}	Leaf node containing code token t_i
$p_i \in [0,1]^{\lvert \mathcal{V} \rvert}$	Predicted probability of each token in vocabulary $\mathcal{V}$ at i^{th} position
$p^{ast}_{ik} \in [0,1]^{\lvert \mathcal{Y} \rvert}$	Predicted probability of each node type for the k^{th} AST node on the root-l_{t_i} path
$p^{dfg}_{ij} \in [0,1]$	Predicted probability of dataflow from t_j to t_i
$y_{ij} \in \{0,1\}$	Ground truth dataflow indicator from t_j to t_i
$\mathcal{L}_{lm}$	Language modeling loss
$\mathcal{L}_{app}$	AST Paths Prediction (APP) loss
$\mathcal{L}_{dfp}$	Data Flow Prediction (DFP) loss
α_1, α_2	DFP loss weight, APP loss weight

3 STRUCTCODER

StructCoder is a Transformer-based encoder-decoder model in which both encoder and decoder are structure aware. We build our model using T5 architecture and add the relevant components for modeling code structure. For code inputs, the encoder (Section 3.2) inputs the tokenized source code sequence along with its AST and DFG and employs structure-aware self-attention. The structure-aware decoder (Section 3.3) simultaneously learns to generate the target code sequence as well as to perform target AST- and DFG-related tasks. The notations used to describe our methodology in this section are summarized in Table 2.

3.1 Preliminaries

A **Code** can be a function or a program, and is represented as a sequence of tokens $S = (s_1, \ldots, s_{\lvert S \rvert})$. A code S has a corresponding **AST** represented as $\mathcal{T} = (N, N_{leaf}, r, p(.), L^{ast})$, where N is the set of nodes in the AST, $N_{leaf} = \{l_1, \ldots, l_{\lvert N_{leaf} \rvert}\} \subset N$ is the subset of leaf nodes, $r \in N$ is the root node, $p : N \backslash \{r\} \longrightarrow N$ is a mapping such that $p(n)$ denotes the parent of node n, and $L^{ast} \in \{0,1\}^{\lvert S \rvert \times \lvert N_{leaf} \rvert}$ is a linking matrix such that $L^{ast}_{ij} = 1$ if and only if token s_i is part of leaf l_j.

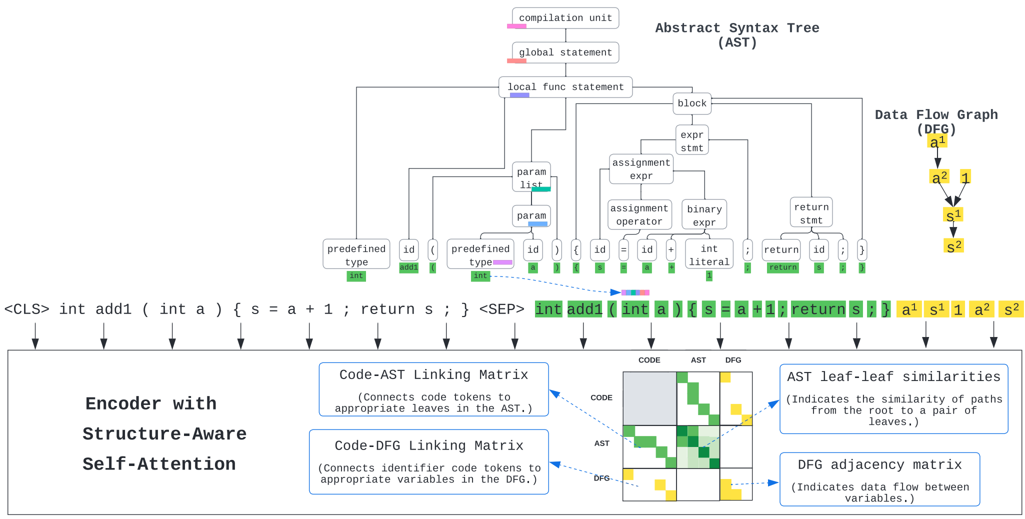


Fig. 1. Structure-aware encoder: The input sequence to the encoder consists of source code concatenated with the AST leaves and DFG variables, where the AST leaves are embedded using the root-leaf paths in the AST. The modified structure-aware self-attention mechanism of this Transformer encoder utilizes code-AST/DFG linking information, leaf-leaf similarities in the AST, and the (asymmetric) DFG adjacency matrix to compute the attention matrix.

Each node $n \in N$ has a type denoted by $n.type$. We use the tree-sitter library[5] to parse codes and generate syntax trees according to a context-free grammar for each programming language.

A code S also has a corresponding **DFG** represented as $\mathcal{G} = (V, D, L^{dfg})$, where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is the set of variables extracted from code S and $D \in \{0,1\}^{|V| \times |V|}$ is the adjacency matrix where $D_{ij} = 1$ if and only if the value of v_i is directly obtained from v_j and $L^{dfg} \in \{0,1\}^{|S| \times |V|}$ is a linking matrix such that $L^{dfg}_{ij} = 1$ if and only if variable v_j is derived from token s_i. For extracting the DFG, we utilize the implementation[6] of Ren et al. [26] where the tree-sitter-generated AST is traversed to recursively identify the variables and dataflow relations between them using a language-specific deterministic function.

The goal of code translation is to transform a code $S = (s_1, \ldots, s_{|S|})$ in a source language to code $T = (t_1, \ldots, t_{|T|})$ in a different target language such that the translated code T is semantically equivalent to the input code S. In text-to-code generation, the goal is to generate target code T from a natural language description.

3.2 Structure-Aware Encoder

Given source code S, its corresponding AST $\mathcal{T}$, and DFG $\mathcal{G}$, the input sequence to the encoder is

$$\langle CLS \rangle, s_1, .., s_{|S|}, \langle SEP \rangle, l_1, \ldots, l_{|N_{leaf}|}, v_1, \ldots, v_{|V|},$$

which consists of the code tokens, special tokens $\langle CLS \rangle$ and $\langle SEP \rangle$, AST leaves, and DFG variables. For text input, the leaves and variables are simply ignored in the input. The encoder architecture is illustrated in Figure 1 and is described in detail below.

[5] https://github.com/tree-sitter/py-tree-sitter
[6] https://github.com/microsoft/CodeXGLUE/blob/main/Code-Code/code-to-code-trans/evaluator/CodeBLEU/parser/DFG.py

3.2.1 Input Embedding. As StructCoder consists of a Transformer encoder, each token in the input sequence has to be embedded in $\mathbb{R}^d$. We embed the code tokens along with special tokens by using a lookup table and use a unique embedding for all DFG variables. The DFG information will be used by the encoder in structure-aware self-attention. We compute the embedding of a leaf l in an AST as a function of the path from the root to the leaf l in the AST.

Let $(r_1, r_2, \ldots, r_{|l|})$ be the nodes on the path from root $r = r_1$ to leaf $l = r_{|l|}$. We utilize node-type embedding $E_{type}(\cdot) \in \mathbb{R}^d$ to encode a node's syntax along with a node-height embedding $E_{height}(\cdot) \in \mathbb{R}^d$ to encode the order of nodes on this path. The leaf embedding $E(l)$ is computed as

$$E(l) = \sum_{i=1}^{|l|} E_{type}(r_i.type) \odot E_{height}(|l| - i) \quad \in \mathbb{R}^d, \tag{1}$$

where $\odot$ denotes element-wise multiplication.

3.2.2 Structure-Aware Self-Attention. Since the input contains a DFG and AST that consist of structural information, the traditional attention computation using (relative) positional embeddings that capture sequential ordering information is not sufficient. Hence, we propose structure-aware self-attention, which computes attention scores between tokens based on the structural relations between them.

Code–code: Following T5, we compute attention scores (before softmax) between code tokens by adding the query-key dot product with weights $W_q, W_k \in \mathbb{R}^{d_k \times d}$ and a lookup embedding $\phi : \mathbb{Z}_{\geq 0} \longrightarrow \mathbb{R}$ for relative position. Denoting embedding of x by E_x, we have that

$$A(s_i, s_j) = E_{s_i}^T W_q^T W_k E_{s_j} + \phi(|i - j|). \tag{2}$$

Leaf–leaf: To calculate attention scores between leaves, we introduce a similarity-based transformation to replace the relative positional embedding in Equation (2). Let $(r_1^i, \ldots, r_{|l_i|}^i)$ be the nodes on the path from root to leaf l_i. We define similarity between two leaves l_i and l_j as

$$sim(l_i, l_j) = log\left(1 + \frac{\left(\sum_{k=1}^{min(|l_i|,|l_j|)} \mathbb{1}(r_k^i = r_k^j)\right)^2}{|l_i||l_j|}\right), \tag{3}$$

which is based on the number of common nodes on the paths from root to leaves l_1 and l_2. The log transformation is used to reduce the skewness of the distribution of similarity values. The attention scores between leaves are then computed as follows.

$$A(l_i, l_j) = E_{l_i}^T W_q^T W_k E_{l_j} + (w_a \; sim(l_i, l_j) + w_b), \tag{4}$$

where $w_a, w_b \in \mathbb{R}$ are learnable parameters.

Variable–variable: Following Guo et al. [7], the attention scores between DFG nodes are computed using only the query-key dot product. They are set to $-\infty$ if corresponding edges are absent in the DFG.

$$A(v_i, v_j) = \begin{cases} E_{v_i}^T W_q^T W_k E_{v_j} & \text{if } D_{ij} = 1 \\ -\infty & else \end{cases} \tag{5}$$

Code–leaf/variable: For interaction between code tokens and AST leaves (or DFG variables), we only compute the query-key dot product and do not use any positional information. Inspired by the work of Guo et al. [7], we set the attention score to $-\infty$ for cases in which the leaf (or variable)

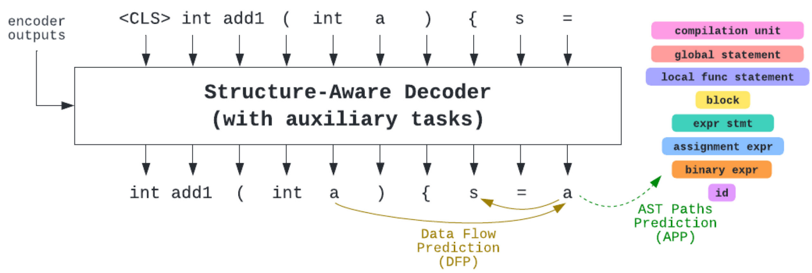


Fig. 2. Structure-aware decoder generates the next token in the target code and predicts the node types on the root–leaf path to the leaf containing this token in the target AST as well as the DFG edges incident on this token.

is not linked to the code token. We show the equations only for interactions between code tokens and leaves as those for interactions between code tokens and variables are similar.

$$A(s_i, l_j) = \begin{cases} E_{s_i}^T W_q^T W_k E_{l_j} & \text{if } L_{ij}^{ast} = 1 \\ -\infty & else \end{cases} \quad ; \quad A(l_j, s_i) = \begin{cases} E_{l_j}^T W_q^T W_k E_{s_i} & \text{if } L_{ij}^{ast} = 1 \\ -\infty & else \end{cases} \tag{6}$$

3.3 Structure-Aware Decoder

The decoder in StructCoder constitutes the original T5 decoder with additional layers at the end for AST path prediction and dataflow prediction tasks that are introduced in this section. Figure 2 illustrates the structure-aware decoder that predicts the next target code token along with the AST root-leaf path to this token and the dataflow relations between this token and all past tokens. The addition of these auxiliary tasks does not increase the number of generated tokens, which is important since the decoding is done in an autoregressive manner.

Let $h_1, h_2, \ldots, h_{|T|}$ be the hidden states generated by the Transformer decoder. Decoders of existing Transformer models, including T5, employ a linear layer with weights $W \in \mathbb{R}^{|\mathcal{V}| \times d}$ followed by softmax transformation to extract a probability distribution p_i on the token vocabulary space $\mathcal{V}$ for the i^{th} position.

$$p_i = softmax\,(Wh_i) \tag{7}$$

The sequence generation task is trained using language modeling loss as shown below for one sample.

$$\mathcal{L}_{lm} = -\frac{1}{|T|} \sum_{i=1}^{|T|} log\; p_i(t_i), \tag{8}$$

where $p_i(t_i)$ refers to the predicted probability for true target token t_i at the i^{th} position.

In addition to sequence generation, StructCoder also learns target syntax using the AST path prediction task and learns to match the target DFG using a dataflow prediction task.

3.3.1 ***AST Path Prediction (APP).*** In this task, the goal is to encourage the decoder to be aware of all root–leaf paths in the target AST. Since the type attribute of a node captures important syntactic information, we predict the type of each ancestor on each root–leaf path.

Let l_{t_i} be the leaf node containing the i^{th} target token t_i and let $(r_1^i, \ldots, r_{|l_{t_i}|}^i)$ be the nodes on the root-l_{t_i} path. To predict type of node r_k^i (which is at height $|l_{t_i}| - k$ in the tree), we use a linear layer with weights $W_{ast\,(|l_{t_i}|-k)} \in \mathbb{R}^{|\mathcal{Y}| \times d}$ followed by a softmax transformation to predict a probability distribution on the set of node types $\mathcal{Y}$:

$$p_{ik}^{ast} = softmax(W_{ast\,(|l_{t_i}|-k)} h_i). \tag{9}$$

The APP cross-entropy loss for a sample is given by

$$\mathcal{L}_{app} = -\frac{1}{|T|(\sum_i |l_{t_i}|)} \sum_{i=1}^{|T|} \sum_{k=1}^{|l_{t_i}|} log\ p_{ik}^{ast}(r_k^i.type). \tag{10}$$

3.3.2 Data Flow Prediction (DFP). In this task, the decoder learns to predict all the dataflow edges in the target code. The probability p_{ij}^{dfg} of dataflow from j^{th} to i^{th} position in target code sequence is computed using an asymmetric transformation (since dataflow is directed) as

$$p_{ij}^{dfg} = \sigma\,(h_i^T U_{dfg}^T V_{dfg} h_j + u_{dfg}^T h_i + v_{dfg}^T h_j + w_{dfg}), \tag{11}$$

where $\sigma(.)$ denotes the sigmoid function. Suppose that $\mathcal{G} = (V, D, L)$ is the true target DFG. There is a dataflow from the j^{th} to i^{th} position in target sequence if and only if "target DFG contains variables $v_{j'}$, $v_{i'}$ such that variable $v_{j'}$ is derived from t_j, variable $v_{i'}$ is derived from t_i, and value of variable $v_{i'}$ is derived from $v_{j'}$". Thus, the DFP loss for a sample can be written as

$$\mathcal{L}_{dfp} = -\frac{1}{|T|^2} \sum_{i=1}^{|T|} \sum_{j=1}^{|T|} \left\{ y_{ij}\, log\, p_{ij}^{dfg} + (1 - y_{ij})\, log\,(1 - p_{ij}^{dfg}) \right\},$$

$$\text{where} \quad y_{ij} = \mathbb{1}(\exists\, v_{i'}, v_{j'} \in V \text{ such that } D_{i'j'} = L_{ii'}^{dfg} = L_{jj'}^{dfg} = 1). \tag{12}$$

The overall loss function for training StructCoder (given below) is a combination of the language modeling objective, and the APP and DFP losses with weights α_1 and α_2, i.e., $\mathcal{L} = \mathcal{L}_{lm} + \alpha_1 \mathcal{L}_{app} + \alpha_2 \mathcal{L}_{dfp}$.

3.4 Pretraining

We pretrain StructCoder on the CodeSearchNet [9] dataset[7] containing about 2M code and comment pairs, with a structure-based DAE task along with NL-PL bimodal dual generation to generate code from text and vice versa. For the denoising task, we corrupt random spans in the code sequence by replacing them with $\langle MASK \rangle$ or a random token or deleting them. The span lengths are sampled from a Poisson distribution with a mean of 12 tokens. We corrupt 35% of the code tokens in total, similar to [1]. To improve the understanding of code structure, we also randomly drop 35% of the DFG variables and AST leaves, and 35% of the ancestors for each leaf from the input to StructCoder. The model is then trained to predict the uncorrupted code along with the AST root–leaf paths and dataflow edges. We initialize our model for pertaining with CodeT5's weights (for faster pretraining) except for the AST- and DFG-related weights, which are randomly initialized.

3.5 Implementation Details

We implement StructCoder by extending the CodeT5-base architecture containing 12 T5 blocks with hidden dimension 768, and 12 attention heads in each block. StructCoder comprises a total of 224M trainable parameters, whereas CodeT5-base contains 223M. We employ the AdamW [17] optimizer with a learning rate of 2e-4 for pretraining and 1e-5 for fine-tuning. We ran the pretraining for 175K batches with a batch size of 20 code–comment pairs. For fine-tuning, we used batch sizes of 25, 32, and 20 for CodeXGLUE translation, CONCODE, and APPS datasets, respectively. The fine-tuning was run for 50K, 300K, and 40K batches for the three datasets, respectively. The

[7] https://github.com/github/CodeSearchNet

Table 3. Results on Code Translation Tasks from CodeXGLUE Benchmark

	Java-C#			C#-Java		
	BLEU	xMatch	CodeBLEU	BLEU	xMatch	CodeBLEU
Naïve Copy	18.54	0.00	42.20	18.69	0.00	34.94
Transformer	55.84	33.00	63.74	50.47	37.90	61.59
RoBERTa (code)	77.46	56.10	83.07	71.99	57.90	80.18
CodeBERT	79.92	59.00	85.10	72.14	58.80	79.41
GraphCodeBERT	80.58	59.40	-	72.64	58.80	-
PLBART	83.02	64.60	87.92	78.35	65.00	85.27
CodeT5*	83.88	64.70	87.38	79.71	67.50	85.51
StructCoder	**84.43**	**66.90**	**88.19**	**80.43**	**68.70**	**85.98**

*Since CodeT5 is a competitive baseline and did not report CodeBLEU in their paper, we tested this model using their finetuned checkpoint and provided the results.

loss weights of auxiliary tasks α_1 and α_2 are both set to 0.1. To facilitate minibatch training with available resources, we set the maximum number of DFG variables in the input to 65, the maximum number of AST leaves to 250, and the maximum root–leaf path length to 17 (by trimming paths from the root's side). We set the maximum source length (no. of code/text tokens) to 400 for pretraining, 320 for translation, and 320 and 600 for text-to-code generation on CONCODE and APPS, respectively. We set the maximum target length to 400 for pretraining, 256 for translation, and 150 and 512 for text-to-code generation on CONCODE and APPS, respectively. We implement our model using the PyTorch [21] and Hugging Face [31] libraries. Additional implementation and experimental setup details are provided in the Appendix.

4 EXPERIMENTS

We evaluate StructCoder on the code translation and text-to-code generation tasks from the CodeXGLUE [8] [18] benchmark and on the text-to-code generation task from the APPS benchmark [8], and compare with previously published results on these tasks.[9] For CodeXGLUE tasks, we use the metrics from the CodeXGLUE leaderboard. These metrics include (i) BLEU [20] score, which measures n-gram overlap; (ii) exact match (xMatch), which checks whether the prediction is the same as ground truth; and (iii) CodeBLEU [26], which combines BLEU score with keywords-based weighted n-gram match as well as syntax and semantic matches based on the AST and DFG. APPS evaluates generated codes based on test cases in which the evaluation metrics include (i) 'test case average,' which is the average percentage of test cases passed; and (ii) 'strict accuracy,' which is the percentage of generated codes that pass all test cases.

4.1 Code Translation

The code translation dataset from CodeXGLUE consists of two tasks for translating between Java and C# functions in either direction and contains 10K training samples, 500 validation samples, and 1,000 test samples. Table 3 presents the results of StructCoder alongside the baselines on the two code translation tasks. The Naïve Copy baseline simply copies source code to target, and the Transformer model does not include any pretraining. RoBERTa (code) [18], CodeBERT, and GraphCodeBERT involve encoder-only pretraining whereas PLBART and CodeT5 incorportate

[8] https://github.com/microsoft/CodeXGLUE
[9] We did not include CODEGEN [19] and Incoder [6] in the baselines because these models are trained on much bigger datasets and/or use much larger architectures. Thus, it is unfair to compare them with our model and the other baselines used in this article.

Table 4. Results on Text-to-Code Generation Task from CodeXGLUE Benchmark

	BLEU	xMatch	CodeBLEU
GPT-2	25.37	17.35	29.69
CodeGPT	28.69	18.25	32.71
CodeGPT-adapted	32.79	20.10	35.98
PLBART	36.69	18.75	38.52
CoTexT	37.40	20.10	40.14
CodeT5	40.73	22.30	43.20
StructCoder	**41.35**	**22.40**	**44.44**

Table 5. Results on the APPS Dataset Along with Model Size in #Billion Parameters

		Test case average			Strict accuracy		
	Model size	Intro	Interview	Competition	Intro	Interview	Competition
GPT-2	0.1B	5.64	6.93	4.37	1	0.33	0
CodeT5	0.2B	9.50	6.03	2.51	1.70	0.37	0
StructCoder	0.2B	**10.22**	7.50	3.18	**2.5**	**0.70**	**0.2**
GPT-2	1.5B	7.40	**9.11**	**5.05**	1.3	**0.70**	0

The results for GPT-2 models were obtained from [8].

encoder-decoder pretraining like StructCoder. StructCoder achieves the best results on the two translation tasks, which can be attributed to the structure-aware encoder-decoder design of our model. From Table 3, we observe that the encoder-decoder pretraining of PLBART, CodeT5, and StructCoder is very beneficial to code translation. Also, the encoder-only pretrained models improve over Transformer by a huge margin. GraphCodeBERT, which utilizes a DFG, offers minor improvements over CodeBERT and we also observed in our ablation study that DFG-related components contribute less to the performance gains of StructCoder compared with AST-related components.

4.2 Text-to-Code Generation

The text-to-code generation task from CodeXGLUE uses the CONCODE [10] dataset; the goal here is to generate a Java function given a natural language description. This dataset contains 100K training samples, 2K validation samples, and 2K test samples. Table 4 presents the results of our model alongside the baselines on the text-to-code generation task. Among the baselines, GPT-2 [24] is pretrained on natural language to predict next token, CodeGPT [18] is pretrained from scratch like GPT-2 but using code data, CodeGPT-adapted [18] is pretrained from GPT-2 initialization using code data, and CoTexT [22] pretrains the T5 model further on code data using the MSP objective. The decoder-only baselines that include GPT-2-based models are outperformed by the rest, which are all encoder-decoder models. StructCoder again achieves the best performance on all metrics for the text-to-code generation task.

APPS [8] is a text-to-code generation benchmark in Python that evaluates generated codes based on test cases. The inputs here contain detailed questions and possibly some starter code as well. The dataset contains 10K problems equally divided into train and test splits. The test set contains 1K introductory-level, 3K interview-level, and 1K competition-level problems. Table 5 shows the results of StructCoder, CodeT5, and GPT-2 [8] models of two sizes. These GPT-2 models were pretrained exclusively on Python code from GitHub, which gives them an edge in this particular task. The 'strict accuracy' metric is more important than the 'test case average' as it does not give

Table 6. CodeBLEU and Its Different Components on Java-C# and C#-Java Translation for the Validation Sets by Adding the Proposed Structure-Based Components to a Smaller T5 Model

	BLEU		wBLEU		AST match		DF match		CodeBLEU	
	J-C	C-J	J-C	C-J	J-C	C-J	J-C	C-J	J-C	C-J
(i) No structure (baseline)	60.00	54.46	61.85	55.64	78.10	74.79	72.41	64.92	68.09	62.45
(ii) DFG (enc)	59.20	54.38	61.66	55.60	78.20	75.56	73.20	66.89	68.07	63.11
(iii) DFG (dec)	61.25	54.45	62.78	55.58	78.72	76.11	73.08	66.39	68.96	63.13
(iv) AST (enc)	60.78	54.70	62.21	55.87	79.15	76.67	73.69	67.02	68.96	63.57
(v) AST (dec)	<u>61.76</u>	<u>56.40</u>	<u>63.16</u>	<u>57.42</u>	78.72	75.65	73.91	64.81	69.39	63.57
(vi) DFG (enc, dec), AST (enc, dec)	61.51	55.43	62.89	56.43	<u>79.71</u>	<u>77.36</u>	<u>74.12</u>	<u>67.80</u>	<u>69.56</u>	<u>64.26</u>
(vii) DFG (enc, dec), AST (enc, dec), & structure-based DAE pretraining	**80.58**	**77.09**	**81.17**	**77.62**	**89.03**	**89.28**	**86.87**	**87.24**	**84.50**	**82.67**

'enc' and 'dec' indicate whether the proposed structure-based components/tasks were included in the encoder and decoder, respectively. AST stands for Abstract Syntax tree, DF for Data Flow, and 'wBLEU' for weighted BLEU.

partial credit to a generated code that does not pass all test cases. StructCoder achieves the best strict accuracy on all subsets, notably outperforming the bigger GPT-2 model, which is about 7 times the size of StructCoder.

4.3 Model Analysis

4.3.1 Ablation Study. To emphasize the importance of the novel structure–based components introduced in this work, we conducted an ablation study on the two code translation tasks from CodeXGLUE. For this experiment, we used a smaller T5 architecture with hidden dimension 256, 5 encoder and decoder layers, and 8 heads in each multi-head attention layer. The ablated models we tested here include the smaller T5 model (i) without any of the proposed structure-based components (no structure (baseline)); (ii) enabling DFG in the encoder (DFG (enc)); (iii) enabling Data Flow Prediction task in the decoder (DFG (dec)); (iv) enabling AST in encoder (AST (enc)); (v) enabling AST Path Prediction task in the decoder (AST (dec)); (vi) enabling all proposed structure-based components/tasks; and (vii) adding structure-based DAE pretraining to (vi). We report the CodeBLEU metric along with its different components for each of these models in Table 6. Among the different components of the CodeBLEU metric, weighted BLEU gives more weight to programming language keywords, AST match computes the percentage of subtrees in the ground truth target AST that occur in the generated code, and DFG match computes the percentage of DFG edges in the ground truth that occur in the generated code.

Enabling each of the four [(ii)–(v)] structure-based components individually results in an increase in AST match and dataflow match metrics over the baseline [(i)] in most of the cases. The DFG components in the model [(ii),(iii)], however, do not seem to always increase BLEU and weighted BLEU scores. Among the four components [(ii)–(v)], enabling the AST path prediction task [(v)] yields the best BLEU and weighted BLEU, and modeling AST in the input [(iv)] yields the best AST match. Enabling all the components [(vi)] gives the best results on AST match, dataflow match, and overall CodeBLEU scores. We also observed that structure-based DAE pretraining [(vii)] led to significant performance gains on both tasks.

4.3.2 Auxiliary Tasks. We measure the performance of StructCoder on the auxiliary tasks of APP (AST Path Prediction) and DFP (Data Flow Prediction) as follows. When predicting the next target token, we use the ground truth for target sequence until the previous step as input to the decoder. The decoder then predicts the next token as well as the DFG edges incident on this token and the types of nodes on the path from the root to the leaf node containing this token in the AST. On Java-C# translation, StructCoder achieves 94% accuracy on the APP task and 94.7% average precision on the DFP task, in which positive class prevalence is just 0.8%. On C#-Java translation, StructCoder achieves 96.3% accuracy on the APP task and 82.9% average precision on the DFP task, in which positive class prevalence is just 0.5%. For both translation tasks, there are 298 classes for node type in the APP task.

4.3.3 Case Study. Figure 3 shows an example from the Java-C# translation task with predictions from StructCoder and the best baseline CodeT5. We observe that our structure-aware encoder-decoder architecture is able to generate better target code than CodeT5. Referring to Figure 3, CodeT5 generates both the 'for' loops with variable 'i', leaving variable 'c' undefined. It also misses the first 'if' statement and creates a syntax error from unbalanced braces. CodeT5 also translates the type of argument 'remap' as an integer instead of an integer array. On the other hand, StructCoder generates the 'for' loops by defining variable 'c' and the model predicts (with a probability greater than the 97^{th} percentile) most of the DFG edges incident on the variable 'c' inside these 'for' loops as well as in the first 'if' statement. The only error in StructCoder's output is the treatment of '@in.cells' as an array of 'Cell' objects instead of a Dictionary with Values of type 'Cell'. Such errors motivate the design of better models that align the variables and functions between source and target for code translation. Also, for token '[]' in args, StructCoder correctly predicts the parent node type 'array rank specifier'. More examples are included in the Appendix.

4.3.4 Inference Time. To analyze the impact of adding the proposed structure-based components on the overall computational complexity experimentally, we measured the inference times on CodeXGLUE translation tasks by including/excluding the different proposed structure-based components. We report the results by running inference on a GPU (NVIDIA Tesla P40 with 12,288 MiB memory) for 200 samples from the test set using the maximum batch size that can fit on the GPU with a beam size of 10. The batch sizes used are 6 when the AST is included in the encoder, 8 when the DFG but not the AST is included in the encoder, 10 when only the code tokens are fed to the encoder. We run decoding till the maximum target length is reached so that the model's decoded sequence lengths do not impact the inference times. We did not include preprocessing (tokenization, AST, and DFG construction) time while measuring the inference time because preprocessing took negligible time compared with the forward pass.[10]

Figure 4 shows the average inference time per sample and average input length per batch for model versions including and excluding AST- and DFG-related components in the encoder. Since the decoder's structure-based components are inactive during inference, they do not impact inference time; hence, they are not considered here. Note that excluding both the AST and DFG is equivalent to CodeT5. Adding both the AST and DFG to the encoder increased the inference time by 28% to 29% compared with using no structures in the encoder, whereas the input sequence length increased by 75% to 83%. (The increase in inference time being much less than expected may be due to efficient matrix manipulations on GPUs and implementation-specific details of PyTorch/Hugging Face, which are out of the scope of this work.) In our implementation, we compute the full squared attention matrix with diagonal size being equal to the total input length (number of code tokens + number of DFG variables + number of AST leaves), and then mask the attention scores that we want to be zero. However, the attention between code tokens and DFG variables/AST leaves, and among the DFG variables is sparse, which motivates more efficient implementations of our method.

4.4 Performance on Code Summarization

While the primary focus of this work is on code generation, we have also tested the performance of StructCoder on three languages in the CodeXGLUE summarization benchmark, which is a

[10]For all 200 samples combined, tokenization took 0.27 s (0.30 s), and AST and DFG construction took 0.62 s (0.88 s) for codes in Java (C#).

Source (Java)

```java
public boolean eat(Row in, int remap[]) {
    int sum = 0;
    for (Iterator<Cell> i =
in.cells.values().iterator(); i.hasNext();) {
        Cell c = i.next();
        sum += c.cnt;
        if (c.ref >= 0) {
            if (remap[c.ref] == 0) {
                c.ref = -1;
            }
        }
    }
    int frame = sum / 10;
    boolean live = false;
    for (Iterator<Cell> i =
in.cells.values().iterator(); i.hasNext();) {
        Cell c = i.next();
        if (c.cnt < frame && c.cmd >= 0) {
            c.cnt = 0;
            c.cmd = -1;
        }
        if (c.cmd >= 0 || c.ref >= 0) {
            live |= true;
        }
    }
    return !live;
}
```

Gold (C#)

```csharp
public bool Eat(Row @in, int[] remap){
    int sum = 0;
    foreach (Cell c in @in.cells.Values){
        sum += c.cnt;
        if (c.@ref >= 0){
            if (remap[c.@ref] == 0){
                c.@ref = -1;
            }
        }
    }
    int frame = sum / 10;
    bool live = false;
    foreach (Cell c in @in.cells.Values){
        if (c.cnt < frame && c.cmd >= 0){
            c.cnt = 0;c.cmd = -1;
        }
        if (c.cmd >= 0 || c.@ref >= 0){
            live |= true;
        }
    }
    return !live;
}
```

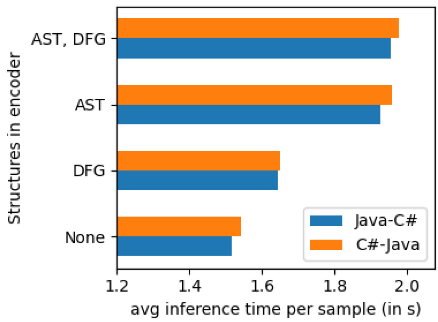
CodeT5

```csharp
public override bool Eat(Row @in, int remap){
    int sum = 0;
    for (IEnumerator<Cell> i =
@in.cells.Values.GetEnumerator(); i.MoveNext();){
        sum += c.cnt;
        if (remap[c.@ref] == 0){
            c.@ref = -1;
        }
    }
    }
    int frame = sum / 10;
    bool live = false;
    for (IEnumerator<Cell> i =
@in.cells.GetEnumerator(); i.MoveNext();){
        if (c.cnt < frame && c.cmd >= 0){
            c.cnt = 0; c.cmd = -1;
        }
        if (c.cmd >= 0 || c.@ref >= 0){
            live |= true;
        }
    }
    return !live;
}
```

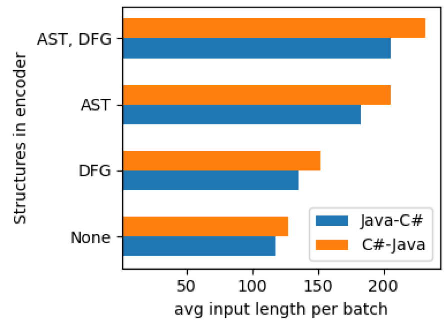
StructCoder

```csharp
public bool Eof(Row @in, int[] remap){
    int sum = 0;
    for (int i = 0; i < @in.cells.Count; i++){
        Cell c = @in.cells[i];
        sum += c.cnt;
        if (c.@ref >= 0){
            if (remap[c.@ref] == 0){
                c.@ref = -1;
            }
        }
    }
    int frame = sum / 10;
    bool live = false;
    for (int i = 0; i < @in.cells.Count; i++){
        Cell c = @in.cells[i];
        if (c.cnt < frame && c.cmd >= 0){
            c.cnt = 0;c.cmd = -1;
        }
        if (c.cmd >= 0 || c.@ref >= 0){
            live |= true;
        }
    }
    return !live;
}
```

Fig. 3. Case study: An example from the Java-C# translation task comparing the outputs from StructCoder and CodeT5. StructCoder only makes one error by assuming that 'cells' is an array of 'Cell' objects instead of a dictionary with values of type 'Cell'. CodeT5, however, misses the first 'if' statement, produces an unbalanced '}', and does not define variable 'c'. The blue arrows in the StructCoder output show the correctly predicted (probability > 97^{th} percentile) dataflow edges incident on variable 'c'.)



(a)



(b)

Fig. 4. (a) Inference time (in seconds) per sample averaged over 200 samples and (b) average input length per batch for the 200 samples in the CodeXGLUE translation tasks for model versions including/excluding AST/DFG-related components in the encoder. Since the decoder's structure-based components are not active during inference, we did not consider them in this plot.

Table 7. Results on CodeXGLUE Summarization

	Go	Java	PHP
CodeT5	19.56	20.31	**26.03**
StructCoder	**24.18**	**20.39**	25.16

code-to-text generation task. The results are shown in Table 7. StructCoder outperforms CodeT5 by a substantial margin in the case of Go but not in the case of the other languages.

5 CONCLUDING DISCUSSION

This work proposes a structure-aware Transformer encoder-decoder model called StructCoder for code generation. Our encoder modifies traditional input embeddings and employs a structure-aware self-attention mechanism to model AST and DFG relations in source code. The decoder is trained to recognize target syntax and dataflow using two novel auxiliary tasks to predict the node types on all root–leaf AST paths and dataflow edges of target code. We also pretrained our model using a structure-based DAE task to improve its performance. Experiments on code translation and text-to-code generation tasks demonstrate the performance gains of StructCoder over state-of-the-art baselines. We believe that this work will encourage future research in this field to give careful consideration to code structure while building models for code generation.

While automated code generation holds the potential to benefit software development and migration, it comes with inherent risks. The models cannot consider constraints such as security, efficiency, and modularization when generating code, which makes their deployment and maintenance challenging. Also, the performance improvements in code generation models largely rely on the scaling up of both the model and the training, which requires significant computational resources. Thus, future research in this area can look into designing more efficient models and models that generate code conforming to certain preset standards.

APPENDIX

A MORE IMPLEMENTATION DETAILS

We use the CodeT5 tokenizer with a vocabulary size of 32,100. As we build upon CodeT5 architecture, both the encoder and decoder of StructCoder contain 12 T5 blocks with hidden dimension

768, and 12 attention heads in each block. During implementation, we only used the first 16 bits of the last hidden representation from the decoder to predict DFG links and the next 128 bits for AST path prediction. This is done because the model learns the DFP task more easily than the APP task and using few bits for these auxiliary tasks prevents overfitting on these tasks.

A.1 Fine-tuning

For code translation, we ran fine-tuning with a batch size of 25 for 50K steps. For text-to-code generation using the CONCODE dataset, we ran fine-tuning with a batch size of 32 for 300K steps. To fine-tune on the APPS dataset, we used a batch size of 20 for 40K steps. For new AST node types seen during fine-tuning, we initialized the weights corresponding to these new node types randomly. We used beam search with a beam size of 10 for decoding in all fine-tuning tasks except for the APPS dataset, in which the beam size was set to 5. We ran validation every 500 steps for CodeGLUE translation and every 3,000 steps for CONCODE, and chose the checkpoint with the best CodeBLEU+xMatch score on the validation set for testing. For APPS, which has no validation set, the checkpoint at the end of the training was used for inference. Since CodeT5 does not have published results on the APPS dataset, we fine-tuned it using the same hyperparameters used by our model.

For the ablation study, the learning rate was set to 2e-4 when training from scratch and 1e-5 for fine-tuning, and the beam size was set to 5. For the auxiliary tasks of DFP and APP, we use the first 8 and next 32 bits of the last hidden state representation, respectively, for the ablation study.

A.2 Sequence Lengths

The main article lists the maximum lengths used for the source and target. We used the same sequence lengths as StructCoder for fine-tuning CodeT5 on APPS. The results of CodeT5 on CodeXGLUE tasks were borrowed from Wang et al. [30], in which the maximum source and target lengths were set to 512 and 256, respectively. On the code translation tasks, GraphCodeBERT [7] sets the maximum source and target lengths to 256 and the maximum number of DFG variables to 64.

A.3 Other Details

All the hyperparameters discussed above were set either based on CodeT5's implementation or, in rare cases, by observing the progression in validation performance for a few steps or by choosing the ones with best validation performance after a few trials. The code for generating ASTs and DFGs is built using tree-sitter [11] and is also adapted from https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT. The random generators were seeded in the 'set_seed()' function for each experiment. We ran our experiments on an Ubuntu 18.04 server with 4 RTX 8000 GPUs with 48 GB memory on each GPU. We used all 4 GPUs for pretraining and 2 GPUs for the fine-tuning experiments.

A.4 Model Size

Table A1 shows the number of parameters in different pretrained models for code. Note that StructCoder is built by adding additional components to CodeT5 for modeling the AST and DFG in input and output, with the majority of additional parameters coming from the encoder's AST leaves embedding module (381K) and the classification layer of the APP (AST Path Prediction) task (743K).

[11] https://github.com/tree-sitter/py-tree-sitter

Table A1. Number of Parameters in Various Pretrained Models

Pretrained model	# parameters
CodeBERT	125M
GraphCodeBERT	125M
CodeGPT-small-java	126M
PLBART	139M
CodeT5	223M
CoTexT	223M
StructCoder	224M

B EXAMPLES

In this section, we illustrate a few examples of text-to-code generation along with the predicted DFG links and AST paths (see Figures B1–B3). The DFG predictions are visualized as a matrix where the ij^{th} cell denotes the probability of dataflow from the j^{th} to i^{th} token. To visualize predicted AST paths, for each predicted token, we indicate the predicted node types on the path starting from the root (top) to the leaf (bottom) containing this token, vertically using colored discs.

TEXT:

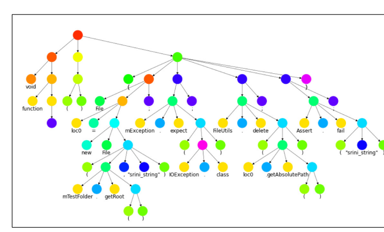
```
tests the fileutils #delete string method to throw an exception when trying to delete anon-existent file . concode_field_sep
ExpectedException mException concode_elem_sep TemporaryFolder …
```

ACTUAL CODE:

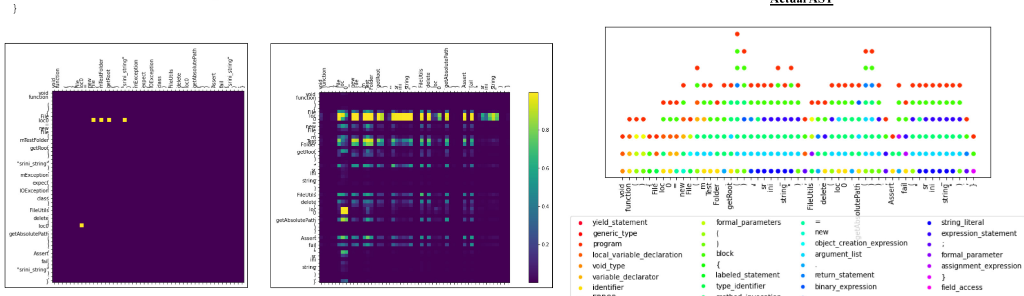
```
void function ( ) {
    File loc0 = new File ( mTestFolder . getRoot ( ) , "srini_string" ) ;
    mException . expect ( IOException . class ) ;
    FileUtils . delete ( loc0 . getAbsolutePath ( ) ) ;
    Assert . fail ( "srini_string" ) ;
}
```

GENERATED CODE:

```
void function ( ) {
    File loc0 = new File ( mTestFolder . getRoot ( ) , "srini_string" ) ;
    FileUtils . delete ( loc0 . getAbsolutePath ( ) ) ;
    Assert . fail ( "srini_string" ) ;
}
```



Actual AST



Actual DFG links

Predicted DFG links

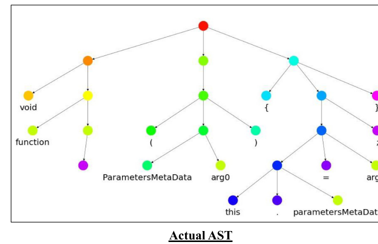
Predicted node types on root-leaf paths

Fig. B1. An example from the CONCODE dataset with BLEU=78.85.

TEXT INPUT:
```
sets the value of the parametersmetadata property . concode_field_sep MetaData.Template template concode_elem_sep
MetaData.WorkflowImpl workflowImpl concode_elem_sep String workflowImplId …
```

ACTUAL CODE:
```
void function ( ParametersMetaData arg0 ) {
     this . parametersMetaData = arg0 ;
}
```

GENERATED CODE:
```
void function ( ParametersMetaData arg0 ) {
     this . parametersMetaData = arg0 ;
}
```



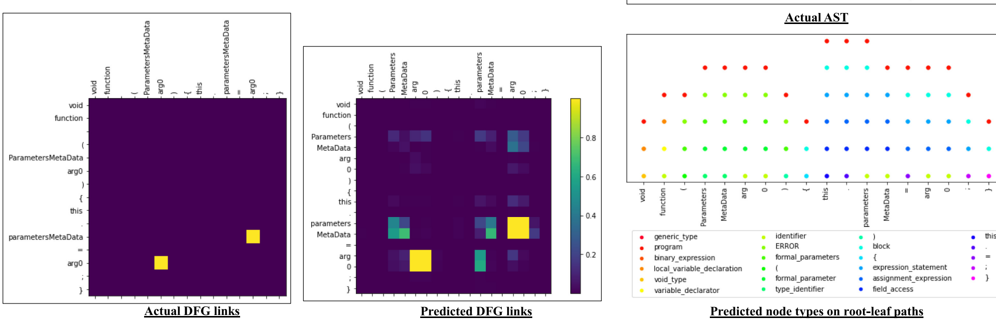
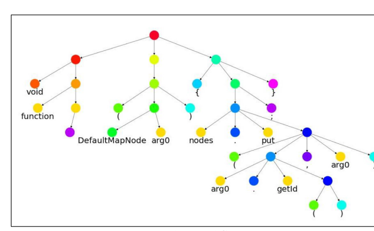


Fig. B2. An example from the CONCODE dataset with BLEU = 100.

SOURCE:
```
adds a node to the central node hashtable . concode_field_sep BoundingBox boundingBox concode_elem_sep
Hashtable<Long,MapNode> nodes concode_elem_sep Hashtable<Long,MapWay> …
```

ACTUAL CODE:
```
void function ( DefaultMapNode arg0 ) {
     nodes . put ( arg0 . getId ( ) , arg0 ) ;
}
```

GENERATED CODE:
```
void function ( MapNode arg0 ) {
     nodes . put ( arg0 . getId ( ) , arg0 ) ;
}
```



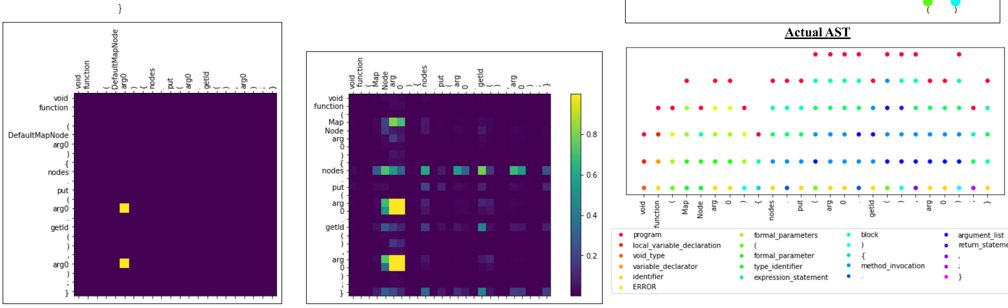


Fig. B3. An example from the CONCODE dataset with BLEU = 87.25.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of NAACL-HLT 2021*. 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International Conference on Machine Learning*. PMLR, 245–256.

[3] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative code modeling with graphs. In *7th International Conference on Learning Representations, ICLR 2019*. https://openreview.net/forum?id=Bke4KsA5FX

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[5] 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*. 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[6] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[7] 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 B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021*. https://openreview.net/forum?id=jLoC4ez43PZ

[8] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021*. https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

[9] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[10] 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*. 1643–1652. https://doi.org/10.18653/v1/d18-1192

[11] Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. 2021. An AST structure enhanced decoder for code generation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 30 (2021), 468–476.

[12] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su. 2021. Exploring dynamic selection of branch expansion orders for code generation. *arXiv preprint arXiv:2106.00261* (2021).

[13] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[14] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. In *NeurIPS 2021*. 14967–14979. https://proceedings.neurips.cc/paper/2021/hash/7d6548bdc0082aacc950ed35e91fcccb-Abstract.html

[15] 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, IJCAI 2018*. 4159–4165. https://doi.org/10.24963/ijcai.2018/578

[16] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[17] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021*. https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[19] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[20] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. 311–318.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An

imperative style, high-performance deep learning library. In *NeurIPS 2019*. 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[22] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645* (2021).

[23] 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, ACL 2017*. 1139–1149. https://doi.org/10.18653/v1/P17-1105

[24] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[25] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[26] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[27] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *NeurIPS 2020*. https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[28] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[30] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[31] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, RÃľmi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[32] Binbin Xie, Jinsong Su, Yubin Ge, Xiang Li, Jianwei Cui, Junfeng Yao, and Bin Wang. 2021. Improving tree-structured decoder training for code generation via mutual learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 14121–14128.

[33] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R. Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. *Advances in Neural Information Processing Systems* 32 (2019).

[34] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017*. 440–450. https://doi.org/10.18653/v1/P17-1041

[35] Ming Zhu, Karthik Suresh, and Chandan K. Reddy. 2022. Multilingual code snippets training for program translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11783–11790.

[36] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *9th International Conference on Learning Representations, ICLR 2021*. https://openreview.net/forum?id=Xh5eMZVONGF

Received 29 May 2023; revised 20 November 2023; accepted 29 November 2023