Outline, Then Details: Syntactically Guided Coarse-To-Fine Code Generation

Wenqing Zheng [1] **S P Sharan** [1] **Ajay Kumar Jaiswal** [1] **Kevin Wang** [1] **Yihan Xi** [1] **Dejia Xu** [1] **Zhangyang Wang** [1]

Abstract

For a complicated algorithm, its implementation by a human programmer usually starts with outlining a rough control flow followed by iterative enrichments, eventually yielding carefully generated syntactic structures and variables in a hierarchy. However, state-of-the-art large language models generate codes in a single pass, without intermediate warm-ups to reflect the structured thought process of "outline-then-detail". Inspired by the recent success of chain-of-thought prompting, we propose **ChainCoder**, a program synthesis language model that generates Python code progressively, i.e. *from coarse to fine* in *multiple passes*. We first decompose source code into layout frame components and accessory components via abstract syntax tree parsing to construct a hierarchical representation. We then reform our prediction target into a multi-pass objective, each pass generates a subsequence, which is concatenated in the hierarchy. Finally, a tailored transformer architecture is leveraged to jointly encode the natural language descriptions and syntactically aligned I/O data samples. Extensive evaluations show that ChainCoder outperforms state-of-the-arts, demonstrating that our progressive generation eases the reasoning procedure and guides the language model to generate higher-quality solutions. Our codes are available at: `https://github.com/VITA-Group/ChainCoder`.

1. Introduction

The goal of automatic program synthesis has been established and extensively researched for decades (Waldinger & Lee, 1969; Backus et al., 1957). Recently, with the rapid advancements of Large Language Models (LLMs), a surge of methods leverage transformers to model programs as sequences, and have demonstrated prospects to automatically analyze, annotate, translate, or synthesize code (Li et al., 2022; Austin et al., 2021; Fried et al., 2022; Chen et al., 2021a; Kanade et al., 2020; Feng et al., 2020; Clement et al., 2020; Svyatkovskiy et al., 2020; Wang et al., 2021). These LLMs are trained on a large corpus of code so as to imbibe syntactic and semantic understanding into model weights, and develop logical reasoning for program synthesis.

Though current LLMs have gained success in code understanding and generation, their inherent design, as well as the training/finetuning techniques, are largely borrowed from the *natural language* modeling. This limits their potential in code generation modeling due to two reasons.

First, for a given problem specification, LLMs generate end-to-end code solutions autoregressively in a single pass regardless of the logical complexity involved. Even from a programmer's perspective, it is difficult to write good code in a single shot without laying out logical and hierarchical thinking. In other words, composing code involves molding an overall "warm-up" proceeded by lower-level atomic algorithms and variable definitions[1].

Second, LLMs tokenize the code using tokenizers meant for everyday-natural-language strings and disregard syntactical awareness as structural priors (Chen et al., 2022; Hendrycks et al., 2021; Li et al., 2022; Chen et al., 2021a; Austin et al., 2021; Athiwaratkun et al., 2022; Nijkamp et al., 2022; Xu et al., 2022; Wang et al., 2022; Chen et al.; Li et al., 2023; Mao et al., 2022). Notably, programming languages contain complex rules and conform to more rigorous layouts compared to their natural language counterparts (Casalnuovo et al., 2019; Naur, 1975). They have fewer choices of synonyms, stringent syntax requirements, and diverse control flows (sequential statement execution, selection/branching, and repetition such as for and while loops).

In our ChainCoder, we investigate if addressing the aforementioned flaws is indeed promising. As shown in Figure 1, we aim to remodel the code generation that previous works treat as a no-warm-up, single-shot textual sequence comple-

[1]VITA Group, The University of Texas At Austin, Austin, TX, US. Correspondence to: Wenqing Zheng, Zhangyang Wang <w.zheng@utexas.edu, atlaswang@utexas.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[1]This is usually termed as a *top-down practice* in programming: beginning with creating an outline or high-level view of the program, and then successively breaking it down into smaller and smaller components until the entire program is complete.

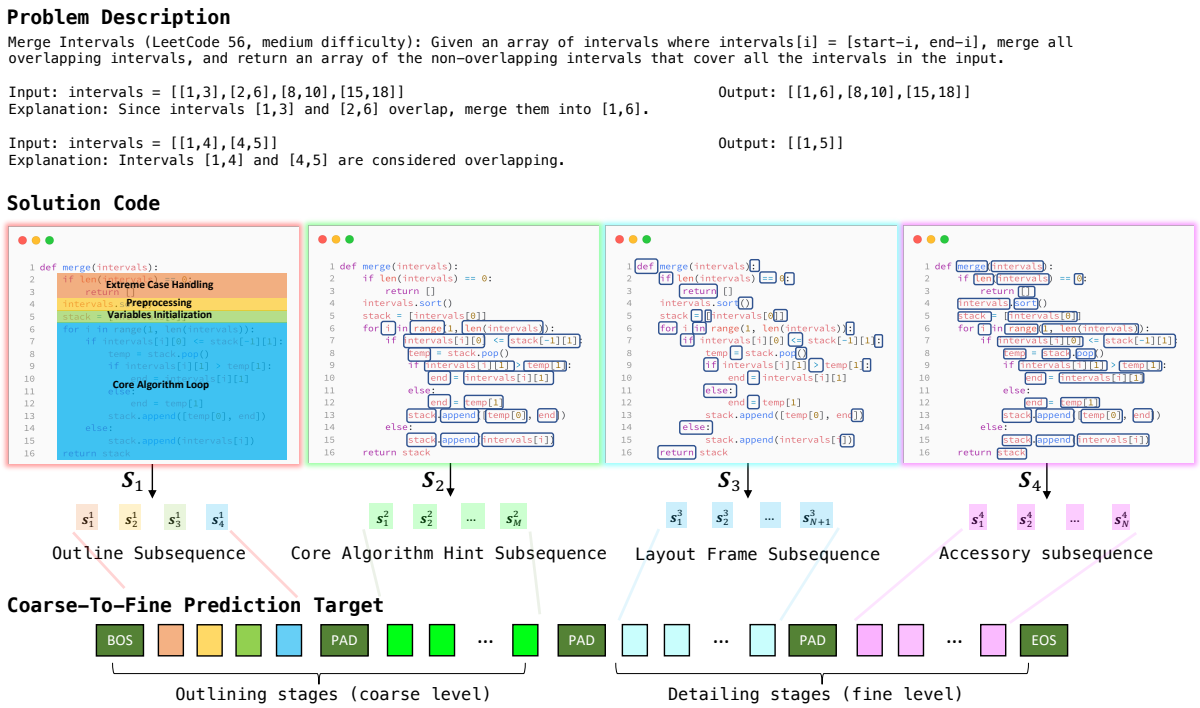


Figure 1. Our prediction target formulation illustrated with an example. In the *Merge Intervals* problem (one medium difficulty problem on leetcode), the solution requires first tackling extreme cases, initializing variables, then entering the main algorithm loop. A programmer needs to first come up with this outline, then think deeper on how to implement the algorithm loop. They may then think about using a stack to keep track of the biggest end time and to decide whether to close or extend the current interval. Finally, after being clear with these ideas, they may write the formal answer with carefulness on both syntax and variable names. This reasoning procedure happens from coarse to fine. We therefore construct the prediction target into four subsequences. Note that the boxed substrings only roughly indicate the tokens, while our actual tokens take the forms of the grouped nodes in the syntax tree.

tion task into a multi-step intermediate reasoning strategy through syntactical decomposition.

In a similar context of harnessing multi-step reasoning capability of LLMs (on natural language), recent works discover that chain-of-thought (CoT) prompting (Zhang et al., 2022; Wei et al., 2022; Shi et al., 2022) significantly boosts performance. CoT prompting advances LLMs to naturally develop a series of intermediate reasoning steps before providing the final answer through exemplar demonstrations. This exposes a profound potential of refactoring LLMs to explicitly perform **coherent intermediate step-wise reasoning**, rather than directly concluding final solutions.

Inspired by the success of chain-of-thought prompting for LLMs, we design a novel approach that follows a *step-wise*, **outline-then-detail** thought process. We decompose the source code into a predefined hierarchy of code logic, and prepare the LLM generation target in each level ($\boldsymbol{S}_{1-4}$ in Figure 1) so that the model predicts them in multiple passes separately. We refer to our model as ChainCoder, and such hierarchical prediction as "*coarse-to-fine*" generation, similar to Dong & Lapata (2018). Specifically, ChainCoder first builds a logical skeleton of the code, then move to step-wise implementation of lower-level atomic details/algorithms. This is inherently different from vanilla CoT prompting, as CoT methods require external prompting and are mainly developed in the context of natural language reasoning, while ChainCoder is the first program synthesis model that progressively generates code by itself.

To build such a multi-level hierarchical representation, appropriate disentanglement of the code is required. To this end, we tailor the tokenization step to better display complex programming language rules. We parse the code into the Abstract Syntax Tree (AST), which naturally exposes extensive domain knowledge and structures expressed in the tree nodes and edges. We then disentangle the syntax tree into two components: the *layout frame component*, which tells more clause type and other syntactical information, and the *accessory component*, which includes concrete variable and function names, etc. The syntax tree-based tokenizer brings advantages in two-fold: its domain knowledge eases the comprehension of source code, and would possibly boost

the syntactical correctness. Unlike previous research on applying syntax aware tokenizer (Jimenez et al., 2018) or simplification tricks during the tokenization step (Gupta et al., 2017; Chirkova & Troshin, 2020), ChainCoder is the first approach to leverage syntax knowledge to disentangle the code and build hierarchical prediction targets.

In brief, our work aims to reformulate program synthesis from a sequential text completion task into a multi-pass coarse-to-fine refinement strategy. As depicted in Figure 1, this comprises of four progressive steps – outlining, core algorithm hinting, layout framing, and accessory subsequencing inspired from the "warm-up" thought process of programmers. We then leverage a tailored transformer architecture that better encodes the syntactically aligned I/O data to ease comprehension of problem specifications. To the best of our knowledge, ChainCoder is the first large language model to perform multi-pass *coarse-to-fine* code generation to reflect intermediate warm-up procedure, in contrary to existing single-shot sequential completion works. In summary, our technical contributions are as follows:

- Inspired by the chain-of-thought process, we leverage the unique syntax hierarchy to build the multi-pass coarse-to-fine code generation framework. ChainCoder is the first program synthesis approach that harnesses progressive generation to elicit the step-wise reasoning capability of LLMs and improve accuracy and syntactic coherency.
- To enable progressive generation, we propose a syntax-aware tokenizer that neatly disentangles the code and outputs multi-level prediction targets accordingly. We further develop a transformer to better leverage the structure of the syntactically aligned data.
- Evaluations on the competition-level datasets show that ChainCoder performs better than other state-of-the-art models even with smaller model sizes. Ablation studies verified the effectiveness of the coarse-to-fine guidance and other design choices.

2. Related Works

Synthesizing program from the description and I/O pairs (Chen et al., 2018; Bunel et al., 2018; Devlin et al., 2017; Gulwani et al., 2012; Fox et al., 2018; Ganin et al., 2018; Chen et al., 2019; Hong et al., 2021; Le et al., 2022) or other modality inputs (Sun et al., 2018; Liu & Wu, 2019; Tian et al., 2019) is a well-received benchmark, yet is challenging due to the indefinite program space.

The classical approaches to program synthesis date back to *rule-based program synthesis* which use formal grammar to derive programs from well-defined specifications (Waldinger & Lee, 1969; Manna & Waldinger, 1971; 1980). Later on, symbolic and neuro-symbolic techniques (Balog et al., 2017; Odena & Sutton, 2019; Ellis et al., 2018; 2020; Devlin et al., 2017; Panchekha et al., 2015) have also been explored to generate code with higher quality. However, these methods are mainly applied to domain-specific languages (DSLs), which limits the applicability in more advanced programming languages.

Program Synthesis With LLMs. Recently, there has been a huge surge in exploiting language models for program synthesis with extension to general-purpose programming languages (Austin et al., 2021; Hendrycks et al., 2021; Chen et al., 2021b; Clement et al., 2020; Wang et al., 2021). These models are trained on massive codes and natural language datasets, and learn to condition the code on the natural language descriptions or existing code fragments. For example, Devlin et al. (2017) uses an encoder-decoder network formulating synthesis process as a sequence generation problem. CodeT5 (Wang et al., 2021) prominently focuses on understanding tasks such as code defect detection, translation, and clone detection. Codex (Chen et al., 2021b) uses GPT-3 architecture, evaluating its synthesis performance on a new benchmark of simple programming problems. CodeBERT (Feng et al., 2020) is pre-trained for natural language and programming languages like Python, Java, JavaScript, etc., and captures the semantic connection between natural language and programming language.

More recent state-of-the-art works such as APPS (Hendrycks et al., 2021) and AlphaCode (Li et al., 2022) have shown promising results over competition-level problems. Another related program synthesis framework InCoder (Fried et al., 2022) is able to refine the code via infilling under a bidirectional context. Similarly, Parsel (Zelikman et al., 2022) generates programs by combining hierarchical generations of sub-programs during interactions between the LLM and an external environment. However, these methods still adopt natural language tokenizers that overlook the special syntax structures of the programs and do not execute the necessary warm-up before generation.

Multi-Step Reasoning In Language Modeling. Recent years have witnessed a shift of LLM paradigm from "pre-train, fine-tune" procedure into "pre-train, prompt, and predict" (Liu et al., 2021). Among the prompting methods, the chain-of-thought (CoT) has become the research highlight (Zhang et al., 2022; Wei et al., 2022; Shi et al., 2022). Chain of thought refers to techniques that provide intermediate reasoning steps as prompting demonstrations for language models (Zhang et al., 2022).

The CoT methods allow LLMs to decompose problems into intermediate steps. They provide an interpretable window into the model behaviors to ease the debugging, and can be readily elicited by adding examples in the few-shot prompting (Wei et al., 2022). They have also been demonstrated

to improve model performance in-situ for multiple choice question answering tasks (Lewkowycz et al., 2022), and to better perform in multi-step reasoning task such as the math word problems (Chowdhery et al., 2022).

Generating Programs Hierarchically From Abstractions Recently, a few papers aims to improve the scalability by generating certain forms of sketches first, then convert to full operational programs. Specifically, Murali et al. (2017) starts with generating sketches that leave out names and operations. They then infer a posterior distribution over sketches, from which to samples type-safe programs using combinatorial techniques. Nye et al. (2019) follows previous strategies to break code generation into two steps: to generate sketches and to fill full in them. Previous methods assign the generation step with pattern recognition or perform symbolic search (Zheng et al., 2022a;b) in a fixed way. This paper proposed a novel algorithm that learns when to switch between the two ways without direct supervision. Brockschmidt et al. (2018) uses a graph to represent the intermediate state of the generated output. It generates code by interleaving grammar-driven expansion steps with graph augmentation and neural message-passing steps. Similarly, Zhong et al. (2023) separately learned two stages: (1) a high-level module to comprehend the task specification from long programs into task embeddings, and (2) a program decoder that can decode a task embedding into a program.

In comparison, ChainCoder uses a unified space to represent the middle step sketches and the final representations, instead of specifically curating different forms of representations. On the other hand, the entire decoding mechanism of ChainCoder is learned by a single LLM, and the results are also directly sampled from this LLM. The decomposition of the program into syntax and components is explicitly done via a novel tokenizer, also in the form of a unified string.

3. Coarse-To-Fine Programming Language Modeling With ChainCoder

This section discusses how ChainCoder tokenizes and predicts code in a hierarchical fashion. In brief, the ChainCoder first lays out a coarse-level outline of the code, then focuses on fine-grained details. The coarse-level layout can be thought of as the summarized representation of constituent fine-level parts. In order to better leverage code structure and ease learning, our tokenization does not base on code as raw text but rather as a specialized data structure parsed from source code – namely, the abstract syntax tree (AST). In the following, we first discuss the preliminaries for AST parsing in Section 3.1, and formulate the tokenization algorithm in Section 3.2. We then discuss the construction of multi-step prediction targets in Section 3.3 and finally describe ChainCoder's model architecture in Section 3.4.

3.1. Preliminaries of Parsing and Abstract Syntax Tree

Codes are written by humans, but are executed by computers. Therefore, humans write them in a form that they can understand, then the software transforms them in a way that can be used by the computer. Such transformation is referred to as parsing, which determines a structured template from the source code text. After parsing, the programming language is converted to a better-formatted data structure, usually a tree-type format called the abstract syntax tree.

Parsing the code into the tree-type format is common to almost all programming languages. Specifically in Python, the leaf nodes of the abstract syntax tree are the function/class/-variable/operator names and constant values. The branching nodes of the tree represent syntactic relationships between the leaf nodes. For example, consider the line of code `x = 0`. Its syntax tree is roughly a tree with three nodes, with the root node being the *assignment*, and two leaf nodes being `x` and `0`. The root node conveys syntax information ("="), which tells that this line is an "assignment" type sequential statement, there will be two placeholders as its children nodes, and one child node will assign its value to the other child node. The precise syntax tree is more complex though, which we show in Figure 7 and Figure 8 in the Appendix. The abstract syntax tree has better-formatted structures than the raw text, but is overlooked by previous LLM research on program synthesis. Therefore, we seek to ease the learning of the LLM by tokenizing based on the parsed tree rather than raw code text.

3.2. Syntax-Aware Tokenization: Layout Frame and Accessory Components

The code contains complex structures both syntactically and semantically. Writing algorithmic code is generally difficult, partially due to the fact that both syntactic and semantic information need to be handled simultaneously. To ease this difficulty, we propose to disentangle the code into a component that conveys more syntax information and another component that conveys more semantic information and generate them separately. We refer to these two components as the *layout frame subsequence* and the *accessory subsequence*. The disentanglement procedure is based on the syntax tree representation, as it enables more flexible treatments of the code in the domain of tree nodes and edges.

The layout frame subsequence is responsible for indicating the clause type, assigning placeholders in the sequel, and maintaining syntactic correctness. On the other hand, the accessory subsequence contains the concrete variable/class/-function/operator names, and constant values. In the above example of `x = 0`, the syntax subtree of this line of code could be unambiguously decoupled into two layout frame tokens and two accessory tokens. First, the two accessory tokens are easily identified as the leaf nodes, `x` and `0`. Then,

the first layout frame token could be the root node *assignment (=)* with the corresponding edge between it and the leaf node `x`, while the second layout frame token could be the edge between the root node and the other leaf node `0`. When these tokens are interleaved together, they become the serialized representation of the syntax tree.

The training procedure of ChainCoder heavily relies on an *encoding* step, which converts the source code into decoupled layout frame and accessory subsequences. In the encoding step, the code is first parsed into an AST, which is then partitioned into nodes and edges. These nodes and edges are then re-grouped to generate these two subsequences. The resulting subsequences are used for training and inference. On the other hand, we use the term *decoding* to refer to the reverse procedure. Given the decoupled subsequences, the decoding procedure recovers the original source code. It is noteworthy that the encoded two subsequences will be less human-interpretable than the original code, yet they convey precise information to the Python interpreter.

As revealed in Figure 1, the layout frame and the accessory subsequences constitute the fine level part in the coarse-to-fine hierarchy, and are referred to as $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$. They contain complete information to reconstruct the original code by themselves. There are other parts of coarse level subsequences ($\boldsymbol{S}_1$ and $\boldsymbol{S}_2$), which we discuss in Section 3.3. Next we discuss the encoding and decoding steps in detail.

Encoding Step. The encoding step is used to preprocess the training samples to construct the prediction targets. Prior to training, each code sample in the training set is converted to the coarse-to-fine hierarchical subsequences, and the ChainCoder is trained to generate these new subsequences.

The encoding step first parses the source code into the abstract syntax tree, then disentangles the serialized syntax tree into the layout frame subsequence ($\boldsymbol{S}_3$) and the accessory subsequence ($\boldsymbol{S}_4$). It constructs the other two coarse-level subsequences at the same time. The encoding algorithm of ChainCoder is displayed in Algorithm 1.

In general, the encoding step is completed during the depth-first pre-order traversal for the parsed syntax tree. Starting from the root node, it first visits the current node, then visits all the children nodes from left to right. Every time it meets a leaf node, the traversal pauses, and it groups all branching nodes seen so far into a token, and appends the grouped branching nodes to the layout frame subsequence. It then appends the leaf node into the accessory subsequence. By grouping the branching nodes, the algorithm efficiently compresses token sequence length. The following relationship always satisfies for the tokenized subsequences: `layout frame subsequence length = accessory subsequence length + 1`, where the "`+1`" comes from the ending brackets in the layout frame subsequence after the last accessory token is appended.

Algorithm 1 The Encoding Step Of ChainCoder

Require: Source code
Ensure: Four subsequences: outline $\boldsymbol{S}_1$, core algorithm hint $\boldsymbol{S}_2$, layout frame $\boldsymbol{S}_3$, accessory $\boldsymbol{S}_4$

Parse the source code into syntax tree ϕ
Initialize $\boldsymbol{S}_1, \boldsymbol{S}_2, \boldsymbol{S}_3, \boldsymbol{S}_4$ with empty sequences
Branching nodes group $t \leftarrow$ empty string
while Traversal not finished **do**
  Get next node $\boldsymbol{\xi}$ using pre-order traversal for ϕ
  if $\boldsymbol{\xi}$ is branching node **then**
    Group with previous branching nodes: $t \leftarrow t + \boldsymbol{\xi}$
  else if $\boldsymbol{\xi}$ is leaf node **then**
    Append grouped branching nodes: $\boldsymbol{S}_3 \leftarrow \boldsymbol{S}_3 + t$
    Append leaf node: $\boldsymbol{S}_4 \leftarrow \boldsymbol{S}_4 + \boldsymbol{\xi}$
    if $\boldsymbol{\xi}$ is within a loop or user-defined function **then**
      Append packed branching nodes: $\boldsymbol{S}_2 \leftarrow \boldsymbol{S}_2 + t$
    end if
    if $\boldsymbol{\xi}$ is the first leaf node in the current line and line indent level is 1 **then**
      Add branching nodes to outline: $\boldsymbol{S}_1 \leftarrow \boldsymbol{S}_1 + t$
    end if
    $t \leftarrow$ empty string
  else if $\boldsymbol{\xi}$ is the starting token of a new line **then**
    Remove the doc-strings and comments, if any
    for All variable names v_i in this line **do**
      if v_i is not linked to any imported name, and is not a built-in name **then**
        Replace v_i in $\boldsymbol{S}_4$ with a name pool candidate
      end if
    end for
  end if
end while

The encoding algorithm offers two optional ways to further simplify the generation task: the *name replacement* and the *doc-strings/comments removal*. The name replacement feature detects all user-defined names in the parsed tree, and replaces them from a name pool. In this way, the code logic is maintained, while the extra learning burden caused by naming style differences can be avoided. The doc-strings/comments removal also naturally takes advantage of tree-based representation, they reduce the sequence length without hurting the functionality.

By parsing and encoding, the program space is reduced and the learning task is simplified. On one hand, thanks to the degenerated variable names, the vocabulary could be shrinked while still leading to clearer meanings. For example, if there are two variables called `student_num` and `student_nums` in one program, ChainCoder might tokenize them into two distinct words `user_defined_var_7`

and `user_defined_var_8` drawn from the name pool, rather than using the original names that might cause confusion. The name pool size is also much smaller than the natural language vocabulary which includes all possible sub-words of variable names. On the other hand, the token sequence could be shortened by removing the doc-strings and comments without hurting the functionality. Some sentences that would be tokenized into long sequences by natural language tokenizer can now be represented very concisely in the syntax tree domain, such as `for i in range(num)`, `res = a if x is None else b`. What's more, the names of variables/functions/classes/etc are never broken into multiple sub-words in the syntax tree representation.

Decoding Step. The decoding step is used during the inference time. As ChainCoder generates the hierarchical subsequences rather than the source code, this step is applied to convert the resulting subsequences back to the human-readable source code.

Given the design of the encoded subsequences, the decoding algorithm is straightforward. We first interleave the tokens in $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$ and glue them together, so that the serialized syntax tree is obtained. Then, an abstract syntax tree unparse tool is used to convert the syntax tree into the source code.

3.3. Multi-Level Coarse-To-Fine Prediction Targets

As visualized in Figure 1, the prediction target sequence is composed of four components: the outline ($\boldsymbol{S}_1$), the core algorithm structure hint ($\boldsymbol{S}_2$), the layout frame ($\boldsymbol{S}_3$) and the accessory ($\boldsymbol{S}_4$) subsequences. These four subsequences are concatenated by the system-level `<PAD>` token.

In the coarse outline $\boldsymbol{S}_1$, each token essentially corresponds to a line of code with the smallest indent. For each of these lines, one layout frame token is selected, which is the first token, the one that contains the root node of the subtree of this line. The $\boldsymbol{S}_1$ is a discontinuous subset of the entire layout frame subsequence $\boldsymbol{S}_3$.

The core algorithm structure hint $\boldsymbol{S}_2$ is designed to warm-up the model before generating the final solution ($\boldsymbol{S}_3$ and $\boldsymbol{S}_4$). Since most algorithms contain loops, and the core functional parts are usually implemented in the loops or user-defined functions, we identify the implementations within the loop or user-defined functions as the core algorithms and use their layout frame tokens to construct $\boldsymbol{S}_2$. This part is a continuous subset of $\boldsymbol{S}_4$.

The construction of the last two subsequences $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$ has been discussed in Section 3.2. With these subsequences, ChainCoder breaks the code generation task from a single pass generation into four turns. In each turn, ChainCoder focuses on a simpler subtask, i.e., only generates a specific subsequence.

The functionalities of $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ are analogous to the chain-of-thought prompting (Zhang et al., 2022; Wei et al., 2022; Shi et al., 2022), except that they are generated by the model itself rather than external prompting. They only exist to hint the model generation procedure. When the entire solution has been generated, they will be removed before checking the correctness of this answer. Such design follows the simple philosophy: an outline of the algorithm shall be built before writing down the concrete answer, and a layout frame shall be constructed before the detailed variable names / constants / etc are filled in.

We adopt the same autoregressive generation recipe as other LLMs, which asks ChainCoder to predict the next token given all previously generated tokens. The inference procedure is also the same: ChainCoder completes the target sequence token by token until a `<EOS>` token is generated. As $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ need not to be complete, we apply token dropout to let the model occasionally generate $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$ with information in $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ missed out. We achieve this by randomly dropping tokens in $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ with probabilities 0.05 and 0.2, respectively.

3.4. The Model Architecture of ChainCoder

The inputs of program synthesis contain the problem description in natural language and a group of I/O data. Thanks to the syntax tree parsing, the I/O data could be well-aligned across samples based on the values' syntax roles. After alignment and padding, the tokens of I/O data are arranged into a regular matrix with both sample and token dimensions. The natural language descriptions can be well handled by LLMs such as BERT. However, the one-dimensional sequence modeling of LLMs fails to fully leverage the structures of the aligned I/O data. We accommodate this gap with a new transformer architecture, which has a sample embedder submodule that attends to the regular matrix-shaped I/O data. The overall model architecture consists of four components: a sample embedder, a description embedder, a token encoder, and a program decoder. The architecture is shown in Figure 2.

Sample Embedder. With the syntax-based encoding, alignment, and padding, the I/O data in each instance is now represented as a matrix $W \in \mathbb{R}^{N_{\text{sample}} \times N_{\text{token}} \times E}$, where N_{token} is the maximum number of tokens across different I/O data samples, N_{sample} is the number of samples for this instance, and E represents the token embedding dimension. The sample embedder processes the matrix-shaped aligned I/O data. To track the locations of tokens in the matrix, the tokens are added with two sets of positional embeddings along the token and sample dimensions. At the output side of the sample embedder, it uses first element pooling to reduce the embedding size of each instance to $N_{\text{token}} \times E$.

Description Embedder. Each instance in the I/O data cor-

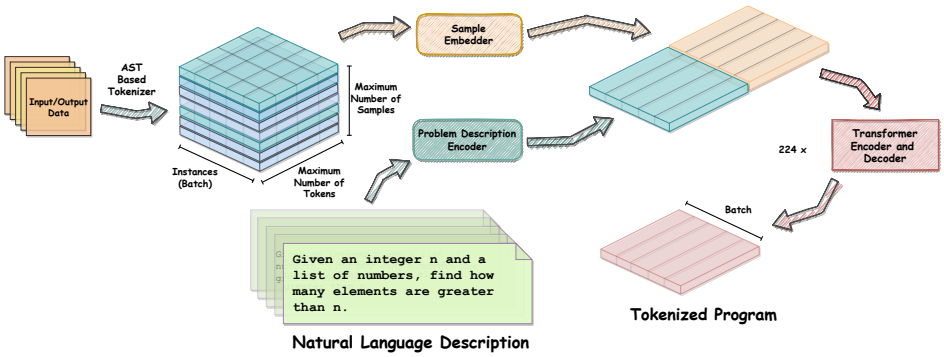


Figure 2. Overall Model Architecture of ChainCoder, which consists of the sample embedder, the problem description embedder, the traditional transformer encoder, and decoder.

responds to a natural language problem description. We utilize the BERT model (Devlin et al., 2018) to perform natural language understanding, and distill its outputs into four tokens: the first token, the last token, the minimum pooling token, and the maximum pooling token. The output dimensions for one instance is $4 \times E$.

Token Encoder, Program Decoder And Training Loss. From the sample embedder and the description embedder, we get the embeddings for the I/O data and the problem descriptions separately. We concatenate these embeddings and get a sequence with dimensions $(N_{\text{token}} + 4) \times E$. We then follow the traditional transformer encoder and decoder layers and training recipes. At the output of the decoder, we ask the model to predict the coarse-to-fine subsequences described in Section 3.3 using cross-entropy loss.

4. Experimental Results

In our work, we target improving the performance of language models on code generation tasks particularly on competition-level problem-solving. Following previous works in this domain (Hendrycks et al., 2021; Li et al., 2022; Chen et al., 2021a; Austin et al., 2021), we leverage the CodeParrot GitHub-Code (CodeParrot, 2022) dataset for model-pretraining on general purpose source code. We then fine-tune on the training sets of competitive programming benchmarks and evaluate on their test sets respectively. To prevent the data leakage, we execute extra carefulness and strictly followed the datasets as well as the processing steps of AlphaCode and CodeRL (Le et al., 2022).

Datasets and Experimental Overview. The CodeParrot GitHub-Code pre-training dataset contains 7.2 million Python files with a total size of 52.03 GB. These pre-training codes do not contain corresponding I/O data pairs, and many also lack natural language descriptions of their functionalities. In order to approach the absence of I/O data, our sample-embedder submodule output is initially neglected and is only updated during the fine-tuning phase. Meanwhile, to combat the lack of natural language descriptions, we employ a pre-trained CodeT5-based (Wang et al., 2021) Python code explanation model to generate natural language descriptions for each code sample. The natural-language-embedder-submodule is a pre-trained BERT (Devlin et al., 2018) and is fixed during the pre-training phase, and optimized during fine-tuning.

Sequence length	APPS (Hendrycks et al., 2021)			Contests (Li et al., 2022)		
	ChainCoder	BERT	Raw String	ChainCoder	BERT	Raw String
Mean	155.808	176.312	416.419	145.676	176.509	432.566
Median	113.0	124.0	278.0	111.0	124.0	289.0
Std.	272.911	261.470	618.100	82.547	292.139	780.052

Table 1. The tokenization sequence length statistics. The syntax-aware tokenizer sequence length is calculated by the summation of layout frame and accessory subsequences. The ChainCoder tokenizer leads to shorder sequence thanks to the domain knowledge built-in to syntax tree.

For the fine-tuning and evaluation, we choose two carefully curated datasets, CodeContests (Li et al., 2022) and APPS (Hendrycks et al., 2021), which are both massive and contain problems of varied difficulties (see Table 2). The CodeContests dataset comprises of data scraped from Codeforces along with existing data from Description2Code (Caballero et al., 2016). Meanwhile, APPS is scraped from of Codewars, AtCoder, Kattis, and Codeforces. After pre-trained on the GitHub codes, we use two copies of the model to fine-tune on the training sets of CodeContests and APPS, and evaluate on their test sets respectively.

Tokenization. Before pre-training, we sweep across the APPS and CodeContests datasets to collect *vocabulary*, yielding $212,892$ distinct layout frame tokens. These tokens follow a long tail distribution as illustrated in Figure 3. Our tokenization step filters out some code files in the dataset that yield too long or incompatible sequences. After our tokenization step, we obtain $81,339$ code files from APPS,

and $1,337,655$ code files from CodeContests.

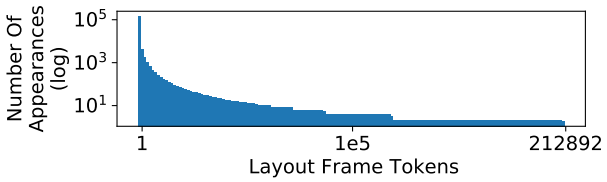


Figure 3. Token frequency distribution. Visibly, the majority of tokens are rarely met. From our inspection, these highly infrequent tokens are not necessarily tied to the code solutions but rather just happened to occur randomly. Therefore, it is safe to assume that ChainCoder can be trained on the more-frequent end of the distribution alone without facing any performance losses.

Architectural Details and Metrics. ChainCoder contains two sample-embedder transformer blocks, two transformer encoder blocks, 224 transformer decoder blocks, with the 512 hidden dimensions for these submodules, yielding 1.09 billion parameters in total. The natural-language-embedder inherits weights of the pre-trained BERT model, while all other submodules of ChainCoder are trained from scratch. We diversify the learning difficulty during the fine-tuning phase by periodically changing the number of I/O data pairs (sweeping between 1 and 32, with a 15 epoch period) and the number of programs that the model predicts (sweeping between 1 and 8, with a 37 epoch period). At inference time, we set our beam-search width to 5. We adopt the same evaluation metric as AlphaCode (Li et al., 2022): the `n@k`, which measures the fraction of problems the model can solve when allowed to generate k solutions but only submit the best n of them for evaluation. The results are shown in Table 2, and the ablation study results are shown in the bottom section of Table 2. We make the following observations.

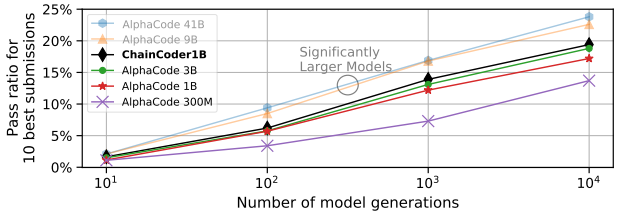


Figure 4. Results on CodeContests dataset compared with AlphaCode ChainCoder with 1 billion parameters outperforms AlphaCode with similar sizes.

ChainCoder Achieves Better Results. As seen in Table 2 and Figure 4, ChainCoder reaches state-of-the-art performance. ChainCoder with one billion parameters outperforms the AlphaCode with similar size, and slightly outperforms AlphaCode with three billion parameters. With regard to the syntax-error-free rate, ChainCoder also stays competitive. As shown in Table 2, ChainCoder generated codes showcase impressive syntax pass rates on the APPS test set. This implies that the disentanglement of the layout frame and the accessory helps the model to better learn the syntax structure.

ChainCoder Tokenizer Leads To Shorter Sequences Than Natural Language Tokenizer. Shorter sequence lengths intuitively points towards easier optimization and prediction for LLMs. We collect the sequence lengths of different tokenizers in Table 1. The sequence length calculation of the ChainCoder tokenizer takes into account the components with complete information: `len(`$\boldsymbol{S}_3$`) + len(`$\boldsymbol{S}_4$`)`. We found that the ChainCoder's tokenizer encodes the program into shorter sequences, compared with the BERT tokenizer. For example, in the APPS dataset, the average sequence length is 78.404 layout frame tokens + 77.404 accessory tokens (155.808 in total), whereas BERT tokenizer on average generates 176.312 tokens. Though the sequence length will become longer if adding the coarse level $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ subsequences together, they do not add new prediction difficulty as they are subsets of $\boldsymbol{S}_3$.

Syntax Aware Tokenization And I/O Aligned Sample Embedder Both Improve Model Performance. In this ablation study, we design ablation studies to justify the contributions of the design choices used in the tokenization step. Specifically, we testify the coarse-to-fine syntax aware tokenization, the sample embedder with I/O data cross-sample alignment, the variable name replacement, the title summation based descriptor pre-training, and show the results in Table 2. The *ChainCoder with GPT code tokenizer* uses the natural language tokenizer for codes at the model output side, while the model input side uses the same tokenizer as the ChainCoder. The second variant, *ChainCoder I/O not aligned* refers to the ablation study for the matrix-shaped aligned I/O data processed by the sample embedder. In this case, the I/O data is not aligned; instead, it is tokenized into a one-dimensional sequence with a natural language tokenizer. This sequence is then passed into the embedder as usual but with only one set of positional encodings. The third variant *ChainCoder no var-name replacement* does not apply the variable name replacement step. The fourth variant *ChainCoder no destriptor during pre-train* do not feed input natural language summation to the model during the pre-training stage.

As can be seen from the results, *ChainCoder with GPT code tokenizer* significantly underperforms the original ChainCoder. And when the I/O data alignment is dropped, the model also degrades. These results verify the contributions of the core model designs, i.e., the syntax-aware tokenization and the induced I/O data cross-sample alignment. When no variable name replacement is applied, the performance

	Filtered From(k)	Attempts (n)	Introductory		Interview		Competition	
			n@k	syntax-error-free	n@k	syntax-error-free	n@k	syntax-error-free
GPT-3 few shot	N/A	1	0.20%	31.0%	0.03%	42.0%	0.00%	40.0%
GPT-Neo 2.7B	N/A	1	3.90%	87.9%	0.57%	87.9%	0.00%	85.0%
GPT-Neo 2.7B	N/A	5	5.50%	97.4%	0.80%	96.0%	0.00%	95.4%
AlphaCode 1B	1,000	5	14.36%	N/A	5.63%	N/A	4.58%	N/A
ChainCoder 1B	1,000	5	**17.52%**	**100%**	**7.36%**	**100%**	**5.48%**	**100%**
ChainCoder I/O not aligned	1,000	5	14.85%	**100%**	5.90%	**100%**	4.82%	**100%**
ChainCoder with GPT code tokenizer	1,000	5	14.80%	**100%**	4.55%	**100%**	3.46%	**100%**
ChainCoder no var-name replacement	1,000%	5	17.40%	**100%**	7.25%	**100%**	5.32%	**100%**
ChainCoder no destriptor during pre-train	1,000%	5	15.34%	**100%**	6.29%	**100%**	3.70%	**100%**
ChainCoder tokenizer no warm-up	1,000	5	16.10%	**100%**	5.30%	**100%**	4.22%	**100%**
ChainCoder tokenizer no $\boldsymbol{S}_1$	1,000%	5	16.50%	**100%**	5.80%	**100%**	5.10%	**100%**
ChainCoder tokenizer no $\boldsymbol{S}_2$	1,000%	5	16.90%	**100%**	6.20%	**100%**	5.22%	**100%**
ChainCoder $\boldsymbol{S}_3$/$\boldsymbol{S}_4$ interleaved	1,000	5	16.75%	**100%**	6.80%	**100%**	5.10%	**100%**

Table 2. The results of APPS (Hendrycks et al., 2021), AlphaCode (Li et al., 2022) and ChainCoder on the APPS test set, fine-tuned on the APPS training set.

is slightly worse, which suggests that in our case where the model learns to predict syntax format decoupled from the raw string text, learning the logical relationship between the variables are enough, and adding the semantic meaning of the variable names do not add performance to it. The performance for no descriptor is also unstable, which suggests the need for natural language from the pre-training stage.

Coarse-To-Fine Generation Leads To Better Performance. In this ablation study, we replace our multi-step coarse-to-fine strategy with a single-shot final answer generation scheme. In other words, the ChainCoder only predicts steps $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$, and omits $\boldsymbol{S}_1$ and/or $\boldsymbol{S}_2$. The result is represented as *ChainCoder tokenizer no warm-up* (it has neither $\boldsymbol{S}_1$ nor $\boldsymbol{S}_2$), *ChainCoder tokenizer no* $\boldsymbol{S}_1$ and *ChainCoder tokenizer no* $\boldsymbol{S}_2$ in Table 2. As shown in the results, once the outline subsequence $\boldsymbol{S}_1$ and/or the core algorithm hint subsequence $\boldsymbol{S}_2$ are dropped, ChainCoder's performance takes a hit. This validates our assumption that warm-up with certain coarse information before generating the final results helps smooth the reasoning chain and improve the performance. The performance drop is especially prominent for more difficult benchmarks (the competition level benchmark), which further implies that difficult problems need more reasoning warm-up.

Disentangling Layout Frame And Accessory Leads To Better Performance. In this case, we compare two settings but use the same tokenizer. The first setting is the original one, where $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, $\boldsymbol{S}_3$, $\boldsymbol{S}_4$ are generated in four rounds, i.e., the prediction target is $[\cdots, <\text{PAD}>, \boldsymbol{s}_1^3, \boldsymbol{s}_2^3, \cdots, \boldsymbol{s}_{N+1}^3, <\text{PAD}>, \boldsymbol{s}_1^4, \boldsymbol{s}_2^4, \cdots, \boldsymbol{s}_N^4, <\text{EOS}>]$. For the second setting shown in *ChainCoder* $\boldsymbol{S}_3$*/*$\boldsymbol{S}_4$ *interleaved* in the bottom section of Table 2, the model is trained to generate a different sequence, where $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$ interleaved together, just as originally shown in the syntax tree, i.e., the prediction target becomes $[\cdots, <\text{PAD}>, \boldsymbol{s}_1^3, \boldsymbol{s}_1^4, \boldsymbol{s}_2^3, \boldsymbol{s}_2^4, \cdots, \boldsymbol{s}_N^3, \boldsymbol{s}_N^4, \boldsymbol{s}_{N+1}^3, <\text{EOS}>]$. Comparing the results, we can see that separate generation performs better than interleaved version, validating the benefits of generating the layout frame and accessory separately.

5. Conclusions

In this work, we propose ChainCoder, a program synthesis language model that generates Python code progressively in multiple passes. The ChainCoder decouples the code snippet into the layout frame component and accessory component and uses them to construct a step-by-step, coarse-to-fine representation of the code as the prediction target. We leverage a novel transformer architecture to encode the syntactically aligned samples. Extensive evaluations showed that ChainCoder outperforms state-of-the-art code generation models, which verifies that generating related information before the final results as a warm-up helps to improve the language model in difficult code generation tasks.

Acknowledgement

Z. Wang is in part supported by US Army Research Office Young Investigator Award W911NF2010240 and the NSF AI Institute for Foundations of Machine Learning (IFML).

References

Athiwaratkun, B., Gouda, S. K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W. U., Wang, S., Sun, Q., Shang, M., et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western*

joint computer conference: Techniques for reliability, pp. 188–198, 1957.

Balog, M., Gaunt, A., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR 2017)*. OpenReview. net, 2017.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Caballero, E., OpenAI, ., and Sutskever, I. Description2Code Dataset, 8 2016. URL https://github.com/ethancaballero/description2code.

Casalnuovo, C., Sagae, K., and Devanbu, P. Studying the difference between natural and programming language corpora. *Empirical Software Engineering*, 24(4):1823–1868, 2019.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Chen, T., Zhang, Z., JAISWAL, A. K., Liu, S., and Wang, Z. Sparse moe as the new dropout: Scaling dense and self-slimmable transformers. In *The Eleventh International Conference on Learning Representations*.

Chen, X., Liu, C., and Song, D. X. Towards synthesizing complex programs from input-output examples. *arXiv: Learning*, 2018.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

Chen, X., Song, D., and Tian, Y. Latent execution for neural program synthesis. *Advances in Neural Information Processing Systems*, 2021b.

Chirkova, N. and Troshin, S. A simple approach for handling out-of-vocabulary identifiers in deep learning for source code. *arXiv preprint arXiv:2010.12663*, 2020.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Clement, C. B., Drain, D., Timcheck, J., Svyatkovskiy, A., and Sundaresan, N. Pymt5: Multi-mode translation of natural language and python code with transformers. *ArXiv*, abs/2010.03150, 2020.

CodeParrot. Codeparrot/github-code. https://huggingface.co/datasets/codeparrot/github-code, 2022.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dong, L. and Lapata, M. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*, 2018.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., and Tenenbaum, J. B. Learning libraries of subroutines for neurally-guided bayesian program induction. In *NeurIPS*, 2018.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *ArXiv*, abs/2006.08381, 2020.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Fox, R., Shin, R., Krishnan, S., Goldberg, K., Song, D., and Stoica, I. Parametrized hierarchical procedures for neural programming. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJl63fZRb.

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. *arXiv preprint arXiv:2204.05999*, 2022.

Ganin, Y., Kulkarni, T. D., Babuschkin, I., Eslami, S. M. A., and Vinyals, O. Synthesizing programs for images using reinforced adversarial learning. *ArXiv*, abs/1804.01118, 2018.

Gulwani, S., Harris, W. R., and Singh, R. Spreadsheet data manipulation using examples. volume 55, pp. 97–105, January 2012. Invited to CACM Research Highlights.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*, 2017.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Hong, J., Dohan, D., Singh, R., Sutton, C., and Zaheer, M. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, pp. 4308–4318. PMLR, 2021.

Jimenez, M., Maxime, C., Le Traon, Y., and Papadakis, M. On the impact of tokenizer and parameters on n-gram based code analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 437–448. IEEE, 2018.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *ICML*, 2020.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*, 2022.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

Li, T., Shetty, S., Kamath, A., Jaiswal, A., Jiang, X., Ding, Y., and Kim, Y. Cancergpt: Few-shot drug pair synergy prediction using large pre-trained language models. *arXiv preprint arXiv:2304.10946*, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586*, 2021.

Liu, Y. and Wu, Z. Learning to describe scenes with programs. In *International conference on learning representations*, 2019.

Manna, Z. and Waldinger, R. J. Toward automatic program synthesis. *Commun. ACM*, 14:151–165, 1971.

Manna, Z. and Waldinger, R. J. A deductive approach to program synthesis. In *TOPL*, 1980.

Mao, Z., Jaiswal, A., Wang, Z., and Chan, S. H. Single frame atmospheric turbulence mitigation: A benchmark study and a new physics-inspired transformer model. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XIX*, pp. 430–446. Springer, 2022.

Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

Naur, P. Programming languages, natural languages, and mathematics. *Communications of the ACM*, 18(12):676–683, 1975.

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. *ArXiv preprint, abs/2203.13474*, 2022.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. Learning to infer program sketches. In *International Conference on Machine Learning*, pp. 4861–4870. PMLR, 2019.

Odena, A. and Sutton, C. Learning to represent programs with property signatures. In *International Conference on Learning Representations*, 2019.

Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. Automatically improving accuracy for floating point expressions. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

Shi, F., Suzgun, M., Freitag, M., Wang, X., Srivats, S., Vosoughi, S., Chung, H. W., Tay, Y., Ruder, S., Zhou, D., et al. Language models are multilingual chain-of-thought reasoners. *arXiv preprint arXiv:2210.03057*, 2022.

Sun, S.-H., Noh, H., Somasundaram, S., and Lim, J. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pp. 4790–4799. PMLR, 2018.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: code generation using transformer. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

Tian, Y., Luo, A., Sun, X., Ellis, K., Freeman, W. T., Tenenbaum, J. B., and Wu, J. Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875*, 2019.

Waldinger, R. J. and Lee, R. C. T. Prow: A step toward automatic program writing. In *IJCAI*, 1969.

Wang, X., Li, S., and Ji, H. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810*, 2022.

Wang, Y., Wang, W., Joty, S. R., and Hoi, S. C. H. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859, 2021.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Xu, F. F., Vasilescu, B., and Neubig, G. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–47, 2022.

Zelikman, E., Huang, Q., Poesia, G., Goodman, N. D., and Haber, N. Parsel: A unified natural language framework for algorithmic reasoning. *arXiv preprint arXiv:2212.10561*, 2022.

Zhang, Z., Zhang, A., Li, M., and Smola, A. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*, 2022.

Zheng, W., Chen, T., Hu, T.-K., and Wang, Z. Symbolic learning to optimize: Towards interpretability and scalability. *arXiv preprint arXiv:2203.06578*, 2022a.

Zheng, W., Sharan, S., Fan, Z., Wang, K., Xi, Y., and Wang, Z. Symbolic visual reinforcement learning: A scalable framework with object-level abstraction and differentiable expression search. *arXiv preprint arXiv:2212.14849*, 2022b.

Zhong, L., Lindeborg, R., Zhang, J., Lim, J. J., and Sun, S.-H. Hierarchical neural program synthesis. *arXiv preprint arXiv:2303.06018*, 2023.

A. I/O data Augmentation Approach

The model inputs include the natural language description and the I/O data of the given problem. Our natural language embedder uses a pre-trained BERT model, which has been trained on diverse texts. However, the sample embedder is only trained with I/O data sample tokens, and is only trained in the fine-tuning stage. To train the sample embedder with diverse texts, we carefully augment the I/O data and ensure that for each training instance in APPS and CodeContests, at least 100 program I/O pairs exist. The data augmentation is only applied to the training set within the existing instances and does not cause data leakage.

We apply *input distribution control* and *output distribution control* to ensure the quality of the generated data. The input distribution control aims to estimate the underlying distribution of the cases provided officially and draw more data from it. We implement a wide variety of data generators, and use a mixture of these generators to generate the inputs. We augment for both integer and string intputs, For example, if the program input is a list of integers, the values can be drawn from uniform distribution, quantized Gaussian distribution, almost sorted, all same value, almost having the same values with a few exceptions, or, the first element might be controlling the vector properties, such as the length of the vector, etc. We use different generators for these cases, and empirically assign probabilities to mix these generators. The output distribution control is only applied to bool program outputs. For each generated input sample, we use the existing code in the dataset to run and get the outputs. If one label (true or false) is significantly more than the other in the augmented data, then more is dropped to make sure the output labels are balanced.

B. Token Compositions

The encoded token strings for the $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$ subsequences are visualized in Figure 5.

The ChainCoder leverages a novel tokenizer based on AST instead of natural language. More specifically, the code expert tokenizer is used in the sample embedder to encode the program I/O data, and used in the model output side to encode the program syntax tree.

The vocabulary comprises of two parts: the tokens for the layout frame subsequence, which are grouped branching nodes in the syntax tree, and the tokens for the accessory subsequence, which are the leaf nodes. The layout frame part is collected by sweeping the dataset, as shown in Figure 3, and the accessory part is pre-computed and stored. The composition of the accessory part is shown in Table 3.

C. Visualizations

On the input side, for each given instance, multiple I/O data could be provided as part of the problem specifications. As the I/O data sequence is python-interpretable, it is also processed by the AST based tokenizer. Thanks to the syntax tree structure, they can be aligned across samples based on their syntax roles. The I/O data cross-sample alignment is shown in Table 4.

In addition, the examples of generated code are shown in Figure 6, where we pick the is-palindrome problem and compare the code generated by GPT Neo and ChainCoder.

Type	Definition	Example
Names	The names of variables, functions, classes, operators, etc., replaced from name pool	“var_1”, “func_1”, “class_1”, ...
Build in python vocabs	About 600 common python’s built-in functions and keywords	“__name__”, “__package__”, ...
Digit	Int representation of single digit number	0,1,2,3,4,5,6,7,8,9
ASCII	Character encoding	“a”, “!”, “=”, “0”, “[SPACE]”,...
Common float	Float that appears most common while training	0.1, 0.0001, 0.5, 0.2, ...

Table 3. Types of accessory tokens

```
Module(body=[FunctionDef(name=
,args=arguments(posonlyargs=[],args=[arg(arg=
,annotation=None,type_comment=None)],vararg=None,kwonlyargs=[],kw_defaults=[],kwarg=None,defaults
=[]),body=[Assign(targets=[Name(id=
,ctx=Store())],value=Call(func=Name(id=
,ctx=Load()),args=[Name(id=
,ctx=Load())],keywords=[]),type_comment=None),If(test=Compare(left=Name(id=
,ctx=Load()),ops=[
],comparators=[Constant(value=
,kind=None)]),body=[Expr(value=Call(func=Attribute(value=Name(id=
,ctx=Load()),attr=
,ctx=Load()),args=[],keywords=[])),Expr(value=Call(func=Attribute(value=Name(id=
,ctx=Load()),attr=
,ctx=Load()),args=[],keywords=[])),Assign(targets=[Name(id=
,ctx=Store())],value=Constant(value=
,kind=None),type_comment=None),While(test=BoolOp(op=
,values=[Compare(left=Name(id=
,ctx=Load()),ops=[
],comparators=[Name(id=
,ctx=Load())]),Compare(left=BinOp(left=Subscript(value=Name(id=
,ctx=Load()),slice=Index(value=Name(id=
,ctx=Load())),ctx=Load()),op=
,right=Constant(value=
,kind=None)),ops=[
],comparators=[Name(id=
,ctx=Load())])]),body=[AugAssign(target=Name(id=
,ctx=Store()),op=
,value=Constant(value=
,kind=None))],orelse=[]),Return(value=Name(id=
,ctx=Load()))],orelse=[Return(value=Constant(value=
,kind=None))])],decorator_list=[],returns=None,type_comment=None)],type_ignores=[])
```

```
func_0
var_in_0
var_0
len
var_in_0
var_0
Gt()
0
var_in_0
sort
var_in_0
reverse
var_1
0
And()
var_1
Lt()
var_0
var_in_0
var_1
Sub()
1
GtE()
var_1
var_1
Add()
1
var_1
0
```

```python
def h_index(counts):
    n = len(counts)
    if n > 0:
        counts.sort()
        counts.reverse()
        h = 0
        while (h < n and counts[h]-1>=h):
            h +=1
        return h
    else:
        return 0
```

Figure 5. Token strings demonstration: the strings in the above and below figures are $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$ subsequences for the code snipped in the bottom right, respectively.

Sample	Input	Content	Syntax
sample 1	Input 1	<align-padding>	<align-padding>
		<align-padding>	<align-padding>
		0	Module(body=[Expr(value=List(elts=[List(elts=[List(elts=[Constant(value=
		2	kind=None),Constant(value=
		3	kind=None),Constant(value=
	Input 2	<align-padding>	<align-padding>
		<align-padding>	<align-padding>
		0	kind=None)],ctx=Load()),Constant(value=
		<wait-padding>	kind=None)],ctx=Load())],ctx=Load()))],type_ignores=[])
sample 2	Input 1	0	Module(body=[Expr(value=List(elts=[List(elts=[List(elts=[Constant(value=
		2	kind=None),Constant(value=
		3	kind=None),Constant(value=
		5	kind=None),Constant(value=
		1	kind=None),Constant(value=
	Input 2	<align-padding>	<align-padding>
		<align-padding>	<align-padding>
		2	kind=None)],ctx=Load()),Constant(value=
		<wait-padding>	kind=None)],ctx=Load())],ctx=Load()))],type_ignores=[])
sample 3	Input 1	<align-padding>	<align-padding>
		4	Module(body=[Expr(value=List(elts=[List(elts=[List(elts=[Constant(value=
		5	kind=None),Constant(value=
		6	kind=None),Constant(value=
		7	kind=None),Constant(value=
	Input 2	5	kind=None)],ctx=Load()),Constant(value=
		3	<wait-padding>
		2	<wait-padding>
		<wait-padding>	kind=None)],ctx=Load())],ctx=Load()))],type_ignores=[])’

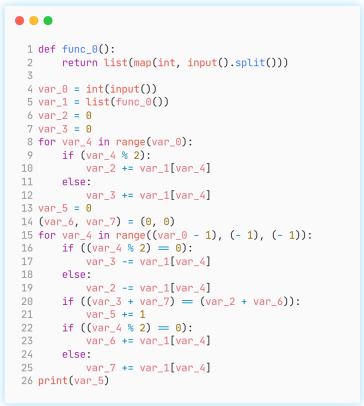
Table 4. One instance with three exsamples of I/O data. `io1 = [[[0,2,3],0], [12, 'abcd']]`, `io2 = [[[0,2,3,5,1], 2], [43, 'm']]`, `io3 = [[[4,5,6,7], 532], [9908, 'ss']]`.

```
n = int(input().strip())
weights = list(map(int, input().strip().split()))
'''
The expected inputs:
7
5 5 4 5 5 5 6
'''
res = 0
sum_even = sum(weights[1::2])
sum_odd = sum(weights[2::2])

for idx in range(n):
    if sum_even == sum_odd:
        res += 1
    if idx == n-1:
        break
    if idx % 2 == 0:
        sum_even += weights[idx]
        sum_even -= weights[idx+1]
    else:
        sum_odd += weights[idx]
        sum_odd -= weights[idx+1]
    # print(sum_odd)

print(res)

```



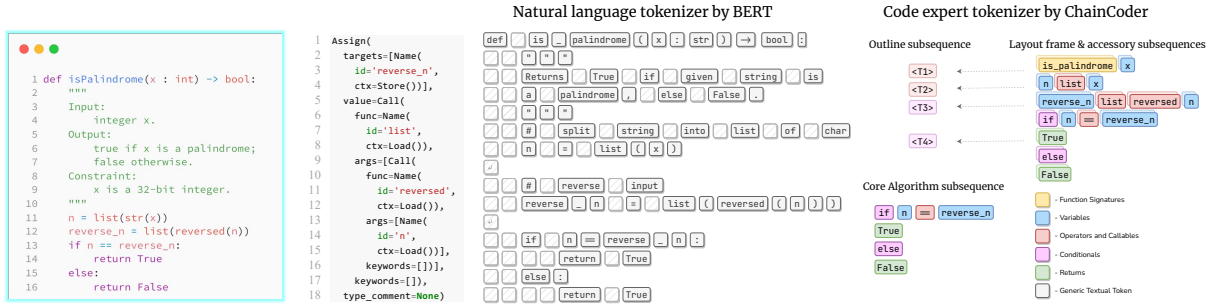
```
def func_0():
    return list(map(int, input().split()))

var_0 = int(input())
var_1 = list(func_0())
var_2 = 0
var_3 = 0
for var_4 in range(var_0):
    if (var_4 % 2):
        var_2 += var_1[var_4]
    else:
        var_3 += var_1[var_4]
var_5 = 0
(var_6, var_7) = (0, 0)
for var_4 in range((var_0 - 1), (- 1), (- 1)):
    if ((var_4 % 2) == 0):
        var_3 -= var_1[var_4]
    else:
        var_2 -= var_1[var_4]
    if ((var_3 + var_7) == (var_2 + var_6)):
        var_5 += 1
    if ((var_4 % 2) == 0):
        var_6 += var_1[var_4]
    else:
        var_7 += var_1[var_4]
print(var_5)
```

```
def a():
    return list(map(int, input().split()))
b = int(input())
c = list(a())
d = 0
e = 0
for f in range(b):
    if (f % 2):
        d += c[f]
    else:
        e += c[f]
g = 0
(h, i) = (0, 0)
for f in range((b - 1), (- 1), (- 1)):
    if ((f % 2) == 0):
        e -= c[f]
    else:
        d -= c[f]
    if ((e + i) == (d + h)):
        g += 1
    if ((f % 2) == 0):
        h += c[f]
    else:
        i += c[f]
print(g)

```

Figure 6. The example outputs of APPS model (GPT-Neo 2.7B, left) and ChainCoder (middle and right, with two different post-naming rules), over the same problem.



(a) Full solution code (b) Syntax tree of one line (c) Illustration of natural language and ChainCoder tokenizers

Figure 7. One Python solution to the is-palindrome example problem (left), and a subtree of the AST parsed syntax tree, from the sentence `reverse_n = list(reversed(n))` (right). The full tree is in Figure 8 in the Appendix with 106 lines in total.

```
body=[
  FunctionDef(
    name='isPalindrome',
    args=arguments(
      posonlyargs=[],
      args=[arg(
        arg='x',
        annotation=Name(
          id='int',
          ctx=Load()),
        type_comment=None)],
      vararg=None,
      kwonlyargs=[],
      kw_defaults=[],
      kwarg=None,
      defaults=[]),
    body=[
      Assign(
        targets=[Name(
          id='n',
          ctx=Store())],
        value=Call(
          func=Name(
            id='list',
            ctx=Load()),
          args=[Call(
            func=Name(
              id='str',
              ctx=Load()),
            args=[Name(
              id='x',
              ctx=Load())],
            keywords=[])],
          keywords=[]),
        type_comment=None),
      Assign(
        targets=[Name(
          id='reverse_n',
          ctx=Store())],
        value=Call(
          func=Name(
            id='list',
            ctx=Load()),
          args=[Call(
            func=Name(
              id='reversed',
              ctx=Load()),
            args=[Name(
              id='n',
              ctx=Load())],
            keywords=[])],
          keywords=[]),
        type_comment=None),
      If(
        test=Compare(
          left=Name(
            id='n',
            ctx=Load()),
          ops=[Eq()],
          comparators=[Name(
            id='reverse_n',
            ctx=Load())]),
        body=[Return(value=Constant(
          value=True,
          kind=None))],
        orelse=[Return(value=Constant(
          value=False,
          kind=None))])],
    decorator_list=[],
    returns=Name(
      id='bool',
      ctx=Load()),
    type_comment=None),
  Assign(
    targets=[Name(
      id='reverse_n',
      ctx=Store())],
    value=Subscript(
      value=Constant(
        value='121',
        kind=None),
      slice=Slice(
        lower=None,
        upper=None,
        step=UnaryOp(
          op=USub(),
          operand=Constant(
            value=1,
            kind=None))),
      ctx=Load()),
    type_comment=None),
  Expr(value=Call(
    func=Name(
      id='print',
      ctx=Load()),
    args=[Call(
      func=Name(
        id='isPalindrome',
        ctx=Load()),
      args=[Name(
        id='reverse_n',
        ctx=Load())],
      keywords=[])],
    keywords=[]))],
type_ignores=[])
```

Figure 8. Complete format of our AST tree, in the example of `isPalindrome` case. The corresponding code is in Figure 7.