Towards Compositional Generalization in Code Search

Hojae Han♠,*, Seung-won Hwang♠†, Shuai Lu◇, Nan Duan◇, Seungtaek Choi♣
♠Seoul National University, {stovecat, seungwonh}@snu.ac.kr
◇Microsoft Research Asia, {shuailu, nanduan}@microsoft.com
♣Riiid, seungtaek.choi@riiid.co

Abstract

We study compositional generalization, which aims to generalize on unseen combinations of seen structural elements, for code search. Unlike existing approaches of partially pursuing this goal, we study how to extract structural elements, which we name a template that directly targets compositional generalization. Thus we propose CTBERT, or Code Template BERT, representing codes using automatically extracted templates as building blocks. We empirically validate CTBERT on two public code search benchmarks, AdvTest and CSN. Further, we show that templates are complementary to data flow graphs in GraphCodeBERT, by enhancing structural context around variables.

1 Introduction

We study compositional generalization in the context of code search, which aims to generalize on unseen combinations of previously seen elements, or building blocks (Qiu et al., 2022). Following the categorization in the NLP domain (Kim and Linzen, 2020), Figure 1 motivates such a goal in the code search setting, where a model is expected to generalize on an unseen code, which is a combination of a seen structure with other lexicons (**lexical generalization**). Moreover, the model should also generalize on another case when an unseen code is a combination of seen structures, though red and blue colored structures were only independently observed while its combination was not (**structural generalization**).

Existing code search models such as Code-BERT (Feng et al., 2020) regarding code as an unstructured sequence (Figure 2a) often fail to generalize when the lexical perturbation without changing structures and labels is applied in test codes, known as AdvTest evaluation (Lu et al., 2021). Towards better generalization, SynCoBERT (Wang

(a) Lexical generalization.

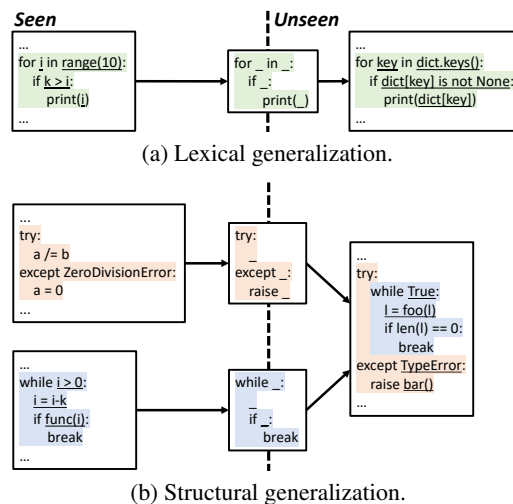(b) Structural generalization.

Figure 1: **Motivation.** Templates enhance compositional generalization between seen and unseen codes.

et al., 2021b) extracts an abstract syntax tree (AST; Figure 2b) to incorporate syntactic structure into the model. On the other hand, GraphCode-BERT (Guo et al., 2021) extracts a data flow graph (DFG; Figure 2c), to represent variable relations within code. From structure-aware pretraining objectives, these two models significantly enhance their accuracy in code search, both in original and adversarial settings. Meanwhile, utilizing composable building blocks, like the colored structures in Figure 1 for code search was not studied.

Inspired by the compositional generalization work in NLP domain, we study how to represent composable building blocks targeted to improve compositional generalization. Specifically, we consider subtrees of AST as such building blocks, which we call templates (Figure 2d). As illustrated in Figure 1, templates share across codes, as a code is a combination of templates. However, it requires an additional non-trivial challenge of mining such a meaningful block.

In this paper, we suggest how to mine templates

10743

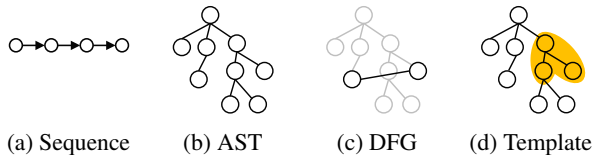(a) Sequence   (b) AST   (c) DFG   (d) Template

Figure 2: **Our distinction.**

and propose CTBERT, Code Template BERT, pre-trained with templates for compositional generalization. Specifically, we group frequently adjacent AST nodes into subtrees, filter uncommon or meaningless subtrees, and regard the remaining ones as templates. Then, we pretrain CTBERT with templates, using our proposed objective teaching models to recognize structural building blocks.

To confirm the effectiveness of CTBERT towards compositional generalization in code search, we validate CTBERT by fine-tuning on a large-scale dataset CSN (Husain et al., 2019; Guo et al., 2021) then testing on an adversarial test set AdvTest (Lu et al., 2021). AdvTest is more appropriate for evaluating compositional generalization than CSN as 1) variable and function names in test codes are normalized (testing lexical generalization) and 2) test time codes are unseen combinations of seen structures (testing structural generalization).

Our experimental results show that CTBERT enhances its compositional generalization with templates, as shown by the performance gains on AdvTest, while performing comparably on CSN. Further, we show that CTBERT can straightforwardly be combined with DFG objectives from GraphCodeBERT, achieving orthogonal and complementary performance gains. All our implementations and data including the mined templates will be publicly released[1].

2 CTBERT

Our goal is to improve the model's compositional generalization capability, by pre-training with automatically mined templates.

2.1 Mining Templates

An assumption in compositional generalization is that there are shared common building blocks in both source and target distributions (Qiu et al., 2022). In the code search task, we regard frequently observed subtrees in abstract syntax trees (ASTs) as such building blocks, namely templates.

Specifically, we first extract candidate templates, by grouping the adjacent AST nodes, according to the probability of the occurrence of a child node y out of other children, given its parent node x, denoted as $P(x \rightarrow y)$. Following (Wang et al., 2021a), we formulate such probability as[2]:

$$P(x \rightarrow y) = \frac{N(x \rightarrow y)}{\sum_{y_i} N(x \rightarrow y_i)}, \qquad (1)$$

where $N(x \rightarrow y)$ is the number of the observed child y given the parent node x in the source distribution. Then, after removing all the edges $x \rightarrow y$ in the ASTs when its probability $P(x \rightarrow y)$ is lower than a threshold T, we treat the remaining sub-trees as candidate templates. From the candidate templates, we filter meaningless subtrees, decided by the number of nodes smaller than S or the depth shallower than D. Finally, we count the number of occurrences for each template, to select the most frequently observed k templates, filtering out uncommon ones[3].

2.2 Pretraining with Templates

To understand unseen codes based on templates, we propose a pretraining strategy utilizing templates.

2.2.1 Preliminary

Following GraphCodeBERT (Guo et al., 2021), we use a Transformer based model BERT (Devlin et al., 2019) as the backbone architecture of CTBERT. Given a natural language query $W = \{w_1, w_2, ..., w_{|W|}\}$ describing a code $C = \{c_1, c_2, ..., c_{|C|}\}$, a basic BERT input format is $X = \{[\text{CLS}]\} \oplus W \oplus \{[\text{SEP}]\} \oplus C$, where $[\text{CLS}]$ and $[\text{SEP}]$ are special tokens in BERT, and $\oplus$ is a concatenate operation. Our focus is to augment the input with structural information, such as $X \oplus \{[\text{SEP}]\} \oplus \mathcal{V}$ for GraphCodeBERT, given by a DFG node sequence $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$, where each $v \in \mathcal{V}$ indicates a variable. However in CTBERT, instead of defining the input format of templates as a sequence of every node in the templates, we define a sequence of templates $\mathcal{T} = \{t_1, t_2, ..., t_{|\mathcal{T}|}\}$, where each template $t \in \mathcal{T}$ is a subsequence of C containing code tokens from the template. The purpose of our design choice is 1) to incorporate the meaning of each template into a single special token, and 2) to reduce the length of the template sequence $|\mathcal{T}|$ to avoid being truncated.

---

[2]See Figure 7 in Appendix C for details.
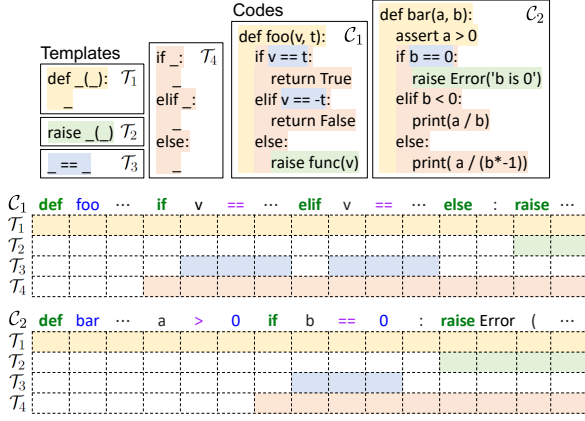[3]Selected hyperparameters are listed in Appendix B.

Figure 3: **Illustration of templates with different combinations in two codes.**

Thus, we use $X \oplus \{[\text{SEP}]\} \oplus \mathcal{T}$ as our input for CTBERT.

2.2.2 Injecting Structure

Inspired by GraphCodeBERT, we inject the dependency between code tokens and templates using masked attention and mask prediction, each of which we contrast with our added challenges.

Masked Attention The purpose of masked attention in GraphCodeBERT is to inject DFGs into the model, interpreted as conditionally allowing non-zero attention in the following two cases only: 1) When the attention between a DFG node token $x_i \in \mathcal{V}$ and a code token $x_j \in C$ refers to the same variable, or 2) when the attention between two node tokens $x_i, x_j \in \mathcal{V}$ are connected in the DFG. We denote these two cases as $x_i \mid x_j$, and zero-out attention by adding $-\infty$, turning attention to zero after softmax when these two cases do not hold. Formally, an attention masking matrix for DFG is defined below[4]:

$$M_{ij}^{DFG} = \begin{cases} 0 & \text{if } x_i \mid x_j \\ -\infty & \text{otherwise.} \end{cases} \quad (2)$$

Now, we employ the idea of masked attention to representing the dependency between code tokens and templates. Figure 3 shows the membership of each code token c to each template t, where we allow attention for highlighted slots, or $c \in t$, by setting 0 in the mask matrix, and $-\infty$, otherwise. More formally, we define a masked attention $M_{ij}^{\mathcal{T}}$

---

[4]Without loss of generality, we omit the details that attention among word tokens and code tokens are not masked.

for the template sequence $\mathcal{T}$ as:

$$M_{ij}^{\mathcal{T}} = \begin{cases} 0 & \text{if } c_i \in t_j \\ -\infty & \text{otherwise,} \end{cases} \quad (3)$$

where $c_i \in C$ is a code token and $t_i \in \mathcal{T}$ is a template.

We note that, as token-DFG and token-template dependencies are represented by the same mechanism, we may replace one with the other, or build on top of another, both of which are found effective empirically (see §3.3).

Pretraining Task To learn dependency among tokens in their representation, GraphCodeBERT also formulates a mask prediction task. For the task, we can first randomly mask the attention of two tokens x_i and x_j satisfying $x_i \mid x_j$, meaning either a node token $x_i \in \mathcal{V}$ and a code token $x_j \in C$ indicate the same variable, or two node tokens $x_i, x_j \in \mathcal{V}$ are connected in DFG. Then we ask the model to predict whether x_i and x_j satisfy $x_i \mid x_j$:

$$\mathcal{L}_{DFG} = -\sum_{x_i \in \bar{\mathcal{V}}} \sum_{x_j \in C \cup \mathcal{V}} [\mathbb{1}(x_i \mid x_j) \log p_{x_i x_j} + (1 - \mathbb{1}(x_i \mid x_j)) \log(1 - p_{x_i x_j})], \quad (4)$$

where $\bar{\mathcal{V}}$ is the sampled set of every node token x_i for masking, and $p_{x_i x_j}$ is the predicted probability of $x_i \mid x_j$, by calculating the dot product with a sigmoid function between the last layer representation of the token x_i and x_j.

This objective can be extended to inject code token membership to template, shown in Figure 3. Our goal is to make the model understand "which template t each token c comes from" or, predicting $c \in t$. We randomly sample templates, and mask attention between each sampled template t and code tokens that is in the template t. Then, for each code token c, we ask whether c is in the template t, or $c \in t$. Formally, this objective is denoted as:

$$\mathcal{L}_{\mathcal{T}} = -\sum_{t \in \bar{\mathcal{T}}} \sum_{c \in C} [\mathbb{1}(c \in t) \log p_{tc} + (1 - \mathbb{1}(c \in t)) \log(1 - p_{tc})], \quad (5)$$

where $\bar{\mathcal{T}}$ is the sampled set of the templates for masking, and p_{tc} is the predicted probability from representation similarity between template and code tokens.

3 Experiments

To confirm whether learning with code templates contributes to compositional generalization in code

| Model | MRR | | Pre-training Corpus | Code Structure |
	AdvTest	CSN		
CodeBERT[†‡]	0.2719	0.672		-
CodeBERT (reproduced)	0.3393	0.6682		-
GraphCodeBERT[†¶]	0.352	0.692	CodeSearchNet	DFG
GraphCodeBERT (reproduced)	0.3709	0.6935		DFG
SynCoBERT[¶]	0.381	**0.724**		AST
CodeBERT (Python)	0.3636	0.6786		-
GraphCodeBERT (Python)	0.3922	0.6878	CodeSearchNet (Python)	DFG
CTBERT	**0.3997**	0.6922		DFG + Template

Table 1: **Code search on AdvTest and CSN.** The results with the footnotes are reported from: [†](Guo et al., 2021), [‡](Lu et al., 2021), and [¶](Wang et al., 2021b). **Bold**/underlined mark signifies the 1st/2nd MRR performance, respectively.

| Template Mining | MRR | |
	AdvTest	CSN
Anonymized code lines	0.3885	0.6900
AST subtrees (**this work**)	**0.3997**	**0.6922**

Table 2: **Comparison of different template mining strategies.**

| Index | DFG | Template | | MRR | |
		$M_{ij}^{\mathcal{T}}$	$\mathcal{L}_{\mathcal{T}}$	AdvTest	CSN
0	✓	-	-	0.3922	0.6878
1	-	✓	-	0.3932	0.6832
2	✓	✓	-	0.3847	0.6870
3	-	✓	✓	0.3983	0.6889
4	✓	-	✓	0.3924	0.6911
5	✓	✓	✓	**0.3997**	**0.6922**

Table 3: **Ablation studies.**

Figure 4: **Template complements DFG.** AdvTest MRR scores over a range of input code lengths.

search, we validate CTBERT on AdvTest (Lu et al., 2021) which can evaluate both lexical and structural generalization. The baselines and implementation details can be found in Appendix B.

3.1 Main Results

For evaluating compositional generalization in code search, we fine-tune code search models on CSN (Husain et al., 2019; Guo et al., 2021) Python, then evaluate on AdvTest (Lu et al., 2021) and CSN Python measured by Mean Reciprocal Rank (MRR). Note that AdvTest is more suitable to validate compositional generalization as both lexical and structural generalization can be tested. Thus, our goal is to outperform on AdvTest while performing comparably on CSN. Indeed, Table 1 demonstrates that CTBERT achieves the best MRR score on AdvTest, while performing comparably to
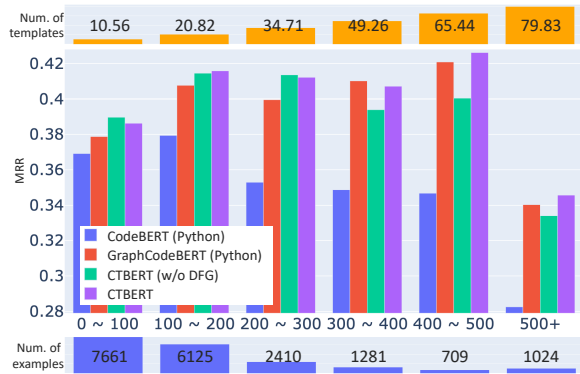
the baselines on CSN.

3.2 RQ1: Do Templates Help Compositional Generalization?

We compare our AST-based template mining with a template selection used in (Oren et al., 2020), using frequent anonymized code as a template. However, applying this baseline "as is" for our problem is too weak, as only 0.28% of AdvTest codes contain 1 or more templates observed from the pretraining corpus (CodeSearchNet (Python)), while our approach leads to 99.67%. Thus, we build a better baseline by regarding a frequent anonymized code *line* as a template (now 92.06% of AdvTest codes contain 1 or more templates). This new baseline is trained in the same setting as CTBERT, including DFG utilization. Table 2 shows that the code-line-based template mining approach underperforms CTBERT capturing a tree structure.

Further, as shown in Table 3, we conduct ablation studies to confirm whether the masked template attention and the template pretraining objec-
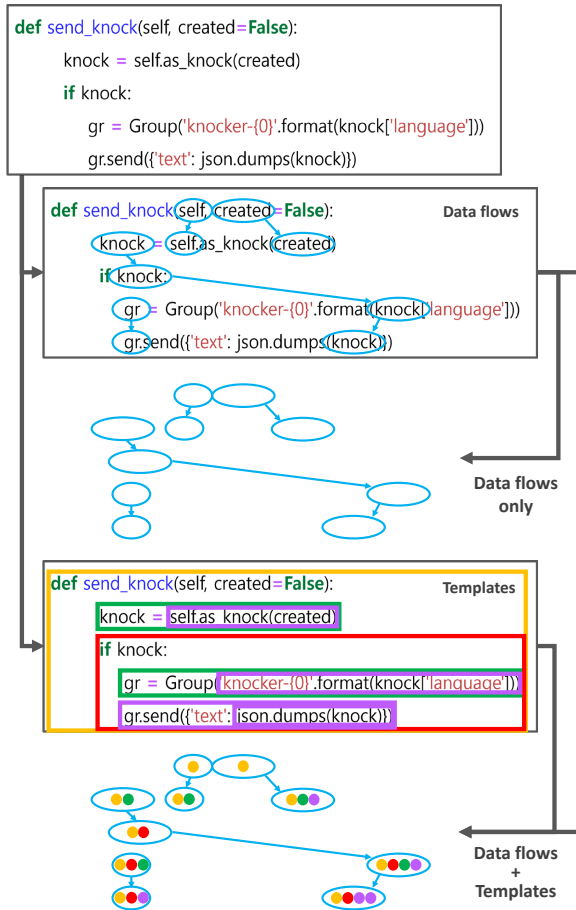
Figure 5: **Illustration of how template complements DFG in CTBERT**. A DFG catches long-range dependencies, e.g., the variable knock in line 5 is declared in line 1. Templates decorate the DFG with local contexts, e.g., the variable gr is declared inside the if statement.

tive contribute to compositional generalization. By comparing the indices 4 and 5 in Table 3, we empirically validate its effectiveness as removing the masked template attention drops both MRR scores of AdvTest and CNN. Also, by comparing the indices pairs (1, 3) and (2, 5) in Table 3, we observe that adding our objective consistently boosts the MRR performances on both datasets.

3.3 RQ2: Do Templates Complement Other Structural Tasks?

The indices 0, 3, and 5 in Table 3 are the results when the code search model utilizes DFG-only, template-only, and when it uses both. We can see that 1) the template-only setting has a higher MRR score than the DFG-only; 2) utilizing both DFG and template shows the best performance.

To explain why they complement each other, Figure 4 reports results, stratified by a range of input lengths. We can observe that DFG has an
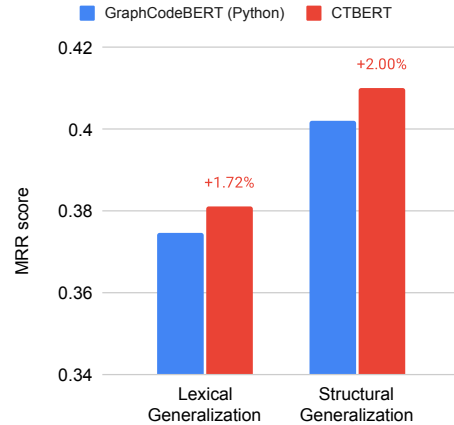

Figure 6: **CTBERT improves compositional generalization.** We split AdvTest into two subsets to measure lexical generalization and structural generalization. The size of each subset is 36.02% and 63.98% respectively (100% is AdvTest size).

advantage of capturing long-distance dependency between variables, but is relatively weak in capturing the local dependency within semantic blocks, as illustrated in Figure 5. We leave an optimal ensemble for multiple structure-aware pretraining tasks, as future directions.

3.4 RQ3: Do Templates Contribute To Lexical/Structural Generalization?

We split AdvTest into two subsets: One contains every test example that the composition of the templates is seen during pretraining and another with unseen. We hypothesize that the former subset can measure lexical generalization as the compositions are already *seen*, and the latter can measure structural generalization as the compositions are *unseen*. Figure 6 shows that, from GraphCode-BERT (Python), CTBERT can contribute +1.72% and +2.00% of MRR performance gain on both subsets, confirming better generalization both lexically and structurally.

4 Conclusion

This paper studied compositional generalization in code search, and proposed CTBERT. Our contributions are three-fold: 1) We proposed templates that are composable building blocks for code, and pretrained CTBERT with templates; 2) We showed CTBERT improves compositional generalization; 3) We empirically confirmed that templates are complementary to DFG in GraphCodeBERT, one of existing structural-aware tasks.

Acknowledgement

Limitations

The limitation of this study is, as we propose a pre-training strategy, our experiments required heavy computations of taking 14 hours on 8 NVIDIA Tesla V100 with 32GB memory.

References

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. 2020. Improving compositional generalization in semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2482–2495, Online. Association for Computational Linguistics.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Pawel Nowak, Tal Linzen, Fei Sha, and Kristina Toutanova. 2022. Improving compositional generalization with latent structure and data augmentation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4341–4362, Seattle, United States. Association for Computational Linguistics.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 784–789, Melbourne, Australia. Association for Computational Linguistics.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas. Association for Computational Linguistics.

Bailin Wang, Wenpeng Yin, Xi Victoria Lin, and Caiming Xiong. 2021a. Learning to synthesize data for semantic parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2760–2766, Online. Association for Computational Linguistics.

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

A Related Work

A.1 Compositional Generalization in NLP

The objective of compositional generalization is to generalize on unseen data composed of seen elements, or building blocks (Qiu et al., 2022), which is not pursued by many standard NLP models (Kim and Linzen, 2020). To overcome this, (Qiu et al., 2022) propose a data augmentation approach for compositional generalization objective in NLP, and ours can be viewed as the first work pursuing such an objective for code search.

A.2 Pre-training with Code Structures

Without considering structural information in code by taking a code as an unstructured sequence, Code-BERT (Feng et al., 2020) is weak in AdvTest, testing lexical and structural generalization.

To overcome this, later work has adopted code structures for representation: As Figure 2 contrasts, SynCoBERT (Wang et al., 2021b) took AST as an additional input to represent the syntactic structure of code. GraphCodeBERT (Guo et al., 2021) used DFG to understand the relation between variables. Our distinction is considering composable building blocks, namely templates, targeted to enhance compositional generalization capability.

A.3 Template Mining

Recently, (Oren et al., 2020) propose a compositional generalization approach in semantic parsing, mining templates by anonymizing frequently observed SQLs from the training corpus.

Our distinction is regarding frequent AST subtrees as templates, which is more suitable for composable building blocks (See Table 2).

B Implementation Details

Template Mining   For mining templates, we set the probability threshold as $T = 0.2$, the minimum number of nodes as $S = 4$ the minimum depth as $D = 2$. Then, we set K to 100, mining 100 frequent templates. The entire mining took 18 hours on CodeSearchNet (Husain et al., 2019) Python corpus, with 25.49 templates per code on average.

Code Search Model   **CodeBERT** (Feng et al., 2020) is one of the popular code search baselines regarding a code as an unstructured sequence. **GraphCodeBERT** (Guo et al., 2021) uses DFG, to inject code/DFG tokens referring to the same variable using $\mathcal{L}_{DFG}$. In our experiment, we report

two versions for each CodeBERT and GraphCode-BERT: 1) **(reproduced)** fine-tuned on CSN (Husain et al., 2019; Guo et al., 2021) Python training set from the published checkpoints; 2) **(Python)** further pretrained on CodeSearchNet (Husain et al., 2019) Python corpus, then fine-tuned on CSN Python. **SynCoBERT** (Wang et al., 2021b) leverages AST to learn syntactic structures in code. We report their published results, as the implementation is unavailable. **CTBERT** is our proposed method, with the same setting as GraphCodeBERT (Python), except for using templates with $\mathcal{L}_{\mathcal{T}}$ but not DFG. **CTBERT+** further utilizes both objectives $\mathcal{L}_{\mathcal{T}}$ and $\mathcal{L}_{DFG}$, to validate whether templates complement other structural objectives.

Pretraining   CTBERT is pretrained on Python language corpus from CodeSearchNet (Husain et al., 2019), with the initial parameters of the pretrained GraphCodeBERT checkpoint. We set the max length of sequences as 512, and the batch size as 256. We train the model 40K batches, including 4K of warm-up batches, for 14 hours on 8 NVIDIA Tesla V100 with 32GB memory. The training is done by AdamW optimizer, where the learning rate is 2e-4. For each step, CTBERT is updated by Masked Language Modeling loss (MLM) (Devlin et al., 2019) added by template loss $\mathcal{L}_{\mathcal{T}}$ and DFG loss $\mathcal{L}_{DFG}$ without weighting. Except for the loss function, all the further pretrained baselines share the same setting for a fair comparison.

Finetuning   We follow the hyperparameters used in (Guo et al., 2021): The maximum length of NL and PL sequences is 128 and 256, respectively. The batch size is 32, trained 10 epochs for 13 hours on 2 NVIDIA Tesla V100 with 24GB memory. We use AdamW optimizer to update the model with 2e-5 as the learning rate. The maximum length of a template sequence is 256. The maximum number of nodes is 64, following (Guo et al., 2021). We truncate node tokens first, then template tokens, then code tokens at last. The model is trained and tested as the bi-encoder format, which takes query text and code sequence (with templates) separately, then predicts the similarity calculated by the dot product of two [CLS] vectors. All the experimental results are single run. Again, all the baselines share the same setting for a fair comparison.

C   Mining Templates

```
assert_statement →                              try_statement →
     comparison_operator   38.53%                    except_clause       48.14%
     call                  22.35%                    block               46.18%
     string                11.44%                    finally_clause        2.73%
     parenthesized_expression 6.60%                  else_clause           2.59%
     identifier             5.00%                    _simple_statements    0.29%
     boolean_operator       4.84%                    ERROR                 0.06%
     binary_operator        4.25%
     not_operator           2.72%               typed_parameter →
     attribute              2.07%                    type                50.00%
     false                  0.95%                    identifier          48.86%
     tuple                  0.54%                    dictionary_splat_pattern 0.57%
     concatenated_string    0.32%                    list_splat_pattern    0.56%
     subscript              0.17%
     integer                0.12%               decorator →
     await                  0.03%                    call                71.29%
     list                   0.03%                    identifier          17.23%
     conditional_expression 0.01%                    attribute           11.48%
     true                   0.01%
     dictionary             0.01%               dictionary_comprehension →
     list_comprehension     0.00%                    for_in_clause       44.91%
     none                   0.00%                    pair                44.19%
     generator_expression   0.00%                    if_clause           10.90%
```

Figure 7: Examples of $P(x \rightarrow y)$ extracted on Code-SearchNet Python corpus.

D   Template Span Task

Idiom task	MRR	
	CSN	AdvTest
Span	**0.6909**	0.3830
Alignment	0.6889	**0.3983**

Table 4: Comparison of CTBERT code search performances across the different template tasks.

This section explores whether the alignment loss can be replaced by a span prediction, inspired by NLP models for the factoid question answering task (Rajpurkar et al., 2016, 2018). Formally, we redefine the template sequence $\mathcal{T}$ into $\mathcal{T}'$, where $t' \in$ is a subsequence of the code token sequence C containing only the first and the last code tokens inside the template. Then, from $\mathcal{L}_{\mathcal{T}}$ in Eq (5), the alignment task $\mathcal{L}_{\mathcal{T}'}$ now asks the span of each template t':

$$\mathcal{L}_{\mathcal{T}'} = - \sum_{t' \in \bar{\mathcal{T}}'} \sum_{c \in C} [\mathbb{1}(c \in t') \log p_{t'c} \\ + (1 - \mathbb{1}(c \in t')) \log(1 - p_{t'c})], \quad (6)$$

The comparison of the two tasks is shown in Table 4, validating that the template alignment loss $\mathcal{L}_{\mathcal{T}'}$ is a more effective way for compositional generalization.

E   Template Span Task

Model	MRR	
	AdvTest	CSN
GraphCodeBERT (Python)	0.3493	0.6652
CTBERT	**0.3625**	**0.6778**

Table 5: **Pretraining from the CodeBERT checkpoint.**

From the CodeBERT checkpoint, we pretrain GraphCodeBERT (Python) and CTBERT on Code-SearchNet Python for 20K batches including 2K warm-up batches with a batch size of 512 and a learning rate of 2e-4. In this setting, GraphCode-BERT (Python) underperforms CTBERT. Not surprisingly, having a good initial point is more desirable in all models, but CTBERT is consistently the winner.

F   Dataset Statistics

Name	Train	Valid	Test
CodeSearchNet	1,099,694	11,108	45,283
CSN	251,820	13,914	14,918
AdvTest	-	9,604	19,210

Table 6: Statistics for the pre-training Python corpus CodeSearchNet (Husain et al., 2019; Guo et al., 2021), down-stream code search Python dataset CSN (Guo et al., 2021), and the normalized code search Python test set AdvTest (Lu et al., 2021).