CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X

Qinkai Zheng[§○*], **Xiao Xia**[§*], **Xu Zou**[§], **Yuxiao Dong**[§†], **Shan Wang**[○], **Yufei Xue**[○], **Zihan Wang**[§], **Lei Shen**[○], **Andi Wang**[○], **Yang Li**[○], **Teng Su**[◇], **Zhilin Yang**[§†], **Jie Tang**[§†‡]

Tsinghua University[§], Zhipu.AI[○], Huawei[◇]

Abstract

Large pre-trained code generation models, such as OpenAI Codex, can generate syntax- and function-correct code, making the coding of programmers more productive and our pursuit of artificial general intelligence closer. In this paper, we introduce CodeGeeX, a multilingual model with 13 billion parameters for code generation. CodeGeeX is pre-trained on 850 billion tokens of 23 programming languages as of June 2022. Our extensive experiments suggest that CodeGeeX outperforms multilingual code models of similar scale for both the tasks of code generation and translation on HumanEval-X. Building upon HumanEval (Python only), we develop the HumanEval-X benchmark for evaluating multilingual models by hand-writing the solutions in C++, Java, JavaScript, and Go. In addition, we build CodeGeeX-based extensions on Visual Studio Code, JetBrains, and Cloud Studio, generating 4.7 billion tokens for tens of thousands of active users per week. Our user study demonstrates that CodeGeeX can help to increase coding efficiency for 83.4% of its users. Finally, CodeGeeX is publicly accessible and in Sep. 2022, we open-sourced its code, model weights (the version of 850B tokens), API, extensions, and HumanEval-X at `https://github.com/THUDM/CodeGeeX`.

1 Introduction

Given the description of a human intent, such as "`write a factorial function`", can the machine automatically generate an executable program that addresses this need? This is the problem of *automatic program writing* that has been explored since the early days of computer science in the 1960s (Waldinger and Lee, 1969; Summers, 1977). From LISP-based pioneering deductive synthesis approaches (Waldinger and Lee, 1969; Summers, 1977) to modern program synthesis systems (Solar-Lezama, 2008; Polozov and Gulwani, 2015), to end-to-end code generation via deep neural networks (Mou et al., 2015; Svyatkovskiy et al., 2020; Sun et al., 2020), tremendous efforts have been made to enable machines to automatically write correct programs as part of the quest to artificial general intelligence.

By treating programs as language sequences, neural sequential architectures, such as recurrent neural networks and transformer (Vaswani et al., 2017), can be naturally applied to code generation. In fact, transformer-based techniques (Svyatkovskiy et al., 2020; Sun et al., 2020) have shown the potential of *automatic program writing* by starting to generate code that is both syntactically correct and

[*]QZ and XX contributed equally. Emails: `{qinkai|xiax19}@tsinghua.edu.cn`

[†]Team Leads: YD, ZY, and JT. Emails: `{yuxiaod|zhiliny|jietang}@tsinghua.edu.cn`

[‡]Corresponding author: JT. Email: `jietang@tsinghua.edu.cn`

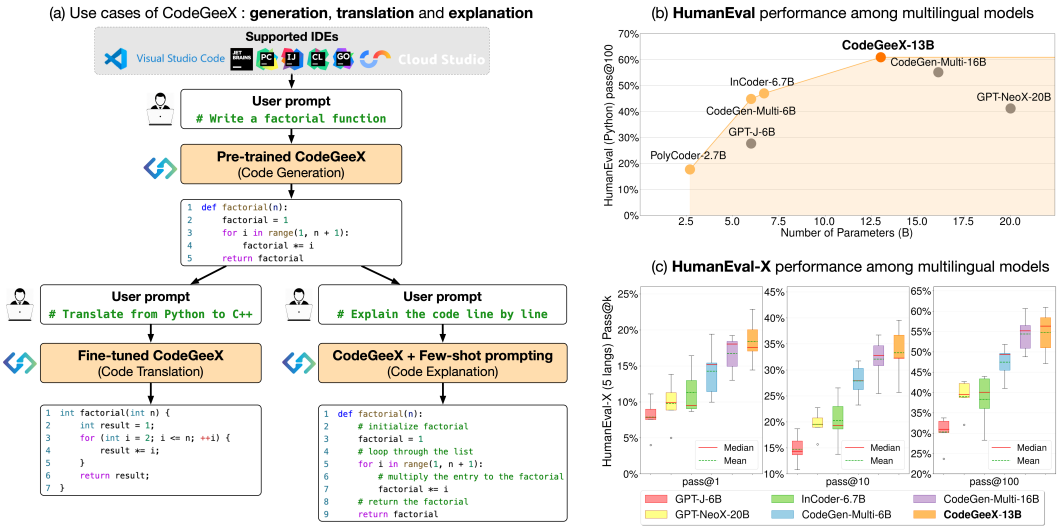


Figure 1: Summary of CodeGeeX. (a): In supported IDEs, users can interact with CodeGeeX by providing prompts. Different models are used to support three tasks: code generation, code translation and code explanation. (b) and (c): In HumanEval and our newly-proposed HumanEval-X, CodeGeeX shows promising multilingual abilities and consistently outperforms other multilingual code generation models.

consistent in 2020. This progress is significantly furthered when large language models (transformers with billions of parameters) meet the massive open-sourced code data.

Notably, the OpenAI Codex (Chen et al., 2021) model (Python only) with 12 billion (12B) parameters pioneered and demonstrated the potential of large code generation models pre-trained on billions lines of public code. By using the generative pre-training (GPT) strategy, Codex can solve introductory-level programming problems in Python with a high probability. Research studies (Ziegler et al., 2022) also show that 88% of users of GitHub Copilot—a paid service powered by Codex—feel more productive when coding with it. Since then, large pre-trained code models have been extensively developed, including DeepMind AlphaCode (Li et al., 2022), Salesforce CodeGen (Nijkamp et al., 2022), Meta InCoder (Fried et al., 2022), and Google PaLM-Coder-540B (Chowdhery et al., 2022).

In this work, we present CodeGeeX, a multilingual code generation model with 13 billion parameters, pre-trained on a large code corpus of 23 programming languages. It was trained on more than 850 billion tokens on a cluster of 1,536 Ascend 910 AI Processors between April and June 2022, and was publicly released in Sep. 2022 (Cf. the GitHub repo). CodeGeeX has the following properties. First, different from Codex in Chen et al. (2021), both CodeGeeX—the model itself—and how such scale of code models can be pre-trained are open-sourced, facilitating the understanding and advances in pre-trained code generation models. CodeGeeX also supports cross-platform inference on both Ascend and NVIDIA GPUs. Second, in addition to code generation and code completion as Codex and others, CodeGeeX supports the tasks of code explanation and code translation between language pairs (Cf. Figure 1 (a)). Third, it offers consistent performance advantages over well-known *multilingual* code generation models of the similar scale, including CodeGen-16B, GPT-NeoX-20B, InCode-6.7B, and GPT-J-6B (Cf. Figure 1 (b) and (c)).

We also build the free CodeGeeX extensions in several IDEs, currently including Visual Studio Code, JetBrains, and Tencent Cloud Studio (a Web IDE). It supports several different modes—code completion, function-level generation, code translation, code explanation, and customizable prompting—to help users' programming tasks in real time. Since its release, there are tens of thousands of daily active users, each of which on average makes 250+ API calls per weekday. As of this writing, the CodeGeeX model generates 4.7 billion tokens per week. Our user survey suggests that 83.4% of users feel the CodeGeeX extensions improve their programming efficiency.

Finally, we develop the HumanEval-X benchmark for evaluating multilingual code models as 1) HumanEval (Chen et al., 2021)—developed by OpenAI for evaluating Codex—and other bench-

Table 1: Large pre-trained language models related to programming languages in the literature.

	Model Properties			Dataset			Evaluation		
	Open	Multi-lingual	# Params	Source	Languages	Size	Multilingual Evaluation	Translation	Benchmark
Codex (Chen et al., 2021)	✗	✗	12B	Collected	Python	Code: 159GB	✗	✗	HumanEval, APPS
AlphaCode (Li et al., 2022)	✗	✓	41B	Collected	12 langs	Code: 715.1GB	✓	✗	HumanEval, APPS CodeContest
PaLM-Coder (Chowdhery et al., 2022)	✗	✓	8B, 62B, 540B	Collected	Multiple	Text: 741B tokens Code: 39GB (780B tokens trained)	✓	✓	HumanEval, MBPP TransCoder, DeepFix
PolyCoder (Xu et al., 2022)	✓	✓	2.7B	Collected	12 langs	Code: 253.6GB	✗	✗	HumanEval
GPT-Neo (Black et al., 2021)	✓	✓	1.3B, 2.7B	The Pile	Multiple	Text: 730GB Code: 96GB (400B tokens trained)	✗	✗	HumanEval
GPT-NeoX (Black et al., 2022)	✓	✓	20B	The Pile	Multiple	Text: 730GB Code: 95GB (473B tokens trained)	✗	✗	HumanEval
GPT-J (Wang and Komatsuzaki, 2021)	✓	✓	6B	The Pile	Multiple	Text: 730GB Code: 96GB (473B tokens trained)	✗	✗	HumanEval
Incoder (Fried et al., 2022)	✓	✓	1.3B, 6.7B	Collected	28 langs	Code: 159GB StackOverflow: 57GB (60B tokens trained)	✗	✗	HumanEval, MBPP CodeXGLUE
CodeGen-Multi (Nijkamp et al., 2022)	✓	✓	6.1B, 16.1B	BigQuery	6 langs	Code: 150B tokens Text: 355B tokens (1000B tokens trained)	✗	✗	HumanEval, MTPB
CodeGen-Mono (Nijkamp et al., 2022)	✓	✗	6.1B, 16.1B	BigPython	Python	Code: 150B tokens Text: 355B tokens (1300B tokens trained)	✗	✗	HumanEval, MTPB
CodeGeeX	✓	✓	13B	The Pile CodeParrot Collected	23 langs	Code: 158B tokens (850B tokens trained)	✓	✓	HumanEval-X, HumanEval MBPP, CodeXGLUE, XLCoST

marks (Austin et al., 2021; Hendrycks et al., 2021; Nijkamp et al., 2022) only consist of programming problems in a single language and 2) existing multilingual datasets (Ren et al., 2020; Lu et al., 2021; Zhu et al., 2022) use string similarity metrics like BLEU (Papineni et al., 2002) for evaluation rather than really verify the functional correctness of generated code. Specifically, for each problem—defined only for Python—in HumanEval, we manually rewrite its prompt, canonical solution, and test cases in C++, Java, JavaScript, and Go. In total, HumanEval-X covers 820 hand-written problem-solution pairs (164 problems, each having solutions in 5 languages). Importantly, HumanEval-X support the evaluation of both code generation and code translation between different languages.

The contributions of this work can be summarized as follows:

- We develop and release CodeGeeX, a 13B pre-trained 23-language code generation model that demonstrates consistent outperformance on code generation and translation over its multilingual baselines of the same scale.
- We build the CodeGeeX extensions on VS Code[4], JebBrains[5], and Tencent Cloud Studio. Compared to Copilot, it supports more diverse functions, including code completion, generation, translation, and explanation. According to the user survey, CodeGeeX can improve the coding efficiency for 83.4% of its users.
- We hand-craft the HumanEval-X benchmark to evaluate multilingual code models for the tasks of code generation and translation in terms of functional correctness, facilitating the understanding and development of pre-trained (multilingual) code models.

2 The CodeGeeX Model

CodeGeeX is a multilingual code generation model with 13 billion (13B) parameters, pre-trained on a large code corpus of 23 programming languages. As of June 22, 2022, CodeGeeX has been trained on more than 850 billion tokens on a cluster of 1,536 Ascend 910 AI Processors for over two months.

We introduce the CodeGeeX model and its design choices. The consensus reality is that it is computationally unaffordable to test different architectural designs for large pre-trained models (Brown et al., 2020; Chowdhery et al., 2022; Zhang et al., 2022; Zeng et al., 2022), though they define the inductive bias of models.

[4] `https://marketplace.visualstudio.com/items?itemName=aminer.codegeex`
[5] `https://plugins.jetbrains.com/plugin/20587-codegeex`

2.1 CodeGeeX's Architecture

The Transformer Backbone. Similar to recent pre-trained models, such as GPT-3 (Brown et al., 2020), PaLM (Chowdhery et al., 2022), and Codex (Chen et al., 2021), CodeGeeX follows the generative pre-training (GPT) architecture (Radford et al., 2018) with the decoder-only style for autoregressive (programming) language modeling. The core architecture of CodeGeeX is a 39-layer transformer decoder. In each transformer layer (in Figure 2), we apply a multi-head self-attention mechanism (Vaswani et al., 2017) followed by MLP layers, together with layer normalization (Ba et al., 2016) and residual connection (He et al., 2016). We use an approximation of GELU (Gaussian Linear Units) operation (Hendrycks and Gimpel, 2016), namely FastGELU, which is more efficient under the Ascend 910 AI Processor:

$$\text{FastGELU}(X_i) = \frac{X_i}{1 + \exp(-1.702 * |X_i|) * \exp(0.851 * (X_i - |X_i|))} \tag{1}$$

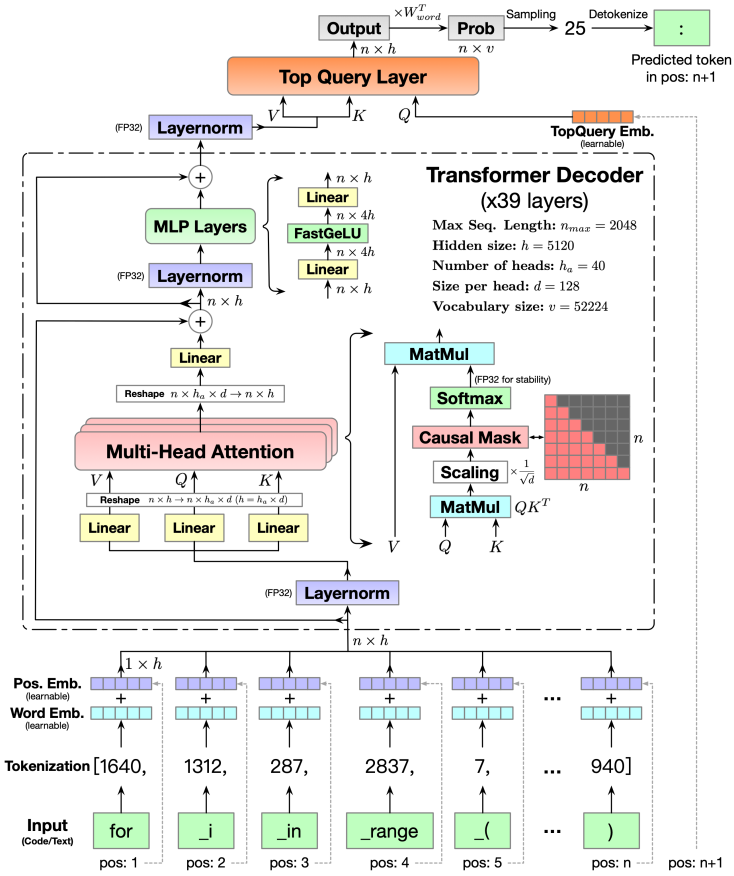


Figure 2: CodeGeeX's model architecture. CodeGeeX is a code generation model with 13B parameters, consisting of 39-layer left-to-right transformer decoders and a top query layer. It takes text/code tokens as input and outputs the probability of the next token autoregressively.

Generative Pre-Training Objective. By adopting the GPT paradigm (Radford et al., 2019; Chen et al., 2021), we train the model on a large amount of unlabeled code data. The principle is to iteratively take code tokens as input, predict the next token, and compare it with the ground truth. Specifically, for any input sequence $\{x_1, x_2, ..., x_n\}$ of length n, the output of CodeGeeX is a probability distribution of the next token $\mathbb{P}(x_{n+1}|x_1, x_2, ..., x_n, \Theta) = p_{n+1} \in [0, 1]^{1\times v}$, where Θ represents all parameters of the model and v is the vocabulary size. By comparing it with the real distribution, *i.e.*, a one-hot vector $y_{n+1} \in \{0, 1\}^{1\times v}$ of the ground-truth token, we can optimize the

cumulative cross-entropy loss:

$$\mathcal{L} = -\sum_{n=1}^{N-1} y_{n+1} \log \mathbb{P}(x_{n+1}|x_1, x_2, ..., x_n, \Theta) \tag{2}$$

The Top Query Layer and Decoding. The original GPT model uses a pooler function to obtain the final output. We use an extra query layer (Zeng et al., 2021) on top of all other transformer layers to obtain the final embedding through attention. As shown in Figure 2, the input of the top query layer replaces the query input X_{in} by the query embedding of position $n + 1$. The final output is multiplied by the transpose of word embedding matrix to get the output probability. For decoding strategies, CodeGeeX supports greedy, temperature sampling, top-k sampling, top-p sampling, and beam search. Finally, detokenization will turn the selected token ID into an actual word.

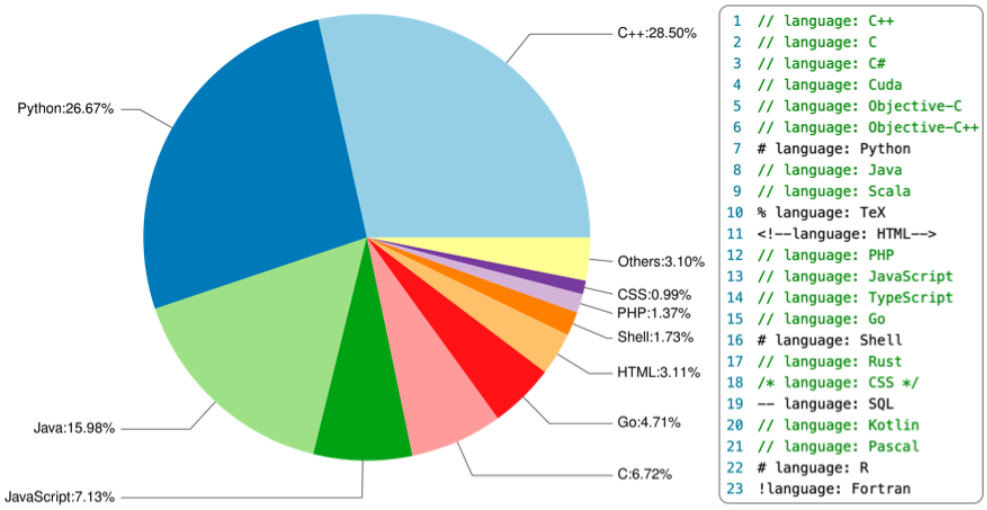


Figure 3: Language distribution and tags of CodeGeeX's data.

2.2 Pre-Training Setup

Code Corpus. The training corpus contains two parts. The first part is from open source code datasets, the Pile (Gao et al., 2020) and CodeParrot[6]. The Pile contains a subset of public repositories with more than 100 stars on GitHub, from which we select files of 23 popular programming languages including C++, Python, Java, JavaScript, C, Go, and so on. We identify the programming language of each file based on its suffix and the major language of the repository it belongs to. CodeParrot is another public Python dataset from BigQuery. The second part is supplementary data of Python, Java, and C++ directly scraped from GitHub public repositories that do not appear in the first part. We choose repositories that have at least one star and a total size within 10MB, then we filter out files that: 1) have more than 100 characters per line on average, 2) are automatically generated, 3) have a ratio of alphabet less than 40%, 4) are bigger than 100KB or smaller than 1KB. We format Python code according to the PEP8 standards.

Figure 3 shows the composition of the 158B-token training data, containing 23 programming languages. We divide the training data into segments of equal length. To help the model distinguish between multiple languages, we add a language-specific tag before each segment in the form of `[Comment sign]language: [LANG]`, *e.g.*, `# language: Python`.

Tokenization. The first step is to convert code snippets into numerical vectors. Considering that 1) there is a large number of natural language comments in code data, 2) the naming of variables, functions, and classes are often meaningful words, we treat code data the same as text data and apply the GPT-2 tokenizer (Radford et al., 2019). It is a BPE (Byte Pair Encoding) (Sennrich et al.,

[6] `https://huggingface.co/datasets/transformersbook/codeparrot`

2015) tokenizer that deals with the open-vocabulary problem using a fixed-size vocabulary with variable-length characters. The initial vocabulary size is 50,000, we encode multiple whitespaces as extra tokens following Chen et al. (2021) to increase the encoding efficiency. Specifically, `L` whitespaces are represented by `<|extratoken_X|>`, where `X=8+L`. Since the vocabulary contains tokens from various natural languages, it allows CodeGeeX to process tokens in languages other than English, like Chinese, French, Russia, Japanese and more. The final vocabulary size is $v = 52,224$. After tokenization, any code snippet or text description can be transformed into a vector of integers. More details can be found in Appendix A.2.

The Input Word and Positional Embeddings. Given the tokens, the next step is to associate each token with a word embedding. By looking up the token ID in a word embedding matrix $W_{word} \in \mathbb{R}^{v \times h}$, where $h = 5120$ is the hidden size, a learnable embedding $x_{word} \in \mathbb{R}^h$ is obtained for each token. To capture positional information, we also adopt learnable positional embedding that maps the current position ID to a learnable embedding $x_{pos} \in \mathbb{R}^h$, from $W_{pos} \in \mathbb{R}^{n_{max} \times h}$, where $n_{max} = 2048$ is the maximum sequence length. Then, two embeddings are added to obtain the input embeddings $x_{in} = x_{word} + x_{pos}$ for the model. Finally, the entire sequence can be turned into input embeddings $X_{in} \in \mathbb{R}^{n \times h}$, where n is the input sequence length.

2.3 CodeGeeX Training

Parallel Training on Ascend 910. CodeGeeX was trained on a cluster of the Ascend 910 AI processors (32GB) with Mindspore (v1.7.0). We faced and addressed numerous unknown technical and engineering challenges during pre-training, as Ascend and Mindspore are relatively new compared to NVIDIA GPUs and PyTorch/TensorFlow. The entire pre-training process takes two months on 192 nodes with 1,536 AI processors, during which the model consumes 850B tokens, equivalent to 5+ epochs (213,000 steps). Detailed configurations can be found in Table 2.

Table 2: Training configuration of CodeGeeX.

Category	Parameter	Value
Environment	Framework	Mindspore v1.7.0
	Hardwares	1,536x Ascend 910 AI processors
	Mem per GPU	32GB
	GPUs per node	8
	CPUs per node	192
	RAM per node	2048GB
Model	Model parameters	13B
	Vocabulary size	52224
	Position embedding	Learnable
	Maximum sequence length	2048
	Hidden size h	5120
	Feed-forward size $4h$	20480
	Feed-forward activation	FastGELU
	Layernorm epsilon	1e-5
	Layernorm precision	FP32
	Number of attention heads h_n	40
	Attention softmax precision	FP32
	Dropout rate	0.1
Parallelism	Model parallel size	8
	Data parallel size	192
	Global batch size	3072
Optimization	Optimizer	Adam
	Optimizer parameters	$\beta_1 = 0.9, \beta_2 = 0.999$
	Initial/final learning rate	1e-4/1e-6
	Warm-up step	2000
	Decay step	200000
	Learning rate scheduler	cosine decay
	Loss function $\mathcal{L}$	Cross entropy
	Loss scaling	Dynamic
	Loss scaling window	1000
	Trained steps	213000

To increase training efficiency, we adopt an 8-way model parallel training together with 192-way data parallel training, with ZeRO-2 (Rajbhandari et al., 2020) optimizer enabled to further reduce the memory consumption of optimizer states. Finally, the micro-batch size is 16 per node and the global batch size reaches 3,072.

Specifically, we use Adam optimizer (Kingma and Ba, 2014) to optimize the loss in Equation 2. The model weights are under FP16 format, except that we use FP32 for layer-norm and softmax for higher precision and stability. The model takes about 27GB of GPU memory. We start from an initial learning rate 1e-4, and apply a cosine learning rate decay by:

$$lr_{current} = lr_{min} + 0.5 * (lr_{max} - lr_{min}) * (1 + \cos(\frac{n_{current}}{n_{decay}}\pi)) \tag{3}$$

During the two-month training, the training loss of CodeGeeX continues to decrease as the training goes on. We evaluate the checkpoints on HumanEval-X code generation task and observe that the performance is continuously increasing. See details in Figures 13 and 14 in Appendix A.3.

Training Efficiency Optimization. Over the course of the training, we actively attempted to optimize the Mindspore framework to release the power of Ascend 910. Notably, we adopt the following techniques that significantly improve training efficiency:

- Kernel fusion: We fuse several element-wise operators to improve calculation efficiency on Ascend 910, including `Bias+LayerNorm`, `BatchMatmul+Add`, `FastGeLU+Matmul`, `Softmax`, etc. We also optimize `LayerNorm` operator to support multi-core calculation.
- Auto Tune optimization: When loading models, Mindspore first compiles them to static computational graphs. It uses the Auto Tune tool to optimize the choice of operators (*e.g.*, matrix multiplication in different dimensions). And it applies graph optimization techniques to deal with operator fusion and constant folding.

Table 3 shows the comparison of training efficiency before and after our optimization. The overall efficiency is measured by trained tokens per day. We observe that the efficiency per processor was improved 3× compared to the non-optimized implementation and the overall token throughput of 1,536 GPUs was improved by 224%.

Table 3: Training efficiency (before and after optimization).

	Before	**After**
Device	Ascend 910	Ascend 910
#GPUs	1536	1536
Parallelism	Data parallel + Model parallel	Data parallel + Model parallel
Sequence length	2048	2048
Global batch size	2048	3072
Step time(s)	15s	10s
Overall efficiency	24.2B tokens/day	54.3B tokens/day

2.4 Fast Inference

To serve the pre-trained CodeGeeX, we implement a pure PyTorch version of CodeGeeX that supports inference on NVIDIA GPUs. To achieve fast and memory-efficient inference, we apply both quantization and acceleration techniques to the pre-trained CodeGeeX.

Quantization. We apply post-training quantization techniques to decrease memory consumption of CodeGeeX during inference. We transform weights W in all linear transformations from FP16 to INT8 using the common absolute maximum quantization:

$$W_q = \text{Round}(\frac{W}{\lambda}), \lambda = \frac{\text{Max}(|W|)}{2^{b-1}-1} \tag{4}$$

where b is the bitwidth and $b = 8$. λ is the scaling factor. This quantization transform FP16 values in $[-\text{Max}(|W|), \text{Max}(|W|)]$ to integers between $[-127, 127]$.

As in Table 4, the memory consumption of CodeGeeX decreases from ∼26.9GB to ∼14.7GB (down by 45.4%), allowing CodeGeeX inference on one RTX 3090 GPU. Importantly, Figure 4 shows that

the quantization only slightly affects the performance on the code generation task (Cf Section 3.2 for details about HumanEval-X.).

Table 4: GPU memory and inference time of CodeGeeX w/ and w/o quantization on different GPUs and frameworks.

Implementation	GPU	Format	L=128		L=256		L=512		L=1024		L=2048	
			Mem (G)	Time (s)	Mem (G)	Time (s)	Mem (G)	Time (s)	Mem (G)	Time (s)	Mem (G)	Time (s)
Pytorch	3090	FP16	OOM									
Pytorch	A100	FP16	26.9	3.66	27.1	7.16	27.6	14.35	28.9	29.95	34.6	63.20
Megatron	A100	FP16	26.9	4.55	27.1	9.40	27.6	18.65	28.9	37.63	34.6	75.02
Megatron	2xA100	FP16	17.9	5.11	22.1	10.17	22.1	20.42	22.1	41.04	22.1	82.93
Megatron	4xA100	FP16	8.0	5.25	11.1	10.35	11.1	20.89	11.1	41.86	11.1	84.95
Megatron	8xA100	FP16	4.8	5.47	5.7	11.04	6.3	22.38	6.5	45.50	6.5	90.47
Pytorch	3090	INT8	14.7	13.82	15.7	27.10	16.1	55.42	17.1	110.83	18.7	228.67
Pytorch	A100	INT8	14.7	9.40	15.7	18.65	16.1	37.38	17.1	75.60	18.7	155.01
LLM.int8()	A100	INT8	14.7	20.65	15.1	35.86	15.6	72.76	16.7	147.59	22.3	301.93
Oneflow	A100	FP16	25.9	2.61	26.2	5.25	27.0	10.89	29.0	22.49	33.6	47.54
Oneflow	A100	INT8	13.6	1.85	13.9	3.73	14.4	7.83	15.9	16.24	21.1	35.98
FastTrans	A100	FP16	26.0	2.43	26.1	4.93	26.3	10.21	26.7	22.60	27.5	50.09
FastTrans	A100	INT8	14.9	**1.61**	15.0	**3.24**	15.2	**6.35**	15.6	**14.32**	17.4	**34.96**

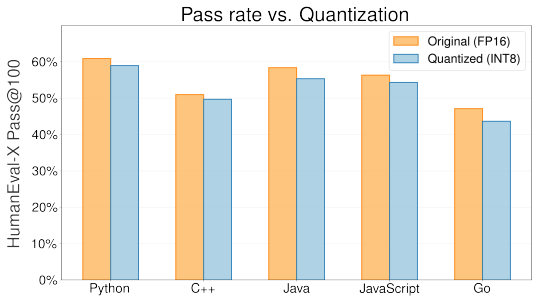


Figure 4: CodeGeeX vs. its quantized version on code generation of HumanEval-X.

Acceleration. After quantization, we further implement a faster version of CodeGeeX using the NVIDIA FasterTransformer (FastTrans). It supports highly-optimized operations by using layer fusion, GEMM autotuning, and hardware-accelerated functions. For INT8 quantized version, we also implement a custom kernel that accelerates the mixed precision matrix multiplication between INT8 weights and FP16 activation vectors. According to Table 4, the INT8 quantization plus FastTrans implementation achieves the fastest inference speed and the lowest GPU memory consumption on a single GPU. The inference time per token is within 13ms (1.61 seconds / 128 tokens). We also compare the inference speed with implementations in LLM.int() (Dettmers et al., 2022) and Oneflow (Yuan et al., 2021).

3 The HumanEval-X Benchmark

We develop the HumanEval-X benchmark[7] for evaluating multilingual code models. There are 164 code problems defined for five major languages: C++, Java, JavaScript, Go, and Python, resulting in 164×5=820 problem-solution pairs. For each problem, it supports both code generation and code translation. Examples of the problems can be found in Appendix A.5.

3.1 HumanEval-X: A Multilingual Benchmark

HumanEval (Chen et al., 2021) has been developed to evaluate Codex by OpenAI. However, similar to MBPP (Austin et al., 2021) and APPS (Hendrycks et al., 2021), it only consists of handcrafted programming problems in Python, thus cannot be directly applied to systematically evaluate the performance of multilingual code generation.

[7]The HumanEval-X dataset and docker image are at https://hub.docker.com/r/codegeex/codegeex.

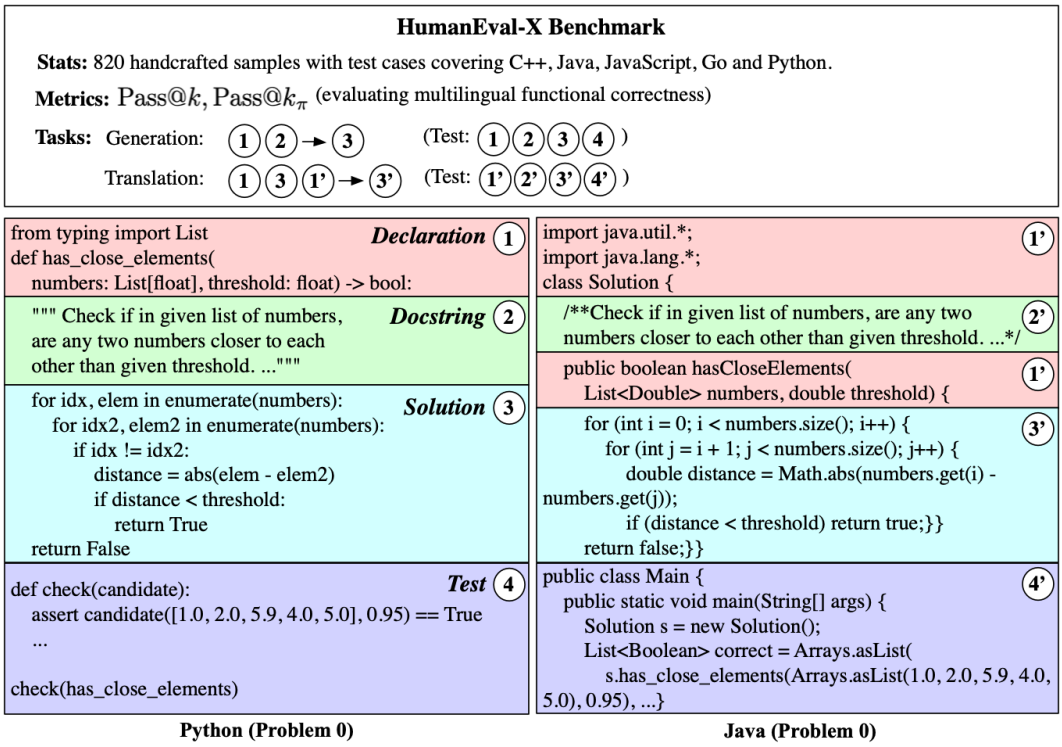


Figure 5: An illustration of code *generation* and *translation* tasks in HumanEval-X. Declarations, docstrings, solutions, and test cases are marked with red, green, blue, and purple respectively. *Generation* uses declaration and docstring as input to generate the solution. *Translation* uses declaration in both languages and solution in source language as input, to generate solution in the target language (docstring is not used to prevent models from directly solving the problem).

To this end, we propose to develop a multilingual variant of HumanEval, referred to as HumanEval-X. This is not trivial. For each problem, defined only for Python, in HumanEval, we manually rewrite its prompt, canonical solution, and test cases in the other four languages—C++, Java, JavaScript, and Go. Altogether, we have 820 problem-solution pairs in total in HumanEval-X, each comprising the following parts:

- **task_id**: programming language and numerical problem id, *e.g.*, `Java/0` represents the 0-th problem in Java;
- **declaration**: function declaration including necessary libraries or packages;
- **docstring**: description that specifies the functionality and example input/output;
- **prompt**: function declaration plus docstring;
- **canonical_solution**: a verified solution to the problem;
- **test**: test program including test cases.

Each problem-solution pair in HumanEval-X supports both code generation code translation. An illustrative example is shown in Figure 5. We take the following efforts to make sure that the rewritten code conforms to the programming style of the corresponding language. First, we use the customary naming styles, like `CamelCase` in Java, Go, and JavaScript, and `snake_case` in C++. Second, we put the docstrings before the function declaration in Java, JavaScript, C++, and Go. Symbols in docstrings are modified, *e.g.*, single quotes are replaced by double quotes in some languages, and keywords like `True/False`, `None` are also replaced. Third, we refine test cases according to language-specific behaviors, rather than forcing the programs to return the same result for different languages. For example, when converting an integer to a binary string, Python method `bin` adds a prefix `''0b''` before the string while Java method `Integer.toBinaryString` does not, so we remove such prefix in Java test cases. Last, we also take care of the rounding function. In Python, `round` converts half to the

closest even number, unlike in other languages. Thus, we change the test cases to match the rounding implementations in each language.

3.2 HumanEval-X: Tasks

In HumanEval-X, we evaluate two tasks: code generation and code translation.

Code Generation. The task of code generation takes a problem description (e.g., ‘‘`write a factorial function`’’) as input and generates the solution in the selected languages (Cf Figure 1 (a)). Specifically, the model takes in the prompt including declaration and docstrings, and generates the implementation of the function. Note that HumanEval-X uses the same problem set for all the five languages, thus, for solving each problem, it supports either one single language or multiple languages simultaneously.

Code Translation. The task of code translation takes the implementation of a problem in the source language and generates its counterpart implementation in the target language. Precisely, its input includes the function declaration and a canonical solution in the source language (e.g., Python). The model should translate the solution to the target language. Adding declaration in the target language restricts function names and variable types, making the evaluation easier, especially under the zero-shot setting. To prevent the models from directly solving the problem rather than translating, we do not include the docstrings. HumanEval-X supports the translation between all pairs of 5 languages, that is, in total 20 source-target language pairs.

Metric. For both tasks, we use test cases to evaluate the exact functional correctness of the generated code, measuring the performance with pass@k (Kulal et al., 2019), making it real-world useful and also completely different from the string similarity metrics like BLEU (Papineni et al., 2002), and CodeBLEU (Ren et al., 2020; Lu et al., 2021; Zhu et al., 2022). Specifically, we use the unbiased method to estimate pass@k (Chen et al., 2021):

$$\text{pass@}k := \mathbb{E}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}], n = 200, k \in \{1, 10, 100\} \tag{5}$$

where n is the total number of generation (n=200 in this work), k is the sampling budget (typically $k \in \{1, 10, 100\}$) and c is the number of samples that pass all test cases. We average over the problem set to get the expectation. $1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$ is the estimated pass@k for a single problem, and $\mathbb{E}$ is the expectation of pass@k over all problems. In practice, we average single-problem pass@k among all test-set problems to get the expectation.

Multilingual Metric with Budget Allocation. Unlike mono-lingual models, multilingual code models can solve problems by allocating generation budgets to various languages to increase the sampling diversity and improve the solve rate. Given a budget k, we can distribute part of it n_i to each language with the assignment

$$\pi = (n_1, n_2, ..., n_m), \sum_{i=1}^{m} n_i = k, \tag{6}$$

where n_i is the generation budget assigned to language i, m is the number of candidate languages. Under an assignment $\pi = (n_1, ...n_m)$, for a problem p, the pass@k_π can be estimated by:

$$\text{pass@}k_\pi = \mathbb{E}[1 - \prod_{i=1}^{m} \frac{\binom{n-c_i}{n_i}}{\binom{n}{n_i}}], \tag{7}$$

where n is the total number of generation, n_i is the sampling budget and c_i is the number of samples that pass all test cases for language i. We show in Section 4.3 that multilingual models can benefit from budget allocation strategies and have higher solve rate than using any single language.

4 Evaluating CodeGeeX on HumanEval-X

We evaluate CodeGeeX for the code generation and translation tasks on the multilingual benchmark HumanEval-X. By inheriting from HumanEval, the HumanEval-X results on Python are equivalent to the evaluation on HumanEval.

Table 5: Results of **code generation** task in HumanEval-X.

Language	Metric	GPT-J -6B	GPT-NeoX -20B	InCoder -6.7B	CodeGen -Multi-6B	CodeGen -Multi-16B	CodeGeeX -13B (ours)
Python	pass@1	11.10%	13.83%	16.41%	19.41%	19.22%	**22.89%**
	pass@10	18.67%	22.72%	26.55%	30.29%	34.64%	**39.57%**
	pass@100	30.98%	39.56%	43.95%	49.63%	55.17%	**60.92%**
C++	pass@1	7.54%	9.90%	9.50%	11.44%	**18.05%**	17.06%
	pass@10	13.67%	18.99%	19.30%	26.23%	30.84%	**32.21%**
	pass@100	30.16%	38.75%	36.10%	42.82%	50.90%	**51.00%**
Java	pass@1	7.86%	8.87%	9.05%	15.17%	14.95%	**20.04%**
	pass@10	14.37%	19.55%	18.64%	31.74%	**36.73%**	36.70%
	pass@100	32.96%	42.23%	40.70%	53.91%	**60.62%**	58.42%
JavaScript	pass@1	8.99%	11.28%	12.98%	15.41%	**18.40%**	17.59%
	pass@10	16.32%	20.78%	22.98%	27.92%	**32.80%**	32.28%
	pass@100	33.77%	42.67%	43.34%	48.81%	**56.48%**	56.33%
Go	pass@1	4.01%	5.00%	8.68%	9.98%	13.03%	**14.43%**
	pass@10	10.81%	15.70%	13.80%	23.26%	25.46%	**25.68%**
	pass@100	23.70%	32.08%	28.31%	41.01%	**48.77%**	47.14%
Average	pass@1	7.90%	9.78%	11.33%	14.28%	16.73%	**18.40%**
	pass@10	14.77%	19.55%	20.25%	27.89%	32.09%	**33.29%**
	pass@100	30.32%	39.06%	38.48%	47.24%	54.39%	**54.76%**

4.1 Evaluation Settings

Baselines. We compare CodeGeeX with five competitive open-source baselines: GPT-J-6B (Wang and Komatsuzaki, 2021), GPT-NeoX-20B (Black et al., 2022), InCoder-6.7B (Fried et al., 2022), and CodeGen-Multi-6B/16B (Nijkamp et al., 2022). These models are all trained on multilingual code data, but is previously only evaluated in HumanEval (Python). And they are closer to the scale of CodeGeeX or even larger, while smaller models in the literature are ignored. For all baselines, we use the versions available on HuggingFace (Wolf et al., 2019). We follow the experimental settings of HumanEval-X in Section 3.2. Further details can be found in Appendix A.3.

Environment. Experiments are conducted by using the NVIDIA A100-SXM-40GB GPUs with Linux system. We design a distributed framework for generation based on ZeroMQ to balance GPU loads. All generated codes are tested in language-specific environments with necessary packages installed.

Decoding Strategy. We use temperature sampling ($t \in [0, 1]$) and nucleus sampling ($p \in [0, 1]$) for generation. For CodeGeeX in code generation, we use $t = 0.2, p = 0.95$ for pass@1 and $t = 0.8, p = 0.95$ for pass@10 and pass@100 (except for Go and JavaScript, where $p = 0.9$). For CodeGeeX in code translation, we use $t = 0.2, p = 0.95$ for pass@1 and $t = 0.8, p = 0.95$ for pass@10 and pass@100 for all language pairs. For the fine-tuned CodeGeeX-13B-FT used for code translation, we use $p = 0.95$. For all baselines in both tasks, we use $t = 0.2, p = 0.95$ for pass@1, $t = 0.8, p = 0.95$ for pass@10 and pass@100. All pass@k, $k \in \{1, 10, 100\}$ results are estimated with $n = 200$. The maximum number of generated tokens is set to 1024 for all models.

4.2 Results of Code Generation and Translation

Multilingual Code Generation. Table 5 and Figure 6 report the code generation results in terms of the pass@k, $k \in \{1, 10, 100\}$ for CodeGeeX and five baseline models on five programming languages. CodeGeeX significantly outperforms models trained with mixed corpora (GPT-J-6B and GPT-NeoX-20B), even though GPT-NeoX-20B has much more parameters. For models trained on codes, CodeGeeX outperforms those with smaller scales (InCoder-6.7B, CodeGen-Multi-6B) by a large margin, and is competitive with CodeGen-Multi-16B with a larger scale. CodeGeeX achieves the best average performance among all models, even slightly better than the larger CodeGen-Multi-16B in all three metrics (0.37%$\sim$1.67% improvements). When considering individual languages, models have preferences highly related to the training set distribution. For example, the best language

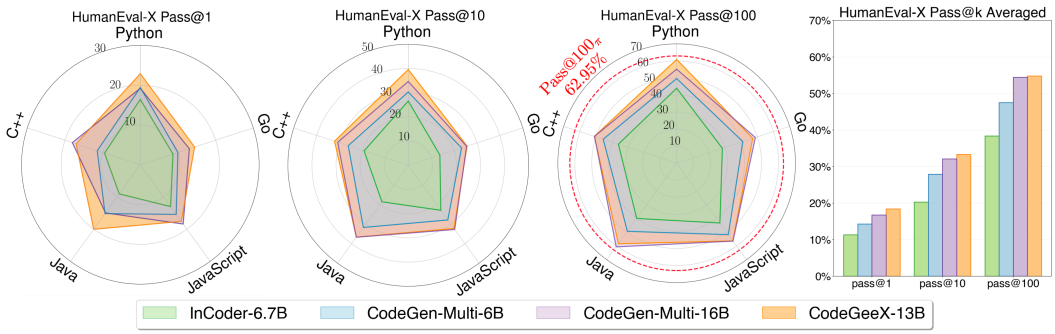


Figure 6: Results of **code generation** task in HumanEval-X. Left: Detailed pass@k performance in five languages. Right: CodeGeeX achieves the highest average performance compared with other open-sourced multilingual baselines. We also find that it gains performance when the sampling budgets are properly distributed to multiple languages.

Table 6: Results of **code translation** task in HumanEval-X.

	Model	Target Language: Python			C++			Java			JavaScript			Go		
		@1	@10	@100	@1	@10	@100	@1	@10	@100	@1	@10	@100	@1	@10	@100
Py	InCoder-6.7B	-	-	-	26.11	41.00	54.25	26.74	42.66	61.20	37.05	58.85	78.91	15.69	27.57	43.67
	CodeGen-Multi-16B	-	-	-	**35.94**	**47.81**	59.37	29.27	45.70	64.45	**43.40**	**66.26**	**82.55**	**28.87**	**41.01**	**57.72**
	CodeGeeX-13B	-	-	-	26.54	43.56	56.48	25.84	41.52	59.72	23.22	47.33	65.87	9.56	23.83	33.56
	CodeGeeX-13B-FT	-	-	-	34.16	46.86	**61.22**	**41.98**	**58.17**	**72.78**	34.81	53.05	66.08	16.41	30.76	46.37
C++	InCoder-6.7B	34.37	58.41	78.57	-	-	-	34.04	57.02	68.70	37.05	65.05	79.61	25.54	39.11	58.02
	CodeGen-Multi-16B	33.83	55.37	76.64	-	-	-	43.20	69.84	**88.82**	**54.51**	**71.50**	**83.14**	**27.94**	**49.73**	**68.32**
	CodeGeeX-13B	27.18	49.02	67.69	-	-	-	22.56	40.91	64.08	30.23	55.68	75.58	8.64	18.79	31.76
	CodeGeeX-13B-FT	**62.79**	**80.39**	**87.10**	-	-	-	**71.68**	**81.62**	85.84	50.83	64.55	74.57	16.71	34.18	52.98
Java	InCoder-6.7B	42.76	65.55	80.43	40.01	55.17	70.39	-	-	-	43.20	**68.24**	**84.39**	21.58	35.20	54.97
	CodeGen-Multi-16B	52.73	69.30	82.74	41.42	54.68	65.50	-	-	-	**57.65**	67.90	79.22	**34.00**	**48.49**	**67.94**
	CodeGeeX-13B	43.41	68.46	84.03	39.33	58.48	72.36	-	-	-	44.19	64.22	82.89	17.17	32.74	47.71
	CodeGeeX-13B-FT	**75.03**	**87.71**	**95.13**	**49.67**	**65.65**	**75.40**	-	-	-	49.95	62.82	79.64	18.85	32.92	48.93
JS	InCoder-6.7B	23.18	50.47	67.26	35.47	54.48	70.71	30.67	50.90	71.03	-	-	-	25.79	42.96	61.47
	CodeGen-Multi-16B	35.52	52.23	69.78	35.41	53.12	64.47	33.79	56.06	74.00	-	-	-	**33.38**	**49.08**	**64.14**
	CodeGeeX-13B	31.15	54.02	72.36	30.32	51.63	69.37	24.68	48.35	69.03	-	-	-	11.91	26.39	39.81
	CodeGeeX-13B-FT	**67.63**	**81.88**	**89.30**	**46.87**	**60.82**	**73.18**	**56.55**	**70.27**	**80.71**	-	-	-	16.46	32.99	50.29
Go	InCoder-6.7B	34.14	54.52	70.88	30.45	48.47	62.81	34.52	53.95	69.92	39.37	**63.63**	**80.75**	-	-	-
	CodeGen-Multi-16B	38.32	50.57	68.65	32.95	45.88	59.56	36.55	59.12	78.70	38.93	56.68	70.68	-	-	-
	CodeGeeX-13B	35.92	56.02	77.32	29.83	41.98	58.15	22.89	41.04	61.46	25.24	46.50	69.93	-	-	-
	CodeGeeX-13B-FT	**57.98**	**79.04**	**93.57**	**38.97**	**53.05**	**63.92**	**54.22**	**69.03**	**79.40**	**43.07**	59.78	74.04	-	-	-

for CodeGeeX is Python while the best language for CodeGen-Multi-16B is Java. Examples of CodeGeeX generation can be found in Appendix A.5.

Cross-Lingual Code Translation. Table 6 illustrates the results on code translation. For CodeGeeX, we evaluate both the original version CodeGeeX-13B and the fine-tuned CodeGeeX-13B-FT. CodeGeeX-13B-FT is first fine-tuned using the training set of code translation task in XLCoST (Zhu et al., 2022), and then continuously fine-tuned by a small amount of Go data (since Go is missing in XLCoST). Among all translation pairs, CodeGeeX-13B-FT performs the best on pass@100 in 11 out of the 20, while CodeGen-Multi-16B is the best on 7 of them. We also observe a clear preference of languages by different models. CodeGeeX performs the best when translating other languages to Python and C++, while CodeGen-Multi-16B performs better when translating to JavaScript and Go.

Test Result Analysis. We group the samples' test results into five categories: passing, wrong answer, runtime error, syntax/semantic error and unfinished generation, and calculate the proportion of results for each model. Runtime error includes out-of-bound index, wrong string format, etc; syntax/semantic error indicates errors detected by syntax or semantic check, like compilation error in compiled languages and syntax, undefined or type error in interpreted languages; unfinished generation means failing to complete one function within maximum length.

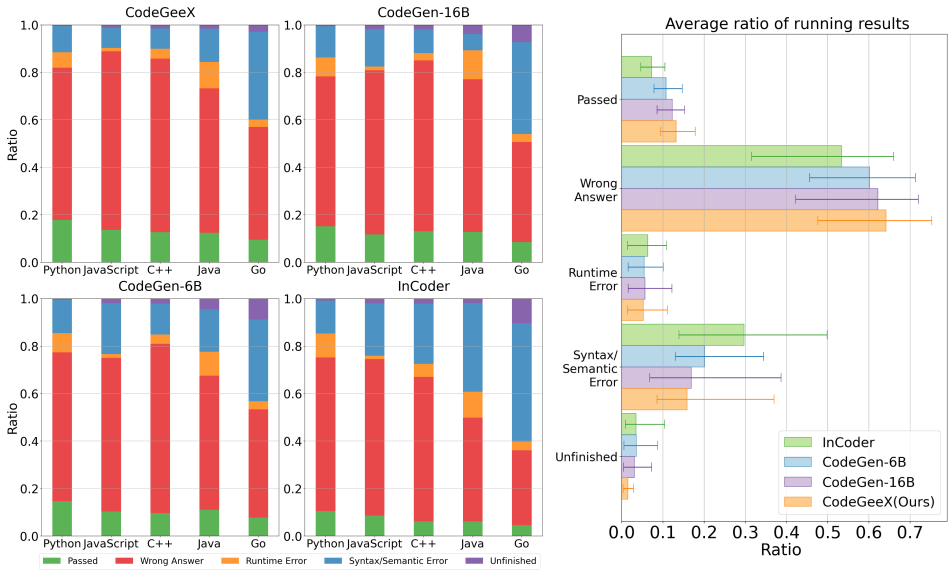


Figure 7: **Left**: the proportions of running results of four models for each language. **Right**: the average result ratios across four models, with lines representing minimum and maximum values. For each model and each language, we study 200 samples generated under $t = 0.8$ and $p = 0.95$.

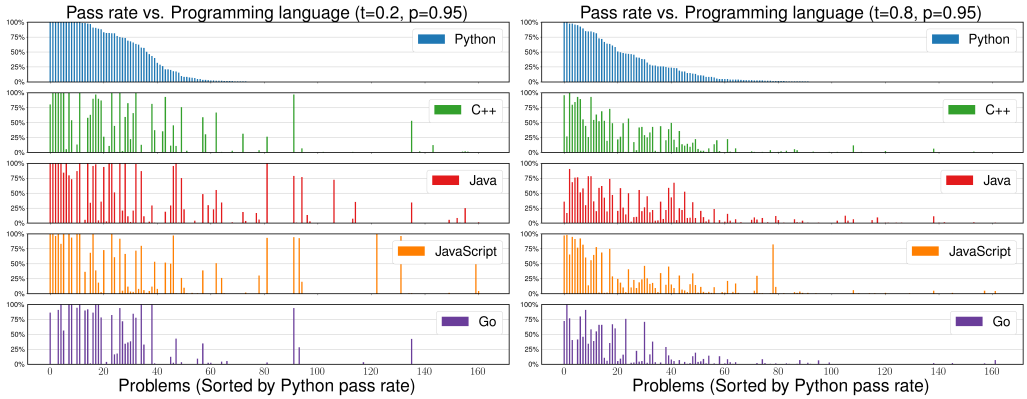


Figure 8: In HumanEval-X, each problem's pass rate varies when generating in different programming languages with CodeGeeX. **Left:** $t = 0.2, p = 0.95$; **Right:** $t = 0.8, p = 0.95$.

Figure 7 shows the proportions of running results of four models. For all languages, the most common error type is wrong answer, with ratio ranging from 0.44 to 0.75 except for Go, showing that code generation models at the current stage mainly suffer from incorrect code logic rather than semantics. Go samples have a high syntax error rate, which may be due to Go having strict restrictions on syntax and forbidding unused variables and imports, failing to compile many logically correct codes. CodeGeeX has less rate to generate code that produces runtime, syntax, or semantic errors.

4.3 The Multilingual Pre-Training Helps Problem Solving

We perform studies to understand whether and how multilingual pre-training can benefit problem-solving of CodeGeeX.

Exploration vs. Exploitation under Fixed Budgets. Given a fixed budget k, pass@k evaluates the ability of models generating at least 1 correct solution under k generations. Previous works (Chen et al., 2021; Li et al., 2022) have already discovered that there's a trade-off between exploration and exploitation: When the budget is small, it is better to use a low temperature to ensure accuracy on

Table 7: Results for fixed-budget multilingual generation on HumanEval-X.

Metric	Method	GPT-J -6B	GPT-NeoX -20B	InCoder -6.7B	CodeGen -Multi-6B	CodeGen -Multi-16B	CodeGeeX -13B
pass@k_π ($k = 100$)	Best Single	33.77%	42.67%	43.95%	53.19%	60.62%	**60.92%**
	Uniform	36.40%	44.75%	43.89%	53.47%	61.01%	**62.41%**
	Weighted	*36.76%*	*44.97%*	*45.60%*	*53.94%*	*61.34%*	***62.95%***

easy problems. When the budget is large, instead, adjusting a higher temperature is vital, as it makes the model more likely to find at least one solution for difficult problems.

Pass Rate Distribution vs. Languages. Unlike monolingual models, multilingual models can solve problems using various programming languages. In Figure 8, we observe that the pass rate distribution of problems against different languages are diverse. This inspires us to use budget allocation methods to help improve the diversity of the generated solutions.

Budget Allocation Strategies. We compare three basic strategies: *Best Single* chooses a single language with the best performance; *Uniform* allocates the budget uniformly; *Weighted* allocates the budget to multiple languages based on their proportions in the training corpus (detailed weights can be found in Appendix Table 9). Table 7 illustrates how budget allocation improves the multilingual generation. Both **Uniform** and **Weighted** outperform **Best Single** by promoting a more diverse generation, which gives a higher chance of solving problems. **Weighted** is slightly better due to the prior knowledge on the model. For model-wise comparison, CodeGeeX shows up a decent advantage over other baselines in both strategies, which suggests that it might have a more diverse solution set under multiple languages. Programming languages are created with a specific purpose and unique design; in real-world scenarios, multilingual models might take this advantage for certain tasks.

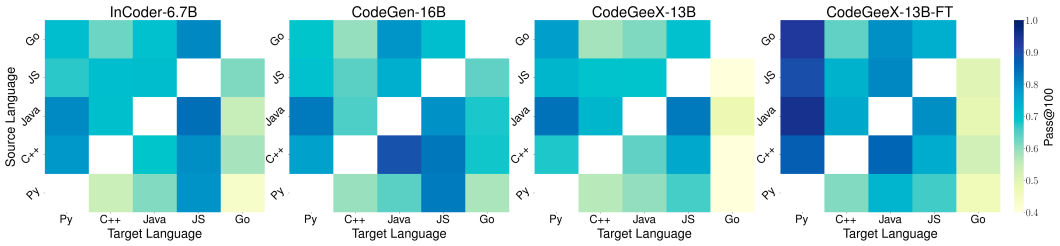


Figure 9: The performance of translating A-to-B is negatively correlated with B-to-A. Such asymmetry indicates that multilingual models still lack of high-level understanding between languages.

Negative Correlations in Pair-Language Translation. When evaluating the translation ability in HumanEval-X, an interesting observation is that the performance of A-to-B and B-to-A are usually negatively-correlated, shown in Figure 9. Such asymmetry suggests that multilingual code generation models may have imbalanced focus on source and target languages during code translation. We provide two possible explanations. First, language distributions in training corpus differ a lot, resulting in different level of generation ability. For example, the ratio of Python is 26.6% (vs. Go 4.7%) in CodeGeeX training corpus, and average pass@100 of *Others-to-Python* reaches ~90% (vs. *Others-to-Go* only ~50%). Second, some languages are themselves harder to automatically write with syntactic and semantic accuracy due to language-dependent features, affecting translation performance as target languages. For instance, Go, which models translate poorly into, has more constraints on syntax level, forbidding unused variables or imports.

5 The CodeGeeX Tools and Users

Based on CodeGeeX, we build open-source extensions for IDEs including VS Code, JetBrains and Cloud Studio. The extensions support code generation, completion, translation and explanation, aiming at improving the development efficiency of programmers. As of this writing, CodeGeeX has served tens of thousands of users, with an average of 250+ API calls per active user per weekday. It currently generates 4.7+ billion tokens per week, which has been steadily growing since its release.

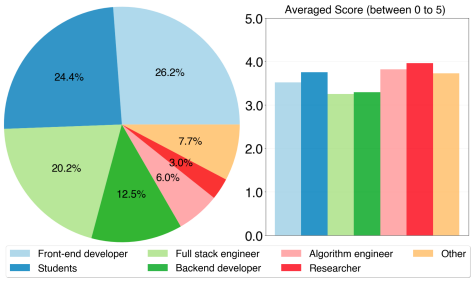


Figure 10: Profession vs. satisfaction. Left: Profession distribution. Right: Averaged rating score of CodeGeeX extensions.

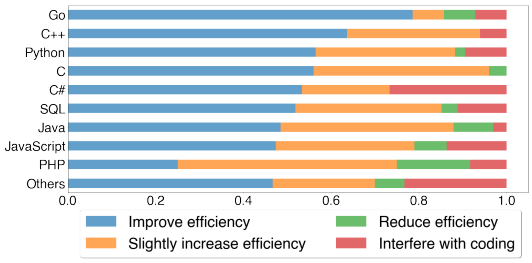


Figure 11: Survey on "Has CodeGeeX improved your coding efficiency?". Over 83.4% of users have positive answers.

We perform a survey on CodeGeeX's user experience from 168 users covering *front-end developer*, *backend developer*, *full stack engineer*, *algorithm engineer*, *students*, *researcher*, and *other programmers*. Figure 10 illustrates users' profession distribution and the satisfaction score. We evaluate the satisfaction considering five dimensions, "Ease of Use", "Reliability", "Feature", "Visual", "Speed", each scored from 0 to 5. Figure 10 shows that the majority of users have positive experiences with CodeGeeX, especially for researchers and students, while there is still room for improvement for professional developers. This can be interpreted by our training code corpus: open-sourced repositories contain many introductory or research projects, while production codes are often close-sourced. To increase the CodeGeeX's capability in professional domain, these codes are needed in the future.

We further investigate how multilinguality of CodeGeeX help coding. Figure 11 illustrates how users evaluate the helpfulness of CodeGeeX during development. There are on average over 83.4% of users think CodeGeeX can improve or slightly increase their coding efficiency, especially for mainstream programming languages like Go, C++, Python, C, C#, etc. Note that these well-performing programming languages also appear more frequently in the training data (Figure 3), which encourages us to train CodeGeeX on more language-specific data to enhance its capability.

6 Conclusion

We introduce CodeGeeX, a 13B pre-trained 23-language code generation model, as well as we build HumanEval-X, to fill the gap of multilingual code generation. CodeGeeX consistently outperforms open-sourced multilingual baselines of the same scale on code generation and translation tasks. The extensions built on CodeGeeX bring significant benefits in increasing coding efficiency. The multilinguality of CodeGeeX brings the potential of solving problems with an ubiquitous set of formalized languages. We open sourced CodeGeeX aiming to help researchers and developers to widely take benefit of large pre-trained models for code generation.

The multilingual ability of CodeGeeX shows the potential of solving problems with a ubiquitous set of formalized languages. Here, we share three of our observations as the future directions.

First, we find that the model capacity is essential for multilingual programming ability. It is not trivial for the model to benefit from learning multiple languages. Human programmers can abstract the high-level concept of programming, thus learning one language can help them master the others. On the contrary, the model seems to require a large capacity to concurrently store the knowledge of each language. How to help the model extract the most essential knowledge of programming remains a research challenge.

Second, similar to others, CodeGeeX shows the reasoning potential as a model though its lack of strong generality. We demonstrate that CodeGeeX can solve problems in different languages. However, the pass rate distribution varies a lot among languages, *i.e.*, it is not able to solve the same problem using different languages on occasion. We assume that this could probably be related to some language-specific features (*e.g.*, some problems are easier to solve in Python), or it could be simply due to the appearance of a similar language-specific implementation in training data. Either case, there is a long way to go for the model to have a reliable reasoning ability.

Third, the few-shot ability of CodeGeeX is worth exploration. Instead of using costly fine-tuning approaches, we may do priming using a few examples and make the model achieve comparable performance. Recent works like chain-of-thought (CoT) prompting (Wei et al., 2022) have shown impressive results using such an approach, inspiring us to examine CoT in code models.

Acknowledgement

This research was supported by Natural Science Foundation of China (NSFC) for Distinguished Young Scholars No. 61825602, NSFC No. 62276148 and a research fund from Zhipu.AI. We give our special thanks to Wenguang Chen from Tsinghua, the Peng Cheng Laboratory, and Zhipu.AI for sponsoring the training and inference GPU resources. We thank all our collaborators and partners from Tsinghua KEG, IIIS, Peng Cheng Laboratory, and Zhipu.AI, including Aohan Zeng, Wendi Zheng, Lilong Xue, Yifeng Liu, Yanru Chen, Yichen Xu, Qingyu Chen, Zhongqi Li, Gaojun Fan, Yifan Yao, Qihui Deng, Bin Zhou, Ruijie Cheng, Peinan Yu, Jingyao Zhang, Bowen Huang, Zhaoyu Wang, Jiecai Shan, Xuyang Ding, Xuan Xue, and Peng Zhang.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

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*.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, and et al. 2022. Palm: Scaling language modeling with pathways.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*.

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*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645*.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126.

Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, et al. 2021. Prophetnet-x: large-scale pre-training models for english, chinese, multi-lingual, dialog, and code generation. *arXiv preprint arXiv:2104.08006*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre training.

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):9.

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*, pages 1–16. IEEE.

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*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

Phillip D Summers. 1977. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175.

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*, volume 34, pages 8984–8991.

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*, pages 1433–1443.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2022. *Natural language processing with transformers*. " O'Reilly Media, Inc.".

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Richard J Waldinger and Richard CT Lee. 1969. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252.

Ben Wang and Aran Komatsuzaki. 2021. Gpt-j-6b: A 6 billion parameter autoregressive language model.

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. *arXiv preprint arXiv:2109.00859*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. 2021. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032*.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*.

Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. 2021. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29.

A Appendix

Contents

A.1 Statistics of Code Corpus

Table 8 summarizes the composition of CodeGeeX's code corpus.

Table 8: Composition of our code corpus for pre-training.

Language	# Tokens (B)	% Tokens (%)	Language Tag
C++	45.2283	28.4963	// language: C++
Python	42.3250	26.667	# language: Python
Java	25.3667	15.9824	// language: Java
JavaScript	11.3165	7.13	// language: JavaScript
C	10.6590	6.7157	// language: C
Go	7.4774	4.7112	// language: Go
HTML	4.9355	3.1096	<!–language: HTML–>
Shell	2.7498	1.7325	# language: Shell
PHP	2.1698	1.3671	// language: PHP
CSS	1.5674	0.9876	/* language: CSS */
TypeScript	1.1667	0.7351	// language: TypeScript
SQL	1.1533	0.7267	– language: SQL
TeX	0.8257	0.5202	% language: TeX
Rust	0.5228	0.3294	// language: Rust
Objective-C	0.4526	0.2851	// language: Objective-C
Scala	0.3786	0.2385	// language: Scala
Kotlin	0.1707	0.1075	// language: Kotlin
Pascal	0.0839	0.0529	// language: Pascal
Fortran	0.077	0.0485	!language: Fortran
R	0.0447	0.0281	# language: R
Cuda	0.0223	0.014	// language: Cuda
C#	0.0218	0.0138	// language: C#
Objective-C++	0.0014	0.0009	// language: Objective-C++

A.2 Tokenization of CodeGeeX

Given a code snippet as in Figure 12, it is first separated into token pieces by the tokenizer. Then, each token is mapped to an integer according to its ID in the pre-defined dictionary. For example, 4 or 8 whitespaces (one or two indents in Python) are concatenated to `<|extratoken_12|>` or `<|extratoken_16|>`, respectively. Note that in Figure Figure 12, tokens are starting with `"▁"`, which represents whitespace and is often used to indicate if the token appears in the middle of a sentence. After tokenization, any code snippet or text description can be transformed into a vector of integers.

A.2.1 Details of Budget Allocation Strategies

We compare three strategies: **Best Single**, choose a single language with the best performance; **Uniform**, allocate the budget uniformly; **Weighted**, allocate the budget to multiple languages based their proportions in the training corpus. Detailed weights can be found in Table 9. The allocation of CodeGen-Multi-16B and InCoder-6.7B are extracted from the training corpora description in the original papers. The allocation of GPT-J-6B/GPT-NeoX-20B are from the number of tokens in the GitHub section of the Pile.

Table 9: Detailed assignment of budget allocation strategies. Given budget $k = 100$, **Weighted** distribute the budgets according to the proportions of language in the training corpus of each model.

Strategy	Model	Python	C++	Java	JavaScript	Go
Uniform	All	20	20	20	20	20
Weighted	GPT-J-6B	17	36	11	22	14
	GPT-NeoX-20B	17	36	11	22	14
	InCoder-6.7B	45	12	5	34	4
	CodeGen-Multi-6B/16B	17	38	29	8	8
	CodeGeeX-13B (ours)	32	33	20	9	6

```go
// language: Go
// Return list of all prefixes from shortest to longest of the input string
// >>> AllPrefixes('abc')
// ['a', 'ab', 'abc']
func AllPrefixes(str string) []string{
    result := []string{}
    for i = 1; i <= len(str); i ++ {
        result = append(result, str[0: i])
    }
    return result
}
```

Transform a given code into token pieces

```
['//', '_language', ':', '_Go', '\n', '//', '_Return', '_list', '_of', '_all',
'_prefix', 'es', '_from', '_shortest', '_to', '_longest', '_of', '_the', '_inpu
t', '_string', '\n', '//', '_>>>', '_All', 'Pref', 'ix', 'es', "('", 'abc', "')"
, '\n', '//', "_['", 'a', "',", "_'", 'ab', "',", "_'", 'abc', "']", '\n', 'fun
c', '_All', 'Pref', 'ix', 'es', '(', 'str', '_string', ')', '_[]', 'string', '{'
, '\n', '<|extratoken_12|>', 'result', '_:=', '_[]', 'string', '{', '}', '\n',
'<|extratoken_12|>', 'for', '_i', '_=', '_1', ';', '_i', '_<=', '_len', '(', 'st
r', ');', '_i', '_++', '_{', '\n', '<|extratoken_16|>', 'result', '_=', '_appen
d', '(', 'result', ',', '_str', '[', '0', ':', '_i', '])', '\n', '<|extratoken_1
2|>', '}', '\n', '<|extratoken_12|>', 'return', '_result', '\n', '}']
```

Map token pieces to IDs in the dictionary

```
[1003, 3303, 25, 1514, 198, 1003, 8229, 1351, 286, 477, 21231, 274, 422, 35581, 28
4, 14069, 286, 262, 5128, 4731, 198, 1003, 13163, 1439, 36698, 844, 274, 10786, 39
305, 11537, 198, 1003, 37250, 64, 3256, 705, 397, 3256, 705, 39305, 20520, 198, 20
786, 1439, 36698, 844, 274, 7, 2536, 4731, 8, 17635, 8841, 90, 198, 50268, 20274,
19039, 17635, 8841, 90, 92, 198, 50268, 1640, 1312, 796, 352, 26, 1312, 19841, 188
96, 7, 2536, 1776, 1312, 19969, 1391, 198, 50272, 20274, 796, 24443, 7, 20274, 11,
965, 58, 15, 25, 1312, 12962, 198, 50268, 92, 198, 50268, 7783, 1255, 198, 92]
```

Figure 12: Illustration of tokenization in CodeGeeX. "_" represents a whitespace, and "<|extratoken_X|>" represents concatenated whitespaces of different lengths.

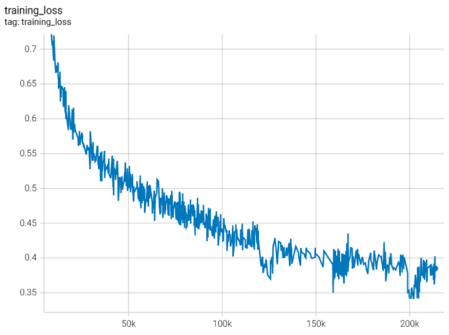


Figure 13: Training loss of CodeGeeX.

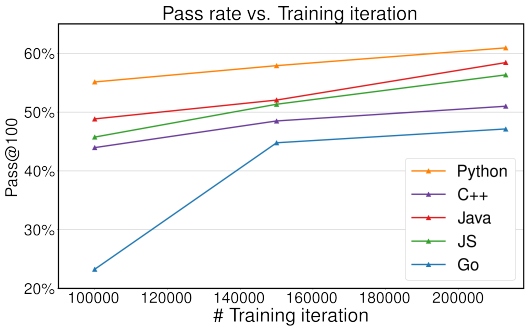


Figure 14: HumanEval-X pass rate vs. iteration.

A.3 Evaluation on HumanEval-X (Additional)

Pass rate vs. number of training iterations. We show in Figure 13 that the cross entropy loss decreases steadily during training while the pass rate in HumanEval-X continues to improve for different languages in Figure 14.

Pass rate distribution vs. Languages for other code generation models. We show in Figure 15 that other code generation models also have various pass rate distribution for different languages.

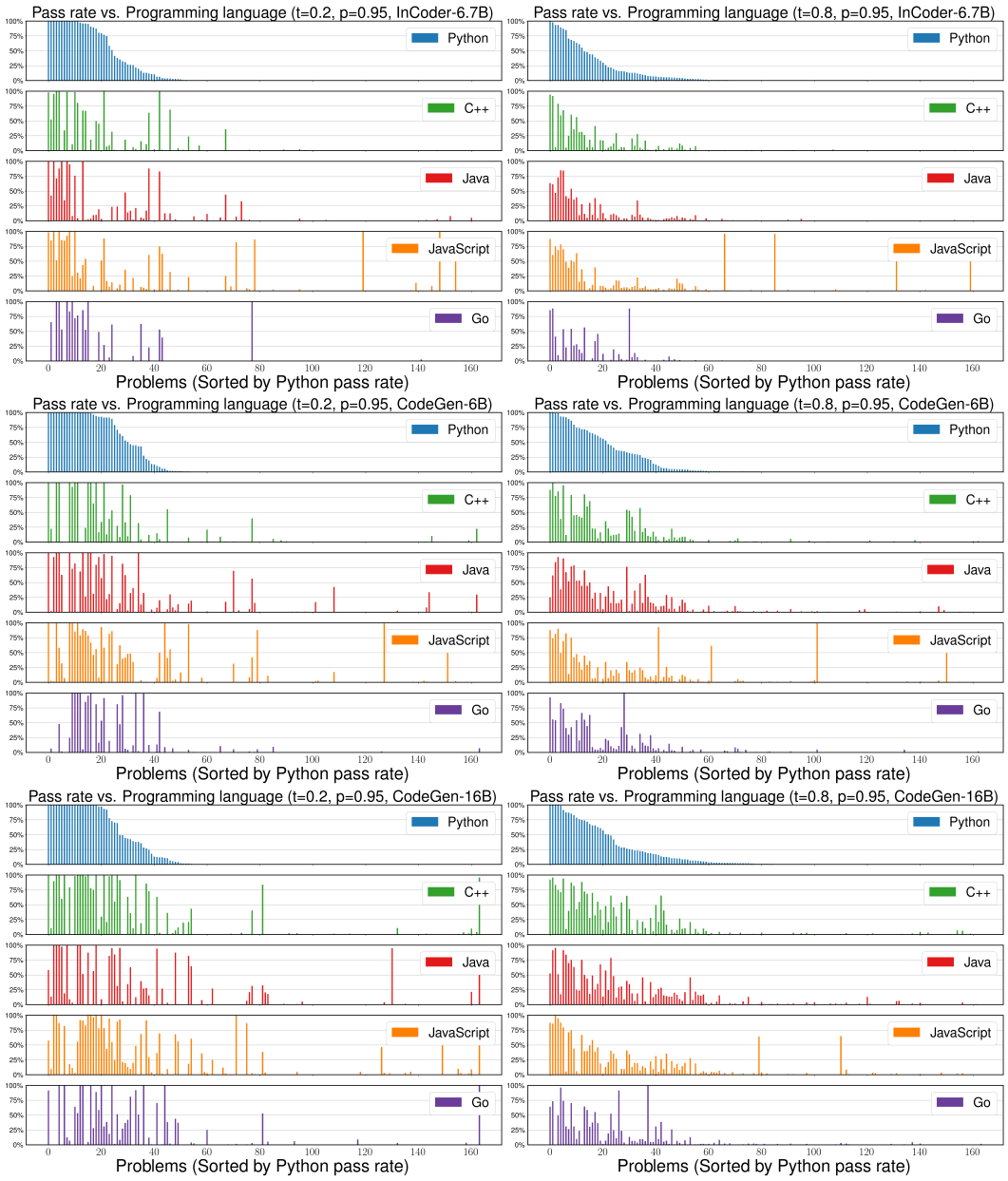


Figure 15: In HumanEval-X, each problem's pass rate varies when generating in different programming languages. **Left:** $t = 0.2, p = 0.95$; **Right:** $t = 0.8, p = 0.95$. **From top to bottom**: InCoder-6.7B, CodeGen-Multi-6B, CodeGen-Multi-16B.

A.4 Evaluation on Other Benchmarks

A.4.1 Evaluation on HumanEval

The evaluation setting on HumanEval is the same as HumanEval-X. We show that among multilingual code generation models, CodeGeeX achieves the second highest performance on HumanEval, reaching 60% in pass@100 (surpassed by PaLMCoder-540B). We also notice that monolingual models outperforms multilingual ones with a large margin, indicating that multilingual models require a larger model capacity to master different languages.

Table 10: The results of CodeGeeX on HumanEval benchmark. The metric is pass@k introduced in Chen et al. (2021) (* use the biased pass@k from Chowdhery et al. (2022)). Nucleus sampling is used with top-p=0.95 and sampling temperature being 0.2/0.6/0.8 for @1/@10/@100 respectively.

Model	Size	Type	Available	pass@1	pass@10	pass@100
CodeParrot (Tunstall et al., 2022)	1.5B	Multi	Yes	4.00%	8.70%	17.90%
PolyCoder (Xu et al., 2022)	2.7B	Multi	Yes	5.60%	9.80%	17.70%
GPT-J (Wang and Komatsuzaki, 2021)	6B	Multi	Yes	11.60%	15.70%	27.70%
CodeGen-Multi (Nijkamp et al., 2022)	6.1B	Multi	Yes	18.16%	27.81%	44.85%
InCoder (Fried et al., 2022)	6.7B	Multi	Yes	15.20%	27.80%	47.00%
GPT-NeoX (Black et al., 2022)	20B	Multi	Yes	15.40%	25.60%	41.20%
LaMDA (Thoppilan et al., 2022)	137B	Multi	No	14.00%*	-	47.30%*
CodeGen-Multi (Nijkamp et al., 2022)	16.1B	Multi	Yes	19.22%	34.64%	55.17%
PaLM-Coder (Chowdhery et al., 2022)	540B	Multi	No	**36.00%***	-	**88.40%***
Codex (Chen et al., 2021)	12B	Mono	No	28.81%	46.81%	72.31%
CodeGen-Mono (Nijkamp et al., 2022)	16.1B	Mono	Yes	29.28%	**49.86%**	75.00%
CodeGeeX (ours)	13B	Multi	Yes	22.89%	39.57%	60.92%

A.4.2 Evaluation on MBPP

MBPP dataset is proposed by Austin et al. (2021), containing 974 problems in Python. Due to specific input-output format, MBPP need to be evaluated under a few-shot setting. We follow the splitting in the original paper and use problems 11-510 for testing. Under 1-shot setting, we use problem 2 in prompts. Under 3-shot setting, we use problem 2,3,4 in prompts. The metric is pass@k, $k \in \{1, 10, 80\}$. For pass@1, the temperature is 0.2 and top-p is 0.95; for pass@10 and pass@ 80, the temperature is 0.8 and top-p is 0.95. For baselines, we consider LaMDA-137B, PaLM-540B, Code-davinci-002 (online API version of OpenAI Codex), PaLMCoder-540B and InCoder-6.7B.

The results indicate that the model capacity is essential for multilingual code generation model. With significantly more parameters, PaLM and Codex outperform CodeGeeX with a large margin. Meanwhile, we find that more shot in the prompts harm the performance of CodeGeeX, the same phenomenon have also been discovered in InCoder (Fried et al., 2022). We assume that it is because smaller models do not have enough reasoning ability to benefit from the few-shot setting.

A.4.3 Evaluation on CodeXGLUE

CodeXGLUE is a benchmark proposed by Lu et al. (2021), containing multiple datasets to support evaluation on multiple tasks, using similarity-based metrics like CodeBLEU, BLEU and accuracy

Table 11: The results of CodeGeeX on MBPP dataset (Austin et al., 2021).

Method	Model	Pass@1	Pass@10	Pass@80
3-shot	**LaMDA-137B** (Austin et al., 2021)	14.80	-	62.40
	PaLM-540B (Chowdhery et al., 2022)	36.80	-	75.00
	Code-davinci-002 (Chen et al., 2021)	**50.40**	-	**84.40**
	PaLMCoder-540B (Chowdhery et al., 2022)	47.00	-	80.80
	CodeGeeX-13B (ours)	22.44	43.24	63.52
1-shot	**InCoder-6.7B** (Fried et al., 2022)	19.40	-	-
	CodeGeeX-13B (ours)	24.37	47.95	68.50

for generation tasks. We test the performance of CodeGeeX on the **code summarization** task of CodeXGLUE. We first fine-tune the parameters of CodeGeeX on the given training set, mixing the training data in all languages to get one fine-tuned model. Then, we test the performance of the fine-tuned model on each language, using BLEU score for evaluation because the models generate natural language in summarization tasks.

For all languages, we set temperature to 0.2 and top-p to 0.95, and generate one summarization for each sample in the test set. We report the results in Table 12. CodeGeeX obtains an average BLEU score of 20.63, besting all baseline models. It is worth noting that CodeGeeX is not pre-trained on Ruby, and after removing the results on Ruby for all models, CodeGeeX outperforms the best baseline model (DistillCodeT5 from Wang et al. (2021)) by 1.88 in the average BLEU score.

Table 12: The results of CodeGeeX on **code summarization** in CodeXGLUE benchmark (Lu et al., 2021). Six languages are considered, Ruby, JavaScript, Go, Python, Java, PHP. The metric is the BLEU score. * We don't have Ruby in the pretraining corpus.

Model	All	Ruby	JavaScript	Go	Python	Java	PHP
CodeBERT (Feng et al., 2020)	17.83	12.16	14.90	18.07	19.06	17.65	25.16
PLBART (Ahmad et al., 2021)	18.32	14.11	15.56	18.91	19.30	18.45	23.58
ProphetNet-X (Qi et al., 2021)	18.54	14.37	**16.60**	18.43	17.87	19.39	24.57
CoTexT (Phan et al., 2021)	18.55	14.02	14.96	18.86	19.73	19.06	24.68
PolyglotCodeBERT (Feng et al., 2020)	19.06	14.75	15.80	18.77	18.71	20.11	26.23
DistillCodeT5 (Wang et al., 2021)	20.01	**15.75**	16.42	20.21	20.59	**20.51**	26.58
CodeGeeX (ours)	**20.63**	10.05*	16.01	**24.62**	**22.50**	19.60	**31.00**

A.4.4 Evaluation on XLCoST

XLCoST is a benchmark proposed by Zhu et al. (2022), containing parallel multilingual code data, with code snippets aligned among different languages. For generation tasks, XLCoST uses CodeBLEU, BLEU for evaluation. We choose the **code translation** task of XLCoST for CodeGeeX evaluation. We first fine-tune the parameters of CodeGeeX on the given training set, combining the training data in all 42 languages pairs to obtain one fine-tuned model. Then, we test the performance of the fine-tuned model on each language pair with CodeBLEU score.

For all language pairs, we set temperature to 0.2 and top-p to 0.95, and generate one translation for each sample in the test set. We report the results in Table 13. CodeGeeX performs better than all baseline models on all language pairs except for: PHP to Python on program level, C++ to Python on snippet level, and PHP to Python on snippet level. On average, CodeGeeX outperforms the baseline by 4.10 on program level and by 1.99 on snippet level.

Table 13: The results of CodeGeeX on **code translation** in XLCoST benchmark. Six languages are considered, C++, Java, Python, C#, JavaScript, PHP, C. The metric is CodeBLEU (Ren et al., 2020). The results of baselines are adopted from the original paper (Zhu et al., 2022).

		Snippet-level							**Program-level**						
	Model	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**
C++	CodeBERT	–	84.94	74.55	84.99	82.79	68.56	45.46	–	74.73	24.96	76.35	72.95	50.40	21.84
	PLBART	–	83.85	74.89	84.57	83.19	68.62	83.95	–	75.26	70.13	78.01	61.85	67.01	72.59
	CodeT5	–	86.35	**76.28**	85.85	84.31	69.87	90.45	–	80.03	71.56	81.73	79.48	70.44	85.67
	CodeGeeX	–	**86.99**	74.73	**86.63**	**84.83**	**70.30**	**94.04**	–	**84.40**	**73.89**	**84.49**	**82.20**	**71.18**	**87.32**
Java	CodeBERT	87.27	–	58.39	92.26	84.63	67.26	39.94	79.36	–	8.51	84.43	76.02	51.42	21.22
	PLBART	87.31	–	58.30	90.78	85.42	67.44	72.47	81.41	–	66.29	83.34	80.14	67.12	63.37
	CodeT5	88.26	–	74.59	92.56	86.22	69.02	82.78	84.26	–	69.57	87.79	80.67	69.44	78.78
	CodeGeeX	**89.08**	–	**74.65**	**92.94**	**86.96**	**69.77**	**88.44**	**87.07**	–	**73.11**	**91.78**	**84.34**	**70.61**	**81.07**
Py	CodeBERT	80.46	58.50	–	54.72	57.38	65.14	10.70	68.87	28.22	–	17.80	23.65	49.30	18.32
	PLBART	80.15	74.15	–	73.50	73.20	66.12	62.15	74.38	67.80	–	66.03	69.30	64.85	29.05
	CodeT5	81.56	78.61	–	78.89	77.76	67.54	68.67	78.85	73.15	–	73.35	71.80	67.50	56.35
	CodeGeeX	**82.91**	**81.93**	–	**81.30**	**79.83**	**67.99**	**82.59**	**82.49**	**79.03**	–	**80.01**	**77.47**	**68.91**	**71.67**
C#	CodeBERT	86.96	90.15	56.92	–	84.38	67.18	40.43	78.52	82.25	10.82	–	75.46	51.76	21.63
	PLBART	84.98	6.27	69.82	–	85.02	67.30	75.74	80.17	81.37	67.02	–	79.81	67.12	57.60
	CodeT5	88.06	91.69	73.85	–	85.95	68.97	81.09	83.59	85.70	69.52	–	80.50	69.63	77.35
	CodeGeeX	**88.70**	**93.03**	**74.55**	–	**86.44**	**69.49**	**86.69**	**87.11**	**90.46**	**72.89**	–	**83.83**	**70.58**	**80.73**
JS	CodeBERT	84.38	84.42	52.57	84.74	–	66.66	33.29	75.43	72.33	9.19	75.47	–	52.08	19.79
	PLBART	84.45	84.90	69.29	85.05	–	67.09	72.65	80.19	76.96	64.18	78.51	–	67.24	67.70
	CodeT5	85.06	85.48	73.15	85.96	–	68.42	80.49	82.14	79.91	68.42	81.77	–	68.76	74.57
	CodeGeeX	**86.72**	**86.96**	**73.25**	**86.41**	–	**69.00**	**83.85**	**85.84**	**83.85**	**72.11**	**85.35**	–	**69.80**	**79.41**
PHP	CodeBERT	82.58	81.57	69.29	80.96	79.94	–	28.45	50.13	46.81	16.92	49.75	48.12	–	22.19
	PLBART	83.87	81.66	71.17	78.00	82.94	–	57.39	79.40	72.77	61.26	74.16	44.26	–	56.23
	CodeT5	86.33	85.12	**73.22**	84.56	83.56	–	79.30	85.55	82.09	**72.26**	83.79	81.72	–	65.86
	CodeGeeX	**86.75**	**86.24**	71.37	**85.58**	**84.17**	–	**83.89**	**87.23**	**83.90**	71.02	**85.34**	**82.81**	–	**78.76**
C	CodeBERT	45.84	39.69	13.55	39.71	29.85	38.88	–	21.70	21.27	21.10	19.50	15.64	31.71	–
	PLBART	82.53	72.35	49.16	75.78	75.05	60.86	–	78.42	13.45	5.53	45.15	31.47	25.17	–
	CodeT5	90.26	81.81	63.81	83.05	79.73	66.32	–	88.17	76.12	56.32	80.20	76.50	64.28	–
	CodeGeeX	**91.30**	**85.58**	**71.52**	**87.52**	**84.91**	**68.52**	–	**88.21**	**82.46**	**69.78**	**85.56**	**81.21**	**68.80**	–

A.5 Examples of CodeGeeX Generation

```cpp
// language: C++
/*
Check if in given vector of numbers, are any two numbers closer to each other than
given threshold.
>>> has_close_elements({1.0, 2.0, 3.0}, 0.5)
false
>>> has_close_elements({1.0, 2.8, 3.0, 4.0, 5.0, 2.0}, 0.3)
true
*/
#include<stdio.h>
#include<vector>
#include<math.h>
using namespace std;
bool has_close_elements(vector<float> numbers, float threshold){
/////////////// Generation ///////////////
    for(int i=0; i<numbers.size(); i++){
        for(int j=i+1; j<numbers.size(); j++){
            if(abs(numbers[i]-numbers[j])<=threshold){
                return true;
            }
        }
    }
    return false;
}
```

```java
// language: Java
import java.util.*;
import java.lang.*;

class Solution {
    /**
    Check if in given list of numbers, are any two numbers closer to each other than giv
en threshold.
    >>> has_close_elements(Arrays.asList(1.0, 2.0, 3.0), 0.5)
    false
    >>> has_close_elements(Arrays.asList(1.0, 2.8, 3.0, 4.0, 5.0, 2.0), 0.3)
    true
     */
    public boolean has_close_elements(List<Double> numbers, double threshold) {
/////////////// Generation ///////////////
        Map<Double, Integer> numberToIndex = new HashMap<>();
        for (int i = 0; i < numbers.size(); ++i) {
            numberToIndex.put(numbers.get(i), i);
        }
        for (int i = 0; i < numbers.size(); ++i) {
            for (int j = i + 1; j < numbers.size(); ++j) {
                if (Math.abs(numbers.get(i) - numbers.get(j)) <= threshold) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

```javascript
// language: JavaScript
/* Check if in given list of numbers, are any two numbers closer to each other than
  given threshold.
  >>> hasCloseElements([1.0, 2.0, 3.0], 0.5)
  false
  >>> hasCloseElements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
  true
  */
const hasCloseElements = (numbers, threshold) => {
/////////////// Generation ///////////////
    let found = false;
    for (let n = 0; n < numbers.length - 1; n++) {
    for (let m = n + 1; m < numbers.length; m++) {
        if (Math.abs(numbers[n] - numbers[m]) < threshold) {
        found = true;
        break;
        }
    }
    if (found) {
        break;
    }
    }
    return found;
}
```

```go
// language: Go
import (
    "math"
)

// Check if in given list of numbers, are any two numbers closer to each other than give
n threshold.
// >>> HasCloseElements([]float64{1.0, 2.0, 3.0}, 0.5)
// false
// >>> HasCloseElements([]float64{1.0, 2.8, 3.0, 4.0, 5.0, 2.0}, 0.3)
// true
func HasCloseElements(numbers []float64, threshold float64) bool {
/////////////// Generation ///////////////
    for i := range numbers {
        for j := i + 1; j < len(numbers); j++ {
            if math.Abs(numbers[i]-numbers[j]) < threshold {
                return true
            }
        }
    }
    return false
}
```

Figure 16: Solutions (Problem 0 in HumanEval-X) generated by CodeGeeX. Prompt and generated codes are separated by the 'Generation' line (added after the generation as an indicator).

```java
code translation
Java:
import java.util.*;
import java.lang.*;

class Solution {
    public boolean checkDictCase(Map<Object, Object> dict) {
        if (dict.isEmpty()) {
            return false;
        }
        String state = "start";
        for (Map.Entry entry : dict.entrySet()) {
            if (!(entry.getKey() instanceof String key)) {
                state = "mixed";
                break;
            }
            boolean is_upper = true, is_lower = true;
            for (char c : key.toCharArray()) {
                if (Character.isLowerCase(c)) {
                    is_upper = false;
                } else if (Character.isUpperCase(c)) {
                    is_lower = false;
                } else {
                    is_upper = false;
                    is_lower = false;
                }
            }
            if (state.equals("start")) {
                if (is_upper) {
                    state = "upper";
                } else if (is_lower) {
                    state = "lower";
                } else {
                    break;
                }
            } else if ((state.equals("upper") && !is_upper) || (state.equals("lower") && !is_lower)) {
                state = "mixed";
                break;
            }
        }
        return state.equals("upper") || state.equals("lower");
    }
}
JavaScript:

const checkDictCase = (dict) => {
/////////////// Translation ///////////////
    if (dict.isEmpty) {
        return false;
    }
    let state ='start';
    for (const key of Object.keys(dict)) {
        if (typeof key!='string') {
            state ='mixed';
            break;
        }
        if (state =='start') {
            if (key.toUpperCase() == key) {
                state = 'upper';
            } else if (key.toLowerCase() == key) {
                state = 'lower';
            }
        } else if (state == 'upper' && key.toUpperCase()!= key) {
            state ='mixed';
            break;
        } else if (state == 'lower' && key.toLowerCase()!= key) {
            state ='mixed';
            break;
        }
    }
    return state == 'upper' || state == 'lower';
}
```

```go
code translation
Go:

func CheckDictCase(dict map[interface{}]interface{}) bool {
    if len(dict) == 0 {
        return false
    }
    state := "start"
    key := ""
    ok := false
    for k := range dict {
        if key, ok = k.(string); !ok {
            state = "mixed"
            break
        }
        if state == "start" {
            if key == strings.ToUpper(key) {
                state = "upper"
            } else if key == strings.ToLower(key) {
                state = "lower"
            } else {
                break
            }
        } else if (state == "upper" && key != strings.ToUpper(key)) || (state == "lower" && key != strings.ToLower(key)) {
            state = "mixed"
            break
        } else {
            break
        }
    }
    return state == "upper" || state == "lower"
}
Java:
import java.util.*;
import java.lang.*;

class Solution {
    public boolean check_dict_case(Map<Object, Object> dict) {
/////////////// Translation ///////////////
        if (dict == null) return false;
        String state = "start";
        for (Object k : dict.keySet()) {
            Object v = dict.get(k);
            if (state == "start") {
                if (k.equals(String.valueOf(k).toUpperCase())) {
                    state = "upper";
                } else if (k.equals(String.valueOf(k).toLowerCase())) {
                    state = "lower";
                } else {
                    return false;
                }
            } else if (state == "upper") {
                if (!(k.equals(String.valueOf(k).toUpperCase()))) {
                    return false;
                }
            } else if (state == "lower") {
                if (!(k.equals(String.valueOf(k).toLowerCase()))) {
                    return false;
                }
            } else {
                return false;
            }
        }
        return state == "upper" || state == "lower";
    }
}
```

Figure 17: Solutions (Problem 95 in HumanEval-X) translated by CodeGeeX. Prompt and generated codes are separated by the 'Translation' line (added after the generation as an indicator).

```go
// language: Go
import (
    "math"
    "sort"
)

// We have an array 'arr' of N integers arr[1], arr[2], ..., arr[N].The
// numbers in the array will be randomly ordered. Your task is to determine if
// it is possible to get an array sorted in non-decreasing order by performing
// the following operation on the given array:
// You are allowed to perform right shift operation any number of times.
//
// One right shift operation means shifting all elements of the array by one
// position in the right direction. The last element of the array will be moved to
// the starting position in the array i.e. 0th index.
//
// If it is possible to obtain the sorted array by performing the above operation
// then return true else return false.
// If the given array is empty then return true.
//
// Note: The given list is guaranteed to have unique elements.
//
// For Example:
//
// MoveOneBall([3, 4, 5, 1, 2])==>true
// Explanation: By performin 2 right shift operations, non-decreasing order can
// be achieved for the given array.
// MoveOneBall([3, 5, 4, 1, 2])==>false
// Explanation:It is not possible to get non-decreasing order for the given
// array by performing any number of right shift operations.
func MoveOneBall(arr []int) bool {
/////////////// Generation ///////////////
    var min, max int
    for i := 0; i < len(arr)-1; i++ {
        if arr[i] > arr[i+1] {
            min = arr[i+1]
            max = arr[i]
            break
        }
    }
    if len(arr) == 0 {
        return true
    }
    var i int
    for i = 0; i < len(arr)-1; i++ {
        if arr[i] > max && arr[i+1] > min {
            arr[i] = max
            arr[i+1] = min
            break
        }
    }
    sort.Ints(arr)
    return arr[0] == min
}
```

```cpp
// language: C++
/*
We have a vector "arr" of N integers arr[1], arr[2], ..., arr[N].The
numbers in the vector will be randomly ordered. Your task is to determine if
it is possible to get a vector sorted in non-decreasing order by performing
the following operation on the given vector:
    You are allowed to perform right shift operation any number of times.

One right shift operation means shifting all elements of the vector by one
position in the right direction. The last element of the vector will be moved to
the starting position in the vector i.e. 0th index.

If it is possible to obtain the sorted vector by performing the above operation
then return true else return false.
If the given vector is empty then return true.

Note: The given vector is guaranteed to have unique elements.

For Example:

move_one_ball({3, 4, 5, 1, 2})==>true
Explanation: By performing 2 right shift operations, non-decreasing order can
            be achieved for the given vector.
move_one_ball({3, 5, 4, 1, 2})==>false
Explanation:It is not possible to get non-decreasing order for the given
            vector by performing any number of right shift operations.

*/
#include<stdio.h>
#include<vector>
using namespace std;
bool move_one_ball(vector<int> arr){
/////////////// Generation ///////////////
    int index = arr.size()-1;
    int k = arr.size()-1;
    while(k>0){
        k--;
        if(arr[k]<=arr[k+1]){
            continue;
        }
        else{
            arr[index]=arr[k];
            arr[k]=arr[k+1];
            arr[k+1]=arr[k+2];
            index--;
            //cout<<"arr[index]:"<<arr[index]<<endl;
            //cout<<"arr[k+1]:"<<arr[k+1]<<endl;
        }
    }
    for(int i=1;i<arr.size();i++){
        if(arr[i]<arr[i-1]){
            return false;
        }
    }
    return true;
}
```

```javascript
// language: JavaScript
/*We have an array 'arr' of N integers arr[1], arr[2], ..., arr[N].The
  numbers in the array will be randomly ordered. Your task is to determine if
  it is possible to get an array sorted in non-decreasing order by performing
  the following operation on the given array:
      You are allowed to perform right shift operation any number of times.

  One right shift operation means shifting all elements of the array by one
  position in the right direction. The last element of the array will be moved to
  the starting position in the array i.e. 0th index.

  If it is possible to obtain the sorted array by performing the above operation
  then return true else return false.
  If the given array is empty then return true.

  Note: The given list is guaranteed to have unique elements.

  For Example:

  moveOneBall([3, 4, 5, 1, 2])==>true
  Explanation: By performin 2 right shift operations, non-decreasing order can
               be achieved for the given array.
  moveOneBall([3, 5, 4, 1, 2])==>false
  Explanation:It is not possible to get non-decreasing order for the given
              array by performing any number of right shift operations.

  */
const moveOneBall = (arr) => {
/////////////// Generation ///////////////
  let cnt = 0;
  for (let i = 0; i < arr.length - 1; i++) {
    if (arr[i + 1] < arr[i]) {
      cnt++;
    }
  }
  if (cnt <= 1) {
    return true;
  }
  return false;
}
```

```python
# language: Python

def move_one_ball(arr):
    """We have an array 'arr' of N integers arr[1], arr[2], ..., arr[N].The
    numbers in the array will be randomly ordered. Your task is to determine if
    it is possible to get an array sorted in non-decreasing order by performing
    the following operation on the given array:
        You are allowed to perform right shift operation any number of times.

    One right shift operation means shifting all elements of the array by one
    position in the right direction. The last element of the array will be moved to
    the starting position in the array i.e. 0th index.

    If it is possible to obtain the sorted array by performing the above operation
    then return True else return False.
    If the given array is empty then return True.

    Note: The given list is guaranteed to have unique elements.

    For Example:

    move_one_ball([3, 4, 5, 1, 2])==>True
    Explanation: By performin 2 right shift operations, non-decreasing order can
                 be achieved for the given array.
    move_one_ball([3, 5, 4, 1, 2])==>False
    Explanation:It is not possible to get non-decreasing order for the given
                array by performing any number of right shift operations.

    """
/////////////// Generation ///////////////
    if len(arr) == 0:
        return True
    if len(arr) == 1:
        return True
    count = 0
    for i in range(1, len(arr)):
        if arr[i - 1] > arr[i]:
            count += 1
    if count > 1:
        return False
    else:
        return True
```

Figure 18: Solutions (Problem 109 in HumanEval-X) generated by CodeGeeX. Prompt and generated codes are separated by the 'Generation' line (added after the generation as an indicator).

```python
# language: Python

def greatest_common_divisor(a: int, b: int) -> int:
    """ Return a greatest common divisor of two integers a and b
    >>> greatest_common_divisor(3, 5)
    1
    >>> greatest_common_divisor(25, 15)
    5
    """
################ Generation ###############
    while b:
        a, b = b, a % b
    return a
```

```cpp
// language: C++
/*
Return a greatest common divisor of two integers a and b
>>> greatest_common_divisor(3, 5)
1
>>> greatest_common_divisor(25, 15)
5
*/
#include<stdio.h>
using namespace std;
int greatest_common_divisor(int a, int b){
################ Generation ###############
    int c;
    if(a%b==0) return b;
    else return greatest_common_divisor(b,a%b);
}
```

```javascript
// language: JavaScript
/* Return a greatest common divisor of two integers a and b
  >>> greatestCommonDivisor(3, 5)
  1
  >>> greatestCommonDivisor(25, 15)
  5
  */
const greatestCommonDivisor = (a, b) => {
################ Generation ###############
  if (b === 0) return a;
  return greatestCommonDivisor(b, a % b);
}
```

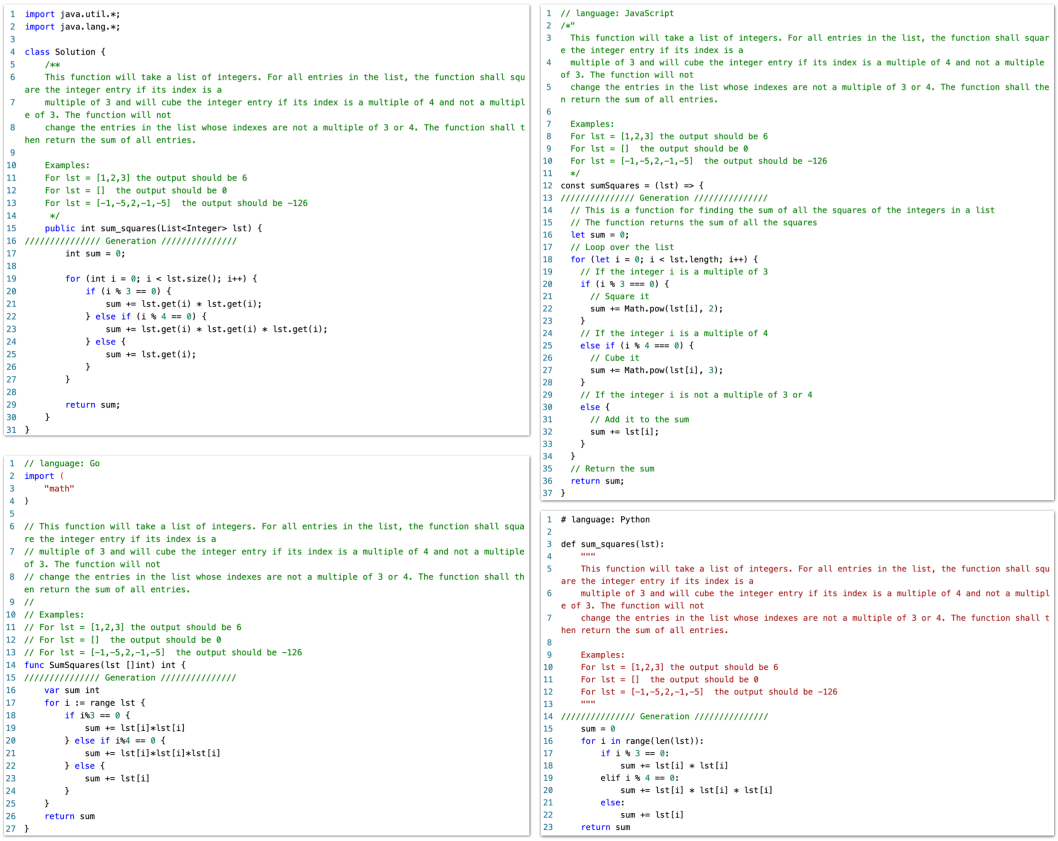
```java
// language: Java
import java.util.*;
import java.lang.*;

class Solution {
    /**
    Return a greatest common divisor of two integers a and b
    >>> greatest_common_divisor(3, 5)
    1
    >>> greatest_common_divisor(25, 15)
    5
     */
    public int greatest_common_divisor(int a, int b) {
################ Generation ###############
        if (a < b) {
            int temp = a;
            a = b;
            b = temp;
        }
        while (b!= 0) {
            int temp = a;
            a = b;
            b = temp % b;
        }
        return a;
    }
}
```

```go
// language: Go

// Return a greatest common divisor of two integers a and b
// >>> GreatestCommonDivisor(3, 5)
// 1
// >>> GreatestCommonDivisor(25, 15)
// 5
func GreatestCommonDivisor(a int,b int) int{
################ Generation ###############
    if b == 0{
        return a
    }
    return GreatestCommonDivisor(b, a%b)
}
```

Figure 19: Solutions (Problem 13 in HumanEval-X) generated by CodeGeeX. Prompt and generated codes are separated by the 'Generation' line (added after the generation as an indicator).



```java
import java.util.*;
import java.lang.*;

class Solution {
    /**
    This function will take a list of integers. For all entries in the list, the function shall square the integer entry if its index is a
    multiple of 3 and will cube the integer entry if its index is a multiple of 4 and not a multiple of 3. The function will not
    change the entries in the list whose indexes are not a multiple of 3 or 4. The function shall then return the sum of all entries.

    Examples:
    For lst = [1,2,3] the output should be 6
    For lst = []  the output should be 0
    For lst = [-1,-5,2,-1,-5]  the output should be -126
     */
    public int sum_squares(List<Integer> lst) {
################ Generation ###############
        int sum = 0;

        for (int i = 0; i < lst.size(); i++) {
            if (i % 3 == 0) {
                sum += lst.get(i) * lst.get(i);
            } else if (i % 4 == 0) {
                sum += lst.get(i) * lst.get(i) * lst.get(i);
            } else {
                sum += lst.get(i);
            }
        }

        return sum;
    }
}
```

```go
// language: Go
import (
    "math"
)

// This function will take a list of integers. For all entries in the list, the function shall square the integer entry if its index is a
// multiple of 3 and will cube the integer entry if its index is a multiple of 4 and not a multiple of 3. The function will not
// change the entries in the list whose indexes are not a multiple of 3 or 4. The function shall then return the sum of all entries.
//
// Examples:
// For lst = [1,2,3] the output should be 6
// For lst = []  the output should be 0
// For lst = [-1,-5,2,-1,-5]  the output should be -126
func SumSquares(lst []int) int {
################ Generation ###############
    var sum int
    for i := range lst {
        if i%3 == 0 {
            sum += lst[i]*lst[i]
        } else if i%4 == 0 {
            sum += lst[i]*lst[i]*lst[i]
        } else {
            sum += lst[i]
        }
    }
    return sum
}
```

```javascript
// language: JavaScript
/*"
  This function will take a list of integers. For all entries in the list, the function shall square the integer entry if its index is a
  multiple of 3 and will cube the integer entry if its index is a multiple of 4 and not a multiple of 3. The function will not
  change the entries in the list whose indexes are not a multiple of 3 or 4. The function shall then return the sum of all entries.

  Examples:
  For lst = [1,2,3] the output should be 6
  For lst = []  the output should be 0
  For lst = [-1,-5,2,-1,-5]  the output should be -126
  */
const sumSquares = (lst) => {
################ Generation ###############
  // This is a function for finding the sum of all the squares of the integers in a list
  // The function returns the sum of all the squares
  let sum = 0;
  // Loop over the list
  for (let i = 0; i < lst.length; i++) {
    // If the integer i is a multiple of 3
    if (i % 3 === 0) {
      // Square it
      sum += Math.pow(lst[i], 2);
    }
    // If the integer i is a multiple of 4
    else if (i % 4 === 0) {
      // Cube it
      sum += Math.pow(lst[i], 3);
    }
    // If the integer i is not a multiple of 3 or 4
    else {
      // Add it to the sum
      sum += lst[i];
    }
  }
  // Return the sum
  return sum;
}
```

```python
# language: Python

def sum_squares(lst):
    """
    This function will take a list of integers. For all entries in the list, the function shall square the integer entry if its index is a
    multiple of 3 and will cube the integer entry if its index is a multiple of 4 and not a multiple of 3. The function will not
    change the entries in the list whose indexes are not a multiple of 3 or 4. The function shall then return the sum of all entries.

    Examples:
    For lst = [1,2,3] the output should be 6
    For lst = []  the output should be 0
    For lst = [-1,-5,2,-1,-5]  the output should be -126
    """
################ Generation ###############
    sum = 0
    for i in range(len(lst)):
        if i % 3 == 0:
            sum += lst[i] * lst[i]
        elif i % 4 == 0:
            sum += lst[i] * lst[i] * lst[i]
        else:
            sum += lst[i]
    return sum
```

Figure 20: Solutions (Problem 142 in HumanEval-X) generated by CodeGeeX. Prompt and generated codes are separated by the 'Generation' line (added after the generation as an indicator).

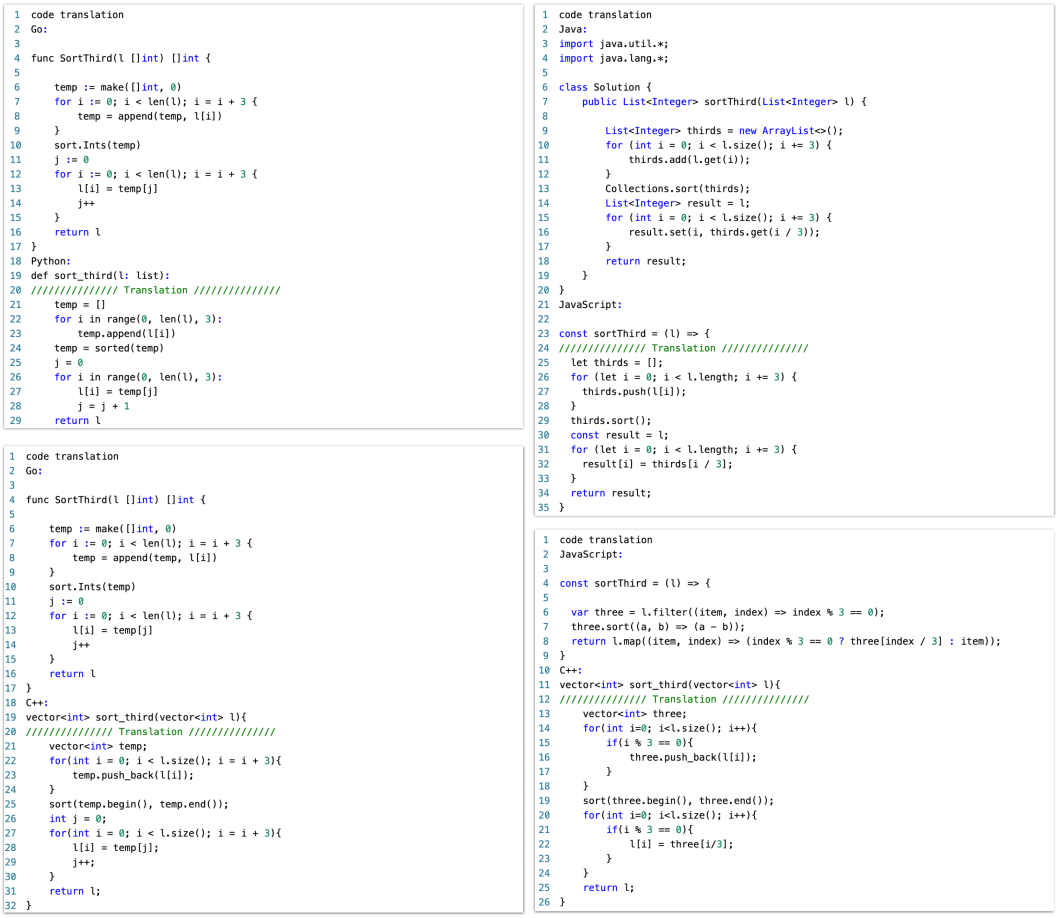


Figure 21: Solutions (Problem 33 in HumanEval-X) translated by CodeGeeX. Prompt and generated codes are separated by the 'Translation' line (added after the generation as an indicator).

Figure 22: Examples of CodeGeeX generation with prompts in Chinese, French, Russia and Japanese. Prompt and generated codes are separated by multiple '#'s (added after the generation as an indicator).