Lyra: A Benchmark for Turducken-Style Code Generation

Qingyuan Liang1,2, Zeyu Sun1, Qihao Zhu1, Wenjie Zhang1, Lian Yu2, Yingfei Xiong1, Lu Zhang1∗

1Key Laboratory of High Confidence Software Technologies (Peking University), MoE; Software Institute, Peking University, 100871, P. R. China
2School of Software & Microelectronics, Peking University, 102600, P. R. China
{lียงqy, szy_, zhuqh, zhang_wen_jie, lianyu, xiongyf, zhanglucs}@pku.edu.cn

Abstract

Recently, neural techniques have been used to generate source code automatically. While promising for declarative languages, these approaches achieve much poorer performance on datasets for imperative languages. Since a declarative language is typically embedded in an imperative language (i.e., the turducken-style programming) in real-world software development, the promising results on declarative languages can hardly lead to significant reduction of manual software development efforts.

In this paper, we define a new code generation task: given a natural language comment, this task aims to generate a program in a base imperative language with an embedded declarative language. To our knowledge, this is the first turducken-style code generation task. For this task, we present Lyra: a dataset in Python with embedded SQL. This dataset contains 2,000 carefully annotated database manipulation programs from real-world projects. Each program is paired with both a Chinese comment and an English comment. In our experiment, we adopted Transformer, BERT-style, and GPT-style models as baselines. In the best setting, the generation performance of GPT-style models is better than others, where the AST exact matching accuracy is 24% and 25.5% when using Chinese and English comments, respectively. Therefore, we believe that Lyra provides a new challenge for code generation. Yet, overcoming this challenge may significantly boost the applicability of code generation techniques for real-world software development.

1 Introduction

With the increase of software complexity, the process of programming is becoming time-consuming and error-prone [Sommerville, 1992]. Code generation is an important artificial intelligence problem that has the potential to release human beings from challenging software development [Dahl et al., 1994; Ling et al., 2016]. It requires both understanding the meaning of development requirements and mapping them to executable source code.

With significant advances in deep learning, many neural-based code generation methods have been proposed. Existing approaches perform well on declarative languages including database query languages [Yu et al., 2018; Zhong et al., 2017] and regular expressions [Li et al., 2021]. For example, Xuan et al. 2021 generate SQL programs with 93% execution accuracy among the test set of WikiSQL [Zhong et al., 2017], while Sun et al. 2020 achieves generate lambda calculus programs with 89% exact match among the test set of ATIS [Price, 1990; Dahl et al., 1994]. These high-accuracy models facilitate end-users to perform various operations, such as querying records in a database, without understanding the syntax of the declarative language. Therefore, it is tempting to directly use such well performed approaches to improve the efficiency of the real-world software development.

However, almost no large-scale software is written with only one declarative language. A declarative language is usually embedded in another imperative language, such as Java and Python. For example, as shown in Figure 1, a SQL statement is typically embedded as a string in Python with parameters specifying the condition being queried. Existing code generation approaches generate imperative programs with much poorer performance than generate declarative programs, making them difficult to apply to real-world software development.

However, almost no large-scale software is written with only one declarative language. A declarative language is usually embedded in another imperative language, such as Java and Python. For example, as shown in Figure 1, a SQL statement is typically embedded as a string in Python with parameters specifying the condition being queried. Existing code generation approaches generate imperative programs with much poorer performance than generate declarative programs, making them difficult to apply to real-world software development.

However, almost no large-scale software is written with only one declarative language. A declarative language is usually embedded in another imperative language, such as Java and Python. For example, as shown in Figure 1, a SQL statement is typically embedded as a string in Python with parameters specifying the condition being queried. Existing code generation approaches generate imperative programs with much poorer performance than generate declarative programs, making them difficult to apply to real-world software development.

However, almost no large-scale software is written with only one declarative language. A declarative language is usually embedded in another imperative language, such as Java and Python. For example, as shown in Figure 1, a SQL statement is typically embedded as a string in Python with parameters specifying the condition being queried. Existing code generation approaches generate imperative programs with much poorer performance than generate declarative programs, making them difficult to apply to real-world software development.

However, almost no large-scale software is written with only one declarative language. A declarative language is usually embedded in another imperative language, such as Java and Python. For example, as shown in Figure 1, a SQL statement is typically embedded as a string in Python with parameters specifying the condition being queried. Existing code generation approaches generate imperative programs with much poorer performance than generate declarative programs, making them difficult to apply to real-world software development.

However, almost no large-scale software is written with only one declarative language. A declarative language is usually embedded in another imperative language, such as Java and Python. For example, as shown in Figure 1, a SQL statement is typically embedded as a string in Python with parameters specifying the condition being queried. Existing code generation approaches generate imperative programs with much poorer performance than generate declarative programs, making them difficult to apply to real-world software development.
In this paper, we present a new code generation dataset, which is specific to the turducken-style code generation task. To ensure that our dataset has satisfactory characteristics, our data have been processed as follows. First, to meet the realism and diversity requirements, the source code snippets are crawled from real development projects on Github. Second, our data have gone through a fine annotation process of 400 human hours. Since most of the original data contain project-related information, they cannot be directly used as independent functions. To address this challenge, we manually modified the crawled data. In the above case, we can put the connection information "conn" as a parameter in the parameter list of the function block. We describe these details in our annotation process. Additionally, to address the quality problem, we designed a quick and efficient checker to check each data. Specifically, we use some rule-based methods to check and modify the annotated data according to the characteristics of the dataset. Finally, we constructed Lyra, which contains 2,000 carefully modified source code snippets, and each code snippet corresponds to one Chinese and one English comment.

Figure 1 shows an example in the Lyra dataset. In this figure, the subfigure (a) is a SQL statement, subfigure (b) is the corresponding source code for executing SQL statement. Each source code in the dataset can be divided into three parts: preparation before SQL execution, executing SQL statements, and processing query results. For the example in subfigure (b), line 4 shows the process of executing SQL statements based on the function parameters; lines 1-3 import the package required by the function, define the function, and prepare for SQL execution; lines 5-7 process the results of the query. Note that our data are not a simple combination of SQL statements and Python code, but there is a close interaction between them. For example, the parameter "password" is used as the judgment condition of line 3 in the base language and also as the parameter to match the SQL string in the embedded language.

In addition, we used currently popular neural network architectures, Transformer [Vaswani et al., 2017] and pre-trained models (BERT-style and GPT-style) [Feng et al., 2020; Radford et al., 2019; Lu et al., 2021] both, to conduct experiments on our dataset. The best performance of the model can reach 24% and 25.5% AST (Abstract Syntax Tree) exact matching accuracy using Chinese and English comments, respectively. Experimental results show that our dataset may improve the practicability of code generation tasks in some specific applications. They also suggest that our dataset is complex and there is a large room for further development and utilization.

2 Related Work

Most closely related to our task is the previous code generation task for generating programs in single programming languages ([Ling et al., 2016; Oda et al., 2015; Tao et al., 2020; Chen et al., 2021], etc) and the text-to-SQL task for generating SQL statements ([Zhong et al., 2017; Yu et al., 2018; Liang et al., 2022], etc). They are all the process through the understanding of natural languages to generate the cor-
responding structured information. Below we discuss the research status of datasets related to these two task.

For the code generation task, several datasets with different programming languages have been created. These datasets include ATIS [Price, 1990; Dahl et al., 1994], Hearthstone [Ling et al., 2016], CONCODE [Iyer et al., 2018], and CoNaLa [Yin et al., 2018]. Although deep learning-based code generation is potentially promising and existing evaluations suggest that such approaches may be more accurate [Sun et al., 2020], they are most evaluated on datasets where requirements are different from real-world requirements in the industry [Liu et al., 2020]. On one hand, the current datasets can hardly be both of high-quality and used in the actual development process. Because the code blocks extracted from real projects often have a lot of project-related dependency information, they cannot be used directly. On the other hand, the generated programs are often significantly different from their references. They often contain syntax or semantic errors, and very little code can pass the test to meet the needs of actual use. Besides, most of the previous code generation datasets only considered one programming language, not the combination of different programming languages.

As to the text-to-SQL task, many datasets using SQL as programs have been created, such as Yelp and IMDB [Yaghmazadeh et al., 2017], WikiSQL [Zhong et al., 2017] and Spider [Yu et al., 2018]. These datasets have been studied by researchers in both communities of NLP [Xuan et al., 2021] and Database [Yaghmazadeh et al., 2017]. Using natural language description to generate SQL statements is easier than generating source code needed in development. However, the current scenarios of text-to-SQL task are bound with end users, and in actual development, SQL statements are often embedded in specific programming languages, such as Python, to perform database manipulation. For example, the SQL statement “select id in user where name = Bob” can only query Bob’s id value. If you replace "Bob" with a variable in the base language, you can query any person specified by the variable.

That is to say, in real-world development, SQL statements are typically embedded in base languages to increase the expressiveness. In the field of code generation, this turducken-style task needs to be considered to improve the usability of generated code. We construct a new dataset, Lyra, about using Python to operate databases to support our task.

3 Dataset Construction

All the code in our dataset is crawled from Github, where each example is an independent function block. The source code reflects how to use Python and SQL to manipulate the database in real development. Ten computer science students as annotators participated in the code modification and Chinese comments annotation process, and two English professionals were responsible for annotating and checking the English comments. As illustrated in Figure 2, we developed our dataset with 2,000 examples, spending around 430 hours of human labor in total. Note that we first explained our annotation task and compensation plan to each candidate annotator. This ensures that candidates who intended to participate in our task knew the specific work and were satisfied with the corresponding compensation. Finally, all participants in the annotation process were well informed and voluntary.

3.1 Dataset Collection

Collecting code snippets with correct functionality, clear logic, and applicable to actual development is hard. The code crawled from Github often has project-related operations or global variables. That results in source code not being an independent executable function block. Besides, the logic of some source code is often not clear enough, and even has bugs. These all bring challenges to construct the dataset.

To solve these challenges, we used the process shown in Figure 2 to our dataset. First, we obtained 8,618 user repositories from Github. Most of them are related to using the SQLAlchemy framework in Python to access the database. Second, we extracted 120,540 source code snippets written in the Python programming language from these repositories, and then obtained 18,047 unique function blocks related to SQL operations. Third, a programmer chose from these nonduplicated function blocks those that seem to have clear code logic and are easy to understand as candidates for annotation. Finally, we used a strict annotation process to modify the source code and annotate the corresponding comments. The final dataset contains 2,000 pieces of source code with related Chinese and English comments.

It is the most time-consuming (400 man-hours) in the code and comment annotation stage. At this stage, we first made careful modifications of the original code so that the final code logic is clear and the functionality is correct. In addition, we used Chinese and English to annotate the code. At the end of this stage, we also checked the quality of the dataset, and conducted a sampled evaluation. The specific process at this stage is shown in Figure 3.

3.2 Code Modification and Comments Annotation

To ensure the correctness and conciseness of the code and the completeness of the comments, we have designed many rules for the annotators to follow. This stage can be divided into two parts: code modification and comments annotation. In
this stage, we randomly distributed 100 pieces of data to each annotator each time. After annotation, the data need to pass the quality checking. If the code passes the quality check, we continued to provide the same amount of new data for the annotator. Everyone was treated fairly in this stage without prejudice. Everyone was compensated according to their workload. Those who gave annotation with more quantity and better quality got more compensation. This setting makes the annotator not ignore the quality of annotation because of too much data. Therefore, while ensuring the data quality, all annotators bore a roughly balanced task pressure.

**Code Modification**

Most of the function blocks extracted from Github are incomplete or incorrect. The most common problem is to refer to variables and class methods outside the function block, that is, project-related information. A function block containing project-related information often makes the function level code in the dataset difficult to understand. Therefore, we must make certain modifications before the code can be used as an independent function block and can pass the static program analysis. Also, we need to ensure that the collected Python code is correct for using SQL to access the database. For this reason, we have added the following restrictions:

- Return the built-in Python data objects (list, dict,...) after executing the SQL statement, instead of objects of other classes.
- Keep the SQL statements whose operations are related to only the SELECT keyword, not INSERT, UPDATE and CREATE.
- Remove project-related information and use parameters to express specific variables or classes outside the function block.

Programmers often need to generate concise code rather than an obscure code, so conciseness is very important for using the generated code in real scenarios. We set the following rules to achieve this goal:

- Simplify the complex variable names (more than 30 characters). This requires the annotator to simplify according to the functionality of the source code and the meaning of the original function name.
- Remove redundant information that does not affect the functionality of the source code, such as redundant spaces, line breaks and comments.
- Focus on the part of the program where Python code embedded with SQL statements. Try to remove Python code that has nothing to do with SQL operations.

**Comments Annotation**

In the task of code generation, the purpose and principle of annotation are that when the programmer sees the annotation, he/she can write the code with the same functionality as the source code. Complete comments can generate better source code, but more complete comments is more difficult to provide. Ideally, we aim to generate the most useful code with the simplest description. In order to assist annotators to annotate the key information in the source code more effectively, we also designed some rules.

First, for the purpose of source code annotation, we set the following principle for comment annotators: correctness, diversity, and clarity. Correctness requires that the annotator can understand the source code well and give the corresponding code description. Diversity and clarity are the requirements of the language level of the annotator. At best, other programmers should be able to write the same code by referring to the annotation.

Second, from the perspective of source code composition, we add some specific requirements. The composition of the source code collected in this paper can be divided into temporary variables, function parameters, import functions, and built-in methods. We set this rule to guide the annotator to decide which content must be described and which content can be omitted. Temporary variables in the source code do not need to be described in the comments because they may not affect the functionality of the code. The parameters in the source code need to be described in the comment and marked with the $ symbol. Import packages and built-in methods in source code generally do not need to be described in comments if they are commonly used. But if there are rarely used, they need to be distinguished in the comments.

Furthermore, specific to our task, we provide more detailed annotation guidance. For example, different SQL execution styles need to be distinguished in the comments, and SQL statements and their related operations should be relatively independent. These rules can make the annotator better understand the data we need for the current task.

Finally, we provided examples for each annotator to learn. In the early stage of annotation, we analyzed the examples of annotation errors and generate the corresponding documents. In addition, we used the way of real-time update and sharing to let the annotator avoid the subsequent annotation errors as much as possible.

### 3.3 Automatic Quality Checker

It is very time-consuming and error-prone to manually check the content annotated by the annotator one by one. In this paper, we designed an automated checker to check the modified code and annotated comments.

As described in Figure 3, our checker contains two basic components, which are content mismatch check for code comments and error detection for modified code. First, we designed a series of check items for the code comments. For example, we automatically check whether the comment completely describe all the parameters in the function, and check whether the comment match the corresponding source code through the characteristics of the code. Second, some function blocks may have specific project-related information, and cannot be used as independent function blocks. We used Pylint ([https://www.pylint.org](https://www.pylint.org)), a Python static code analysis tool, to check the code function. We check all the modified code and found some errors after modification. Finally, we returned all the problematic source code for re-modification.

### 3.4 Data Sampling Check

We randomly selected 100 examples from the dataset each time for checking. We further iteratively improved the rules of the quality checker by analyzing errors from the sampled
data. Until there were no obvious problems with the next random sample of data, we terminated the iterative process.

3.5 Final Review
Finally, we asked the most experienced annotator and English professionals to conduct the final review after the data sampling check. At this stage, we tried to ensure that there are no language problems in the comments and no sensitive information in the source code.

4 Dataset Statistics
We summarize the statistics of Lyra in Table 1. Lyra contains 2,000 source code snippets for database manipulation and their corresponding comments. There are 3 code execution styles in our dataset, which correspond to common SQL processing methods in SQLAlchemy. The first style is to execute raw SQL statements in strings. The second style is to execute SQL statements represented by Python expressions. The third style is to use SQLAlchemy’s ORM (Object Relational Mapper) to execute the SQL statements. To make the style of the dataset consistent, we allow the table objects in the second and third execution-styles to be passed in as parameters. In addition, there is no complex SQL statement in Lyra. The SQL components involved include SELECT, COUNT, WHERE, but no complex keywords like GROUP BY, ORDER BY.

5 Methods
To analyze the quality and demonstrate the purpose of our corpus, we experimented with several code generation models. Although many advanced methods generate code in the form of AST, they are designed to generate code from the syntax of a single programming language. These methods cannot directly be applied to our turducken-style task, so we chose some currently popular and generic neural models as baselines.

5.1 Transformer
Transformer [Vaswani et al., 2017] is a popular encoder-decoder framework that has surpassed RNNs on many sequence-to-sequence tasks. In the encoder, the Transformer first maps the input sequence to word embedding and position embedding. We also use word-level and BPE [Sennrich et al., 2016] methods to tokenize the source code. Then Transformer uses a stack of encoder layers for encoding. Finally, the decoder outputs the word distribution on the vocabulary.

5.2 BERT-Style Models
BERT-style models are pre-trained models based on the encoder in Transformer. Large pre-trained models can learn effective contextual representation from unlabeled data through self-supervised objectives and have brought significant improvement to various NLP tasks [Devlin et al., 2018]. We use CodeBERT [Feng et al., 2020] and GraphCodeBERT [Guo et al., 2020] as BERT-style baselines to generate code snippets. These models are pre-trained for natural language and programming language on the CodeSearchNet [Husain et al., 2019] dataset, which contains more than 2M functions of six programming languages paired with natural language documents. After adding the decoder structure to BERT-style pre-trained models, they can be used in generation tasks.

5.3 GPT-Style Models
GPT-style models are pre-trained models based on the decoder in Transformer. We use GPT-2 [Radford et al., 2019], CodeGPT, and CodeGPT-adapted [Lu et al., 2021] as GPT-style baselines. GPT-2 is pre-trained on WebText dataset with 1.5B parameters. CodeGPT shares the same model architecture with GPT-2, but CodeGPT is pre-trained from scratch on single programming language corpora in CodeSearchNet. CodeGPT-adapted is a domain-adaptive one, which take the GPT-2 model as the starting point and continually trained on code corpus. GPT-style models can perform directly on downstream tasks without adding any additional architecture.

6 Evaluation and Discussion
6.1 Evaluation Metrics
In our experiment, we use three types of metrics to evaluate the generated code on lexical similarity, syntactic similarity, and semantic similarity, respectively. For lexical similarity, we use BLEU (bilingual evaluation understudy) [Papineni et al., 2002] to compare the lexical similarity between the generated code and the reference code. For syntax similarity, we use Code Executable to judge whether the generated code is syntactically correct. For semantic similarity, we use AST Matching to evaluate the functionality of the generated code. Since the generated code usually contains a long SQL string and code for operating SQL, the AST Matching evaluates both the AST elements with and without SQL content, namely AST Exact Matching and AST Matching in Base Language. In particular, we did not use unit testing as a evaluation metric in our paper, because our code snippets are collected from Github, and it is hard to get test cases. Different from text-to-SQL datasets, our code not only comes from multiple projects but also has various parameters for each function. They cannot be executed and unit tested like SQL statements and it potentially takes heavy human involvement to construct test cases for these code snippets.

BLEU. The first quality metrics is BLEU. BLEU was initially proposed to assess the quality of machine translation [Papineni et al., 2002]. For code generation, BLEU scores are calculated to compare the lexical similarity between the generated code and the reference code, where the score is between 0 and 1. We use BLEU 4 to evaluate the generated code, which is also used to evaluate existing code generation techniques.

Code Executable. The second metric is the proportion of generated code that can be executed, in other words, can be successfully compiled. The premise of generating a functionally correct program is to ensure that the program can pass
In the first case, the model cannot correctly generate either SQL or Python parts in the program. In the second case, the model can only correctly generate one of the programming languages in the program, either Python or SQL. This is because the model cannot effectively learn the generation of another syntax or the model cannot learn the interaction between the two languages. For example, forgetting to generate the aggregate function of SQL or ignoring the correspondence of variables between SQL statements and Python code can lead to this situation. In the third case, the model generates code with correct functionality. This is a satisfactory situation, which means to generate the correct base program and embedded program at the same time. With the SQL in the Python program become complex and the processing before and after executing SQL becomes diverse, it is difficult to generate correct code.

Although the performance of the best model in our experiment is still lower than the state-of-the-art performance on text-to-SQL tasks or text-to-Python tasks, we believe that it is promising to achieve much better performance on our dataset in the future. First, no models in our experiment exploit the characteristics or the interactions of the two programming languages. A distinctive feature of our dataset is the involvement of two different languages. While this feature may impose extra difficulty for code generation, it also provides new opportunities for approaches exploiting this feature. Second, the nature of the imperative code in our dataset is different from that in existing imperative code generation datasets. The imperative code in our dataset focuses on preparing the SQL statements and collecting query results, and does not contain complex logic. Thus, we believe that generating the imperative code alone in our dataset may technically be less difficult than generating the imperative code in existing datasets. Achieving around 25% accuracy with a straightforward model in our experiment may have already indicated this trend.

### 7 Conclusion

In this paper, we define a turducken-style code generation task: generating a program in a base language with an embedded language by giving a natural language comment. We also introduce Lyra, a new dataset to support our task. Lyra contains 2,000 carefully annotated database manipulation programs using the Python programming language. These data are crawled from real projects in Github and each source code snippet is paired with both a Chinese comment and an English comment. Experimental results suggest that Lyra provides a new challenge for code generation. In future work, we plan to consider the characteristics of the two programming languages to improve the generation performance for Lyra. We also plan to explore more types of Turducken-style code generation tasks to promote the practical application of code generation, such as generating programs with JavaScript embedded in HTML and SQL embedded in XML or Java. In addition, we also plan to explore the different effects of Chinese and English comments on generation, as well as the use of natural languages other than Chinese and English.

### Acknowledgments

This work is sponsored by the National Key Research and Development Program of China (No. 2017YFB1001803), and the National Science Foundation of China (No. 61872011).
References


[Feng et al., 2020] Zhangxin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, EMNLP, pages 1536–1547, 2020.


