AST-T5: STRUCTURE-AWARE PRETRAINING FOR CODE GENERATION AND UNDERSTANDING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have made significant advancements in code-related tasks, yet many LLMs treat code as simple sequences, neglecting its structured nature. We introduce AST-T5, a novel pretraining paradigm that leverages the Abstract Syntax Tree (AST) for enhanced code generation, transpilation, and understanding. Using dynamic programming, our AST-Aware Segmentation retains code structure, while our AST-Aware Span Corruption objective equips the model to reconstruct various code structures. Unlike other models, AST-T5 avoids intricate program analyses or architectural changes, so it integrates seamlessly with any encoder-decoder Transformer. Evaluations show that AST-T5 consistently outperforms similar-sized LMs across various code-related tasks. Structure-awareness makes AST-T5 particularly powerful in code-to-code tasks, surpassing CodeT5 by 2 points in exact match score for the Bugs2Fix task and by 3 points in exact match score for Java-C# Transpilation in CodeXGLUE. Our code and model are publicly available at *https://anonymized*.

1 INTRODUCTION

We have witnessed the transformative impact of large language models (LLMs) on various aspects of artificial intelligence in recent years (Brown et al., 2020; Ouyang et al., 2022; Touvron et al., 2023), especially in code generation and understanding (Feng et al., 2020; Wang et al., 2021; Rozière et al., 2023). By pretraining on massive code corpora such as the GitHub corpus, LLMs learns rich representations, thereby becoming powerful tools for various downstream applications, including text-to-code generation (Chen et al., 2021a; Austin et al., 2021; Iyer et al., 2018), code-to-code transpilation (Lu et al., 2021; Lachaux et al., 2020; Tufano et al., 2019), and code understanding (mapping code to classification labels) (Zhou et al., 2019; Svajlenko et al., 2014).

Despite these impressive advances, most existing models interpret code as mere sequences of subword tokens, overlooking its intrinsic structured nature. Prior research has shown that leveraging the Abstract Syntax Tree (AST) of code can significantly improve performance on code-related tasks (Guo et al., 2021; Tipirneni et al., 2023). Some studies also use code obfuscation during pretraining to teach models about abstract code structures (Roziere et al., 2021; Wang et al., 2021). However, these models often rely on computationally expensive processes like Control-Flow Analysis (CFA), obfuscation, or even actual code execution. Such dependency limits their scalability and imposes stringent conditions like code executability. Consequently, these methods may struggle with real-world code, especially in intricate languages like C/C++, where comprehensive analysis remains elusive.

In this study, we propose AST-T5, a pretraining paradigm that leverages the Abstract Syntax Tree (AST) structure of code. The key contribution in AST-T5 is a simple yet effective way to exploit code semantics, without the need to run expensive program analysis or execution. Using a lightweight, multi-language parser called Tree-sitter[1], our approach has broad applicability across all syntactically well-defined programming languages. After we parse code into ASTs, we use a dynamic programming-based segmentation algorithm for AST-aware code segmentation to maintain the structural integrity of the input code. Using our novel AST-Aware Span Corruption technique, the model is pretrained to reconstruct various code structures, ranging from individual tokens to entire function bodies. Together, our approach offers three key advantages: (1) enriched bidirectional

[1] `https://tree-sitter.github.io/tree-sitter/`

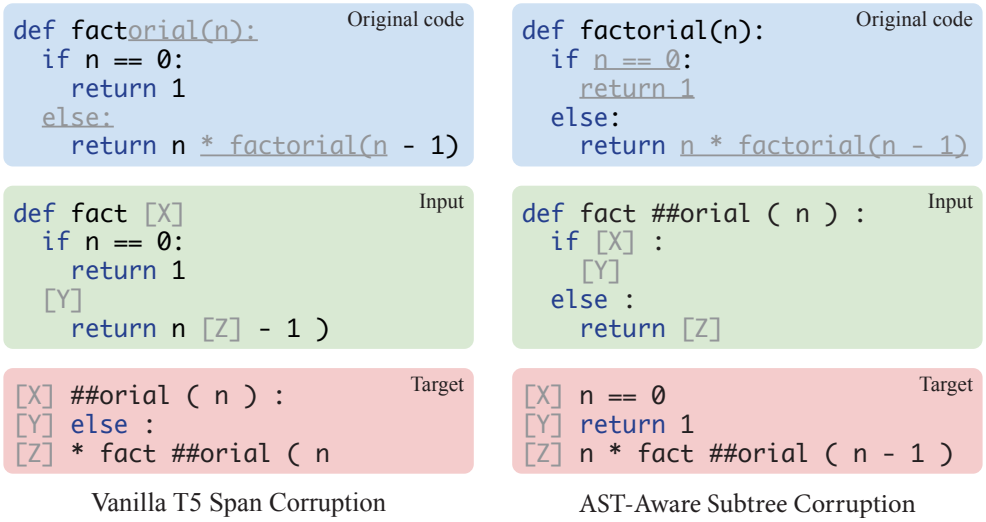

Figure 1: Comparison of AST-Aware Subtree Corruption and Vanilla T5 using a Python factorial function. Both methods replace masked spans with sentinel tokens (special tokens added to the vocabulary, shown as `[X]`, `[Y]`, and `[Z]` in the figure), with output sequences containing the original masked tokens. Inputs and targets are shown in byte-pair encoding (BPE); for instance, "factorial" is encoded into "fact" and "##orial". Unlike Vanilla T5, which masks random spans without considering code structure, our approach specifically targets spans aligned with AST subtrees, like expressions and statements.

encoding for improved code understanding, (2) the ability to coherently generate code structures, and (3) a unified, structure-aware pretraining framework that boosts performance across a variety of code-related tasks, particularly in code transpilation.

In addition, other than our specialized AST-aware masking approach, AST-T5 introduces no architecture changes or additional heads, and our pretraining objective remains the same as Vanilla T5. This compatibility enables seamless integration of our model as a drop-in replacement for any T5 variant.

In our experiments, AST-T5 consistently outperforms baselines in code generation, transpilation, and understanding tasks. Through controlled experiments, we empirically demonstrate that these advancements are attributed to our AST-aware pretraining techniques. Notably, AST-T5 not only outperforms similar-sized models like CodeT5 and CodeT5+ across various benchmarks but also remains competitive with, or occasionally even exceeds, the performance of much larger models using the HumanEval dataset. Furthermore, the inherent AST-awareness of AST-T5 offers unique advantages in structure-sensitive tasks, such as code-to-code transpilation and Clone Detection, highlighting its effectiveness at capturing the structural nuances of code.

2 Related Work

Language Models for Code. Language models (LMs) extended their use from NLP to code understanding and generation. Encoder-only models generally excel in code understanding when fine-tuned with classifiers (Feng et al., 2020), while decoder-only models are optimized for code generation through their autoregressive nature (Chen et al., 2021a; Fried et al., 2023; Nijkamp et al., 2023). However, these models can falter outside their primary domains of expertise or require increased resources for comparable outcomes. Our work focuses on encoder-decoder models, aiming to efficiently balance performance in both understanding and generation tasks without excessive computational demands.

Efforts Toward Unified Models. Extending NLP models like BART (Lewis et al., 2019) and T5 (Raffel et al., 2020), several studies have developed encoder-decoder architectures, such as

PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021), to perform well in diverse code-related tasks. Although these models show broader utility, they struggle with generating coherent, executable code in complex scenarios like HumanEval (Chen et al., 2021a). CodeT5+ (Wang et al., 2023) seeks to address this limitation through an intricate multi-task pretraining strategy across five objectives. In contrast, our proposed model, AST-T5, uses a novel AST-Aware pretraining paradigm to become a unified model capable of generating fluent code and maintaining superior performance in code understanding tasks. Moreover, AST-T5 is more streamlined, because it only uses a single pretraining objective.

Leveraging Code Structure in Pretraining. Code differs from natural language in two key aspects: its executability and strict structural syntax. Previous research leveraged execution traces for improving model performance (Chen et al., 2018; 2021b; Shojaee et al., 2023), but this approach faces scalability challenges when applied to large, web-crawled code datasets used in pretraining. Regarding code's structured nature, various studies have integrated syntactic elements into neural network models. Li et al. (2018), Kim et al. (2021) and Zügner et al. (2021) add AST-Aware attention mechanisms in their models, while Alon et al. (2020) and Rabinovich et al. (2017) focus on modeling AST node expansion operations rather than traditional code tokens. In parallel, Guo et al. (2021) and Allamanis et al. (2017) explore DFG-Aware attention mechanisms and Graph Neural Networks (GNNs), to interpret code based on its Data Flow Graph (DFG). StructCoder (Tipirneni et al., 2023) enriches the code input by appending AST and DFG as additional features. These methods, however, necessitate parsing or static analysis for downstream tasks, which is less feasible for incomplete or incorrect code scenarios like bug fixing.

Our work, AST-T5, aligns with methods that utilize code structure only in pretraining, like DOBF (Roziere et al., 2021) and CodeT5 (Wang et al., 2021), which obfuscate inputs to force the model to grasp abstract structures. Our approach uniquely diverges by using AST-driven segmentation and masking in T5 span corruption during pretraining. This novel approach offers a more refined pretraining signal compared to structure-agnostic T5, equipping our model to proficiently encode and generate semantically coherent code structures.

3 METHOD

In this section, we present AST-T5, a novel pretraining framework for code-based language models that harnesses the power of Abstract Syntax Trees (ASTs). First, AST-T5 parses code into ASTs to enable a deeper understanding of code structure. Leveraging this structure, we introduce AST-Aware Segmentation, an algorithm designed to address Transformer token limits while retaining the semantic coherence of the code. Second, we introduce AST-Aware Span Corruption, a masking technique that pretrains AST-T5 to reconstruct code structures ranging from individual tokens to entire function bodies, enhancing both its flexibility and structure-awareness.

3.1 PARSING CODE INTO ASTS

Unlike traditional language models on code that handle code as simple sequences of subword tokens, AST-T5 leverages the Abstract Syntax Tree (AST) of code to gain semantic insights. For parsing purposes, we assume the provided code is syntactically valid—a reasonable assumption for tasks like code transpilation and understanding. Instead of the often computationally-intensive or infeasible methods of Control-Flow Analysis (CFA) or code execution (Guo et al., 2021; Tipirneni et al., 2023), our method only demands the code to be parsable. We use Tree-sitter, a multi-language parser, to construct the ASTs, where each subtree represents a consecutive span of subword tokens, and every leaf node represents an individual token.

3.2 OUR AST-AWARE SEGMENTATION

In this subsection, we describe our AST-Aware Segmentation method, which splits lengthy code files into chunks in a structure-perserving manner.

Segmentation in language model pretraining is a critical yet often overlooked aspect. Transformer LMs impose token limits on input sequences, making segmentation essential for fitting these

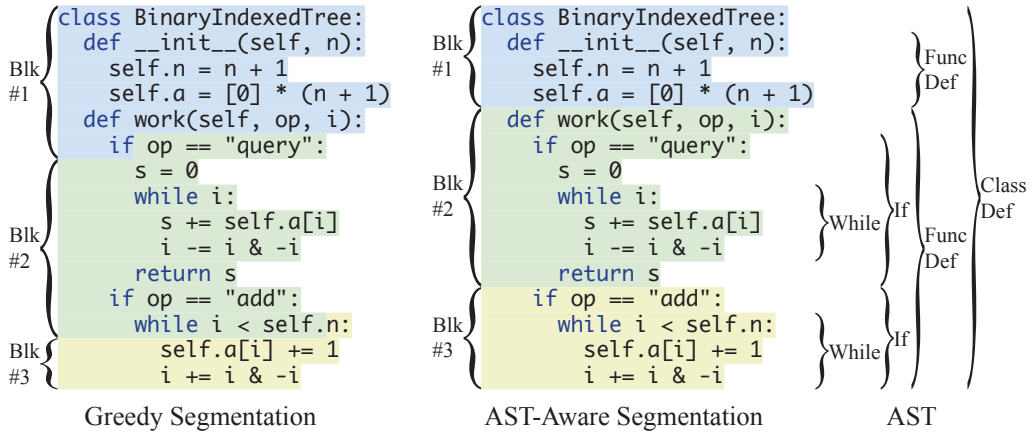

Figure 2: Comparison between Greedy Segmentation and AST-Aware Segmentation: For a 112-token code example with `max_len` set at 48, Greedy Segmentation places the first 48 tokens in Block 1, the next 48 tokens in Block 2, and the remaining in Block 3, disrupting the structural integrity of the code. In contrast, AST-Aware Segmentation uses a dynamic programming algorithm to smartly partition the code, aligning with boundaries of member functions or major function branches, thereby preserving the code's structure. The accompanying AST, with some levels pruned for clarity, corroborates that these segmentations indeed coincide with key subtree demarcations.

inputs within the `max_len` constraint. A naive approach is Greedy Segmentation, where each chunk, except the last, contains exactly `max_len` tokens Figure 2 (Left). This strategy has been widely adopted in previous works, such as CodeT5 (Wang et al., 2021).

Research in NLP by Liu et al. (2019) underscores that segmentation respecting sentence and document boundaries outperforms the greedy strategy. Given programming language's inherently structured nature, which is arguably more complex than natural language, a more sophisticated segmentation approach is even more important. However, this area remains largely unexplored.

AST-Aware Segmentation is our novel approach designed to preserve the AST structure of code during segmentation. Unlike Greedy Segmentation, which can indiscriminately fragment AST structures, our method strategically minimizes such disruptions. As illustrated in the example in Figure 2, Greedy Segmentation leads to nine instances of AST breaks—between Block 1 and Block 2, it breaks `If`, `FuncDef`, and `ClassDef`; between Block 2 and Block 3, it breaks `Attr`, `BinaryExpr`, `While`, `If`, `FuncDef`, and `ClassDef`. In contrast, our AST-Aware approach results in only three breaks: between Block 1 and Block 2, it breaks `ClassDef`, and between Block 2 and Block 3, it breaks `FuncDef` and `ClassDef`.

To identify optimal partition boundaries, we use a dynamic programming (DP) algorithm:

1. We construct an array `cost`, where `cost[i]` denotes the number of AST-structure breaks that would occur if partitioning happened right after token i. This array is populated by traversing the AST and incrementing `cost[l..r - 1]` by 1 for each span $[l, r]$ associated with an AST subtree.
2. We define a 2-D array `dp`, where `dp[k, i]` represents the the minimum total number of AST-structure breaks when k partitions are made for the first i tokens, ending the last partition right after the i-th token. The state transition equation is:

$$\texttt{dp}[k, i] = \texttt{cost}[i] + \min_{i-\texttt{max\_len} \le j < i} \texttt{dp}[k-1, j] \tag{1}$$

3. While the naive DP algorithm has a quadratic time complexity $O(n^2)$ relative to the code file length n, it can be optimized to $O(n^2/\texttt{max\_len})$ by employing a monotonic queue for sliding-window minimum calculations. This allows for efficient computation across most code files. The pseudocode of the optimized dynamic programming algorithm is shown in Algorithm 1. See Appendix A.2 for details about complexity calculations.

4. The algorithm outputs the partition associated with `dp[k_min, n]`, where `k_min` $= \arg\min_k(dp[k, n])$, as the most optimal partition.

Algorithm 1 Dynamic Programming in AST-Aware Segmentation

```
# n: the length of the code file (number of tokens)
# m: the max number of segments; approximately n / max_len
for k in range(1, m + 1):
    q = Queue()  # double ended queue
    for i in range(1, n + 1):
        while q.nonempty() and q.left() < i - max_len:
            q.pop_left()  # pop indices before i - max_len
        while q.nonempty() and dp[k - 1, q.right()] > dp[k - 1, i - 1]:
            q.pop_right()  # maintain monotonicity of values
        q.push_right(i - 1)  # Push i - 1
        best_j = q.left()  # guaranteed to have the smallest value
        prev[k, i] = best_j
        dp[k, i] = cost[i] + dp[k - 1, best_j]
```

In comparing AST-Aware Segmentation with Greedy Segmentation—using the example in Figure 2—we find that the former presents more coherent code segments to the model during pretraining. Conversely, the latter introduces noisy partial expressions near partition boundaries. Consequently, AST-Aware Segmentation not only optimizes the pretraining process but also reduces the mismatch between pretraining and downstream tasks, which often involve complete function definitions as inputs.

3.3 Pretraining with Span Corruption

The pretraining task of AST-T5 is based on *span corruption*, a well-established method for pretraining Transformer encoder-decoder models (Raffel et al., 2020). In this approach, 15% of the input tokens are randomly masked and replaced by unique "sentinel" tokens, distinct within each example. Each unique sentinel token is associated with a specific ID and added to the model's vocabulary.

During pretraining, the encoder processes the corrupted input sequence. The decoder's objective is to reconstruct the dropped-out tokens based on the encoder's output representations. Specifically, the target sequence consists of the masked spans of tokens, demarcated by their corresponding sentinel tokens. This framework effectively trains the model to recover the original text from a corrupted input. Figure 1 (Left) illustrates an example of the input-output pair for span corruption.

3.4 Our AST-Aware Subtree Corruption

Algorithm 2 Subtree Selection in AST-Aware Subtree Corruption

```
def mask_subtree(t: ASTNode, m: int): # mask m tokens in subtree t
    ordered_children = []
    m_remaining = m  # distribute m tokens among children of t
    for child in t.children:
        if child.size > theta:  # a hyperparameter to control granularity
            m_child = m * (child.size / t.size)  # same mask ratio
            mask_subtree(child, m_child)  # mask recursively
            m_remaining -= m_child
        else:
            ordered_children.append(child)
    weighted_shuffle(ordered_children)
    for child in ordered_children:  # greedy allocation
        m_child = min(m_remaining, child.size)
        mask_subtree(child, m_child)
        m_remaining -= m_child
```

AST-T5 augments the traditional span corruption paradigm by incorporating AST-awareness. Rather than arbitrarily masking consecutive token spans, AST-T5 masks code spans corresponding to AST subtrees, ranging from individual expressions to entire function bodies.

Subtree Masking. We use a recursive algorithm, outlined in Algorithm 2, to traverse the AST and select subtrees for masking. The algorithm aims to fulfill two goals:

1. Introduce sufficient randomness across training epochs to enhance generalization.
2. Control the masking granularity via a tunable hyperparameter θ (named `theta` in Algorithm 2, Line 5).

The "mask quota" m denotes the number of tokens to be masked in a subtree rooted at node t. The size of a subtree corresponds to the number of tokens it encompasses, derived from the cumulative sizes of its children. For larger subtrees that exceed the size threshold θ, masking is applied recursively (Lines 5-8). Meanwhile, smaller subtrees undergo a weighted shuffle, and the quota m is then apportioned among t's children in a greedy fashion according to the shuffled order (Lines 11-15). The weights for shuffling are determined by a heuristic function on the size of each child, such that masking probabilities are distributed uniformly across leaf nodes. To create a subtree mask for an AST rooted at t with a mask ratio r (e.g., 15% or 25%), one can use `mask_subtree`$(t, \lfloor |t| \cdot r \rfloor)$.

The parameter θ controls the granularity of masking. For example, with $\theta = 5$, the algorithm has a high probability to mask individual tokens and short expressions. As θ increases to 20, the algorithm is more likely to mask larger constructs such as statements. When $\theta = 100$, the probability increases for masking structures like loops or entire function bodies. To foster diverse training scenarios, θ is randomly sampled within a predefined range (e.g., 5 to 100) for each training example. This allows the pretraining framework to inherently accommodate tasks as varied as single-token completion to full function body generation from a given signature.

The subtree masking strategy is the primary distinction between our AST-Aware Subtree Corruption and the Vanilla T5 Span Corruption, as illustrated in Figure 1. While conventional T5 variants mask random token spans, with an average span length of 3 (Raffel et al., 2020) and neglecting code structures, our method targets the masking of AST subtrees, potentially encompassing up to 100 tokens. This equips AST-T5 for generation of various code structures coherently.

Pretraining Objective. Except for the strategy used to select masked tokens and the segmentation strategy described in Section 3.2 , our approach adheres to the workflow described in Section 3.3. Once subtrees are selected for masking and replaced with sentinel tokens, the encoder processes this modified input. Subsequently, the decoder is tasked with reconstructing the original tokens within the masked subtrees. A side-by-side comparison between our approach and the Vanilla Span Corruption in T5 is presented in Figure 1.

4 Experimental Setup

Model Architecture. AST-T5 has an architecture similar to T5_{BASE} (Raffel et al., 2020), comprising a 12-layer encoder and a 12-layer decoder, where each layer has 768 dimensions and 12 attention heads. In total, the model has 226M parameters.

Pretraining. AST-T5 is pretrained on a mixed corpus consisting of code and natural language. Code is sourced from "GitHub repositories" dataset on Google BigQuery, which includes all code files from repositories with open-source licenses permitting redistribution. For NL, we use Wikipedia and OpenWebText, following Liu et al. (2019). Our corpus consists of 408 GB of code and 64 GB of text, smaller than the corpus used by CodeT5 (Wang et al., 2021) and CodeT5+ (Wang et al., 2023). Detailed statistics are provided in Appendix A.3.

Each code file is first parsed into its AST using the Tree-Sitter multi-language parser, and then tokenized with byte-level Byte-Pair Encoding (BPE) using a 64k BPE token vocabulary. Following AST-Aware Segmentation, these files are partitioned into chunks of 1,024 tokens. Our model is pretrained using the AST-Aware Subtree Corruption objective for 524 billion tokens (1,024 tokens per sequence, 1,024 sequences per batch, and 500k steps). For each training example, we apply AST-Aware Subtree Corruption of it is code, or apply Vanilla T5 Span Corruption of it is natural language. For code, the threshold, θ, is uniformly sampled from 5 to 100. Pretraining uses PyTorch, Fairseq[2] and FlashAttention (Dao et al., 2022) and is conducted on 8 nodes, each with 8x NVIDIA A100 40GB GPUs. Further pretraining hyperparameters are detailed in Appendix A.4.

[2] `https://github.com/facebookresearch/fairseq`

Table 1: Performance comparison of various pretraining configurations for downstream tasks. Each row represents a sequential modification applied to the model in the previous row. Metrics include "Pass@1" rate for HumanEval, "Exact Match" rate for CONCODE, Bugs2Fix (for "Small" and "Medium" code lengths splits), and Java-C# transpilation (both Java-to-C# and C#-to-Java). F1 score is used for Clone Detection, and Accuracy for Defect Detection, consistent with prior studies.

	Generation		Transpilation		Understanding		
Pretraining Config	**HumanEval**	**Concode**	**Bugs2Fix**	**Java-C#**	**Clone**	**Defect**	**Avg**
T5	5.2	18.3	21.2/13.8	65.5/68.4	96.9	64.1	44.2
+ AST. Segmentation	7.2	20.2	22.5/15.1	66.3/69.3	98.3	65.9	45.7
+ AST. Subtree Corrupt	9.6	22.1	23.3/**16.5**	67.3/72.2	**98.6**	**66.0**	47.0
+ Mask 25% (AST-T5)	12.8	**22.9**	**23.8**/16.1	**68.9/72.3**	**98.6**	65.8	**47.7**
+ Mask 50%	**13.0**	22.0	21.9/15.0	66.5/70.1	97.1	64.2	46.2

Evaluation. We evaluate AST-T5 across three types of tasks: text-to-code generation, code-to-code transpilation, and code understanding (classification). Our evaluation encompasses tasks from the CodeXGLUE meta-benchmark (Lu et al., 2021) and also includes HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021). Details about the benchmarks are shown in Appendix A.5.

We finetune AST-T5 on the training datasets of all downstream tasks, adhering to the methodology by Raffel et al. (2020). For the HumanEval task, which lacks its own training dataset, we use CodeSearchNet (Husain et al., 2020), aligning with the approach of Wang et al. (2023). The prompt templates for finetuning are constructed using the PromptSource framework (Bach et al., 2022). The finetuning takes 50k steps, with the peak learning rate set at 10% of the pretraining learning rate. All other hyperparameters from pretraining are retained without further adjustments, and we train only one finetuned model. During inference, rank classification is employed for code understanding tasks and beam search for generative tasks, following Sanh et al. (2021). We evaluate our model on the test set using five prompt templates for each task and report the average performance.

Baselines. We first benchmark AST-T5 against our own T5 baselines to ensure a controlled comparison. All models share identical Transformer architectures, pretraining data, and computational settings, differing only in the use of AST-Aware Segmentation and Subtree Corruption techniques by AST-T5. This setup directly evaluates the efficacy of our proposed methods.

We further benchmark AST-T5 against other language models for code-related tasks. These include decoder-only models such as the GPT variants (Brown et al., 2020; Chen et al., 2021a; Wang & Komatsuzaki, 2021), PaLM (Chowdhery et al., 2022), InCoder (Fried et al., 2023), and LLaMa (Touvron et al., 2023). We also compare with encoder-decoder models, including PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021), StructCoder (Tipirneni et al., 2023), and CodeT5+ (Wang et al., 2023). Notably, $\text{CodeT5}_{\text{BASE}}$ and CodeT5+ (220M) closely resemble our model in terms of architecture and size, but AST-T5 distinguishes itself with its AST-Aware pretraining techniques.

5 Evaluation Results

In this section, we evaluate AST-T5 across multiple benchmarks. First, we analyze the contributions of each component within our AST-aware pretraining framework through controlled experiments. Next, we benchmark AST-T5 against existing models in prior work.

5.1 Pretraining Procedure Analysis

In this subsection, we analyze the key components that contribute to the pretraining of AST-T5 models. Holding the model architecture, pretraining datasets, and computational environment constant, we sequentially add one component at a time to a T5 baseline trained on code, culminating in our finalized AST-T5 model. Table 1 presents the experimental results. These results show that:

AST-Aware Segmentation enhances code language models. A comparison between the first two rows of Table 1 shows that the model trained with AST-Aware Segmentation consistently outper-

Table 2: Results of AST-T5 on downstream tasks compared with reported results of established language models. Evaluation metrics align with those in Table 1. Our focus is primarily on models with similar sizes as AST-T5, specifically the "Base" models (110M to 230M parameters), while comparisons against larger models are depicted in Figure 3. Some models are either encoder-only or decoder-only and are thus not suited for certain tasks. These results are labeled with "N/A" in this table because they are not available in the literature.

	Generation		Transpilation		Understanding	
Model	**HumanEval**	**Concode**	**Bugs2Fix**	**Java-C#**	**Clone**	**Defect**
CodeBERT	N/A	N/A	16.4 / 5.2	59.0/58.8	96.5	62.1
GraphCodeBERT	N/A	N/A	17.3 / 9.1	59.4/58.8	97.1	N/A
PLBART	N/A	18.8	19.2 / 9.0	64.6/65.0	97.2	63.2
CodeT5	N/A	22.3	21.6/14.0	65.9/66.9	97.2	65.8
CodeT5+$_{\text{BASE}}$	12.0	N/A	N/A	N/A	95.2	**66.1**
StructCoder	N/A	22.4	N/A	66.9/68.7	N/A	N/A
AST-T5 (Ours)	**12.8**	**22.9**	**23.8/16.1**	**68.9/72.3**	**98.6**	65.8

forms the T5 baseline that uses Greedy Segmentation across all tasks. The advantage stems from the fact that AST-Aware Segmentation produces less fragmented and thus less noisy training inputs during pretraining. Given that most downstream tasks present coherent code structures, such as entire function definitions, the consistency upheld by AST-Aware pretraining aligns better with these structures, leading to improved generalization.

AST-Aware Span Corruption further boosts generation performance. A comparison between the second and third rows of Table 1 reveals an improvement when shifting from Vanilla T5 Span Corruption to our AST-Aware Subtree Corruption. This performance gain is especially notable in generation and transpilation tasks. Such enhancements stem from the ability of AST-Aware Subtree Corruption to guide the model in generating code with better coherence and structural integrity.

Increasing masking ratio improves generation performance. The typical span corruption mask ratio in T5 is set at 15%. Increasing this ratio could potentially enhance the model's generation capabilities, albeit potentially at the expense of understanding tasks. Essentially, a mask ratio of 100% would emulate a GPT-like, decoder-only Transformer. However, in our experiments (last two rows of Table 1), we observed that raising the mask ratio from 15% to 25% significantly improved generation capabilities without noticeably compromising performance in understanding tasks. Further analysis shows that increasing the masking ratio to 50% yields only a marginal improvement on HumanEval (from 12.8 to 13.0), while adversely impacting transpilation and understanding tasks. Thus, we settled on a 25% mask ratio for our AST-T5 model.

5.2 Main Results

Table 2 shows AST-T5's performance on downstream tasks compared with previously published results of similarly sized models, specifically those within the "Base" scale (110M to 230M parameters). Figure 3 and Figure 4 extends this comparison, comparing AST-T5 with larger models using the HumanEval benchmark and the MBPP benchmark, respectively. These results show that:

AST-T5 excels as a unified and parameter-efficient LM for various code-related tasks. While comparable in size, AST-T5 consistently outperforms similar-sized models such as CodeT5 (Wang et al., 2021) and CodeT5+ (Wang et al., 2023) in code generation, transpilation, and understanding. Notably, while CodeT5 and CodeT5+ are models at the Base scale, they were evaluated across different tasks. Our model, AST-T5, outperforms the best results of these two models across multiple benchmarks at the same time. Moreover, Figure 3 highlights AST-T5's competitiveness against significantly larger models like GPT-J (Wang & Komatsuzaki, 2021) and LLaMa-7B (Touvron et al., 2023) on the HumanEval benchmark, underscoring our model's parameter efficiency.

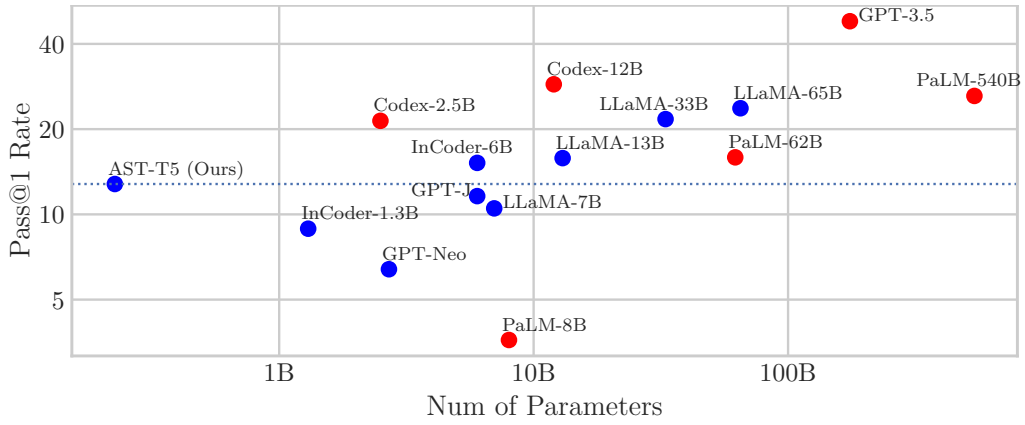

Figure 3: Visualization of AST-T5's performance on HumanEval compared to models exceeding 230M parameters and thus not detailed in Table 2. Each point on the scatter plot represents a model. The x-axis shows the parameter count in log-scale, while the y-axis shows the Pass@1 rate on HumanEval in log-scale. Model open-source status is color-coded: **blue** for open-source and **red** for proprietary.

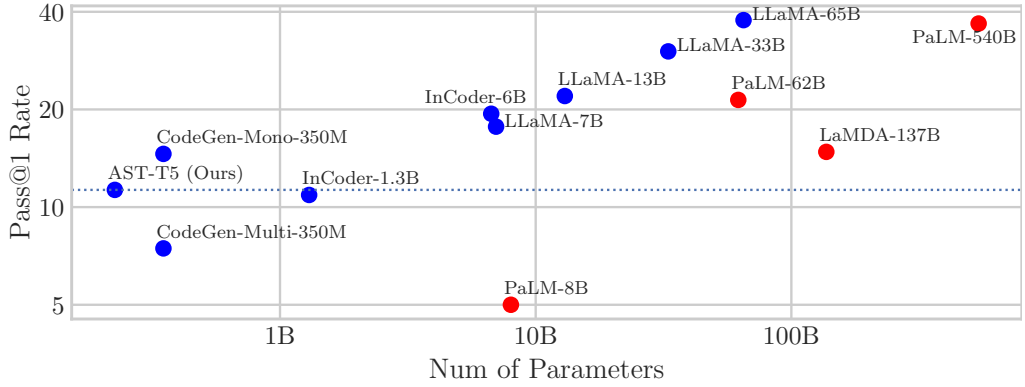

Figure 4: Visualization of AST-T5's performance on MBPP compared to other models. Each point on the scatter plot represents a model.

AST-T5 exhibits unique strengths in transpilation through AST-awareness. Table 2 highlights AST-T5's superior performance in code-to-code transpilation tasks, showcasing gains a substantial gain of 2 to 5 points on Bugs2Fix and Java-C# transpilation. In transpilation, while surface-level code can exhibit significant variability, the intrinsic AST structures of the source and target often maintain a notable similarity. The capability of AST-T5 to exploit this structural similarity is crucial to its effectiveness. The benefits of being structure-aware are further exemplified by AST-T5's leading results in Clone Detection, where it surpasses CodeT5 by 3 points, because AST comparisons yield more precise insights than direct code comparisons.

6 Conclusion and Future Work

In this work, we present AST-T5, a novel pretraining paradigm that harnesses the power of Abstract Syntax Trees (ASTs) to boost the performance of code-centric language models. Using two structure-aware techniques, AST-T5 not only outperforms models of comparable size but also competes favorably against some larger counterparts. The simplicity of AST-T5 lies in its singular pretraining objective and its adaptability as a drop-in replacement for any encoder-decoder LM, highlighting its potential for real-world deployments. Moving forward, we aim to explore the scalability of AST-T5 by training larger models on more expansive datasets.

REFERENCES

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. Apr 2021. doi: 10.48550/arXiv.2103.06333. URL http://arxiv.org/abs/2103.06333. arXiv:2103.06333 [cs].

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. Nov 2017. URL https://arxiv.org/abs/1711.00740. arXiv:1711.00740 [cs].

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. July 2020. doi: 10.48550/arXiv.1910.00577. URL http://arxiv.org/abs/1910.00577. arXiv:1910.00577 [cs, stat].

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. (arXiv:2210.14868), March 2023. URL http://arxiv.org/abs/2210.14868. arXiv:2210.14868 [cs].

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. Aug 2021. doi: 10.48550/arXiv.2108.07732. URL http://arxiv.org/abs/2108.07732. arXiv:2108.07732 [cs].

Stephen H. Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, Taewoon Kim, M. Saiful Bari, Thibault Fevry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged S. Al-shaibani, Shanya Sharma, Urmish Thakker, Khalid Almubarak, Xiangru Tang, Dragomir Radev, Mike Tian-Jian Jiang, and Alexander M. Rush. PromptSource: An integrated development environment and repository for natural language prompts. March 2022. doi: 10.48550/arXiv.2202.01279. URL http://arxiv.org/abs/2202.01279. arXiv:2202.01279 [cs].

BigScience. Bigscience Language Open-science Open-access Multilingual (BLOOM), May 2021. URL https://huggingface.co/bigscience/bloom.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. Jul 2020. doi: 10.48550/arXiv.2005.14165. URL http://arxiv.org/abs/2005.14165. arXiv:2005.14165 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. Jul 2021a. doi: 10.48550/arXiv.2107.03374. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. Sep 2018. URL https://openreview.net/forum?id=H1gfOiAqYm.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis. Jun 2021b. URL https://arxiv.org/abs/2107.00101. arXiv:2107.00101 [cs].

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways. Oct 2022. doi: 10.48550/arXiv.2204.02311. URL http://arxiv.org/abs/2204.02311. arXiv:2204.02311 [cs].

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. June 2022. doi: 10.48550/arXiv.2205.14135. URL http://arxiv.org/abs/2205.14135. arXiv:2205.14135 [cs].

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. Sep 2020. doi: 10.48550/arXiv.2002.08155. URL http://arxiv.org/abs/2002.08155. arXiv:2002.08155 [cs].

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. Apr 2023. doi: 10.48550/arXiv.2204.05999. URL http://arxiv.org/abs/2204.05999. arXiv:2204.05999 [cs].

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. Sep 2021. doi: 10.48550/arXiv.2009.08366. URL http://arxiv.org/abs/2009.08366. arXiv:2009.08366 [cs].

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. Jun 2020. doi: 10.48550/arXiv.1909.09436. URL http://arxiv.org/abs/1909.09436. arXiv:1909.09436 [cs, stat].

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. Aug 2018. doi: 10.48550/arXiv.1808.09588. URL http://arxiv.org/abs/1808.09588. arXiv:1808.09588 [cs].

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. March 2021. doi: 10.48550/arXiv.2003.13848. URL http://arxiv.org/abs/2003.13848. arXiv:2003.13848 [cs].

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. Sep 2020. doi: 10.48550/arXiv.2006.03511. URL http://arxiv.org/abs/2006.03511. arXiv:2006.03511 [cs].

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. Oct 2019. doi: 10.48550/arXiv.1910.13461. URL http://arxiv.org/abs/1910.13461. arXiv:1910.13461 [cs, stat].

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, pp. 4159–4165, July 2018. doi: 10.24963/ijcai.2018/578. URL http://arxiv.org/abs/1711.09573. arXiv:1711.09573 [cs].

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. Jul 2019. doi: 10.48550/arXiv.1907.11692. URL http://arxiv.org/abs/1907.11692. arXiv:1907.11692 [cs].

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. Mar 2021. doi: 10.48550/arXiv.2102.04664. URL http://arxiv.org/abs/2102.04664. arXiv:2102.04664 [cs].

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. Feb 2023. doi: 10.48550/arXiv.2203.13474. URL http://arxiv.org/abs/2203.13474. arXiv:2203.13474 [cs].

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. Mar 2022. doi: 10.48550/arXiv.2203.02155. URL http://arxiv.org/abs/2203.02155. arXiv:2203.02155 [cs].

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. April 2017. doi: 10.48550/arXiv.1704.07535. URL http://arxiv.org/abs/1704.07535. arXiv:1704.07535 [cs, stat].

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. Jul 2020. doi: 10.48550/arXiv.1910.10683. URL http://arxiv.org/abs/1910.10683. arXiv:1910.10683 [cs, stat].

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. (arXiv:2009.10297), September 2020. doi: 10.48550/arXiv.2009.10297. URL http://arxiv.org/abs/2009.10297. arXiv:2009.10297 [cs].

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. Oct 2021. doi: 10.48550/arXiv.2102.07492. URL http://arxiv.org/abs/2102.07492. arXiv:2102.07492 [cs].

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. Aug 2023. doi: 10.48550/arXiv.2308.12950. URL http://arxiv.org/abs/2308.12950. arXiv:2308.12950 [cs].

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M. Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Tali Bers, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. Multitask prompted training enables zero-shot task generalization. arXiv.org, Oct 2021. URL https://arxiv.org/abs/2110.08207v3.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. Jan 2023. URL https://arxiv.org/abs/2301.13816. arXiv:2301.13816 [cs].

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 476–480, Sep 2014. doi: 10.1109/ICSME.2014.77.

Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. StructCoder: Structure-aware transformer for code generation. May 2023. doi: 10.48550/arXiv.2206.05239. URL http://arxiv.org/abs/2206.05239. arXiv:2206.05239 [cs].

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. Feb 2023. doi: 10.48550/arXiv.2302.13971. URL http://arxiv.org/abs/2302.13971. arXiv:2302.13971 [cs].

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. May 2019. doi: 10.48550/arXiv.1812.08693. URL http://arxiv.org/abs/1812.08693. arXiv:1812.08693 [cs].

Ben Wang and Aran Komatsuzaki. GPT-J-6B: 6B JAX-based Transformer, Jun 2021. URL https://arankomatsuzaki.wordpress.com/2021/06/04/gpt-j/.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Sep 2021. doi: 10.48550/arXiv.2109.00859. URL http://arxiv.org/abs/2109.00859. arXiv:2109.00859 [cs].

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. CodeT5+: Open code large language models for code understanding and generation. May 2023. doi: 10.48550/arXiv.2305.07922. URL http://arxiv.org/abs/2305.07922. arXiv:2305.07922 [cs].

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models. (arXiv:2205.01068), June 2022. doi: 10.48550/arXiv.2205.01068. URL http://arxiv.org/abs/2205.01068. arXiv:2205.01068 [cs].

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Sep 2019. doi: 10.48550/arXiv.1909.03496. URL http://arxiv.org/abs/1909.03496. arXiv:1909.03496 [cs, stat].

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. March 2021. doi: 10.48550/arXiv.2103.11318. URL http://arxiv.org/abs/2103.11318. arXiv:2103.11318 [cs].

A Appendix

A.1 Limitations

AST-T5 is specifically designed to enhance code generation performance by exclusively masking code within AST subtrees during pretraining. While this specialized approach is advantageous for code generation tasks, it may result in suboptimal performance in natural language generation. Acknowledging this limitation, future versions of AST-T5 could investigate strategies such as masking docstrings and comments to broaden its applicability. This would potentially improve performance across various tasks, including code summarization.

A.2 More about AST-Aware Segmentation

In Section 3.2, we use a dynamic programming algorithm to calculate the segmentation that results in the least number of AST structure breaks. A naive implementation of the DP algorithm is shown in Algorithm 3.

Algorithm 3 Dynamic Programming in AST-Aware Segmentation (Before Optimization)

```
for k in range(1, m + 1):
    for i in range(1, n + 1):
        best_j = i - max_len
        for j in range(i - max_len + 1, i):
            if dp[k - 1, j] < dp[k - 1, best_j]:
                best_j = j
        prev[k, i] = best_j
        dp[k, i] = cost[i] + min_value
```

Denote the length of the code file (in tokens) by n. In the algorithm, m denotes the maximum number of chunks that the file can be split into, which is approximately $n/\text{max\_len}$. So this implementation has time complexity $O(mn \cdot \text{max\_len}) = O(n^2)$, which is not feasible for longer code files. To optimize this algorithm, we use a monotonic queue to compute the sliding-window minimum, as described in Algorithm 1.

Each element is only pushed into and popped out of the monotonic queue once, so the time complexity of the optimized algorithm is $O(nm) = O(n^2/\text{max\_len})$, making the algorithm $\tilde{1}000$x faster when $\text{max\_len} = 1024$. This allows the algorithm to segment each code file with 100k tokens in milliseconds.

A.3 Pretraining Corpora

Our pretraining corpora consists of two parts: code and natural language, coming from three sources:

- **GitHub (408 GB):** The "GitHub repositories" public dataset available on Google BigQuery [3]. For pretraining, we use all code files in Python (70 GB), C/C++ (195 GB), Java (105 GB), C# (38 GB) from each repo with an open-source license that explicitly permits redistribution.
- **Wikipedia (16 GB):** A natural language corpus widely used for natural language pretraining.
- **OpenWebText (38 GB):** A natural language corpus used by Liu et al. (2019) to train language models.

A.4 Pretraining Hyperparameters

Table 3 shows the pretraining hyperparameters for our proposed AST-T5 model.

[3] https://console.cloud.google.com/marketplace/details/github/github-repos

Encoder Layers	12
Decoder Layers	12
Hidden Dimension	768
Peak Learning Rate	2e-4
Batch Size	1,024
Warm-Up Steps	10,000
Total Steps	500,000
Sequence Length	1,024
Mask Ratio	25%
Min Subtree Corruption Threshold θ	5
Max Subtree Corruption Threshold θ	100
Relative Position Encoding Buckets	32
Relative Position Encoding Max Distance	128
Adam ϵ	1e-6
Adam (β_1, β_2)	(0.9, 0.98)
Clip Norm	2.0
Dropout	0.1
Weight Decay	0.01

Table 3: Pretraining hyperparameters for our AST-T5 model.

Table 4: Overview of our evaluation benchmarks detailing test set size, task type, and evaluation metric for each task. For MBPP, we follow Nijkamp et al. (2023) and evaluate our model on the entire "sanitized" subset without few-shot prompts. For evaluation metrics, "Pass@1" indicates code execution on unit-tests provided in the benchmark using a single generated code per example, with reported pass rates. "Exact Match" evaluates textual equivalence without execution by comparing two canonicalized code pieces. We omit "BLEU scores" because high BLEU values (> 50) can still correspond to unexecutable or significantly flawed code (Lu et al., 2021), which is not useful in real-world applications. We also discuss evaluation results using the CodeBLEU (Ren et al., 2020) metric in Appendix A.7.

	Size	**Type**	**Metric**
HumanEval	164	Text-to-Code Generation	Pass@1
MBPP	427	Text-to-Code Generation	Pass@1
Concode	2,000	Text-to-Code Generation	Exact Match
Bugs2Fix	12,379	Code-to-Code Transpilation	Exact Match
Java-C#	1,000	Code-to-Code Transpilation	Exact Match
BigCloneBench	415,416	Code Understanding	F1
Defect Detection	27,318	Code Understanding	Accuracy

A.5 Evaluation Benchmarks

We evaluate AST-T5 across three types of tasks: text-to-code generation, code-to-code transpilation, and code understanding (classification). Our evaluation encompasses tasks from the CodeXGLUE meta-benchmark (Lu et al., 2021) and also includes HumanEval (Chen et al., 2021a) and MBPP (Austin et al., 2021). Specifically, for text-to-code generation, we assess performance using HumanEval, MBPP, and Concode (Iyer et al., 2018); for transpilation, we use CodeXGLUE Java-C# and Bugs2Fix (Tufano et al., 2019) for evaluation; and for understanding, we use BigCloneBench (Svajlenko et al., 2014) and the Defect Detection task proposed by Zhou et al. (2019). Detailed metrics and statistics of these datasets are provided in Table 4.

A.6 Evaluation Results on Multi-Lingual Code Generation

Table 5 presents a comparative analysis of our AST-T5 model on Python and Java subsets of the multi-lingual HumanEval and MBXP benchmarks (Athiwaratkun et al., 2023). This analysis includes models such as BLOOM (BigScience, 2021), OPT (Zhang et al., 2022), and various con-

Table 5: Results of AST-T5 on multi-lingual HumanEval and MBXP compared with reported results of established language models. The evaluation metric is Pass@1.

	#Params	HumanEval		MBXP	
		Python	Java	Python	Java
CodeGen-multi	350M	7.3	5.0	7.5	8.2
CodeGen-mono	350M	10.3	3.1	**14.6**	1.9
AST-T5 (Ours)	226M	**12.8**	**10.6**	11.3	**9.8**
BLOOM	7.1B	7.9	8.1	7.0	7.8
OPT	13B	0.6	0.6	1.4	1.4
CodeGen-multi	2B	11.0	11.2	18.8	19.5
CodeGen-mono	2B	20.7	5.0	31.7	16.7
CodeGen-multi	6B	15.2	10.6	22.5	21.7
CodeGen-mono	6B	19.5	8.7	37.2	19.8
CodeGen-multi	16B	17.1	16.2	24.2	28.0
CodeGen-mono	16B	22.6	22.4	40.6	26.8

Table 6: Results of AST-T5 on CONCODE with reported results of established language models. The evaluation metric is exact match score and CodeBLEU.

	EM	CodeBLEU
GPT-2	17.4	29.7
CodeGPT-2	18.3	32.7
CodeGPT-adapted	20.1	36.0
PLBART	18.8	38.5
CodeT5-Small	21.6	41.4
CodeT5-Base	22.3	43.2
AST-T5 (Ours)	**22.9**	**45.0**

figurations of CodeGen (Nijkamp et al., 2023), as reported in Athiwaratkun et al. (2023). Our results show AST-T5's superior performance across all benchmarks compared to the CodeGen-multi-350M. Notably, although CodeGen-mono-350M, tailored for Python, surpasses AST-T5 in the MBPP benchmark, it significantly underperforms in the Java subset. Furthermore, AST-T5, having 226M parameters, outperforms larger counterparts like BLOOM-7.1B and OPT-13B.

A.7 Evaluation Results in CodeBLEU

Table 6 presents the performance of various models on the Concode dataset using the CodeBLEU metric, as reported in (Wang et al., 2021). CodeBLEU, specifically designed for evaluating code synthesis, computes a weighted average of three scores: textual match (BLEU), AST match, and Data Flow Graph (DFG) match. Our findings show a clear correlation between CodeBLEU and exact match scores.