AI Coders Are Among Us: Rethinking Programming Language Grammar Towards Efficient Code Generation

Zhensu Sun
Singapore Management University
Singapore
zssun@smu.edu.sg

Xiaoning Du*
Monash University
Australia
xiaoning.du@monash.edu

Zhou Yang
Singapore Management University
Singapore
zyang@smu.edu.sg

Li Li
Beihang University
China
lilicoding@ieee.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

Artificial Intelligence (AI) models have emerged as another important audience for programming languages alongside humans and machines, as we enter the era of large language models (LLMs). LLMs can now perform well in coding competitions and even write programs like developers to solve various tasks, including mathematical problems. However, the grammar and layout of current programs are designed to cater the needs of human developers – with many grammar tokens and formatting tokens being used to make the code easier for humans to read. While this is helpful, such a design adds unnecessary computational work for LLMs, as each token they either use or produce consumes computational resources.

To improve inference efficiency and reduce computational costs, we propose the concept of *AI-oriented grammar*. This aims to represent code in a way that better suits the working mechanism of AI models. Code written with AI-oriented grammar discards formats and uses a minimum number of tokens to convey code semantics effectively. To demonstrate the feasibility of this concept, we explore and implement the first AI-oriented grammar for Python, named Simple Python (SimPy). SimPy is crafted by revising the original Python grammar through a series of heuristic rules. Programs written in SimPy maintain identical Abstract Syntax Tree (AST) structures to those in standard Python. This allows for not only execution via a modified AST parser, but also seamless transformation between programs written in Python and SimPy, enabling human developers and LLMs to use Python and SimPy, respectively, when they need to collaborate. We also look into methods to help existing LLMs understand and use SimPy effectively. In the experiments, compared with Python, SimPy enables a reduction in token usage by 13.5% and 10.4% for CodeLlama and GPT-4, respectively, when completing the same set of code-related tasks. Additionally, these models can maintain or even improve their performance when using SimPy instead of Python for these tasks. With these promising results, we call for further contributions to the development of AI-oriented program grammar within our community.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Philosophical/theoretical foundations of artificial intelligence**; • **Theory of computation** → **Grammars and context-free languages**.

KEYWORDS

Code Generation, Programming Language, Large Language Model

ACM Reference Format:
Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI Coders Are Among Us: Rethinking Programming Language Grammar Towards Efficient Code Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680347

1 INTRODUCTION

High-level programming languages like Python, Java, and C++ are designed with two types of audiences in mind [6]: machines that compile and execute programs and humans who write, read, and comprehend programs. Machines focus on the operational semantics of programs, while humans additionally emphasize programs' readability, which is crucial for understanding source code. For example, one of the code design principles for Python [31] is that "readability counts." As a result, these languages incorporate many *human-centric* design elements within their grammar. For instance, programming languages utilize explicit delimiters to separate code structures, which enhances human readability but may not be essential to convey the program's operational semantics.

Recently, the audiences of programming languages have expanded to include AI models, particularly Large Language Models (LLMs), which can analyze, generate, and execute code. This is evident form the impressive performance that LLMs achieved in code generation [16]. For example, AlphaCode2 [3], a recently released LLM, reportedly outperforms 85% of human participants in a programming competition. Moreover, many LLM-powered assistants, such as ChatGPT [28] and Bard [14], are now equipped with code

*Corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09.
https://doi.org/10.1145/3650212.3680347

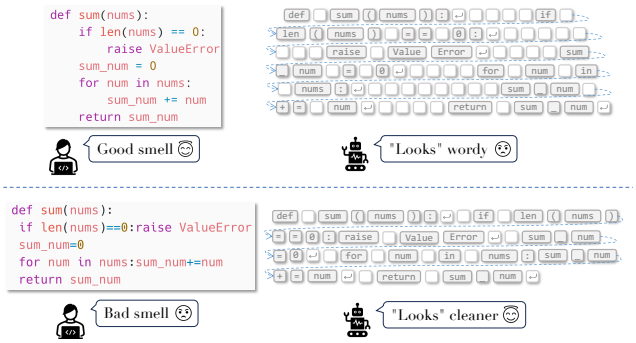

Figure 1: An illustration of how LLMs and human programmers perceive the source code.

execution environments, which enable them to execute generated code and provide responses based on the results. Thus, the role of LLMs has evolved from simply being code generators to actively functioning as "developers" that use programming to complete a wide range of tasks, such as mathematical computations and file processing. This shift in paradigm signifies a new era in which AI models emerge as an important group of programming language users.

While AI models have taken on their new role, the design of code grammar is yet to evolve to accommodate their unique needs. Elements used to improve the readability of source code could impose an additional computational burden on LLMs when they read and generate programs. However, the extra readability-enhancing tokens may not be essential for LLMs to perform coding tasks. Studies have revealed that code models do not capture much information relevant to readability [43], and readability-enhancing symbols like ":" received significantly lower attention compared to other elements such as variable names [50]. We illustrate how humans and AI models perceive a program in Figure 1. When certain elements that improve readability are removed from the code, the program retains its core meaning, but it becomes difficult for humans to understand. However, AI models can process the code more efficiently when they are adapted to this new code representation. This observation leads us to ask: *What is a suitable grammar for AI models?* Exploring this question is vital for optimizing the efficiency of LLMs and reducing energy waste in dealing with unnecessary tokens, especially given that the high operational cost of LLMs sets a big challenge for providers to generate profit [17] from them. As AI models consume and generate source code token-by-token, with one feed-forward process for each token, reducing the tokens in code representation has the potential to reduce the time and energy cost proportionally.

This motivates us to propose the concept of *AI-Oriented Grammar*, a grammar specifically designed for AI models instead of humans. The core idea is to derive grammar rules that keep the code representations concise (with a minimal/reduced number of tokens to AI models). Notably, the code crafted in this grammar can be parsed with its adapted parser and then executed to obtain the same result as the original grammar. A few challenges are in the way of designing such a new grammar and melting it into AI models. The AI models are expected to comprehend code written in this grammar and generate code following its rules to better serve the goal of efficiency. At the same time, human developers, who direct the development, expect to work with grammar that they find friendly and familiar with.

Given these challenges, the realization of this concept remains uncertain. To assess the feasibility of AI-oriented grammar, we embarked on an exploratory study. The study aims to reveal the implications and limitations of integrating AI-oriented grammar into the existing code generation workflow. Three research questions guide it, each addressing a key challenge.

RQ1. *What is the token reduction capacity of AI-oriented grammar in source code?*

Whether and to what extent an AI-oriented grammar can reduce the tokens remains an open question. We fill this gap by implementing a proof-of-concept AI-oriented grammar and assessing its performance. Specifically, we explore a new grammar for Python, named SimPy, by heuristically modifying the Python grammar. Compared to the standard Python grammar, we prohibit using tokens popularly hired to style the code appearance, e.g., whitespace and newline, and simplify keywords, operators, and delimiters to a more compact form. The modifications are designed to be simple, as this is the first attempt to explore such AI-oriented grammar. We also developed an AST parser for SimPy that can parse its code into the same AST as standard Python code, as well as a converter for seamless code transitions between SimPy and Python code. A comparative analysis between the SimPy and Python grammars was conducted using tens of tokenizers employed by existing LLMs. The findings indicate a notable reduction in token usage when employing SimPy, with decreases ranging between 8.6% and 34.7%, thus reducing the time and computational cost during inference by a similar level [19]. For example, the tokenizer of GPT-4 demonstrates a significantly enhanced efficiency with SimPy, achieving a 10.4% reduction in token size.

RQ2. *How can AI models understand AI-oriented grammar?*

Prior research demonstrates that AI models can comprehend human-centric grammars of existing programming languages [16]. However, how these models can learn AI-oriented grammar remains unexplored. We thus further experiment with SimPy to find an effective way. We explored two different training strategies: directly training a model on a SimPy-based code dataset (converted seamlessly from a Python dataset) and fine-tuning a model, originally trained with a Python dataset, on the SimPy dataset. A control group, where a model is directly trained on the Python code dataset, is also included for comparison. The models trained with either strategy should achieve at least equivalent accuracy compared with the control group. Otherwise, it would be impractical to adopt AI-oriented grammar. For each training strategy, we experiment with three models: CodeGen-NL, TinyLlama, and Pythia. The results reveal that models initially trained with Python can adapt effectively to SimPy. For instance, a CodeGen model, initially trained on Python, attains a 7.32% Pass@10 on HumanEval; further fine-tuning it on SimPy witnesses an increase of Pass@10 to 9.15%.

RQ3. *How can AI-oriented grammar support real-world scenarios?*

Given that AI-oriented grammar may compromise human readability, its application is somewhat restricted. Thus, a remaining

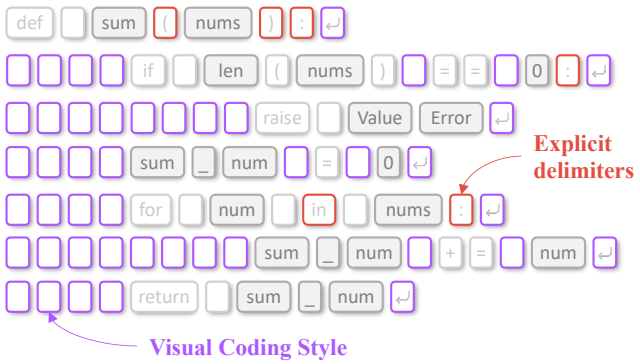

Figure 2: Illustrations of human-centric design elements in Python.

challenge for AI-oriented grammar is how it could be used in real-world scenarios, particularly when human-readable source code is necessary. We first discuss the basic usage scenario of AI-oriented grammar, i.e., scenarios in which the code generated by the AI models is not intended to be displayed to human users. AI agents [40] fall into this category, where the agents solve user-defined problems by generating and executing code in AI-oriented grammar. The code generated in the process is of little interest to the users. However, there are still many scenarios in which human developers need to review the code such as collaborative programming with coding assistants. We thus propose an inference framework for code generation named DualCode. DualCode utilizes a rule-based converter to convert code between these grammars, ensuring that users interact with human-readable code as usual. At the same time, the model benefits from the efficiency of AI-oriented grammar. Our experiments confirm that DualCode introduces negligible latency, with the converter of SimPy processing code under 500 tokens in less than 1.0 ms.

The source code of the paper is available at https://github.com/v587su/SimPy. The contributions of this paper are summarized as follows:

- We propose the concept of AI-oriented grammar and empirically explore its feasibility and potential, paving the way for future improvements in programming language design that prioritize AI efficiency.
- We implement the first AI-oriented grammar for Python, named SimPy, which can reduce at least 8.3% tokens in Python source code.
- We propose a novel code generation framework, DualCode, expanding the applicability of AI-oriented grammar beyond AI-only scenarios with negligible additional latency.

2 MOTIVATION

In this section, we carefully analyze the human-centered aspects found in the grammar of current programming languages and suggest the idea of an AI-oriented grammar. , we introduce the dataset that later will be used in our study when answering the three RQs.

2.1 Human-centric Grammar Design

As discussed in Section 1, modern programming languages are predominantly designed with human-centric grammar. This design philosophy originates from the longstanding reality that humans were the only developers for decades. In the current era of LLMs, this human-centric design philosophy has not been significantly challenged. To better ground this idea, we examine the grammar of widely used programming languages, focusing on lexical and syntactical elements that enhance human readability. Below, we summarize the identified patterns and provide examples in Figure 2:

Visual Coding Style The programming language grammar is deliberately crafted to accommodate diverse coding styles. Although not mandatory, styles like those recommended in the Python PEP8 guide [44] rely on grammatical support. For example, the coding style requires the programs to be written in multiple lines instead of a single extremely long line, easing human code review on screens. This necessitates several lexical elements: line breaks to separate lines, indents to visualize code blocks, and line continuation symbols for splitting long lines. Figure 2 demonstrates these aspects, with line breaks and indents highlighted in purple. Similarly, the coding style suggests surrounding each binary operator with a single white space on either side. Therefore, lexical grammar must accommodate such stylistic elements, even if they may not contribute to the core semantics in parsing.

Intuitive Notations The human-centric syntax of programming languages is designed to be intuitively understandable to humans. Common operators like "+" for addition and "=" for assignment are chosen for their familiarity, and derivations like the augmented assignment operator "+=" maintain this intuitive connection. Although potentially more concise symbols could replace these (e.g., using a brand-new symbol "$" for "+="), they are still deliberately designed to maintain human readability. Similarly, for structural clarity, programming languages often employ explicit delimiters, such as symbols or keywords, to define code structures despite these delimiters not being essential for parsing. For instance, Python's compound statements, such as the if statement and for statement, use a colon to demarcate the header from the body. While a parser might deduce these components from line breaks alone, the colon acts as a visual aid, as illustrated in Figure 2, where colons are highlighted in red. This emphasis on intuitive notation and explicit delimiters, although not essential for parsing, significantly aids human comprehension.

2.2 AI-Oriented Grammar

Grammar is a rule set that defines how the source code should describe the programming language's semantics in aspects of lexis and syntax, using notations such as symbols and keywords. The primary function of the notations in the grammar is two-fold: to define a program's structure for machine execution and to enhance visual comprehension for human readability. Given that AI models do not require assistance in visual comprehension, the focus of AI-oriented grammar is solely on structural definition. We thus consider a notation unnecessary for AI models if it does not contribute to accurate parsing by the parser. AI-oriented grammar is designed with indispensable notations.

In the design process of a programming language, semantics are defined first, followed by the development of a grammar to represent them. Therefore, employing AI-oriented grammar does not alter the fundamental semantics of the programming language. Technically, a programming language grammar and its AI-oriented

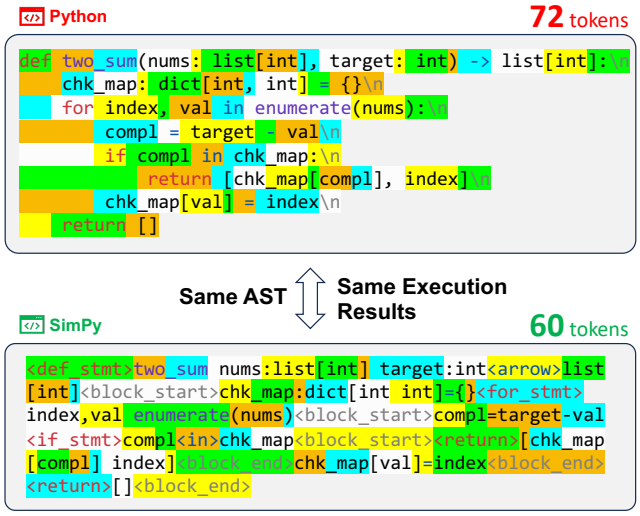

Figure 3: A comparison between Python and SimPy source code, tokenized by GPT-4's tokenizer. Continuous characters with the same background color represent the same token. Notably, there are no line breaks in the SimPy example and we add these line breaks in the figure for our human readers.

design share the AST (abstract syntax tree) definition. Given that an AST uniquely encodes the semantics of a code, it facilitates an equivalent transformation between implementations of the semantics with either grammar.

2.3 Python Code Dataset for Our Study

As a newly proposed concept, we are still unclear whether AI-oriented grammar can be realized and what scenarios it can be applied to. To address these uncertainties and explore the potential of AI-oriented grammar, we conduct an empirical study guided by three critical research questions, respectively introduced in Section 3, Section 4, Section 5. Our study is centered around Python, the main programming language of the execution environment for LLMs like GPT-4 and Bard to address programming-required tasks. We utilize the Python subset of starcoderdata [23], a filtered variant of The Stack dataset [21], a comprehensive collection of over 20 million code files sourced from open-source GitHub repositories. We keep the code files from the repositories with over 100 stars, resulting in 623,887 code files. The dataset is partitioned into training and validation sets in a 95:5 ratio. We do not create a separate testing set, as we plan to evaluate the model's performance using other established evaluation datasets. The code snippets in the evaluation datasets are excluded from the training dataset.

3 TOKEN REDUCTION (RQ1)

In this section, we present an instance of AI-oriented grammar to answer **RQ1**: *What is the token reduction capacity of AI-oriented grammar in source code?* We propose an AI-oriented grammar for Python as a proof-of-concept (Section 3.1) and then proceed to evaluate the extent of token reduction achievable with this grammar (Section 3.2).

3.1 An AI-oriented grammar for Python

We introduce how SimPy, a simplified grammar for Python, is designed by applying the philosophy of AI-oriented grammar in Section 3.1.1. We discuss how SimPy is validated against ambiguity in section 3.1.2 and prove the semantic equivalence in section 3.1.3. Alongside SimPy, we develop a toolkit including a parser to interpret SimPy source code into Python's AST, and a converter for seamless code translation between SimPy and Python.

3.1.1 Design. A straightforward method to reduce the number of tokens in source code is to remove redundant delimiters. For example, consecutive newline symbols or whitespace (neither indent nor dedent) can be reduced to a single instance without affecting the code's semantics. These tokens can be optimized in programs written in any programming language. Inspired by the example in Introduction, we explore slight modifications to an existing grammar to reduce even more tokens.

As the first attempt at AI-oriented grammar design, we limit the changes to terminals in grammar; the semantics of production rules are kept intact. Terminals are notations[1] that cannot be expanded by any production rule. Typical terminals include keywords (e.g., "true"), symbols (e.g., ">=", "(", and ","), literals (e.g., user-defined constants), and identifiers (e.g., variable names). Limiting the changes to terminals additionally helps maintain semantic equivalence between Python and SimPy , minimizes potential ambiguity in SimPy , and keeps implementation efforts at an acceptable level.

In the following, we use Python 3.12 as the base grammar and describe the modifications made to obtain SimPy . SimPy is not guaranteed to be optimal in terms of model-processing efficiency but is sufficient to serve as a proof-of-concept demonstration for AI-oriented grammar. For a quick comparison, an illustration of SimPy and Python code is shown in Figure 3, where both code snippets share the same AST but the SimPy representation consists of fewer tokens (measured by the GPT-4 tokenizer). In Table 1, we compare the grammar specifications of key productions before and after these modifications and count the tokens affected in the modification. Limited by the space, we introduce the two major categories of modifications here; the complete grammar specification is available in our artifact.

Replace terminal notations with tokens. This type of modification is based on two observations: 1) the whitespace around certain terminals, such as "true", is mandatory to ensure recognizability, and 2) some terminals, like "+=", might be tokenized into multiple tokens. If we can find a way to eliminate the need of surrounding whitespace for these terminals and ensure that they are parsed as a single token, more tokens can be removed when squeezing out redundant delimiters. Based on whether the terminals are made of a fixed sequence of characters, they can be divided into two groups, namely constant terminals and user-defined terminals. The instances of user-defined terminals are not available to the grammar definition. Here, we propose to replace the constant terminals by distinct token placeholders. For example, "true" and ">=" will be replaced by "<true>" and "<ge>", respectively. In this way, whitespace around these placeholders is no longer needed. During the

[1]Terminals are tokenized by the compiler lexer. To distinguish the tokenization performed by the lexer and that performed by the tokenizers of LLMs, we use the term "notation" to refer to the tokens produced by the lexer.

Table 1: Comparison of grammar specifications for Python and SimPy, using the official Python grammar notation ([32]). Each terminal notation affected by SimPy is annotated in blue. The table also includes the count of such terminal notations: "N" represents the number of lines, "n" signifies the count of repetitive elements, and "?" indicates that the number of terminals is conditional.

Name	Grammar	Specification	#Terminal
block	Python	NEWLINE INDENT statements DEDENT	3
	SimPy	'<block_start>' statements '<block_end>'	2
function_def	Python	'def' NAME [type_params] '(' [params] ')' ['->' expression] ':' [func_type_comment] block	4+1?
	SimPy	'<def_stmt>' NAME [type_params] [params] ['<arrow>' expression] [func_type_comment] block	1+1?
class_def	Python	'class' NAME ['(' [arguments] ')'] ':' block	2+2?
	SimPy	'<class_stmt>' NAME ['(' [arguments] ')'] block	1+2?
if_stmt	Python	'if' named_expression ':' block elif_stmt	2
	SimPy	'<if_stmt>' named_expression block elif_stmt	1
for_stmt	Python	'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_block]	3
	SimPy	'<for_stmt>' star_targets ~ star_expressions [TYPE_COMMENT] block [else_block]	1
with_stmt	Python	'with' ','.with_item+ ':' [TYPE_COMMENT] block	2+n
	SimPy	'<with_stmt>' ' '.with_item+ [TYPE_COMMENT] block	1
try_stmt	Python	'try' ':' block except_block+ [else_block] [finally_block]	2
	SimPy	'<try_stmt>' block except_block+ [else_block] [finally_block]	1
while_stmt	Python	'while' named_expression ':' block [else_block]	2
	SimPy	'<while_stmt>' named_expression block [else_block]	1
import_from	Python	'from' ('.' | '...')* dotted_name 'import' import_from_targets	2+n?
	SimPy	'<from_import_stmt>' ('.' | '...')* dotted_name import_from_targets	1+n?
simple_stmts	Python	';'.simple_stmt+ [';'] NEWLINE	n+1+1?
	SimPy	['<line_sep>'].simple_stmt+ ['<line_sep>']	n?+1?

tokenization by LLMs, each placeholder will be mapped directly to an entry in the vocabulary of the tokenizer, being treated as a single token. Note that some single-character symbols, like ".", "(", and "=", are not replaced in consideration of the common optimization provided by the popular tokenizers – these symbols are combined into their next token thus contributing no extra token. For example, "(nums" in Figure 2 is treated as a single token by GPT-4.

Simplify notations in the grammar context. Some terminal notations can be further simplified in the context of specific production rules. During the design, we review every production rule and determine if any notations can be removed, merged with or replaced by others. Here, we mainly focus on delimiter symbols, keywords, and other terminals produced by the lexer (e.g., "NEWLINE"). Delimiters separate different program structures, many of which are used to aid humans in reading. In a given grammar context, some delimiters can be removed without affecting the parsing. Taking the "function_def" rule as an example, "(" and ")", surrounding the parameters, and ":", setting aside the function header and body, are discarded. As a result, four terminals plus one terminal in the optional structure in the original Python grammar are simplified into one terminal plus one in the optional structure.

In another example, the "block" statement in Python hires "NEWLINE", "INDENT", and "DEDENT" to indicate the start and end of a block. Here, "NEWLINE" and "INDENT" are merged and replaced by the new token placeholder "<block_start>", while "DEDENT" is replaced by "<block_end>" according to the previous design rule. Three affected terminals in Python grammar are reduced into two. Another design we would like to highlight is that the"NEWLINE" terminal, indicating the line breaks, are replaced with an new optional token placeholder "<line_sep>" in simple_stmts. It permits the omission of "<line_sep>" when the subsequent a token implying the start of a new line, such as "<def_stmt>" (the replacement of "def") for function definitions or "<class_stmt>" (the replacement of "class") for class definitions.

Finally, we leverage the design of existing tokenizers to replace some tokens with those that are combined into their following tokens by the tokenizers. For example, the whitespace character is combined into its following token by GPT-4, as illustrated by " two" and " int" in Figure 2. For some mandatory explicit delimiters, if they can be replaced by a whitespace, the number of tokens can also decrease. For instance, we replace "," with a whitespace in the with_stmt. However, such replacement may cause conflicts, as whitespace is widely used for separation in for many structures. One example is the list structure in Python, which uses "," to separate different elements, e.g., "['1', '2']". When we replace the "," with a whitespace, the parser generator raises a conflict regarding the concatenation of strings separated by whitespace, such as "'hello' 'world'". To resolve this conflict, we introduce the string concatenation with a new token placeholder, "<concat>", as a separator for string concatenation. This approach considers that the list structure is used more frequently in code. Representing more frequent components with fewer tokens is a widely used strategy in content compression. Designing a grammar that considers the frequency of different terminals, structures, and grammar rules could be a very interesting direction for future work.

3.1.2 Unambiguity of SimPy . To determine whether a grammar has ambiguity is theoretically undecidable [13]. In practice, parser generator tools are commonly hired to check for ambiguities in grammar, including those of popular programming languages [15]. A parser generator can find a wide range of ambiguities in the grammar, such as conflicts that arise when the parser has two possible actions at one step. Practically, this is almost the best way to check the ambiguity of SimPy. We have successfully generated parsers for SimPy using the GLR (generalized left-to-right rightmost derivation parser) parsing algorithm [22] from tree-sitter [42], where no ambiguity is detected.

Next, we provide an analytical discussion about why our transformations are unlikely to introduce ambiguity to the grammar. First of all, the transformations are only made to terminal notations, which act as keywords, symbols, or delimiters. Changes made to keywords and symbols are guaranteed to represent its unique semantics, while changes made to delimiters should not affect the recognition of the construct, as well as its precedent and subsequent constructs.

Case I: *New notations are added or introduced as replacements.*

Importantly, different notations are not replaced with the same new notations. To this end, the new notations do not interfere with production rules for which the transformation is not applicable. Given that they are semantically equivalent notations as the original one, the parsing of the affected production rules remains the same. For example, replacing the *'NEWLINE INDENT'* in the production rule of block (see Table 1) with *'<blcok_start>'* conveys the same semantics that a block is about to start.

Case II: *Existing notations are removed.*

Arbitrary removal of notations may introduce ambiguity to the grammar. We carefully design a few heuristics when removing notations so they are unlikely to cause problems.

- **Remove notations with redundant semantics as their adjacent notations.** For example, *':'* in many statements indicates the end of the previous construct and the start of a new construct, e.g., in *'if' named_expression ':' block elif_stmt*. However, the block construct initiates with its starting symbol, making the construct itself distinguishable from any previous construct. Hence, removing *':'* is safe for this case.
- **Remove delimiters used to scope a construct when the scope of its precedent and subsequent constructs is clear.** For example, the *'('* and *')'* for parameters are actually unnecessary in *function_def_raw := 'def' NAME [type_params] '(' [params] ')' ['->' expression] ':' [func_type_comment] block*. *NAME* is an atomic token, thus will not interfere with the beginning of parameters when *type_params* is absent. *type_params* are surrounded by *'['* and *']'*, making their presence not an issue for recognizing params. Hence, *'('* can be safely removed. Now, looking at the subsequent constructs, *['->' expression]*, *':'*, *[func_type_comment]*, or *block* possesses a unique indicator of their beginning. Hence, *')'* can be safely removed as well. Another example is the '*import*' keyword in *import_from := 'from' ('.' | '...')* dotted_name 'import' import_from_targets*. Since *dotted_name* is a must and contains no white spaces, hence the white space between *dotted_name* and *import_from_targets* can perfectly separate these two constructs. Removing '*import*' is also fine.

3.1.3 Semantic equivalence between SimPy and Python. SimPy is designed as a simplified grammar of Python, which means a program written in Python can be equivalently and deterministically transformed to its counterpart in SimPy , and vice versa. In other words, Python and SimPy are semantically equivalent. We prove this statement in Theorem 1.

Formally, we define a grammar G and a grammar G'. G' is obtained via a transformation T to the production rules in G. Given a production rule, T is restricted to adding, replacing, or removing a terminal notation or a sequence of terminal notations. The transformation between Python and SimPy is an instance complying with this restriction. For example, *T(NEWLINE INDENT statements DEDENT) = '<block_start>' T(statements) '<block_end>'*.

Theorem 1. *Python and SimPy are semantically equivalent.*

Proof. Two programs are semantically equivalent if they share the same AST. We assume the Python grammar can be transformed into the SimPy grammar via T.

We give the proof by structural induction on p.

Base case: p is an atomic program construct. According to the design of T, for keywords and symbols used in atomic program construct, a subset of them are transformed to a new token, e.g., $true$ is transformed to <$true$>. Since the mapping is unique and deterministic, a keyword and its mapped token share identical semantics and will be abstracted to the same AST node.

Inductive case: This can be proved by analyzing each compound language construct that is affected by the transformation. Due to the space limitation, we only show one proof for the *if_stmt* construct. Assuming *p = 'if' named_expression ':' block elif_stmt*, we have *p′ = '<if_stmt>' T(named_expression) T(block) T(elif_stmt)*. Both p and p' will be translated into an AST rooted with a node representing the if statement, and with three children representing the *named expression*, *function body*, and *else statement*. By the induction hypothesis, $T(named\_expression)$, $T(block)$, and$T(elif_stmt)$ share the same AST with $named\_expression$, $block$, and $elif\_stmt$, respectively. Hence, p and p' share the same AST.

By proving all other constructs, we can prove that for any program in Python, its counterpart in SimPy is semantically equivalent to it. Similarly, we can prove that for any program in SimPy , its counterpart in Python is semantically equivalent to it as well. Thus, the theorem is proved. □

3.1.4 Implementation. Based on the grammar specifications of SimPy, we develop a toolkit for it, including an AST parser for SimPy code and a converter for seamless translation between SimPy and Python source codes. The parser is built upon tree-sitter [42], a popular parser generator tool. We first describe the grammar specification of SimPy in the configuration file of the tree-sitter and then generate the parser. With the help of the GLR algorithm from the tree-sitter, we ensure SimPy resolves all the conflicts and no ambiguity exists. The generated parser can parse the SimPy source code into the AST of Python. Based on this parser, we further implement a converter, where specific conversion rules are established for each node of the AST. From a pragmatic point of view, we test our implemented toolkits by conducting round-trip transformations, where Python source code is first converted into SimPy code and subsequently retranslated back to Python. Our first tests on the Python dataset revealed that, ignoring all whitespace,

Table 2: Percentage of token reduction achieved with SimPy. The "Code" and "Web" in the "Vocab Source" column represent the sources for constructing the tokenizer's vocabulary: code repositories and internet data, respectively.

Tokenizer	Vocab Source	Vocab Size	Tokens		
			Python	SimPy	
CodeBert	Code	50k	1.33B	0.87B	34.7%↓
GPT2	Web	50k	1.33B	0.87B	34.7%↓
CodeLlama	Web	32k	0.97B	0.84B	13.5%↓
WizardCoder	Web	32k	0.97B	0.84B	13.5%↓
DeepSeek-Coder	Web	32k	0.97B	0.84B	12.9%↓
CodeGen	Web	51k	0.93B	0.82B	12.6%↓
CodeT5+	Web	51k	0.93B	0.82B	12.6%↓
Codex	Web	51k	0.93B	0.82B	12.6%↓
CodeT5	Code	32k	0.91B	0.78B	13.8%↓
StarCoder	Code	49k	0.83B	0.76B	8.6%↓
SantaCoder	Code	49k	0.83B	0.76B	8.8%↓
Replit-code	Code	33k	0.82B	0.75B	8.6%↓
GPT-3.5	Web	100k	0.71B	0.63B	10.4%↓
GPT-4	Web	100k	0.71B	0.63B	10.4%↓

the textual content of the code remains unchanged after the transformation. In addition, we assess its soundness through execution results. We perform the round-trip transformation to the ground-truth code snippets of HumanEval and run the test cases on both the transformed and the original code. The execution results of all the transformed code and the original code are exactly the same, which also indicates the soundness of our implementation.

3.2 Experiments of RQ1

In this section, we detail the tokenizers employed in our experiments and describe the experimental results.

3.2.1 Tokenizers. Our experiments encompass a broad spectrum of tokenizers from various LLMs. The main difference between them is the training corpus, leading to different token vocabularies.
GPT-2 [35], **Codex** [8], **GPT-3.5** [29], **GPT-4** [30]: These tokenizers, released by OpenAI, are trained on a mixed corpus, including both natural language and programming language, with GPT-4 being the latest version offering state-of-the-art performance in various language tasks.
CodeLlama [37], **WizardCoder** [24], **DeepSeek-Coder** [1]: These tokenizers are derived from the tokenizer of Llama 2 [41] which is also trained on the mixed corpus.
SantaCoder [2], **StarCoder** [23], **Replit-code** [36]: These tokenizers are specialized for code, having been trained exclusively on programming language datasets, and are thus more adept at handling source code.
CodeGen [27], **CodeT5** [47], **CodeT5+** [46]: These tokenizers are extended based on the vocabulary of GPT2 with additional tokens representing repeating tokens of tabs and white spaces.

3.2.2 Results. To answer RQ1, we conducted an evaluation involving the representation of code files from our Python dataset in both its original grammar and in SimPy, followed by the tokenization using the same tokenizer for each representation. We created the SimPy dataset by converting the Python dataset with our converter. In tokenizing the SimPy code, we modify the tokenizers to include tokens of SimPy in their vocabularies. In total, 14 tokenizers from popular LLMs are evaluated in our experiments, where each tokenizer's vocabulary source and size are also documented to offer a comprehensive view of SimPy's performance across different models. By examining the variation in token numbers, we evaluated SimPy's effectiveness in reducing token size, thus showcasing the potential benefits of AI-oriented syntax.

As revealed in table 2, SimPy can reduce the number of tokens by 8.6% to 34.7%, depending on the tokenizers. The GPT-4 and GPT-3.5 tokenizers, which are already the most efficient in representing Python source code, show a further reduction of 10.4% in token count with SimPy. For tokenizers trained on code corpora, such as Replit-code and StarCoder, SimPy achieved a token reduction ranging from 8.6% to 13.8%. Tokenizers trained on web-based corpora like CodeGen and CodeT5 also exhibited significant reductions, between 12.6% and 13.5%. The most pronounced impact of SimPy is observed with the least efficient tokenizers, CodeBert and GPT-2, where a remarkable 34.7% reduction in token count was achieved. These promising results highlight SimPy's potential to reduce token count for source code representation. As estimated by OpenAI [19], the Floating-point operations (FLOPS) required for generating each token during inference can be regarded as being only relevant to the model size when the context size is fixed. Therefore, a reduction in token count can be directly translated to a decrease in FLOPS at a similar level, resulting in faster inference speeds given the fixed computing speed of the device.

Answer to RQ1: AI-oriented grammar, exemplified using SimPy, effectively reduces the number of tokens required for source code representation, with models like GPT-4 benefiting from a 10.4% reduction. Correspondingly, it leads to a speed up and a computing saving during inference at a similar level.

4 MODEL TRAINING (RQ2)

In this section, we aim to answer **RQ2**: *How can AI models understand AI-oriented grammar*? We experimentally investigate whether AI models can retain their accuracy when trained with AI-oriented grammar. We describe our training strategies in Section 4.1 and assess their effectiveness on two language models in Section 4.2.

4.1 Training Strategies

Training AI models with AI-oriented grammar is a pivotal step in enabling the model to deal effectively with source code in this new format. Despite the efficiency gains demonstrated by SimPy, such training should not compromise the model's accuracy. To explore the feasibility of such training, we experiment with two different strategies. Next, we introduce the strategies in the experiment, from tokenizer refining to model training.
Tokenizer Refining SimPy introduces 78 new tokens for the tokenizers to recognize. For example, the "def" keyword of the original Python grammar is replaced by a token "<def_stmt>". Given the existing association between the pre-trained model and its tokenizer,

completely retraining the tokenizer on SimPy code to optimize token distribution is impractical. Instead, we opt for a more feasible approach: expanding the tokenizer's vocabulary to include these new tokens. Correspondingly, this modification requires resizing the embedding matrix ([vocab size * embedding size]) and the output layer ([hidden state size * vocab size]) to fit the expended vocab size. This expansion introduces a few new parameters, mainly in the output layer, around 78 * hidden_size parameters. For instance, modifying a CodeGen [27] model with a hidden state size of 2048 introduces around 160 thousand new parameters, a negligible increase (less than 0.01%) in the total parameter count. Moreover, the resizing will randomly initialize both the embedding vector for each new token and the weight of the output layer, which will be updated during the model training.

Model Training Our study explores two basic training strategies: 1) directly training a model on the SimPy code dataset, referred to as **SimPy**, and 2) sequentially training a model first on the Python dataset and then on the SimPy code dataset, referred to as **Python→SimPy**. If such basic strategies work, further improvement in efficiently adapting AI-oriented grammar is completely feasible. Moreover, we construct a control group: directly training a model on the Python code dataset, denoted as **Python**. The performance of the two strategies should match or surpass the model from the control group; otherwise, they are not practical. To control the variable, all training sessions across the two strategies and the control group are conducted under identical conditions, including the training environment, initial model, and training hyper-parameters. Notably, the SimPy dataset is converted from the Python dataset, ensuring no external data is involved. Moreover, for the Python+SimPy setting, we vary the proportion of the SimPy dataset used, i.e., 10%, 20%, 50%, and 100%, to assess the required volume of data for effective fine-tuning.

4.2 Experiments of RQ2

We first present the experimental setup for RQ2, including the models used, evaluation metrics, and implementation details. Then, we report the experimental results and answer the research questions.

4.2.1 Models. We adopt three widely used models in our research community, namely CodeGen-NL, TinyLlama, and Pythia, whose parameter sizes range between 350M and 1.1B. All these models serve as the initial pre-trained model for our experiments. Though these are not the latest state-of-the-art models, they suffice to validate the feasibility of learning AI-oriented grammar like SimPy. We will further discuss the impact of this decision in Section 7.

CodeGen-NL: CodeGen, proposed by Salesfore [27], is an open-sourced language model designed for code generation. It undergoes a multi-phase training process on different datasets, where the model is first trained with natural language datasets and then code datasets. Our experiments utilize its natural language version (CodeGen-350M-nl), produced after the initial phase of its training process, as the foundation model to conduct our experiments.

TinyLlama: TinyLlama [51] is a compact 1.1B language model pre-trained on around 3 trillion tokens, building on the architecture and tokenizer of Llama 2 [41]. It shows competitive performance compared to existing open-source language models of similar sizes.

Table 3: The Pass@1 and Pass@10 of LLMs on Python and SimPy datasets under varied settings. Python and SimPy denote models trained exclusively on respective datasets. Python→SimPy refers to sequential training on both datasets, with the parenthetical numbers indicating the SimPy dataset's proportion involved in the training.

Model	Training Strategy	Pass@1	Pass@10
CodeGen-NL	Python	4.51%	7.32%
	100% SimPy	2.93%	5.49%
	Python → 10% SimPy	3.11%	3.66%
	Python → 20% SimPy	3.66%	4.27%
	Python → 50% SimPy	3.96%	6.71%
	Python → 100% SimPy	**4.82%**	**9.15%**
TinyLlama	Python	10.00%	13.41%
	100% SimPy	5.91%	9.76%
	Python → 10% SimPy	2.07%	3.66%
	Python → 20% SimPy	3.23%	5.49%
	Python → 50% SimPy	5.73%	11.59%
	Python → 100% SimPy	**10.12%**	**14.02%**
Pythia	Python	5.79%	9.76%
	100% SimPy	**7.01%**	9.15%
	Python → 10% SimPy	1.89%	2.44%
	Python → 20% SimPy	3.11%	4.27%
	Python → 50% SimPy	4.21%	7.32%
	Python → 100% SimPy	5.67%	**10.00%**

Pythia: Pythia [4] is a suite of LLMs, which is expected to be used as the baseline for research studies and thus is designed close to currently accepted common practices. Considering the capacity of our computing resources, we use its 1B version.

4.2.2 Evaluation Metrics. We evaluate the model's performance on the code generation task with the Pass@k metric on HumanEval. To compute Pass@k, k code samples are generated for each problem, and a problem is considered solved if any of the k samples pass the unit tests. We report the fraction of problems being successfully solved. The HumanEval dataset [8] comprises 164 programming problems, each with a function signature, a docstring, and multiple test cases. Given the function signature and docstring, the model is required to generate the code, which is then tested by executing the test cases. Notably, the function signatures are written using Python's original grammar. When evaluating the model adapted to SimPy, we convert the function signature into SimPy using the code converter. Similarly, the model-generated SimPy code is subsequently converted into Python to run test cases since the existing testing framework is implemented for Python source code.

4.2.3 Implementation Details. In our experiments, we use the Huggingface Transformers library [48] with Pytorch to implement the models. The experiments of CodeGen-NL are performed on a machine with 48 vCPUs, 512GB RAM, and four RTX A5000 GPUs (24GB RAM), while the other two models are trained on a machine with 28 vCPUs, 200GB RAM, and two RTX A6000 GPUs (48GB RAM). The hyper-parameters of the training are set referring to CodeGen's hyper-parameters: 8 batch size, 1.8e-4 learning rate, 0.1

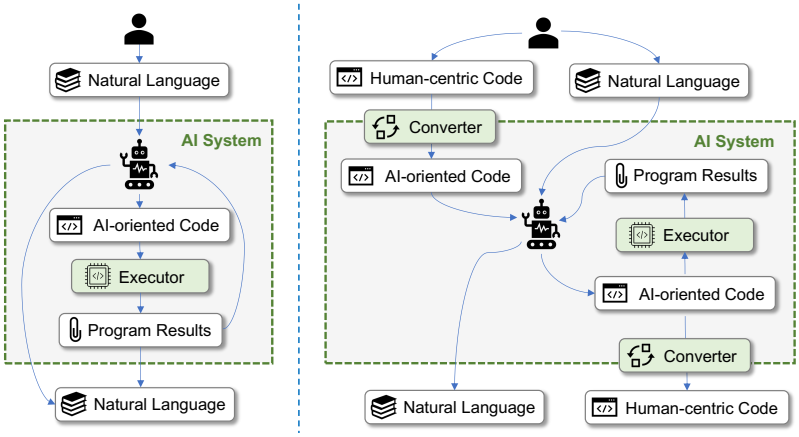

Figure 4: LEFT: the workflow of the basic usage scenarios of AI-oriented grammar. RIGHT: the workflow of the extended usage scenarios of AI-oriented grammar under DualCode, where the code executor of the AI system in the figure is not necessary.

weight decay, and 512 context length. During the inference for evaluation, we set the temperature to 0.2 and the top-p to 0.95.

4.2.4 Results. Following the settings of the two strategies (SimPy and Python→SimPy) and the control group (Python), we train the CodeGen-NL, TinyLlama, and Pythia models, respectively. Finally, for each of our initial models, we have six variations: one each for Python and SimPy, and four models for Python→SimPy incorporating 10%, 20%, 50%, and 100% of the SimPy dataset. The performance of these models is evaluated through Pass@1 and Pass@10 metrics on the HumanEval dataset.

We report the results in Table 3. Notably, the models trained with SimPy lag behind the Python baseline in terms of accuracy. For example, the Pass@1 and Pass@10 of CodeGen (SimPy) are respectively 2.93% and 5.49%, lower than the ones of CodeGen (Python), which are 4.51% and 7.32%. This could be attributed to SimPy's limited expressiveness, constraining the models from leveraging knowledge acquired from natural language datasets during pre-training. Consequently, direct training with AI-oriented grammar appears to be an impractical approach.

However, the sequential training strategy, starting with Python and then incorporating SimPy, yields comparable or even superior accuracy to the control group. Specifically, CodeGen-NL, TinyLlama, and Pythia models trained with Python→100%SimPy achieve Pass@10 scores of 9.15%, 14.02%, and 10.00%, respectively, outperforming the control group's 7.32%, 13.41%, and 9.76%. This suggests a successful training with SimPy, demonstrating the feasibility of AI models learning AI-oriented grammar. Interestingly, we observe that the Pythia model, when trained exclusively with 100% SimPy, surpasses the Python baseline on Pass@1. This highlights the possibility of learning SimPy without relying on the sequential training strategy. By varying the proportion of the SimPy dataset in the Python→SimPy setting, we found that a substantial dataset is still required by the fine-tuning with SimPy. For instance, TinyLlama (Python→50%SimPy) scored 5.73% in Pass@1 and 11.59% in Pass@10, still trailing behind the TinyLlama (Python) scores. We will further discuss this finding in Section 8.

Answer to RQ2: AI models, when initially trained with the original grammar and then the AI-oriented grammar, can successfully learn the AI-oriented grammar while retaining their accuracy. For instance, the CodeGen model, originally trained with Python and achieving a 7.32% Pass@10, improved to a 9.15% Pass@10 after the additional training with SimPy.

5 USAGE SCENARIO (RQ3)

In this section, we address **RQ3**: *How can AI-oriented grammar support real-world scenarios?* We first demonstrate the basic application scenario of AI-oriented grammar, and subsequently introduce a novel inference framework designed to broaden the applicability of AI-oriented grammar, followed by an evaluation of the framework's additional latency.

5.1 Basic usage scenario

The source code, when written in AI-oriented grammar, becomes challenging for human interpretation and is therefore not intended for human display. Consequently, the application of AI-oriented grammar is limited to scenarios where human users do not have access to the generated code. A typical scenario is the AI agents, such as AutoGPT [40] and LangChain [7], for regular users rather than developers. For instance, an AI agent tasked with data collection from a website would generate the required crawler script, execute it to gather data, and present the outcomes to the user. End users generally care more about the results than understanding the underlying script since they lack programming knowledge. Therefore, even without additional enhancement, models trained with AI-oriented grammar can be effectively utilized in real-world scenarios. We demonstrate this scenario on the left of Figure 4. In this scenario, an AI-oriented code generated by the model can be executed in two ways: 1) being translated into human-centric code and then executed by its executor; 2) directly being executed by a specific executor for the AI-oriented grammar. Notably, implementing an executor specifically for AI-oriented grammar demands only lightweight engineering efforts as the AI-oriented grammar and its original grammar differ only at the syntax level. Thus, the second method offers a more efficient solution.

Table 4: Comparison of average conversion times between Python and SimPy, and the processing speed of the StarCoder tokenizer, based on Huggingface Tokenizers.

Token num	Huggingface		Converter	
	Encode	Decode	To SimPy	To Python
[0, 100)	0.2ms	0.1ms	0.2ms	0.2ms
[100, 500)	0.7ms	0.6ms	0.9ms	0.8ms
[500, 2000)	2.4ms	2.2ms	3.4ms	3.1ms
[2000, 5000)	6.7ms	6.4ms	12.2ms	10.8ms
[5000, +∞)	23.0ms	23.7ms	75.4ms	57.4ms

5.2 Extended usage scenario

Despite the effectiveness of AI-oriented grammar in certain contexts, many code generation scenarios still require the involvement of humans, where human-readable code is required. To fill this gap, we propose an inference framework for code generation named DualCode. DualCode enables human users to interact with code in human-centric grammar, while the model still leverages the efficiency of AI-oriented grammar during the inference process. The fundamental concept of DualCode is to convert the code between AI-oriented grammar and the original grammar of the same programming language. To achieve this goal, a rule-based code converter should be employed to convert source code into AI-oriented grammar for model comprehension and, inversely for user readability. Such a converter is feasible since both the AI-oriented grammar and original grammar describe the same AST. The identical AST allows the code written in the two grammars to be equivalently converted into each other based on the grammar rules.

We illustrate the workflow of DualCode on the right of Figure 4. It employs two "gates": an input converter and an output converter. The input converter translates code written in human-centric grammar into AI-oriented grammar for model processing. Similarly, the output converter reverts AI-generated code into human-readable code for user comprehension. Notably, this environment is only for the code, where other inputs, such as natural language, are unaffected. DualCode is a not complicated framework, enabling the lightweight integration of AI-oriented grammar into existing workflows of AI systems. Though being straightforward, it is proposed and investigated for the first time, bridging the gap between efficient AI-oriented code generation and human readability.

5.3 Experiments of RQ3

Given that the DualCode converter adds extra latency to the inference process, a significant concern arises: excessive latency could render the system impractical for real-world applications. To address the concern, we conduct experiments focusing on the converter's performance. Specifically, we measure the time taken to convert Python code files into SimPy and then back to Python using the converter. As a reference, we evaluate the processing speed of the StarCoder tokenizer, which is based on the widely acknowledged Huggingface Tokenizers library [26]. For this experiment, we categorized Python code files into five distinct groups, based on their token counts, as follows: [0, 100), [100, 500), [500, 2000), [2000, 5000), and [5000, +∞). These token counts are determined using the StarCoder tokenizer [23] on the Python code. We calculate the average processing time for each group.

The findings, presented in Table 4, indicate that the converter's speed is comparable to that of Huggingface Tokenizers. For code files with fewer than 100 tokens, the converter's processing time for each conversion is a mere 0.2 ms, only 0.1 ms slower than the Huggingface Tokenizers. For files containing 100 to 500 tokens, the conversion is completed within 1.0 ms. This is not a significant concern, given that over 95% of the dataset's code files (sourced from real-world repositories) are within the 5000-token range. Therefore, we deduce that the latency induced by the converter is acceptably minimal in most practical scenarios.

Answer to RQ3: Beyond the basic scenarios where human interaction is not required, the application of AI-oriented grammar can be substantially extended by incorporating the DualCode framework. DualCode enables humans to continue using human-centric grammar while AI models leverage the efficiency of AI-oriented grammar. Notably, it imposes negligible latency (under 1 ms for code up to 500 tokens).

6 RELATED WORK

Program Simplification Program simplification has emerged as a valuable approach to enhance the efficiency of code models [5, 18, 33, 34, 39, 49]. This approach typically involves the elimination of less critical code tokens to streamline model processing. For example, DietCode [52] removes the code tokens that receive the fewest attention weights by CodeBert. Sivand [34] and P2IM [53] simplify the input code according to the outputs of a supplementary model. While these methods considerably boost efficiency, they unavoidably compromise accuracy due to the removal of certain code elements. In contrast, models with AI-oriented grammar, though perhaps less efficient, are able to preserve or even improve accuracy. Most importantly, existing simplification techniques are irreversible, limiting their application to code understanding tasks like summarization and retrieval, rather than code generation. Conversely, code in AI-oriented grammar can be effortlessly reverted to its original form, thus suitable for various code-related tasks.

Tokenization of Source Code Modern LLMs usually preprocess textual datasets using an open-vocabulary tokenization method, Byte-Pair Encoding (BPE) [38]. BPE tokenizes text into subwords based on their frequency in the text corpus, offering a balance between the granularity of tokens and vocabulary breadth. Karampatsis et al. [20] first identify the effectiveness of BPE on source code. CodeT5 reveals that BPE trained on source code corpus can reduce over 30% of tokens for code generation, compared with the one trained on natural language corpus. Subsequently, all major LLMs for code generation, such as CodeBERT [12], CodeT5 [47], SantaCoder [2], StarCoder [23] and CodeLlama [37], adopt BPE as the tokenization method. Further enhancements to BPE for source code have been proposed. For example, Chirkova [10] suggests that clustering punctuation characters into single tokens can reduce average token length by 17% without impacting model performance. Notably, even though the tokenizers are optimized for source code, they still need to deal with the unnecessary tokens introduced by the human-centric grammar. AI-oriented grammar optimizes the representation of source code in a more fundamental way, which is orthogonal to these existing tokenization methods.

7 THREATS TO VALIDITY

Constrained Model Selection Our experimental scope in RQ2 is restricted by our computational resources, limiting our evaluation to models with around 1B parameters. These models are relatively modest in scale. However, while the model size is expanding, the fundamental issue of computation waste caused by human-centric code grammar remains unaddressed. Therefore, the insights derived from our experiments with smaller models are still highly relevant for understanding inefficiency issues in larger models.

Limited Programming Language Our research primarily investigates the implementation of AI-oriented grammar in Python, a language widely utilized by existing LLMs for programming tasks. This initial exploration has shown that AI-oriented grammar effectively reduces computational costs during inference. However, the conclusions drawn from Python may not generalize to other programming languages. We thus leave the exploration of its implementation in other languages as future work.

Inefficient Implementation We implement a proof-of-concept converter to convert the code between SimPy and Python. While this converter provides seamless translation, its efficiency is not optimized. For instance, it is developed in Python, which is less efficient compared to languages like C++. This aspect could potentially result in an underestimation of the converter's performance in our experimental evaluations.

8 DISCUSSION

Limitations in practice Though extending the applicability of AI-oriented grammar, DualCode relies on a rule-based converter. The converter, we implemented for SimPy, is AST-based, which implicitly requires the input and output code of LLMs under the DualCode framework to satisfy the grammar correctness. For the output, grammar correctness is a fundamental expectation for a qualified LLM-based assistant. Thus, this requirement from DualCode is not an additional constraint set to the model but aligns with the goal of a reliable AI service. However, it poses challenges when dealing with user-provided input, which may not always be grammatically correct. It is not a concern to models handling natural-language-to-code tasks. However, the requirement may limit the application of SimPy when some tasks involve partial source code as input, such as LLM-based code completion. Addressing this limitation could involve developing an error-tolerant converter or grammar, which is a crucial direction for future research.

Learning the AI-oriented grammar The learning of AI-oriented grammar could be a tricky task. In our experiments, we demonstrate the effectiveness of fine-tuning AI models with SimPy using the next token prediction task. However, this simple fine-tuning strategy requires a large number of SimPy samples, 100% of the dataset in our experiments. A more efficient adaptation process would significantly enhance the utility of AI-oriented grammar. However, current research on how AI models learn code grammar is still limited. Although studies [9, 25, 45] have shown that LLMs typically grasp code grammar knowledge in their initial layers, the exact learning mechanism remains unclear. Therefore, a thorough analysis in this area is much needed.

Utility of AI-oriented grammar In this paper, we demonstrate the effectiveness of the sequential training scheme, where the model is initially trained with the original grammar and then the AI-oriented grammar. It achieves an equivalent, or even improved, performance compared to the model trained merely with the original grammar. Such a training method incurs an increase in the cost of the model training. For example, training CodeGen on the original Python dataset costs 183,628 training steps, and 100,288 additional steps are taken during the further finetuning on the 100% SimPy dataset. Nevertheless, mastering AI-oriented grammar still reduces energy consumption in the long run. Training is performed only once or occasionally, while inference tasks can be continuous and massive after the system is deployed. The post-deployment operational cost is a primary component of the overall cost, sometimes reaching 90% of total expenses [11]. Consequently, despite the additional costs incurred during training, implementing AI-oriented grammar remains highly beneficial from a practical standpoint.

9 CONCLUSION AND FUTURE WORK

In this paper, we, for the first time, propose the concept of AI-oriented grammar to address the inefficiency of AI coders in processing the code written in human-centric grammar. Through an empirical study guided by three research questions, we successfully demonstrate the feasibility and potential of this novel concept. During our research, we have developed the first-ever AI-oriented Python grammar. Additionally, we introduced an inference framework designed to empower models to effectively process both AI-oriented and human-centric grammars.

As an emerging field, AI-oriented grammar presents numerous unexplored questions. For example, an interesting finding from our experiments is that models trained with AI-oriented grammar can even improve the model's accuracy in code generation tasks. This emphasizes the critical role of grammar as a foundational element for LLMs in grasping code semantics. Designing grammars that are inherently more comprehensible to AI models could significantly enhance their performance. Our current research provides a preliminary insight into this aspect, opening doors for in-depth future studies. Additionally, the process of simplifying grammar, as exemplified by our manual creation of SimPy, raises the question of whether an automated approach could create optimal grammar rules for AI models. A potential solution for simplifying the grammar could be iteratively searching for grammar tokens/structures that can be removed with the help of a parser generator. Moreover, saving the training cost for teaching LLMs AI-oriented grammar is also of great practical value, where a more efficient training method for LLMs to learn new programming grammar is urgently needed. We, therefore, call for the software engineering community to engage further with this promising topic, recognizing its potential to revolutionize the field of AI coders.

ACKNOWLEDGMENTS

We thank Dr Zhe Hou for the helpful discussion when preparing the manuscript. This research / project is supported by Xiaoning Du's Google Research Scholar Program Award and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] DeepSeek AI. 2023. DeepSeek Coder: Let the Code Write Itself. https://github.com/deepseek-ai/DeepSeek-Coder.

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[3] Google DeepMind AlphaCode Team. 2023. AlphaCode 2 Technical Report. https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf.

[4] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*. PMLR, 2397–2430.

[5] Nghi D. Q. Bui, Y. Yu, and Lingxiao Jiang. 2019. AutoFocus: Interpreting Attention-Based Neural Networks by Code Perturbation. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), 38–41. https://api.semanticscholar.org/CorpusID:208877064

[6] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A Theory of Dual Channel Constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 25–28. https://doi.org/10.1145/3377816.3381720

[7] Harrison Chase. 2022. *LangChain*. https://github.com/langchain-ai/langchain

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *ArXiv* abs/2107.03374 (2021). https://api.semanticscholar.org/CorpusID:235755472

[9] Nuo Chen, Qiushi Sun, Renyu Zhu, Xiang Li, Xuesong Lu, and Ming Gao. 2022. CAT-probing: A Metric-based Approach to Interpret How Pre-trained Models for Programming Language Attend Code Structure. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:252780909

[10] Nadezhda Chirkova and Sergey Troshin. 2023. CodeBPE: Investigating Subtokenization Options for Large Language Model Pretraining on Source Code. *ArXiv* abs/2308.00683 (2023). https://api.semanticscholar.org/CorpusID:252600018

[11] Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. 2023. Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning. *Sustainable Computing: Informatics and Systems* 38 (2023), 100857.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *ArXiv* abs/2002.08155 (2020). https://api.semanticscholar.org/CorpusID:211171605

[13] Seymour Ginsburg and Joseph S. Ullian. 1966. Ambiguity in context free languages. *J. ACM* 13 (1966), 62–89. https://api.semanticscholar.org/CorpusID:5851601

[14] Google. 2023. ChatGPT. https://bard.google.com/.

[15] Dick Grune and Ceriel J. H. Jacobs. 2007. Parsing Techniques - A Practical Guide. In *Monographs in Computer Science*. https://api.semanticscholar.org/CorpusID:33077869

[16] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE]

[17] https://www.wsj.com/tech/ai/ais-costly-buildup-could-make-early-products-a-hard-sell-bdd29b9f [n. d.].

[18] Yufan Huang, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin B. Clement, and Neel Sundaresan. 2023. Program Translation via Code Distillation. *ArXiv* abs/2310.11476 (2023). https://api.semanticscholar.org/CorpusID:264289043

[19] Jared Kaplan, Sam McCandlish, T. J. Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeff Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *ArXiv* abs/2001.08361 (2020). https://api.semanticscholar.org/CorpusID:210861095

[20] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 1073–1085. https://api.semanticscholar.org/CorpusID:211161525

[21] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[22] Bernard Lang. 1974. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *International Colloquium on Automata, Languages and Programming*. https://api.semanticscholar.org/CorpusID:27069587

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *ArXiv* abs/2305.06161 (2023). https://api.semanticscholar.org/CorpusID:258588247

[24] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *ArXiv* abs/2306.08568 (2023). https://api.semanticscholar.org/CorpusID:259164815

[25] Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiexin Zhang, Wenhan Wang, and Yang Liu. 2022. Is Self-Attention Powerful to Learn Code Syntax and Semantics? *ArXiv* abs/2212.10017 (2022). https://api.semanticscholar.org/CorpusID:254877330

[26] Anthony Moi and Nicolas Patry. 2023. *HuggingFace's Tokenizers*. https://github.com/huggingface/tokenizers

[27] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations*. https://api.semanticscholar.org/CorpusID:252668917

[28] OpenAI. 2023. ChatGPT. https://chat.openai.com/.

[29] OpenAI. 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5.

[30] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023). https://api.semanticscholar.org/CorpusID:257532815

[31] Tim Peters. 2023. PEP 20 – The Zen of Python. https://peps.python.org/pep-0020/.

[32] Python. 2023. Full Grammar specification. https://docs.python.org/3/reference/grammar.html.

[33] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021). https://api.semanticscholar.org/CorpusID:235359051

[34] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (2022). https://api.semanticscholar.org/CorpusID:249191662

[35] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[36] Repilt. 2023. ReplitLM. https://github.com/replit/replitLM.

[37] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *ArXiv* abs/2308.12950 (2023). https://api.semanticscholar.org/CorpusID:261100919

[38] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural Machine Translation of Rare Words with Subword Units. *ArXiv* abs/1508.07909 (2015). https://api.semanticscholar.org/CorpusID:1114678

[39] Chaoxuan Shi, Tingwei Zhu, Tian Zhang, Jun Pang, and Minxue Pan. 2023. Structural-semantics Guided Program Simplification for Understanding Neural Code Intelligence Models. *Proceedings of the 14th Asia-Pacific Symposium on Internetware* (2023). https://api.semanticscholar.org/CorpusID:263672536

[40] Significant Gravitas. [n. d.]. *AutoGPT*. https://github.com/Significant-Gravitas/AutoGPT

[41] Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti

Bhosale, Daniel M. Bikel, Lukas Blecher, Cristian Cantón Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony S. Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel M. Kloumann, A. V. Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, R. Subramanian, Xia Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zhengxu Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *ArXiv* abs/2307.09288 (2023). https://api.semanticscholar.org/CorpusID:259950998

[42] tree sitter. 2023. tree-sitter/tree-sitter: An incremental parsing system for programming tools. https://github.com/tree-sitter/tree-sitter.

[43] Sergey Troshin and Nadezhda Chirkova. 2022. Probing Pretrained Models of Source Codes. *ArXiv* abs/2202.08975 (2022). https://api.semanticscholar.org/CorpusID:246996634

[44] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. 2023. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/.

[45] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hairong Jin. 2022. What Do They Capture? - A Structural Analysis of Pre-Trained Language Models for Source Code. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), 2377–2388. https://api.semanticscholar.org/CorpusID:246823289

[46] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *ArXiv* abs/2305.07922 (2023). https://api.semanticscholar.org/CorpusID:258685677

[47] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv* abs/2109.00859 (2021). https://api.semanticscholar.org/CorpusID:237386541

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

[49] Zhou Yang, Zhensu Sun, Terry Yue Zhuo, Prem Devanbu, and David Lo. 2024. Robustness, Security, Privacy, Explainability, Efficiency, and Usability of Large Language Models for Code. *ArXiv* abs/2403.07506 (2024). https://api.semanticscholar.org/CorpusID:268364103

[50] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-Trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/3533767.3534390

[51] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. arXiv:2401.02385 [cs.CL]

[52] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: simplifying programs for pre-trained models of code. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022). https://api.semanticscholar.org/CorpusID:250113729

[53] Yunhui Zheng, Sahil Suneja, Yufan Zhuang, Alessandro Morari, and Jim Laredo. 2020. Probing model signal-awareness via prediction-preserving input minimization. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020). https://api.semanticscholar.org/CorpusID:227227733