AST-T5: Structure-Aware Pretraining for Code Generation and Understanding

Linyuan Gong [1] **Mostafa Elhoushi** [2] **Alvin Cheung** [1]

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 complex 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 including HumanEval and MBPP. 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://github.com/gonglinyuan/ast_t5`.

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 such as 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.

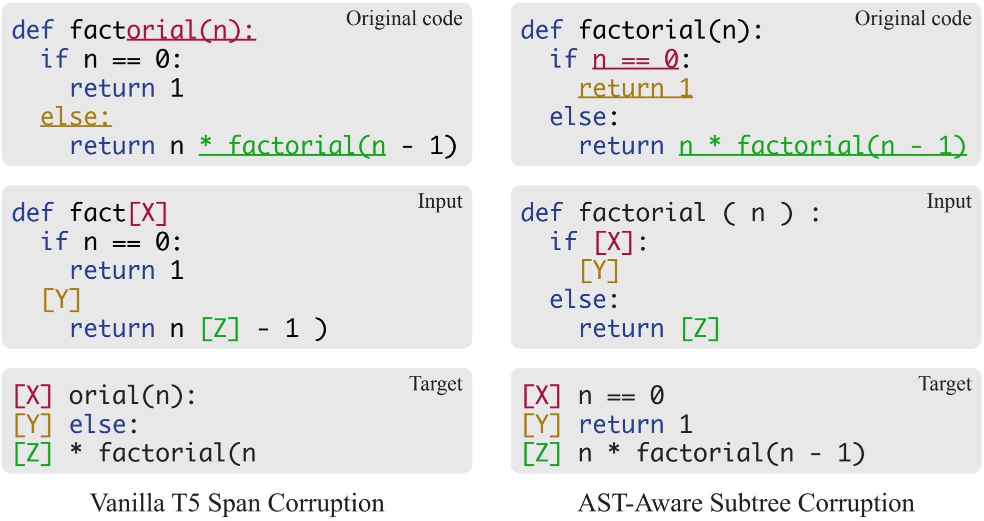
In this study, we propose AST-T5, a pretraining paradigm that leverages the Abstract Syntax Tree (AST) structure of code. The key contribution in AST-T5 is a simple yet effective way to exploit code semantics, without the need to run expensive program analysis or execution. Using a lightweight, multi-language 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 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

[1] Department of EECS, University of California at Berkeley, Berkeley, California, USA [2] AI at Meta, USA. Correspondence to: Linyuan Gong <gly@berkeley.edu>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

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



```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
```

```
def fact[X]
  if n == 0:
    return 1
  [Y]
    return n [Z] - 1 )
```

```
[X] orial(n):
[Y] else:
[Z] * factorial(n
```

Vanilla T5 Span Corruption

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
```

```
def factorial ( n ) :
  if [X]:
    [Y]
  else:
    return [Z]
```

```
[X] n == 0
[Y] return 1
[Z] n * factorial(n - 1)
```

AST-Aware Subtree Corruption

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.

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 (Chen et al., 2021a) and the MBPP (Austin et al., 2021) benchmarks. 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 finetuned 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., 2023b). 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 requires 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. 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.

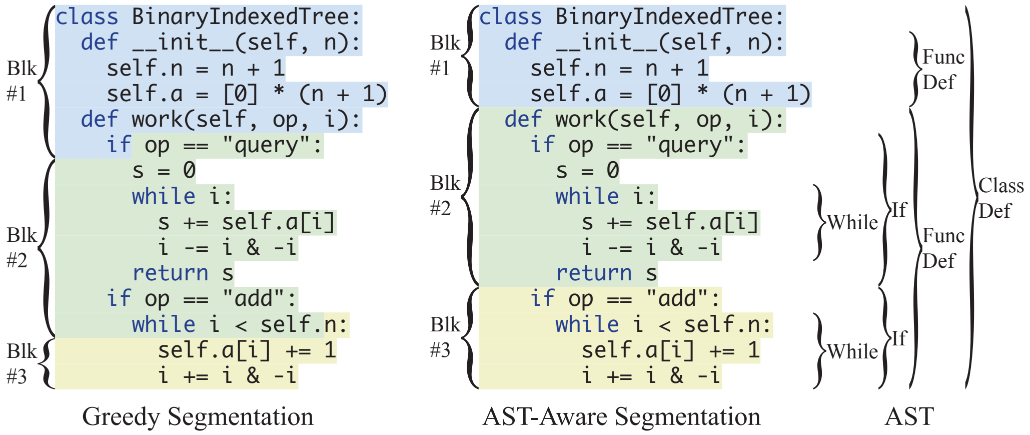
Segmentation in language model pretraining is a critical yet often overlooked aspect. Transformer LMs impose token limits on input sequences, making segmentation essential for fitting these inputs within the `max_len` constraint. A naive approach is Greedy Segmentation, where each chunk, except the last, 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 developed the following dynamic programming (DP)-based 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



```
class BinaryIndexedTree:
  def __init__(self, n):
    self.n = n + 1
    self.a = [0] * (n + 1)
  def work(self, op, i):
    if op == "query":
      s = 0
      while i:
        s += self.a[i]
        i -= i & -i
      return s
    if op == "add":
      while i < self.n:
        self.a[i] += 1
        i += i & -i
```

```
class BinaryIndexedTree:
  def __init__(self, n):
    self.n = n + 1
    self.a = [0] * (n + 1)
  def work(self, op, i):
    if op == "query":
      s = 0
      while i:
        s += self.a[i]
        i -= i & -i
      return s
    if op == "add":
      while i < self.n:
        self.a[i] += 1
        i += i & -i
```

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.

equation is:

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

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 $O(n^2/\mathsf{max\_len})$ 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 $\mathsf{k\_min} = \arg\min_k(\mathsf{dp}[k,n])$, as the most optimal partition.

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

AST-T5's pretraining 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.

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):
      # pop indices before i - max_len
      q.pop_left()
    while (q.nonempty() and
           dp[k-1, q.right()] > dp[k-1, i-1]):
      # maintain monotonicity of values
      q.pop_right()
    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]
```

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

Algorithm 2 Subtree Selection in AST-Aware Subtree Corruption

```
1  def mask_subtree(t: ASTNode, m: int):
2    """mask m tokens in subtree t"""
3    ordered_children = []
4    m_remaining = m
5    # distribute m tokens among children of t
6    for child in t.children:
7      # theta: a hyperparameter to control
8      #        masking granularity
9      if child.size > theta:
10       # same mask ratio as the current subtree
11       m_child = m * (child.size / t.size)
12       mask_subtree(child, m_child)  # recurse
13       m_remaining -= m_child
14     else:
15       ordered_children.append(child)
16   weighted_shuffle(ordered_children)
17   # greedy allocation of remaining mask quota
18   for child in ordered_children:
19     m_child = min(m_remaining, child.size)
20     mask_subtree(child, m_child)
21     m_remaining -= m_child
```

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. AST-Aware Subtree Corruption

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 9).

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 9-13). 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 17-21). 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$_{\text{BASE}}$ (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 277M parameters.

Pretraining. AST-T5 is pretrained on a subset of The Stack Dedup corpus (Kocetkov et al., 2022), a near-deduplicated version of The Stack—a 3.1TB collection of permissively licensed source code from GitHub cutoff at April 2022, spanning 358 programming languages. For our experiments, AST-T5's training involves Python, Java, C, C++, C#, Markdown, and reStructuredText subsets, compris-

Table 1: Pretraining hyperparameters for AST-T5.

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
Min Text Corruption Span Length	1
Max Text Corruption Span Length	10
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

ing a 588GB dataset with 93M code and natural language files.

Each 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 byte-level 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. For text, the length of each masked span is uniformly sampled from 1 to 10. Pretraining uses PyTorch, Fairseq[2] and FlashAttention (Dao et al., 2022) and is conducted on 8 nodes, each with 8x NVIDIA A100 40GB GPUs. We use the `cl100k_base` byte-level BPE vocabulary from `tiktoken`[3], which consists of 100k tokens. Table 1 shows the pretraining hyperparameters for AST-T5.

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). 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 2.

Table 2: Overview of our evaluation benchmarks about test set size, task type, and evaluation metric for each task. "Generation" tasks involve mapping natural language to code, "Transpilation" tasks involve translating code from one programming language to another, and "Understanding" tasks involve classifying code into categorical labels. For MBPP, we follow Nijkamp et al. (2023b) 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. "EM" (Exact Match) evaluates textual equivalence without execution by comparing two canonicalized code pieces. "Acc" means accuracy in classification tasks. 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.5.

	Size	Type	Metric
HumanEval	164	Generation	Pass@1
MBPP	427	Generation	Pass@1
Concode	2,000	Generation	EM
Bugs2Fix	12,379	Transpilation	EM
Java-C#	1,000	Transpilation	EM
BigCloneBench	415,416	Understanding	F1
Defect Detect	27,318	Understanding	Acc

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 is used for generative tasks, following Sanh et al. (2021). For CodeXGLUE, we evaluate our model on the test set using five prompt templates for each task and

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

[3] https://github.com/openai/tiktoken

Table 3: 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)	14.0	**22.9**	**23.8**/16.1	**68.9/72.3**	**98.6**	65.8	**47.9**
+ Mask 50%	**14.3**	22.0	21.9/15.0	66.5/70.1	97.1	64.2	46.4

report the average performance; for HumanEval and MBPP, we evaluate the top-1 generated output from beam search.

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; Black et al., 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, 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 3 presents the experimental results. These results show that:

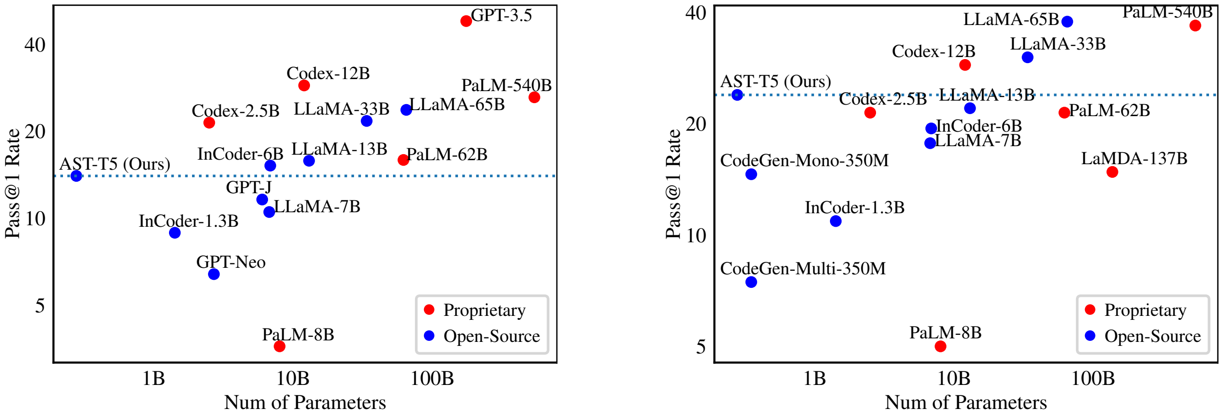
AST-Aware Segmentation enhances code language models. A comparison between the first two rows of Table 3 shows that the model trained with AST-Aware Segmentation consistently outperforms the T5 baseline that uses Greedy Segmentation across all tasks. The advantage stems from the fact that AST-Aware Segmentation 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 3 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 3), 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

Table 4: 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 (100M to 300M 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)	**14.0**	**22.9**	**23.8/16.1**	**68.9/72.3**	**98.6**	65.8



(a) HumanEval

(b) MBPP

Figure 3: Visualizations of AST-T5's performance on HumanEval and MBPP compared to other models compared to models exceeding 300M parameters. Each point on each 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 or MBPP in log-scale. Model open-source status is color-coded: **blue** for open-source and **red** for proprietary.

50% yields only a marginal improvement on HumanEval (from 14.0 to 14.3), 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 4 shows AST-T5's performance on downstream tasks compared with previously published results of similarly sized models, specifically those within the "Base" scale (100M to 300M parameters). Figure 3a and Figure 3b extends this comparison, comparing AST-T5 with larger models using the HumanEval benchmark and the MBPP benchmark, respectively. Additional results on EvalPlus are shown in Appendix A.3. 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 3a 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. Similarly, Figure 3b demonstrates AST-T5's advantages over LLaMa-7B and Codex-2.5B (Chen et al., 2021a) on the MBPP benchmark, showing the effectiveness of AST-T5.

AST-T5 exhibits unique strengths in transpilation through AST-awareness. Table 4 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.

Acknowledgements

We thank the reviewers and metareviewers at ICML for their constructive feedback and support. This work is supported in part by gift from Meta, the U.S. National Science Foundation through grants IIS-1955488, IIS-2027575, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE awards DE-SC0016260, DE-SC0021982. The AI training platform and computation resources supporting this work were provided by High-Flyer AI Fundamental Research Co. Ltd, Hangzhou, China. We thank Sida Wang, Chris Cummins, Volker Seeker, and Hugh Leather for their valuable discussions.

Impact Statement

In this paper, we introduce AST-T5, a language model aimed at automated generation, transpilation, and understanding of code. The advancement of LLMs in code generation raises concerns about automated code production's security, privacy, and potential misuse. There is a risk that improved code generation capabilities could be exploited for malicious purposes, such as automating the creation of software vulnerabilities or facilitating the development of harmful software. Our research emphasizes the importance of responsible AI development and use, advocating for continuous monitoring, ethical guidelines, and safeguards to mitigate these risks.

References

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. 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].

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

Alon, U., Sadaka, R., Levy, O., and Yahav, E. 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].

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., Gonugondla, S. K., Ding, H., Kumar, V., Fulton, N., Farahani, A., Jain, S., Giaquinto, R., Qian, H., Ramanathan, M. K., Nallapati, R., Ray, B., Bhatia, P., Sengupta, S., Roth, D., and Xiang, B. Multi-lingual evaluation of code generation models. (arXiv:2210.14868), March 2023. URL http://arxiv.org/abs/2210.14868. arXiv:2210.14868 [cs].

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. 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].

Bach, S. H., Sanh, V., Yong, Z.-X., Webson, A., Raffel, C., Nayak, N. V., Sharma, A., Kim, T., Bari, M. S., Fevry, T., Alyafeai, Z., Dey, M., Santilli, A., Sun, Z., Ben-David, S., Xu, C., Chhablani, G., Wang, H., Fries, J. A., Alshaibani, M. S., Sharma, S., Thakker, U., Almubarak, K., Tang, X., Radev, D., Jiang, M. T.-J., and Rush, A. M. 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.

Black, S., Gao, L., Wang, P., Leahy, C., and Biderman, S. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. URL https://doi.org/10.5281/zenodo.5297715.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. 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].

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. 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].

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

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

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. 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].

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. 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].

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. 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].

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. 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].

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. 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].

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. 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].

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. 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].

Kim, S., Zhao, J., Tian, Y., and Chandra, S. 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].

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The Stack: 3 TB of permissively licensed source code. (arXiv:2211.15533), November 2022. doi: 10.48550/arXiv.2211.15533. URL http://arxiv.org/abs/2211.15533. arXiv:2211.15533 [cs].

Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. Sep 2020. doi: 10.48550/arXiv.2006.03511. URL http://arxiv.org/abs/2006.03511. arXiv:2006.03511 [cs].

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. 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].

Li, J., Wang, Y., Lyu, M. R., and King, I. 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].

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In Thirty-seventh Conference on Neural Information Processing Systems, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. 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].

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. 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].

Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., and Zhou, Y. CodeGen2: Lessons for training LLMs on programming and natural languages. (arXiv:2305.02309), July 2023a. doi: 10.48550/arXiv.2305.02309. URL http://arxiv.org/abs/2305.02309. arXiv:2305.02309 [cs].

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. CodeGen: An open large language model for code with multi-turn program synthesis. Feb 2023b. doi: 10.48550/arXiv.2203.13474. URL http://arxiv.org/abs/2203.13474. arXiv:2203.13474 [cs].

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. 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].

Rabinovich, M., Stern, M., and Klein, D. 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].

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. 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].

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. 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].

Roziere, B., Lachaux, M.-A., Szafraniec, M., and Lample, G. 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].

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. 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].

Sanh, V., Webson, A., Raffel, C., Bach, S. H., Sutawika, L., Alyafeai, Z., Chaffin, A., Stiegler, A., Scao, T. L., Raja, A., Dey, M., Bari, M. S., Xu, C., Thakker, U., Sharma, S. S., Szczechla, E., Kim, T., Chhablani, G., Nayak, N., Datta, D., Chang, J., Jiang, M. T.-J., Wang, H., Manica, M., Shen, S., Yong, Z. X., Pandey, H., Bawden, R., Wang, T., Neeraj, T., Rozen, J., Sharma, A., Santilli, A., Fevry, T., Fries, J. A., Teehan, R., Bers, T., Biderman, S., Gao, L., Wolf, T., and Rush, A. M. Multitask prompted training enables zero-shot task generalization. arXiv.org, Oct 2021. URL https://arxiv.org/abs/2110.08207v3.

Shojaee, P., Jain, A., Tipirneni, S., and Reddy, C. K. Execution-based code generation using deep reinforcement learning. Jan 2023. URL https://arxiv.org/abs/2301.13816. arXiv:2301.13816 [cs].

Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M. 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.

Tipirneni, S., Zhu, M., and Reddy, C. K. 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].

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. 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].

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. 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].

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

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. H. 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].

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., and Hoi, S. C. H. 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].

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. 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].

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. 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].

Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., and Günnemann, S. 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 $\sim$ 1000x faster when $\text{max\_len} = 1024$. This allows the algorithm to segment each code file with 100k tokens in milliseconds.

A.3. Evaluation Results on EvalPlus

We extend our evaluation to include EvalPlus (Liu et al., 2023), a more rigorous benchmark that enhances the original HumanEval and MBPP datasets with a substantial number of additional test cases. EvalPlus is designed to provide a more accurate evaluation of the correctness of programs produced by LLMs.

For our tests on HumanEval+ and MBPP+, we use the same hyperparameters used in our evaluations of HumanEval and MBPP. It is important to note that the hyperparameter configurations used in our study are not directly comparable to those used for the models listed on the EvalPlus leaderboard[4]. Our results are compared against established models including GPT-Neo, GPT-J, InCoder, and CodeGen-2 (Nijkamp et al., 2023a).

As shown in Table 5, our 277M-parameter AST-T5 outperforms larger models like InCoder-6.7B and CodeGen2-1B, showing the effectiveness and parameter efficiency of AST-T5.

A.4. Evaluation Results on Multi-Lingual Code Generation

Table 6 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 configurations of CodeGen (Nijkamp et al., 2023b), as reported in Athiwaratkun et al. (2023). Our results show AST-T5's superior performance across all benchmarks compared to the CodeGen-multi-350M.

[4] https://evalplus.github.io/leaderboard.html

Table 5: Performance of AST-T5 on HumanEval+ and MBPP+ benchmarks, compared with reported numbers of language models listed on the EvalPlus leaderboard. The evaluation metric used is Pass@1.

	#Params	**HumanEval+**	**MBPP+**
GPT-Neo	2.7B	6.7	7.9
GPT-J	6B	11.0	12.2
InCoder-1.3B	1.3B	11.0	12.2
InCoder-6.7B	6.7B	12.2	15.9
CodeGen2-1B	1B	9.1	11.0
CodeGen2-3B	3B	12.8	15.9
CodeGen2-7B	7B	17.7	18.3
CodeGen2-16B	16B	16.5	19.5
AST-T5 (Ours)	277M	12.8	19.3

Table 6: 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)	277M	14.0	**10.6**	**23.9**	**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

Furthermore, AST-T5, having 277M parameters, outperforms larger counterparts like BLOOM-7.1B and OPT-13B.

A.5. Evaluation Results in CodeBLEU

Table 7 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.

Table 7: 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**