GrammarT5: Grammar-Integrated Pretrained Encoder-Decoder Neural Model for Code

Qihao Zhu
Key Laboratory of HCST (PKU), MoE
SCS, Peking University
Beijing, China
Zhuqh@pku.edu.cn

Qingyuan Liang
Key Laboratory of HCST (PKU), MoE
SCS, Peking University
Beijing, China
liangqy@stu.pku.edu.cn

Zeyu Sun
Science & Technology on Integrated Information System Laboratory,
Institute of Software, Chinese Academy of Sciences
Beijing, China
szy_@pku.edu.cn

Yingfei Xiong*
Key Laboratory of HCST (PKU), MoE
SCS, Peking University
Beijing, China
xiongyf@pku.edu.cn

Lu Zhang
Key Laboratory of HCST (PKU), MoE
SCS, Peking University
Beijing, China
zhanglucs@pku.edu.cn

Shengyu Cheng
ZTE Corporation
Chengdu, China
cheng.shengyu@zte.com.cn

ABSTRACT

Pretrained models for code have exhibited promising performance across various code-related tasks, such as code summarization, code completion, code translation, and bug detection. However, despite their success, the majority of current models still represent code as a token sequence, which may not adequately capture the essence of the underlying code structure.

In this work, we propose GrammarT5, a grammar-integrated encoder-decoder pretrained neural model for code. GrammarT5 employs a novel grammar-integrated representation, Tokenized Grammar Rule Sequence (TGRS), for code. TGRS is constructed based on the grammar rule sequence utilized in syntax-guided code generation and integrates syntax information with code tokens within an appropriate input length. Furthermore, we suggest attaching language flags to help GrammarT5 differentiate between grammar rules of various programming languages. Finally, we introduce two novel pretraining tasks—Edge Prediction (EP), and Sub-Tree Prediction (STP) to learn syntactic information.

Experiments were conducted on five code-related tasks using eleven datasets, demonstrating that GrammarT5 achieves state-of-the-art (SOTA) performance on most tasks in comparison to models of the same scale. Additionally, the paper illustrates that the proposed pretraining tasks and language flags can enhance GrammarT5 to better capture the syntax and semantics of code.

*Corresponding author.

HCST: High Confidence Software Technologies.
SCS: School of Computer Science.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639125

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

KEYWORDS

neural networks, pretrained model, text tagging

ACM Reference Format:
Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. 2024. GrammarT5: Grammar-Integrated Pretrained Encoder-Decoder Neural Model for Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639125

1 INTRODUCTION

Recently, the field of artificial intelligence has experienced noticeable advancements, largely driven by the development and deployment of pretrained models. Pretrained models are trained with self-supervised tasks over a large corpus of data. In both the domains of natural language processing [6, 24, 30, 31] and programming language processing [8, 10, 11, 25, 37, 41, 43], large pretrained language models such as T5 [31] and CodeT5 [43] have significantly outperformed the existing non-pretrained models.

However, most existing pretrained models on programming languages follow the models on natural languages and treat a program as a token sequence, i.e., a unique integer ID is assigned for each token in the training set and a program is represented as a sequence of such IDs. Different from natural languages, programming languages have a well-defined syntactic structure and usually follows a context-free grammar. Representing programs as token sequences make the syntactic structure implicitly, potentially jeopardizing the understanding of syntactic structures and permitting the generation of syntactically incorrect code.

To overcome this problem, multiple existing approaches [29, 35, 36, 44, 46] represent programs as grammar rule sequences. These approaches assign a unique ID to each grammar rule, parse the program as an AST, traverse the AST in a certain order (e.g., pre-order), and record the ID of the grammar rule used to expand each traversed non-terminal as a sequence to represent the program.

This representation makes the syntactic structure explicit, making it easier for the model to understand the syntactic structure. When generating code with this representation, it is ensured that no syntactic error will be introduced. Consequently, this representation leads to better performance: models adopting the representation of grammar rule sequences achieve the state-of-the-art performance in multiple benchmarks among non-pretrained models [49].

Despite being successful in non-pretrained models, the representation of grammar rule sequences has never been used in pretraining models as far as we are aware. Therefore, in this paper we ask a question: *Can we represent programs as grammar rule sequences in pretrained models?*

Answering this question is not easy, as there are several technical challenges of adapting this representation to pretrained models. **The first challenge is the big vocabulary.** In a typical grammar of a programming language, some terminals such as ⟨*identifier*⟩ or ⟨*constants*⟩ represent many possible lexical tokens, called *multi-value terminals*. Existing approaches collect the tokens represented by multi-value terminals in the training set, and add a grammar rule for each collected token, such as *identifier* → *isodd*. However, in pretraining, the training set is much larger, adding such rules would lead to a too big vocabulary. Existing pretraining models use Byte Pair Encoding (BPE) [34] to find a relatively small set of subtokens whose concatenations could represent the large token set. However, how to integrate BPE with grammars is still unclear. **The second challenge is heterogeneous grammars.** An existing non-pretrained model deals with only one programming language, but a pretrained model is typically trained over multiple programming languages, and how to deal with grammars from different programming languages is yet unknown. **The third challenge is pretraining tasks.** Pretraining requires self-supervised tasks for the training. It is yet unknown what pre-training tasks should be used when programs are represented as grammar rule sequences. Especially, since the syntactic structure is explicitly represented, the pretraining tasks should guide the model to learn the syntactic structure of the code.

In this work, we propose GrammarT5, a grammar-integrated encoder-decoder pretrained model for programming languages to support both code understanding and generation tasks. GrammarT5 uses a variant of grammar rule sequence to represent the code, called *Tokenized Grammar Rule Sequence (TGRS)*. To address the first challenge, **our first contribution** is an integration of BPE with the representation of grammar rule sequence. BPE uses a set of sub tokens to represent the original tokens. For example, `is` and `odd` may be used to represent `isodd`. Then, we can extend the grammar with the following rules.

$$\textit{identifier} \rightarrow \texttt{is}\ \textit{identifier} \mid \texttt{odd}\ \textit{identifier} \mid \cdots$$
$$\textit{identifier} \rightarrow \varepsilon$$

However, this grammar introduces an additional node ε for each leaf node with a multi-value terminal in the AST, non-trivially increasing the length of the representation. Since compact representation often leads to better model performance, we rewrite the second grammar rule by adding a special flag "#" to the last sub-word:

$$\textit{identifier} \rightarrow \texttt{\#odd} \mid \texttt{\#number} \cdots$$

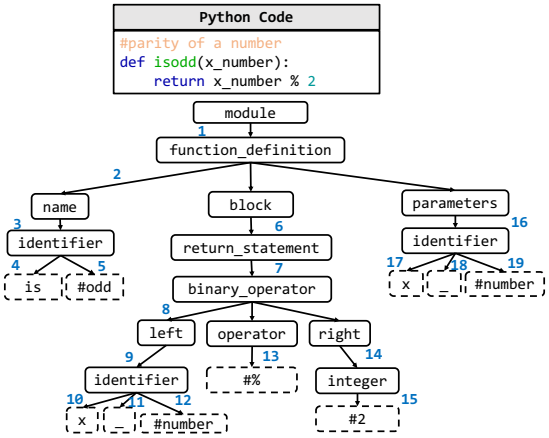

Figure 1: A Python code snippet with its tokenized AST.[1]

Figure 1 shows an example of the AST in our new grammar, called *tokenized AST*. As we can see from the figure, there are five nodes starting with "#", indicating that the original representation will increase the size by 5/19=26%.

To tackle the second challenge, **our second contribution** is an empirical study of the effect of combining the grammar rules of different programming languages. The main consideration is that different programming languages may have the same grammar rules and can be shared. Since context-free languages are closed under union, a possible approach is to build a new grammar by combining all grammars and sharing grammar rules as much as possible. While this approach may reduce the total number of grammar rules, it might impose a challenge for the neural model to learn the syntactical information from this mixed grammar. Another possible approach is to share no grammar rules. To achieve this, we attach a special language flag to non-terminals for each programming language in order to disable sharing. For instance, the symbol *return_statement* in Figure 1 would be modified to *return_statement@py*. We empirically compare the two approaches, and find the latter leads to higher performance.

To overcome the third challenge, **our third contribution** is two novel pre-training tasks, *Edge Prediction (EP)* and *Sub-Tree Prediction (STP)* for GrammarT5 to learn the structural information of the AST. First, EP requires the model to predict the parent node in the decoder step by step, given the TGRS in the encoder. Second, STP is inspired by Masked Span Prediction, a denoising task for token sequences. This task is to predict the randomly masked spans in the input sequence with arbitrary lengths. Due to its randomness, the masked spans can potentially destroy the structural integrity of the input code. Hence, we propose STP, which randomly masks several sub-trees in the AST. STP requires GrammarT5 to regenerate the masked sub-tree based on the surrounding context to learn the code dependency. These two pre-training task guides the model to capture the syntactic structure of the code effectively.

Our fourth contribution is a series of experiments to thoroughly evaluate the performance of GrammarT5 on code-related tasks. The experiments took in total 50 days, and were conducted

[1]For better illustration, we omit some *identifier* nodes in the AST.

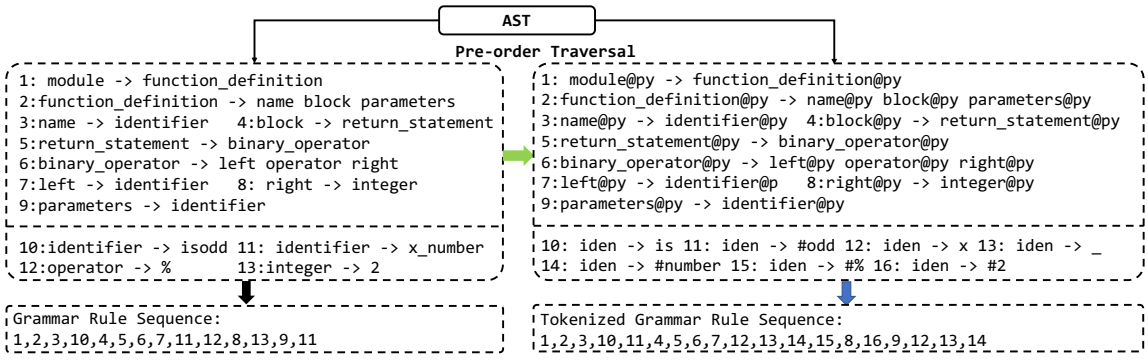

Figure 2: A Python code snippet represented by the grammar rule list and TGRS.

on five tasks over 11 datasets, including two understanding tasks: code search and comment generation, three generation tasks: code generation, code translation, and code refinement. To compare with the SOTA pretrained model of the same scale, CodeT5 [43], we use a subset of the training set of CodeT5 to train GrammarT5, and set all hyper-parameters the same as CodeT5-base. The results show that GrammarT5 achieves state-of-the-art performance on most of the tasks compared with models of the same scale, including CodeT5-base. Moreover, GrammarT5 also exhibits a competitive performance compared with CodeT5-large, a 3x larger model. Further analysis reveals that all the above technical novelties of GrammarT5 enhance its performance.

2 RELATED WORK

2.1 Pretrained Models for Code

Recent advances in pretrained NLP models have inspired the development of numerous pretrained models [1, 8, 10, 11, 19, 25, 37, 41, 43] for programming languages. However, most of these models represent code as token sequences, potentially hindering the learning of the syntactic structure and not ensuring the syntactic correctness during code generation.

Realizing this problem, a few existing models (SynCoBERT [41], TreeBert [19], and UniXcoder [10]) try to explicitly capture the syntactic structure by representing code as AST sequences, which are obtained by traversing the AST in pre-order and recording the symbol of each traversed node. Since a non-terminal may be expanded by multiple grammar rules, it is hard to recover the tree structure by recording only the symbol. These approaches add extra nodes to maintain the tree structure. For example, the sub-tree of the node *name* in Figure 1 can be represented as *name, identifier, isodd, bt, bt, bt*, where *bt* is added upon completing each sub-tree traversal. The program has a length of 48 when represented as an AST sequence, and has a length of 19 when represented as a grammar rule sequence. The AST representation results in significantly longer sequences, incurring substantial GPU memory overhead and potentially lowering model performance. As our evaluation will show later, GrammarT5 outperforms all these models. Furthermore, all the existing models use AST representation only in the encoder and cannot ensure syntactic correctness in code generation.

2.2 Grammar-Integrated Code Generation

Several non-pretrained code generation models [29, 35, 36, 44, 46] have used grammar rule sequences to represent code. The code generation task can be modeled as a series of classification problems of grammar rules, by parsing the programs as AST and decomposing into several context-free grammar rules.

For the AST of a Python code snippet in Figure 1, We focus on the sub-tree for the statement "return x_number % 2", where dotted boxes represent terminal symbols and solid boxes represent non-terminal symbols. The process of AST-based code generation involves the iterative expansion of non-terminal nodes using grammar rules until only terminal nodes are left. The first non-terminal to be expanded is "return_statement", which is expanded by the grammar rule "return_statement → binary_operator". Following a pre-order traversal, the next node to expand is "binary_operator", with the corresponding rule being "binary_operator → left operator right". In this way, the statement is depicted as a list of these grammar rules. By assigning a unique ID to each grammar rule, we can represent the statement as a list of these IDs. The number list representation of the program is illustrated in Figure 2.

The integration of grammar rules boosts the performance. However, non-pretrained models still lag behind pre-trained ones. GrammarT5, inspired by these advances, incorporates novel techniques to address the challenges of grammer-integrated pretrained models as outlined in the introduction, and bridge the performance gap.

3 GRAMMART5

GrammarT5 is a grammar-integrated pretrained model for multimodal data (programming language (PL) and natural language (NL)) for code understanding and generation. The model is based on the encoder-decoder framework as T5 [31], and aims to generate generic representations given the input composed of PL and NL.

3.1 TGRS

As introduced by the syntax-guided decoder [36, 44, 46, 48], each program can be represented as a sequence of grammar rules. TGRS first parses the program into an AST and obtains the grammar rules by traversing the AST. For example, Figure 2 shows the grammar rule sequence of the code snippet illustrated in Figure 1.

3.1.1 Terminal Tokenization. As shown in Figure 2, the grammar rule sequence in the existing syntax-guided decoder approaches [36,

44, 46, 48] directly represents the identifiers in the AST using unique grammar rules, such as *identifier* → *isodd*. These approaches are all experimented on small training sets, they are not exposed to the potential big vocabulary problem. However, when it moves to the pretraining scenarios with a large code corpus, the big vocabulary problem will be exposed due to the significant number of identifiers.

Approaches such as those proposed by Karampatsis et al. [20] and Wang et al. [43] alleviate the issue of representing code as token sequences by employing the BPE (Byte Pair Encoding) algorithm [28] for pretraining. This algorithm enables words to be represented as sub-word lists. However, a drawback of this method is the resulting lack of syntactic information.

To leverage the benefits of both approaches, we tokenize all terminals in the grammar and obtain the tokenized AST via the grammar extended with new rules for these terminals. For example, the following grammar rules are added for the code in Figure 1, where each terminal is tokenized into sub-tokens (i.e., "isodd"/"x_number" is tokenized to "is" and "odd"/"x", "_", and "number").

$$\textit{identifier} \rightarrow \texttt{is}\ \textit{identifier} \mid \texttt{odd}\ \textit{identifier} \mid \cdots$$
$$\textit{identifier} \rightarrow \texttt{\#odd} \mid \texttt{\#number} \cdots$$
$$\textit{operator} \rightarrow \texttt{\#\%} \quad \textit{integer} \rightarrow \texttt{\#2}$$

To identify the last sub-word, we attach a special flag "#" to it. TGRS is derived by traversing the tokenized AST in pre-order.

In Figure 2, we illustrate how the "parameters" sub-tree can be transformed into four distinct rules: *parameters*→*identifier*, *identifier*→x *identifier*, *identifier*→_ *identifier*, and *identifier* → #number. Each grammar rule is assigned a unique ID. For example, *parameters*→*identifier* is represented as 9. Therefore, the code token "x_number" is represented as a sequence of numbers: 12, 13, 14. These numbers are subsequently converted into real-valued vectors through word embedding. These vectors serve as the input for a neural network. This method effectively compresses the length of the code input, making it more manageable for the models to process, while still retaining essential syntactical information.

3.1.2 Language Flag. GrammarT5 is designed to process code from various programming languages. Different programming languages may share identical grammar rules. For example, both Java and Python grammars contain the rule *name* → *identifier*. Directly combining grammars from different programming languages might hinder learning the unique syntax information. To investigate the effect, we conduct an empirical study by training two separate models. One model uses the grammar of tree-sitter [38], which is a parser generator tool for all mainstream languages. In the grammars defined in tree-sitter, many rules are already shared among different programming languages. The other uses an extended grammar, attaching a specific programming language flag to the non-terminals in the tree-sitter grammars. To prevent introducing too many extra grammar rules, we only modify the original non-terminals but not those we introduced for multi-value terminals. As depicted in Figure 2, each non-terminal in the grammar rule is appended with a flag *@py* to identify the language. We evaluate the two models on several downstream tasks, and find the latter one achieves higher performance. We will clarify these results in the evaluation section.

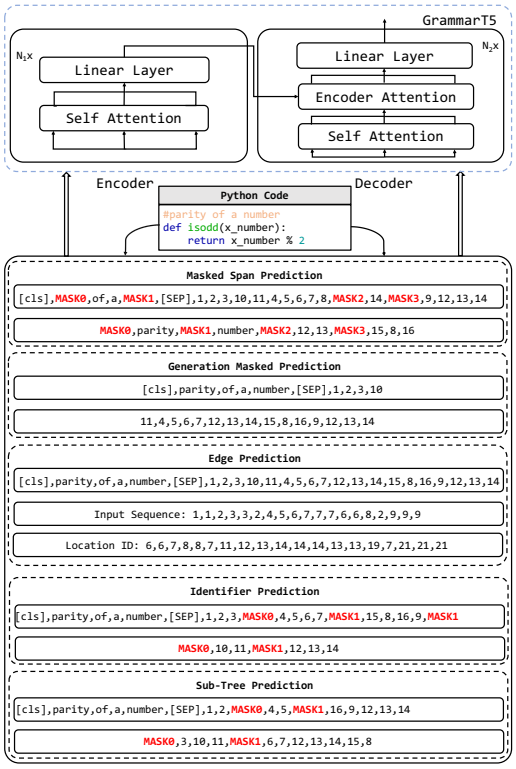

Figure 3: Overview of GrammarT5.

3.1.3 Format of TGRS. Following the previous work [36, 44, 46, 48], we assign each production rule in the extended grammar a unique ID as shown in Figure 2. The TGRS is formulated as a list containing these unique IDs, where each one corresponds to a particular grammar rule, permitting individual recognition and reference of each rule.

3.2 Input Representation

Based on TGRS, we describe the input format of the multi-modal data for GrammarT5, which would take either code or code comment as input. Therefore, there are two formats of the input, PL-only and NL-PL, depending on whether the code snippet has a corresponding comment. Given a code snippet c and its corresponding NL comment w, GrammarT5 first transforms c into a TGRS, denoted as $c_1, c_2, \cdots, c_n$. Then, GrammarT5 converts w into a token sequence, denoted as $w_1, w_2, \cdots, w_m$, using BPE. Finally, GrammarT5 takes the concatenation of these two parts as input:

$$\boldsymbol{x} = \{[\text{CLS}], w_1, w_2, \cdots, w_m, [\text{SEP}], c_1, c_2, \cdots, c_n, [\text{SEP}]\} \quad (1)$$

where n and m denote the length of the TGRS and NL token sequence, respectively. [CLS] is a special token to identify the sequence beginning, and [SEP] is a special token to split two types of sub-sequences. The NL word sequence is empty for PL-only inputs.

3.3 Model Architecture

Figure 3 displays the model architecture of GrammarT5. GrammarT5 adopts an encoder-decoder framework similar to T5 for

input processing. Each component comprises N transformer blocks based on a self-attention layer. In particular, the self-attention layer is the same as the one used in Transformer [40]. This layer inputs three embeddings, $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}$, and outputs the combined embeddings, $\boldsymbol{o}$, based on the attention score computed by the inputs. In the decoder, each block has an additional encoder-decoder attention layer compared with the encoder. Its computation can be represented as $EncAtt = Att(\boldsymbol{b}, \boldsymbol{e}, \boldsymbol{e})$, where $\boldsymbol{b}$ denotes the output of the previous layer in the decoder and $\boldsymbol{e}$ denotes the output of the encoder.

3.4 pretraining Tasks

In this section, we describe the pretraining tasks used in GrammarT5. As shown in Figure 3, we pretrain GrammarT5 with 5 self-supervised tasks over multi-modal data, including three normal pretraining tasks used in pretrained models and two novel proposed denoising objectives. These tasks are designed to enable GrammarT5 to learn syntactic and semantic information from either PL-only or NL-PL bimodal data.

3.4.1 Masked Span Prediction. Denoising pretraining objective has been shown to be quite effective for encoder-decoder models, such as PLBART and CodeT5. This objective typically first poisons the original sequence with some noising function and then requires the model to recover the sequence. One of the most used denoising objectives is **masked span prediction (MSP)**. This task randomly masks spans in the input sequence with arbitrary lengths. The model should generate these masked spans based on the corrupt input. Inspired by this phenomenon, we utilize a similar denoising objective on the multi-modal data as illustrated in Figure 3.

Specifically, we use the same masked rate, 15%, as the previous work [31, 43] in our current implementation. Moreover, we control the average length of the masked length to 3 via uniformly sampling spans from 1 to 5 tokens. Then, we concatenate these masked spans as the output separated by several special tokens $MASK_i$, where i denotes the ID of the span as shown in Figure 3. We represent it as $\boldsymbol{y} = \{y_0, y_1, \cdots, y_n\}$. Thus, the loss of the masked span prediction can be computed as

$$\mathcal{L}_{MSP}(\theta) = \sum_{i=1}^{n} -log\mathcal{P}_\theta(y_i|\boldsymbol{x}^{mask}, \boldsymbol{y}_{t<i}) \tag{2}$$

where θ is the trainable parameters, $\boldsymbol{x}_{mask}$ is the corrupted input sequence, and $\boldsymbol{y}_{t<i}$ denotes the generated sequence so far.

3.4.2 Generation Masked Prediction. Although the MSP task benefits the code understanding tasks, it differs significantly from code generation objectives, which require generating the whole sequence. To address this issue, we adopt a similar pretraining objective used in decoder-only models [30] for GrammarT5.

Specifically, we randomly select a pivot location in the input sequence. Then, given the preceding sequence, GrammarT5 predicts the succeeding sequence, as shown in Figure 3. In our current implementation, we ensure that the pivot location falls between 10% and 90% of the input sequence to control the sequence length. The loss of this pretraining task can be represented as:

$$\mathcal{L}_{GMP}(\theta) = \sum_{i=1}^{|x^{suc}|} -log\mathcal{P}_\theta(x_i|\boldsymbol{x}^{pro}, \boldsymbol{x}^{suc}_{t<i}) \tag{3}$$

where $\boldsymbol{x}^{suc}$ denotes the succeeding sequence and $\boldsymbol{x}^{pro}$ denotes the proceeding sequence.

3.4.3 Masked Identifier Prediction. The importance of symbolic information in programming languages is critical for understanding code semantics, especially identifiers in code. Accordingly, we adopt a denoising objective, masking all identifiers in the code following CodeT5 [43]. We replace all instances of the i-th identifier in the input sequence with a unique sentinel token $MASK_i$. Then, we construct the target sequence by concatenating all unique identifiers with their sentinel tokens (as depicted in Figure 3). The decoder then predicts the target sequence in an auto-regressive manner from the corrupted sequence. The loss is computed as:

$$\mathcal{L}_{IP}(\theta) = \sum_{i=1}^{|\boldsymbol{y}|} -log\mathcal{P}_\theta(y_i|\boldsymbol{x}^{MI}, \boldsymbol{y}_{t<i}) \tag{4}$$

where $\boldsymbol{x}^{MI}$ is the masked input sequence and $\boldsymbol{y}_{t<i}$ denotes the sequence generated so far.

To help GrammarT5 learn the code-specific structural information, we propose two additional pretraining tasks: *Edge Prediction* and *Sub-tree Prediction*.

3.4.4 Edge Prediction. When converting a code snippet into a TGRS, some crucial structural information might get lost. Existing approaches propose the edge masking technique for encoder-only models [11, 41] to predict the masked edge via the attention score. Inspired by these approaches, we propose an edge prediction objective for GrammarT5. The decoder should predict the parent rule of each rule based on the given sequence. Here, we use a pointer network to predict the location of the parent rule in the original sequence. Especially, we use the parent rule sequence as the input of the decoder. Then, GrammarT5 should output the location of the parent rule based on the parent rule sequence as shown in Figure 3.

Given the output of the encoder $\boldsymbol{o}$, the output of the decoder $\boldsymbol{d}$, and the parent rule location sequence $\boldsymbol{l}$, the loss is computed as:

$$\mathcal{L}_{EP}(\theta) = \sum_{i=1}^{|\boldsymbol{x}|} -log(p_{i,l_i}), p_{i,j} \quad = \frac{exp(\boldsymbol{o}_j \boldsymbol{d}_i)}{\sum_{k=1}^{n} exp(\boldsymbol{o}_k \boldsymbol{d}_i)} \tag{5}$$

3.4.5 Sub-Tree Prediction. Since code possesses a tree-like structure, the most prevalent denoising objective, masked span prediction, indiscriminately masks spans within the sequence, potentially compromising its structural integrity. To address this issue, we propose a new denoising objective, Sub-Tree Prediction, which considers the tree structure of the code.

To corrupt the original sequence, this objective randomly masks several sub-trees and uses a special token $MASK_i$ to replace the TGRS of the sub-trees. In our current implementation, each sub-tree has a consistent masking rate of 15%. To control the input and output lengths, we restrict the length of each masked sub-tree to a range of 10-60 and the total length of the masked sub-trees constitutes less than 15% of the input sequence's length. To create the output sequence, we concatenate all the masked sub-trees with the special token. The loss for this objective can be computed as:

$$\mathcal{L}_{SP}(\theta) = \sum_{i=1}^{|\boldsymbol{y}|} -log\mathcal{P}_\theta(y_i|\boldsymbol{x}^{SP}, \boldsymbol{y}_{t<i}) \tag{6}$$

where $\boldsymbol{x}^{SP}$ denotes the masked input. In this objective, the model needs to reconstruct the masked sub-tree using the context, possibly aiding it in assimilating the AST's structural information.

3.4.6 Aggregation. Our pretraining methodology involves cycling through the five different tasks. At every step, a task is randomly selected from this pool, ensuring that each task has an equal chance of being selected and contributes equally to the model's learning. The total loss of the pretraining process can be expressed as:

$$\mathcal{L}(\theta) = \sum_{i=1}^{5} p_i \mathcal{L}_i(\theta), \boldsymbol{p} = OneHot(Random(1,5)) \tag{7}$$

where $\mathcal{L}_i$ denotes the five pretraining tasks above and $\boldsymbol{p}$ denotes a one-hot vector based on a random integer.

4 EXPERIMENT SETUP

4.1 Research Question

Our experiment aims to answer these research questions:

RQ1: How well does GrammarT5 perform compared with the existing pretrained models? To answer this question, we compare GrammarT5 with the existing pre-trained models on 5 code-related tasks, containing 11 benchmarks.

RQ2: How does TGRS representation affect the model performance? To answer this question, we train a modified model, where we replace the code representation of CodeT5 with TGRS, for comparing TGRS and token sequences. Furthermore, we compare the average length of the code in the pretraining dataset represented by TGRS with token sequences and AST sequences, the two representations used in existing pretrained models.

RQ3: How does the inclusion of language flags in grammar rules affect the model performance? To answer this question, we train an ablated model with the combined grammar of tree-sitter without language flags (such that the common rules in different programming languages are shared) and compare it with the original version of GrammarT5 on the benchmarks.

RQ4: How do the proposed pre-training tasks affect the model performance? To answer this question, we pre-train two ablated versions of GrammarT5 and compare them with the original version of GrammarT5 on the benchmarks.

RQ5: How does GrammarT5 generalize to token sequences? We assess the adaptability of GrammarT5 to tasks like code completion, involving unparseable partial code. To answer this question, we conduct an evaluation of GrammarT5 on code completion and code translation, treating the unparsable code as token sequence.

4.2 Pretraining Dataset

To eliminate the possible effect of training sets on model performance, we choose the training set of CodeT5 [43], a SOTA pretrained model at its scale, for fair comparison with CodeT5. However, due to time and computational resource constraints, we can only use a subset of the training set to train GrammarT5. Please note that this setting favors CodeT5, as it is generally believed that more training data leads to better model performance.

CodeT5 is trained on the CodeSearchNet dataset [16] and the C and CSharp data from the GitHub code dataset [5]. We choose Java and Python data from CodeSearchNet and CSharp data from the GitHub code dataset to pretrain GrammarT5. Table 1 shows the detailed statistics of the utilized pretraining dataset. In total, we use approximately 3.64 million instances for pretraining GrammarT5.

Table 1: Statistics of the pre-training datasets we used.

	CodeSearchNet		GithubCode	Total
Statistics	**Java**	**Python**	**CSharp**	-
# W/ NL	457,380	453,750	422,457	1,333,587
# W/o NL	1,070,265	656,990	581,873	2,309,128
#Rule	963	1105	1913	3981
Total	1,527,645	1,110,740	1,004,330	3,642,715

The line "#W/(o)NL" represents the number of instances with(out) natural language descriptions, and "#Rule" refers to the number of grammar rules in the respective language.

To transform code into the TGRS, we adopt the tree-sitter [38] to convert the code into an AST and then extract the corresponding TGRS. We filter out the code snippets that cannot be parsed into a valid AST from the original dataset.

4.3 Model Configurations

Following the model configuration of CodeT5 for consistency, we construct GrammarT5 using the publicly available PyTorch [7] implementation of T5 in the Huggingface Hub [15]. CodeT5 has three versions of different sizes. Given the computational resource constraints, we implement the two smaller versions for GrammarT5: GrammarT5-small (60M) and GrammarT5-base (220M). All these models are trained from scratch.

The hyperparameters are directly adopted from CodeT5. During the pretraining phase, we set the maximum lengths of the source and target sequences to 512 and 256, the batch size to 2880, and a peak learning rate of $2e^{-4}$ with linear decay. GrammarT5 is pretrained using the five self-supervised tasks for 100 epochs in total. During the fine-tuning phase, we employ a grid search and select the best hyper-parameters based on the validation set following CodeT5. We also directly use the trained BPE tokenizer from CodeT5. To expedite the pre-training process, we employ Accelerate [14] and DeepSpeed [32] with BF16 to train GrammarT5.

All experiments were conducted on two Dell workstations. Each workstation is equipped with 300 GB RAM, Intel Xeon CPU E5-2680 v4 @ 2.40 GHz, and eight 24 GB GeForce RTX 3090 GPUs, running Ubuntu 16.04.6 LTS[1]. The total experiments took 50 days, with 40 days for pretraining and 10 days for finetuning the model for the downstream tasks.

4.4 Downstream Tasks and Metrics

To assess the performance of GrammarT5, we adopt CodeXGLUE benchmark [25], a benchmark dataset and open challenge for code intelligence. It includes a collection of code understanding and generation tasks for model evaluation and comparison. For a fair comparison, we use the same data splits following the previous work [25] for all these tasks. Moreover, we consider four additional

[1]The code is available in https://github.com/GrammarT5/GrammarT5.

Table 2: Statistics of the fine-tuning datasets we used.

	Code Summarization		Code Search		NL-Based Code Generation					Code Refinement		Code Translation
Statistics	**Sum-Java**	**Sum-Python**	**Adv**	**CosQA**	**CONCODE**	**Django**	**Conala**	**MBPP**	**MathQA-Python**	**Refine-S**	**Refine-M**	**Codetrans**
# Train	164,923	251,820	251,820	20,000	100,000	16,000	2,379	374	19208	46,680	52,364	10,300
# Dev	5,183	13,914	9,604	604	2,000	998	-	90	2822	5,835	6,545	500
# Test	10,955	14,918	19,210	500	2,000	1,805	500	500	1882	5,835	6,545	1,000

code generation benchmarks—MBPP [2], Django [27], MathQA-Python [2], and Conala [45]—to further assess the code generation capabilities of GrammarT5. The statistics of these datasets are shown in Table 2. We execute GrammarT5 on these benchmarks five times, each time using a different random seed, to ensure robust results. These downstream tasks are divided into two categories, code understanding and code generation.

4.4.1 Code Understanding. In this part, we focus on two cross-modal downstream tasks: code summarization and code search.

Code summarization aims to generate a NL description given a function-level code snippet. The dataset in CodeXGLUE consists of 6 programming languages. In this experiment, we select the subset of Java and Python to experiment with GrammarT5. Following the existing work [43], we use the smoothed BLEU-4 (Bilingual Evaluation Understudy with 4-grams) score [23] to evaluate the performance. The BLEU score measures the quality of the generated text via calculating the geometric mean of n-gram precision scores compared with the ground truth.

Code search aims to identify the most semantically relevant code snippets based on a natural language functional description. We conduct experiments on two datasets, namely AdvTest [25] and CosQA [13]. AdvTest is constructed from the Python sub-set of CodeSearchNet, filtering the low-quality queries. The test set normalizes Python functions and identifiers to better evaluate model generalization capabilities. CosQA's code base is also derived from the CodeSearchNet corpus, with natural language queries collected from Microsoft Bing search engine logs. We use the Mean Reciprocal Rank (MRR) for evaluation in this task. MRR is a metric for evaluating ranking tasks. It calculates the average of the reciprocal ranks of the first correct answers. The higher the MRR, the better the model is at ranking relevant answers higher.

4.4.2 Code Generation. In code generation, we primarily focus on three related tasks: natural language code generation, code refinement, and code translation.

Natural-Language-Based Code Generation aims to generate code snippets from NL descriptions. We employ three commonly-used benchmarks: Concode [17], Django [27], and CoNaLa [45]. Concode considers NL description and class environment contexts, Django includes Python code lines from the Django Web framework paired with NL descriptions, and CoNaLa features NL questions and Python solutions from `Stack Overflow`. We evaluate performance using BLEU-4, exact match accuracy (EM), which is the percentage of programs that has exactly the same token sequence as the ground truth; and CodeBLEU (C-BLEU) [33], which considers syntactic and semantic matches based on data-flow graphs. Additionally, we assess GrammarT5's program synthesis ability on the MBPP dataset and the MathQA-Python. The first one contains 974 coding problems written in Python with 3 unit tests each. The second one is to generate Python programs to solve mathematical problems described in natural language descriptions, where code correctness is measured based on the execution outputs of the generated programs. We follow existing work [3, 12] and evaluate GrammarT5 using the *pass@k* metric, measuring the percentage of problems solved by generating k programs per problem.

Code refinement converts a buggy function into a correct one. We use two Java benchmarks provided by Tufano et al. [39]. These two benchmarks have different function lengths. Refine-small has fewer tokens (< 50 tokens), while Refine-medium has more tokens (50-100 tokens). We use the same metrics as code generation to evaluate the performance.

Code translation is to translate the code of one programming language into another. We utilize CodeTrans dataset [4], which contains the mutually matched pairs of CSharp and Java. We use the same metrics as code generation for this task. Additionally, we observed that CodeXGlue's original dataset did not employ language-specific tokenization when evaluating the BLEU and CodeBLEU metrics, which may have prevented these metrics from accurately reflecting the models' performance. Therefore, we modified the evaluation scripts for these two metrics.

4.5 Comparison Models

Although some code pre-trained models (CodeGen-6B [26], Incoder-6B [9]) have shown promising performance, they have a much larger size (30×) than GrammarT5 and are difficult to fine-tune on downstream benchmarks due to computation resource limitations. Thus, we compare GrammarT5 with various pretrained models with comparable sizes in four categories: encoder-only, decoder-only, encoder-decoder, and unified-encoder. For **encoder-only** models, we consider CodeBERT [8], trained with masked language modeling and replaced token detection; GraphCodeBERT [11], which uses data flow graphs in code; SynCoBERT [41], which employs ASTs to learn syntactical information. For **decoder-only** models, we consider GPT-C [37], trained on a large corpus of Java, and CodeGPT-adapted [25], trained on code using parameters of GPT-2. Besides these two models, we also consider two larger models, CodeGen-Multi-350M and CodeGen-Multi-2B [26], to figure out whether GrammarT5 can still outperform the sequence models with more parameters and data. For **encoder-decoder** models, we adopt CodeT5 [43], which uses identifier-aware masking denoising objectives; PLBART [1], trained on code with a BART [22] architecture; and CoTexT, trained on code with a T5 [31] architecture; CodeT5+ [42], trained on more data and training objectives. For **unified-encoder** models, we select Unixcoder [10], which employs a unified encoder to encompass the functionality of the three model

Table 3: Results of the code understanding tasks.

	Code Summarization		Code Search	
Sub-Task	**Java**	**Python**	**Adv**	**CosQA**
Model \ **Metric**	**BLEU**	**BLEU**	**MRR**	**MRR**
CodeBERT(110M)	17.25	19.06	27.20	65.90
GraphCodeBERT(110M)	18.93	19.39	35.20	68.55
SynCoBERT(110M)	18.89	18.74	38.30	69.19
GPT-C(110M)	17.18	17.78	24.39	50.32
CodeGPT-adapted(110M)	17.68	18.46	25.97	54.24
CodeGen-multi (350M)	19.41	18.31	35.47	69.22
CodeGen-multi (2B)	20.01	19.31	36.47	70.22
CoTexT(220M)	19.19	19.72	34.13	68.70
PLBART(220M)	18.45	19.30	34.70	65.01
CodeT5-small(60M)	19.92	20.04	30.52	66.74
CodeT5-base(220M)	20.31	20.01	39.30	67.80
CodeT5-large(770M)	**20.74**	**20.57**	42.11	71.29
CodeT5+ (220M)	20.31	20.01	43.3	72.7
Unixcoder(110M)	19.42	18.64	41.30	70.10
GrammarT5-small(60M)	19.93±0.10	19.78±0.11	37.24±0.26	70.34 ±0.12
GrammarT5-base(220M)	20.66±0.16	20.21±0.12	**44.12±0.10**	**73.48±0.05**

styles mentioned above. Moreover, for NL-based code generation, we also draw a comparison between GrammarT5-base and the leading non-pretrained code generation model, *TreeGen+Grape* [49].

The majority of models are pretrained over CodeSearchNet, except for GPT-C, PLBART, CodeT5, CodeGen, and CodeT5+. GPT-C is pretrained using a massive dataset of 1.2 billion lines of source code in Python, CSharp, JavaScript, and TypeScript. PLBART utilizes a larger dataset, comprising 470 million Python and 210 million Java functions, as well as 47 million natural language posts from Stack Overflow, outstripping the size of CodeSearchNet. CodeT5 incorporates additional C-Sharp and C corpus extracted from the GitHub code dataset to ensure coverage of all programming languages used in downstream tasks. CodeGen and CodeT5+ adopt a larger training corpus from the BigQuery dataset, including about 115M code files from GitHub in 32 programming languages. As mentioned before, our pretraining set is a subset of CodeT5, minimizing the effect of the pretraining set when compared with CodeT5.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness of GrammarT5

In this section, we compare GrammarT5 with SOTA pretrained models on code understanding and code generation tasks. If the model has been evaluated on the benchmarks, we directly use results from the original papers. Otherwise, we run the pretrained models on the corresponding benchmarks using the published code.

5.1.1 Code Understanding. Table 3 compares pretrained models on code summarization and code search tasks. GrammarT5-base outperforms similar-sized models, achieving the highest MRR scores of 43.98 in Adv and 73.58 in CosQA for Code Search, indicating its accuracy in generating code summaries and retrieving relevant code snippets. GrammarT5-small, despite having fewer parameters, shows strong performance in both tasks. Note that CodeGen-multi-2B (decoder-only), despite having 10 times more parameters, still shows lower performance, highlighting the importance of bidirectional and syntactic information in code comprehension.

5.1.2 Natural-Language-Based Code Generation. We compare GrammarT5 with decoder-only, encoder-decoder, and unified-encoder models since encoder-only models are ineffective for natural-language-based code generation. Table 4 shows that GrammarT5-base outperforms others across all benchmarks. For Conala and Django benchmark, GrammarT5 outperforms other models. The Concode benchmark, requiring function snippet generation from a programmatic context, is more challenging, but GrammarT5-base model achieves an improvement of 2.45 points in exact match accuracy and a 0.4-point increase in CodeBLEU over the previously leading model, CodeT5-large. While these improvements may appear small, they are significant given the task's complexity. Consider that machine learning models for code generation, such as CodeT5-large [21], currently seen as SOTA and an improvement over the existing published approaches, only achieved a 0.35-point increase in exact match accuracy over CodeT5-base [43]. Further showcasing our model's efficiency is GrammarT5-small(60M), which delivers competitive results despite fewer parameters.

The MBPP dataset, designed for Python program synthesis, aims to generate comprehensive Python code to pass specified tests. With its modest size of 374 training sets, MBPP effectively assesses pretrained models' generalizability in code generation. As shown in Table 4, GrammarT5-base significantly outperforms CodeT5-base by 9.2 points and matches CodeT5-large. The smaller GrammarT5-small (60M) variant competes well on the MBPP dataset, even outperforming CodeT5-base, four times its size, by 2 points. In the MathQA-Python task, GrammarT5 surpasses other pretrained models, showing a 4.32-point advantage over CodeT5-large. The GrammarT5-base model particularly performs well in complex problem-solving, achieving a 63.27% pass rate for problems requiring over 10 reasoning steps, significantly higher than CodeT5-base's 21.23%. Despite more training data for CodeGen-multi-2B and CodeT5+, GrammarT5 consistently performs better, highlighting the importance of syntactic and structural information.

To explore the necessity of representing programs grammatically, we analyzed the syntactic correctness of 40,000 programs generated by baseline models of different sizes on the MBPP dataset. Table 6 shows the occurrence of syntax errors in programs generated by various models. It can be seen that models with a size of 220M have about 10% or more programs with syntax errors, some of which even include incorrect Python indentation. Even the 2B model still has a 6% error rate. Additionally, we found that most of the errors these models make involve mixing syntax from different languages. For example, in Python programs generated for the MBPP dataset, there are many expressions like "if (i == 0 && j == 0 or k == 0):", "return (first_char ? (first_char - 1) : (last_char + 1))", and "if(num > 10){". Since the token sequence does not explicitly tell the model the corresponding programming language, while in the grammar sequence, the language's syntax is explicitly indicated through grammar rules. Also, our method, by generating syntax trees, can never produce programs with incorrect indentation. Lastly, compared to the CodeGen-multi-2B, in the problems where our model performes correctly and CodeGen does not, 62% of these involve CodeGen generating syntactically incorrect programs. Therefore,

Table 4: Results of the natural-language-based code generation tasks.

BenchMark	Concode			Conala		Django		MBPP	MathQA
Model \ Metric	BLEU	EM	C-BLEU	BLEU	EM	BLEU	EM	pass@80	pass@80
TreeGen + Grape(35M)	26.45	17.60	30.05	20.16	2.80	75.86	77.30	2.00	26.58
GPT-C(110M)	30.85	19.85	33.10	30.32	4.80	72.56	68.91	10.40	58.94
CodeGPT-adapted(110M)	35.94	20.15	37.27	31.04	4.60	71.24	72.13	12.60	55.90
CodeGen-multi (350M)	38.23	21.25	40.57	33.14	5.70	74.45	74.23	23.50	62.10
CodeGen-multi (2B)	41.23	22.25	44.57	40.14	9.40	81.45	84.04	32.50	83.10
CoTexT(220M)	19.19	19.72	38.13	31.45	6.20	75.91	78.43	14.00	58.18
PLBART(220M)	36.69	18.75	38.52	32.44	5.10	72.81	79.12	12.00	57.25
CodeT5-small(60M)	38.13	21.55	41.39	31.23	6.00	76.91	81.77	19.20	61.58
CodeT5-base(220M)	40.73	22.30	43.2	38.91	8.40	81.40	84.04	24.00	71.52
CodeT5-large(770M)	42.66	22.65	45.08	39.96	7.40	82.11	83.16	32.40	83.14
CodeT5+(220M)	34.13	22.16	43.45	38.91	8.00	78.45	**85.21**	28	85.6
Unixcoder(110M)	38.73	22.65	40.86	36.09	10.20	78.42	75.35	22.40	70.16
GrammarT5-small(60M)	38.08±0.36	21.05±0.36	40.62±0.56	38.18±0.46	8.20±0.36	80.64±0.36	82.27±0.46	25.00±0.86	83.91±0.46
GrammarT5-base(220M)	**43.30±0.86**	**25.10±0.75**	**45.48±0.35**	**41.92±0.56**	**10.40±0.15**	**82.40±0.56**	84.17±0.16	**33.00±0.20**	**87.26±0.32**

Table 5: Results of the code refinement and code translation tasks.

	Code Refinement						Code Translation					
Sub-task	Small			Medium			Java to CSharp			CSharp to Java		
Model \ Metric	BLEU	EM	C-BLEU	BLEU	EM	C-BLEU	BLEU	EM	C-BLEU	BLEU	EM	C-BLEU
CodeBERT(110M)	78.41	16.40	78.09	86.94	5.20	83.88	85.23	62.10	86.87	85.81	61.80	85.19
GraphCodeBERT(110M)	**79.61**	17.3	**79.68**	87.63	9.10	85.33	86.35	63.10	87.6	86.50	62.10	85.18
SynCoBERT(110M)	78.81	20.32	78.56	88.37	11.17	87.05	87.04	65.10	88.26	87.80	65.20	86.81
GPT-C(110M)	70.06	13.03	71.83	85.41	8.26	82.47	78.90	61.90	81.02	84.48	60.70	83.87
CodeGPT-adapted(110M)	76.07	13.66	77.13	85.28	11.00	84.55	82.11	62.90	83.45	85.68	61.30	84.98
CodeGen-multi (350M)	78.12	20.12	77.23	88.21	13.12	86.84	90.21	66.40	90.24	89.75	65.50	88.65
CodeGen-multi (2B)	79.52	22.12	79.23	89.21	14.12	88.84	91.41	68.40	91.44	89.75	70.60	88.65
CoTexT(220M)	77.28	21.33	77.38	87.13	13.03	85.14	85.57	66.70	86.25	87.21	65.80	87.11
PLBART(220M)	77.02	19.40	77.58	88.48	8.98	86.67	87.95	67.80	88.12	87.19	67.70	87.01
CodeT5-small(60M)	76.23	19.06	76.44	89.20	10.92	87.25	88.23	65.40	88.32	87.22	69.60	87.18
CodeT5-base(220M)	77.43	21.61	77.24	87.64	13.96	87.05	88.55	66.90	88.72	87.03	68.70	86.71
CodeT5-large(770M)	77.38	21.7	77.14	89.22	14.76	87.35	88.89	67.20	88.98	87.20	68.80	87.16
CodeT5+(220M)	78.27	22.18	77.48	88.64	**15.13**	86.28	**91.66**	66.20	**91.51**	89.64	70.20	91.01
Unixcoder(110M)	79.18	19.05	79.45	87.59	13.96	86.23	90.20	67.00	90.15	90.51	70.60	90.32
GrammarT5-small(60M)	76.90±0.15	20.50±0.35	76.98±0.15	89.11±0.35	12.63±0.15	86.86±0.25	91.15±0.15	67.80±0.25	90.29±0.54	89.20±0.25	71.60±0.45	89.49±0.15
GrammarT5-base(220M)	79.39±0.05	**22.60±0.26**	78.88±0.16	**90.57±0.24**	15.32±0.40	**89.18±0.18**	91.31±0.13	**69.10±0.26**	91.11±0.28	**90.53±0.05**	**73.40±0.12**	**91.26±0.38**

we believe that representing programs through GrammarT5 can significantly narrow the search space for programs and increase the probability of generating correct programs.

Table 6: Error Type of MBPP.

Model \ Type	Syntax Error	Indentation Error	Tab Error	Error Rate
CodeT5(220M)	5142	389	15	13.87%
CodeT5+(220M)	4392	294	5	11.73%
CodeT5-large(770M)	4123	229	2	10.89%
CodeGen-multi(2B)	2312	92	7	6.03%
GrammarT5	0	0	0	0%

5.1.3 Code-to-Code Generation. We compare GrammarT5 with other pretrained models on two code-to-code generation tasks: code refinement and code translation. In the code refinement task, the large overlap between the source and target code may result in a high BLEU score but zero exact matches. Consequently, we concentrate on the exact match (EM) metric for this task. As illustrated in Table 5, GrammarT5-base outperforms all baselines on both tasks, including CodeT5-large. Examining the GrammarT5-small model, it also demonstrates solid performance across all tasks and metrics when compared to other models in the same size category.

In the task of code translation, we primarily compare GrammarT5 with CodeT5, CodeT5+, and CodeGen-multi, all of which have been pre-trained on a corpus that includes CSharp. In this scenario, GrammarT5-base surpasses CodeGen-multi-2B by 0.7 and 2.8 points in the Exact Match (EM) metric across two distinct subtasks. Remarkably, GrammarT5-small, which has only 60 million parameters, outperforms CodeT5-large, a model with around 770 million parameters, making it nearly 11 times larger.

To sum up, GrammarT5 exhibits better performance on most metrics and tasks compared with existing models with the same or smaller sizes. Furthermore, GrammarT5-base achieves very similar,

or even better performance compared with larger models, CodeT5-large and CodeGen-multi-2B. Please recall that the other variables are controlled when compared with CodeT5 models. These results suggest that representing programs as grammar rule sequences is beneficial and novel techniques used in GrammarT5 are effective.

Table 7: Ablation study with GrammarT5-small.

Model \ BenchMark	Adv MRR	Java-CSharp EM	CSharp-Java EM	MBPP pass@80
GrammarT5-small	**37.24±0.26**	**67.80±0.25**	**71.60±0.45**	**25.00±0.86**
w/o EP	35.41±0.22	66.80±0.32	71.00±0.16	23.80±0.60
w/o STP	34.03±0.36	66.60±0.22	69.60±0.28	22.40±0.44
w/o LF	35.26±0.21	67.20±0.29	69.40±0.26	23.60±0.36
CodeT5-small	30.52	65.40	68.8	19.20
CodeT5-small + TGRS	33.27±0.12	66.10±0.46	70.10±0.36	22.40±0.13

5.2 RQ2: Effectiveness of TGRS

To underscore the efficacy of TGRS, we trained an ablated CodeT5-small model, with the sole alteration being the switch in representation to TGRS. The key difference between this model and GrammarT5-small is the pre-training objectives, where this model uses the same pre-training tasks as the original paper. As portrayed in Table 7, the ablated model outperforms CodeT5-small across all benchmarks. These results point to TGRS's ability to encapsulate crucial syntactic information in a more organized fashion than token sequences, thereby bolstering the performance of models that are pre-trained on this particular representation.

For further understanding the effect of TGRS, we compute the average input lengths of varying programming languages, as represented by different representations within the pretraining dataset. As demonstrated in Table 8, the TGRS representation navigates a judicious balance between the conciseness of the Token Sequence and the comprehensive structure of the AST. TGRS provides a more compact representation than ASTs, which can be beneficial for tasks requiring efficient processing and reduced memory usage.

Table 8: Average Length of Different Representation.

Representation	Java	CSharp	Python
TGRS	199.20	300.67	242.32
AST	427.66	583.49	476.99
Token Sequence	169.58	247.67	186.27

5.3 RQ3: Effectiveness of Language Flags

As stated in Section 3.1.2, there are two approaches to combining grammar rules, whether to share or not to share the same production rules in different languages. To understand their differences, we train an ablated version of GrammarT5-small using the original combined grammar of tree-sitter without language flags.

As shown in Table 7, the performance of GrammarT5-small without language flags decreases in four tasks. This result suggests that GrammarT5 is difficult to learn syntactical information from mixed grammar rules. It is more effective to provide models with additional language-specific information, such as the language flags used in this paper, to differentiate between languages.

5.4 RQ4: Effectiveness of Pre-training Tasks

To evaluate the effectiveness of the proposed pre-training tasks, we conduct an ablation study to examine their contributions. Due to computational resource limitations, we compare GrammarT5-small on four selected tasks by ablating the two proposed denoising objectives: Edge Prediction (EP), and Sub-Tree Prediction (STP).

As depicted in Table 7, the performance declines across all tasks when each component is removed, demonstrating the importance of each component in the model. The removal of STP results in the most substantial decrease in performance across all tasks, indicating that the sub-tree prediction objective significantly impacts the model's effectiveness. The other components, such as EP, and LF, also contribute to the overall performance, albeit to a lesser extent. In summary, the ablation study underscores the importance of each component in achieving the superior performance of GrammarT5.

```
public static List shunting_yard(ArrayList tokens) {
    ……
    ArrayList rpntokens = new ArrayList(100);
    ArrayDeque opstack = new ArrayDeque();
    for (Object token: tokens) {
        if (Integer.class.isInstance(token)) {
            rpntokens.add((Integer) token);
        } else {
            String operator = (String) token;
        while (!opstack.isEmpty() && precedence.get(operator)
<= precedence.get(opstack.getLast())) {
            rpntokens.add(opstack.pop());
        }
        _mask_line_;
    }
  }
  ……
```

```
GrammarT5-base:
opstack.push(operator);

Incoder-6B:
opstack.addLast(operator);

Codegen-6B:
rpntokens.add(operator);
opstack.push(operator);

ChatGPT:
opstack.push(operator);

GroundTruth:
opstack.push(operator);
```

Figure 4: SHUNTING_YARD bug in QuixBugs.

Specifically, we observe that the STP pretraining task can be applied to the code fill downstream task in the zero-shot setting. Figure 4 illustrates a real bug in the QuixBugs benchmark. The human-written patch inserts a push operation in the placeholder "_mask_line_". We compare GrammarT5-base with three pre-trained models capable of generating code in the designated location. As demonstrated, Incoder-6B and Codegen-6B cannot generate the correct statement, while GrammarT5-base produces the same code as ChatGPT, albeit with a much smaller size (220M). We hypothesize that this objective can assist the model in learning the structural relationship between the sub-tree and its context. This objective also endows GrammarT5 with the ability to complete the code based on the context, indicating potential applications of GrammarT5 for code completion and program repair.

5.5 RQ5: Generalizability to token sequences

We recognize that tasks like code completion often involve partial, syntactically incorrect code, which is challenging to convert into TGRS. However, we find that every program token is encapsulated within the TGRS sequence. We hypothesize that it is plausible to directly use token sequence for fine-tuning when dealing with tasks like code completion. To this end, we test the generalizability of GrammarT5 on the line-level code completion task of CodeXGlue. We also conduct experiments on text sequences in the code translation dataset to compare performance differences with TGRS.

In our comparison, as illustrated in Table 9, GrammarT5 demonstrates its capability by showing similar levels of effectiveness on both benchmarks when the code is represented through token sequences compared with the baselines of comparable size. There

Table 9: Results with Token Sequences.

	Code Completion		Code Translation	
Sub-Task	**Python**	**Java**	**Java-CSharp**	**CSharp-Java**
Model \ Metric	**EM**	**EM**	**EM**	**EM**
GPT-C(110M)	38.37	28.60	61.90	60.70
CodeGPT-adapted(110M)	42.37	30.60	62.90	61.30
CodeGen-multi (350M)	42.47	35.47	66.40	65.50
CodeGen-multi (2B)	**46.32**	**40.47**	68.40	70.60
PLBART(220M)	38.01	26.97	**67.80**	67.70
CodeT5-small(60M)	19.92	20.04	65.40	69.60
CodeT5-base(220M)	36.97	24.80	66.90	68.70
CodeT5-large(770M)	38.74	28.57	67.20	68.80
CodeT5+ (220M)	43.42	35.17	66.20	70.20
Unixcoder(110M)	43.12	32.90	67.00	**70.60**
GrammarT5-small(60M)	38.93±0.42	28.54±0.24	65.12±0.46	67.21 ±0.23
GrammarT5-base(220M)	43.75±0.22	34.32±0.36	67.12±0.23	70.23±0.25

is also a performance drop on code translation tasks when TGRS is not used. This disparity suggests that while GrammarT5 is reliable and effective in handling tasks involving token sequences, its integration with TGRS enhances its performance significantly. Furthermore, this observation opens up intriguing possibilities for future advancements. The integration of token sequence representation with TGRS, potentially through the development of additional pretraining objectives, could lead to more sophisticated and effective models, which we suggest as a direction for future research.

6 DISCUSSION

Margin Improvement of Small Models. We observe that in some instances, especially with smaller datasets and models, our technique may not only appear less promising but can occasionally yield inferior results compared to existing methodologies, specifically "GrammarT5-small(60M)" and "CodeT5-small(60M)," on the Concode dataset. However, it is important to emphasize that as model size increases, our approach demonstrates significantly improved performance. This scalability is particularly notable when dynamic metrics are employed, showcasing the method's robustness and adaptability in more complex scenarios. Therefore, our research indicates that the integration of grammar-aware components becomes increasingly beneficial as the model size expands. This finding suggests a promising direction for future research and development, particularly in the realm of large-scale language models where the nuances of grammar and context play a crucial role.

Selection of Aggregation. In this study, we have adhered to the aggregation strategy utilized by CodeT5, maintaining consistency for a fair and uniform comparative analysis. However, there are also several possible approaches, such as averaging all pre-training tasks. Due to constraints in time and computational resources, an in-depth investigation of these alternatives remains beyond the scope of our current research. We identify this as an area for future work, recognizing the potential it holds for enhancing model performance and providing a more nuanced understanding of aggregation impacts in language model training.

7 THREATS TO VALIDITY

Threats to Internal Validity. A potential threat to our study's internal validity lies in our experiment's implementation. To mitigate this threat, we rely on the performance metrics reported in the original papers [1, 8, 10, 11, 25, 37, 41, 43], and use publicly available models from the Hugging Face Hub [15] for additional benchmarks. Furthermore, we use fixed random seeds in all experiments to eliminate randomness and guarantee reproducibility.

Threats to External Validity. A primary threat to external validity lies in the benchmarks utilized in our experiment. While our pre-training dataset is derived from the widely-used CodeSearchNet benchmark [16] and downstream benchmarks use versions collected by CodeXGlue [25], the performance of GrammarT5 can only be generalized to the evaluated tasks and datasets. Its performance on other software engineering tasks [18, 47, 50] and benchmarks remains uncertain. The variety in pre-training corpora used in different models could also influence performance. However, GrammarT5-base's superiority over CodeT5 in most benchmarks implies that incorporating grammar sequences in GrammarT5 significantly improves performance, even with a subset of the same pretraining dataset. Nonetheless, it is crucial to conduct more experiments with diverse corpora and coding scenarios to further validate these results.

8 CONCLUSION

In conclusion, this paper introduces GrammarT5, a pretrained model for code understanding and generation that leverages multimodal content, including programming languages and natural language. The model utilizes an encoder-decoder framework, employing two Transformers in the same manner as T5. This work proposes a novel code representation, TGRS, which effectively represents programs as grammar rule sequences in pretraining. Moreover, the paper introduces three pre-training tasks designed to help the model learn syntactic structures and semantic information in GrammarT5. Experiments were conducted on five code-related tasks using ten datasets, demonstrating that GrammarT5 achieves state-of-the-art performance on all tasks compared to models of a similar scale. Furthermore, ablation studies show that the three technical novelties are all effective at boosting the model performance.

ACKNOWLEDGMENTS

This work is sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China under Grant Nos. 62161146003, the National Natural Science Foundation of China under Grant No. 62232001 and No. 62232003, and ZTE Industry-University-Institute Cooperation Funds under Grant No.HC-CN-20210319008.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).

[5] Codeparrot. 2023. GitHub Code Dataset. https://huggingface.co/datasets/codeparrot/github-code

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

[7] Facebook. 2023. Pytorch. https://pytorch.org

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

[10] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, and Akul Arora. [n. d.]. Measuring Coding Challenge Competence With APPS. ([n. d.]).

[13] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ Web Queries for Code Search and Question Answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 5690–5700.

[14] Huggingface. 2023. Accelerate. https://github.com/huggingface/accelerate

[15] Huggingface. 2023. Huggingface Transformers. https://huggingface.co

[16] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG]

[17] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 1643–1652.

[18] Yuhe JI, Jing HAN, Yongxin ZHAO, Shenglin ZHANG, and Zican GONG. 2023. Log Anomaly Detection Through GPT Log Anomaly Detection Through GPT-2 for Large Scale Systems. *ZTE COMMUNICATIONS* 21, 3 (2023).

[19] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence (Proceedings of Machine Learning Research, Vol. 161)*, Cassio de Campos and Marloes H. Maathuis (Eds.). PMLR, 54–63. https://proceedings.mlr.press/v161/jiang21a.html

[20] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1073–1085.

[21] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328.

[22] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[23] Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. 501–507.

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

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[27] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[28] Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. 2020. BPE-Dropout: Simple and Effective Subword Regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 1882–1892.

[29] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1139–1149.

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

[31] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[32] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

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

[34] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[35] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7055–7062.

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

[37] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[38] Tree-Sitter. 2023. Tree-Sitter. https://tree-sitter.github.io/tree-sitter

[39] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[41] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556* (2021).

[42] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[43] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[44] Yingfei Xiong and Bo Wang. 2022. L2S: A framework for synthesizing the most probable program under a specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–45.

[45] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 476–486.

[46] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

[47] Zipiao Zhao, Yongli Zhao, Boyuan Yan, and Dajiang Wang. 2022. Auxiliary Fault Location on Commercial Equipment Based on Supervised Machine Learning. *ZTE Communications* 20, S1 (2022), 7–15.

[48] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering. 341–353.

[49] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2022. Grape: Grammar-Preserving Rule Embedding.. In *IJCAI*. 4545–4551.

[50] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1443–1455. https://doi.org/10.1109/ICSE48619.2023.00126