Code-Style In-Context Learning for Knowledge-Based Question Answering

Zhijie Nie[1,3], **Richong Zhang**[1,2 *], **Zhongyuan Wang**[1], **Xudong Liu**[1]
[1]SKLSDE, School of Computer Science and Engineering, Beihang University, Beijing, China
[2]Zhongguancun Laboratory, Beijing, China
[3]Shen Yuan Honors College, Beihang University, Beijing, China
{niezj,zhangrc,wangzy23,liuxd}@act.buaa.edu.cn

Abstract

Current methods for Knowledge-Based Question Answering (KBQA) usually rely on complex training techniques and model frameworks, leading to many limitations in practical applications. Recently, the emergence of In-Context Learning (ICL) capabilities in Large Language Models (LLMs) provides a simple and training-free semantic parsing paradigm for KBQA: Given a small number of questions and their labeled logical forms as demo examples, LLMs can understand the task intent and generate the logic form for a new question. However, current powerful LLMs have little exposure to logic forms during pre-training, resulting in a high format error rate. To solve this problem, we propose a code-style in-context learning method for KBQA, which converts the generation process of unfamiliar logical form into the more familiar code generation process for LLMs. Experimental results on three mainstream datasets show that our method dramatically mitigated the formatting error problem in generating logic forms while realizing a new SOTA on WebQSP, GrailQA, and GraphQ under the few-shot setting. The code and supplementary files are released at https://github.com/Arthurizijar/KB-Coder.

Introduction

Knowledge-Based Question Answering (KBQA) (Yih et al. 2016; Lan and Jiang 2020; Yu et al. 2022; Li et al. 2023b) is a long-studied problem in the NLP community. Due to the complexity and diversity of the questions, the models with good performance usually have complex modeling frameworks or special training strategies. However, these designs also lead to models that require a lot of labeled data to help the parameters converge, making them difficult to apply in the new domains. Recently, Large Language Models (LLMs) have strong generalization capabilities, benefiting from pre-training on vast amounts of natural language corpus and open source code (Figure 1 Top). In addition, the emergence of In-Context Learning (ICL) capabilities (Brown et al. 2020) allows LLMs to accomplish even complex reasoning tasks while observing a small amount of labeled data (Wei et al. 2022; Cheng et al. 2022).

Recently, Pangu (Gu, Deng, and Su 2023) first proposed a KBQA method based on the ICL paradigm, which consists of a symbolic agent and a language model. The agent is responsible for constructing effective programs step-by-step, while the language model is responsible for guiding the search process to evaluate the plausibility of candidate plans. KB-BINDER (Li et al. 2023b) proposes a training-free paradigm, which uses multiple (question, logical form) pairs as the demo examples, and leads the LLM to generate the correct logic form for the new question based on these demo examples (Figure 1 Middle). However, we find that this latest method still has several serious problems: (1) **High format error rate**. Due to the highly specialized nature of the logic forms, all of them can hardly ever appear in the training corpus of LLM. As a result, it is challenging to generate the logical form with the correct formatting with a few demo examples. In our preliminary experiments, the logic forms generated by the powerful GPT3.5-turbo have a more than 15% format error rate on GrailQA (Gu et al.

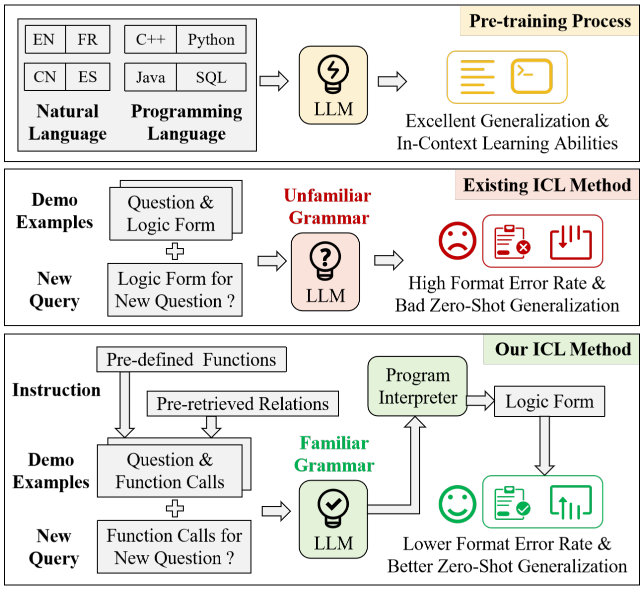


Figure 1: A comparison between our proposed ICL method and the existing method. Intuitively, our method achieves better performance by transforming the original KBQA task into a more familiar code form for the LLM.

*Corresponding author

2021). Without the tedious syntactic parsing steps with error correction, the logic form generated cannot even provide a final answer to the question; (2) **Low zero-shot generalization performance**. In the zero-shot generalization scenario, the prior knowledge of the KB domain schema related to the test question is unable to be obtained through the training set (Gu et al. 2021). In other words, it means that demo examples created from the training set cannot provide enough information for question answering under this scenario. Considering the labeled data usually covers only a small fraction of knowledge in KB, this problem will inevitably affect the performance in practice. To address these problems, the key is to deeply understand the behavior of the LLMs. Thankfully, some empirical observations on LLMs have provided desirable insights, which include (1) converting original tasks to code generation tasks will reduce the difficulty of post-processing (Wang, Li, and Ji 2022); (2) reasoning step-by-step can improve LLMs performance on complex reasoning tasks (Wei et al. 2022); (3) retrieval augmentation is helpful for LLMs in dealing with uncommon factual knowledge (Mallen et al. 2023).

Inspired by the above valuable observations, we propose a novel code-style in-context learning method for KBQA, which converts the one-step generation process of the logic forms to the progressive generation process of the function calls in Python (Figure 1 Bottom). Specifically, we first define seven meta-functions for S-Expression (Gu et al. 2021), a popular logic form for KBQA, and implement these functions in Python. Note that this step enables all S-Expressions to be generated by a finite number of calls of pre-defined meta-functions. For a test question, we sample a few numbers of (question, S-Expression) pairs in the training set and convert them into (question, function call sequence), where the function call sequence can be executed in the Python interpreter to output S-Expression. Finally, the Python implementation of the meta-functions, all (question, function call sequences) pairs, and the test question are reformatted in code form as input to the LLM. And the LLM is expected to complement a complete function call sequence for the new question to obtain the correct logic form. In addition, we find a simple and effective way to improve the performance in the zero-shot generalization scenario: Regard the test question as the query to retrieve a related relation in KB, and provide LLM with the relation for reference.

Our contribution can be summarized as follows:

- We propose a novel code-style in-context learning method for KBQA. Compared to the existing methods, our method can effectively reduce the format error rate of the logic form generated by LLMs while providing additional intermediate steps during reasoning.
- We find that providing a question-related relation as a reference to LLMs in advance can effectively improve the performance in the zero-shot generalization scenario.
- We design a training-free KBQA model, KB-Coder, based on the proposed ICL method. Extensive experiments on WebQSP, GrailQA, and GraphQ show that our model achieves SOTA under the few-shot setting. While allowing access to the full training set, the training-free KB-Coder achieves competitive or better results compared with current supervised SOTA methods.

Related Work

Complex Reasoning with Code-LLMs Codex (Chen et al. 2021) first introduces a code corpus to train LLMs and finds that the obtained code-LLMs have excellent logical reasoning capabilities. Subsequently, code-LLMs have been used for a variety of complex tasks in two ways. The works in the first way only implicitly use the reasoning power that derives from code pre-training and proposes techniques such as chain-of-thought (Wei et al. 2022) and problem decomposition (Zhou et al. 2022), etc. And the works in another way convert the task form into code generation and guide the LLMs to achieve the original task goal by creating instances (Wang, Li, and Ji 2022), complementary code (Li et al. 2023a) or generating SQL (Cheng et al. 2022) or logic forms (Li et al. 2023b) directly.

Question Answering with LLMs We distinguish the different methods according to the type of LLM-generated content. The first class of methods guide LLMs to generate answer directly (Li et al. 2023c; Baek, Aji, and Saffari 2023). In these methods, the knowledge in the external knowledge source is converted into the index, then the questions are used as queries to obtain relevant knowledge from the index through sparse or dense retrieval. Then the questions and related knowledge are spliced together and fed into the LLM to generate the answers directly. The second class of methods (Izacard et al. 2022; Gu, Deng, and Su 2023; Tan et al. 2023) views the LLM as a discriminator and leads LLMs to choose the correct answer or action from a candidate set. The third class of methods (Li et al. 2023b; Cheng et al. 2022) views the LLM as a semantic parser and guides the model to generate intermediate logic forms. Compared to generating the answer directly, the other two classes of methods can eliminate the risk of generating fake knowledge in principle but cannot achieve an end-to-end method. Beyond the above three classes, DECAF (Yu et al. 2022) is a special case that directs LLM to generate both logic forms and the final answer by changing the prompt.

Method

Overview

In general, our proposed KBQA method is a method based on semantic parsing: Given a natural language question q and KB $\mathcal{G} = \{(h, r, t), h, t \in \mathcal{E}, r \in \mathcal{R}\}$, where $\mathcal{E}$ is the entity set and $\mathcal{R}$ is the relation set, our method can be viewed as a function F, which maps q to a semantically consistent logic form $l = F(q)$. Then l is converted into a query to execute, and the queried results are regarded as the answers to q. Specifically, we first design seven meta-functions, which can cover all atomic operations of a specific logic form, and re-define these meta-functions in Python. Finally, for a new question, the following three steps are adopted to get the answer (Figure 3): (1) An LLM is adopted to obtain its function call sequence with the code-style in-context learning method; (2) A dense retriever is utilized to link entities for

Function	Domain	Range	Mapping Descriptions
START	$\{E \mid E \in \mathcal{P}(\mathcal{E})\}$	$\{E' \mid E \in \mathcal{P}(\mathcal{E})\}$	Start from E and return the same set $E' = E$
JOIN	$\{(r, E) \mid r \in \mathcal{R}, E \in \mathcal{P}(\mathcal{E})\}$		Return E' that pointed by r from E
AND	$\{(E_1, E_2) \mid E_1, E_2 \in \mathcal{P}(\mathcal{E})\}$		Return the intersection E' of E_1 and E_2
CMP	$\{(c, r, v) \mid r \in \mathcal{R}, v \in \mathcal{V})\}$		Return $E' \subset E$ whose value pointed by r $</>/\leq/\geq$ v
ARG	$\{(a, E, r) \mid E \in \mathcal{P}(\mathcal{E}), r \in \mathcal{R}\}$	$\{e \mid e \in \mathcal{E}\}$	Return $e \in E$ whose value pointed by r is largest / smallest
COUNT	$\{E \mid E \in \mathcal{P}(\mathcal{E})\}$	$\{n \mid n \in \mathbb{N}\}$	Return the element number n of E
STOP	$\{E \mid E \in \mathcal{P}(\mathcal{E})\}$	–	Stop at E and E is regarded as the predicted answer set

Table 1: Seven meta-functions with their domain, range, and mapping descriptions. $c \in \{<, >, \leq, \geq\}$, $a \in \{\min, \max\}$, $\mathcal{P}(.)$ represents the power set of a given set, $\mathcal{V}$ is a subset of $\mathcal{E}$ containing all entities in value type, and $\mathbb{N}$ represents the set of natural numbers. For the other notations, please refer to the **Overview** section.

the entity mentions extracted from the function calls, while another dense retriever is utilized to match relations for the relation name extracted from the function calls; (3) A program interpreter is used to execute the generated function call sequence to get the logical form l, which will be executed further to get the answer a.

Meta-Function Design

In practice, we use S-Expression defined by Gu et al. (2021) as the logical form l due to its simplicity. S-Expression (McCarthy 1960) is a name-like notation for the nested list (tree-structured) data, which conforms to the grammar of "prefix notation". Specifically, S-Expression usually consists of parentheses and several space-separated elements within them, where the first element is the function name and all remaining elements are the attributes of the function. For example, for the question "how many American presenters in total", its S-Expression is

```
(COUNT (AND (JOIN nationality m.09c7w0)
            (JOIN profession m.015cjr)))
```

where `m.09c7w0` and `m.015cjr` are the unique identifiers of entity `United States of America` and `Presenter`. The corresponding tree structure of this S-Expression is shown on the left side of Figure 2.

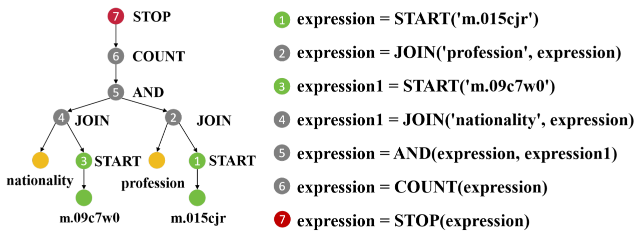
Based on the original syntax of S-Expression (Gu et al. 2021), we define seven meta-functions in total (Table 1), which include their name, domain, range, and mapping description. Compared to the original grammar, we omit the **R** function, remove a call way for the **JOIN** function, and add the **START** and **STOP** functions.

Code-Style In-Context Learning

A typical in-context learning paradigm (Brown et al. 2020) generally includes an instruction I, K demo examples $D = \{d_1, d_2, ..., d_K\}$, and a new query Q. If the output of the LLM is denoted as C, we can express the process of the in-context learning as

$$C = f_{\text{LLM}}(I; D; Q) \tag{1}$$

where f_{LLM} represents a specific LLM. In our method, we use the code style to construct I, D, and Q and expect the model to generate a piece of code - a sequence of meta-function calls - for Q, following the demo example. Next, we describe in detail how to construct each part respectively.

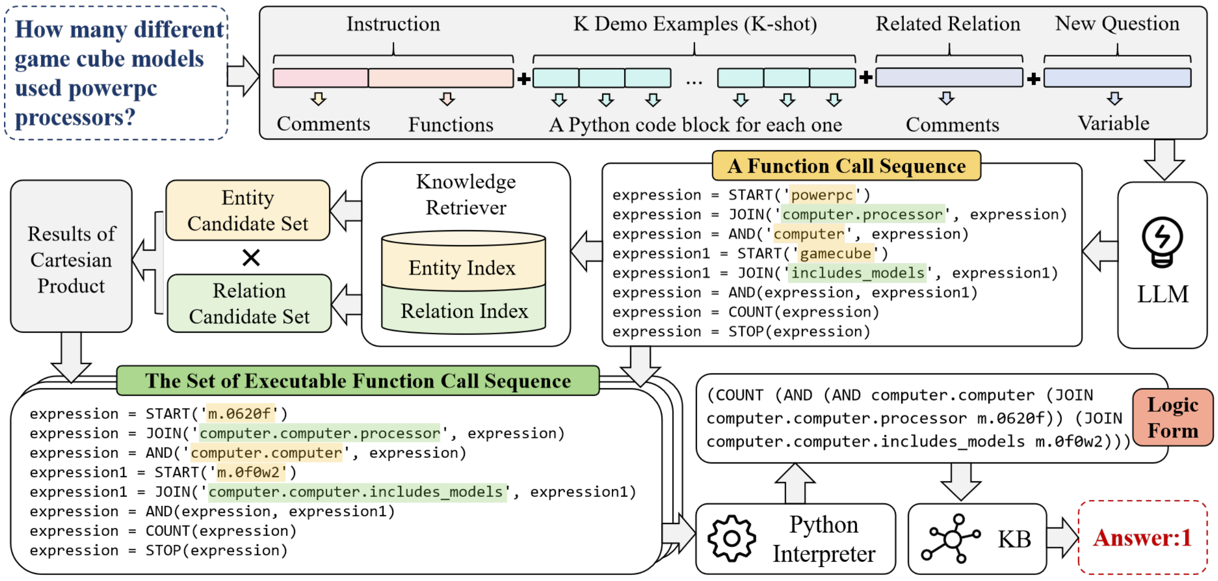


```
1 expression = START('m.015cjr')
2 expression = JOIN('profession', expression)
3 expression1 = START('m.09c7w0')
4 expression1 = JOIN('nationality', expression)
5 expression = AND(expression, expression1)
6 expression = COUNT(expression)
7 expression = STOP(expression)
```

Figure 2: The tree structure (left) and the corresponding function call sequence (right) of S-Expression `(COUNT (AND (JOIN nationality m.09c7w0) (JOIN profession m.015cjr)))`.

Instruction I Similar to previous works on code-LLMs (Cheng et al. 2022; Li et al. 2023a), the instruction consists of two parts: a prompt comment for describing the task and the code implementation of seven meta-functions. Due to the successful practice of Codex (Chen et al. 2021) in Python, we select Python to implement these functions. Then, the complete contents of I are shown in the code from line 1 to line 26 in Figure 4. All seven functions are the implementation of the same-name function in Table 1.

Demo Examples D Each demo example is created from a (question, S-Expression) pair in the training set and contains two parts: a variable named `question` and a function call sequence. Specifically, string variable `question` is assigned by the question, while the function call sequence is converted by the S-Expression. The right side of Figure 2 provides an example of the function call sequence, where each function call can correspond to one non-leaf node in the tree structure. Thus, the function call sequence can also be regarded as the result of a bottom-up parsing of the tree structure. If the meta-function definitions in Instruction are spliced together with the function call sequence as a whole code into the Python interpreter, we can get the complete S-Expression by visiting the value of variable `expression`. Since the entity identifiers in KB, such as `m.03yndmb`, lose



```
expression = START('powerpc')
expression = JOIN('computer.processor', expression)
expression = AND('computer', expression)
expression1 = START('gamecube')
expression1 = JOIN('includes_models', expression1)
expression = AND(expression, expression1)
expression = COUNT(expression)
expression = STOP(expression)
```

```
expression = START('m.0620f')
expression = JOIN('computer.computer.processor', expression)
expression = AND('computer.computer', expression)
expression1 = START('m.0f0w2')
expression1 = JOIN('computer.computer.includes_models', expression1)
expression = AND(expression, expression1)
expression = COUNT(expression)
expression = STOP(expression)
```

```
(COUNT (AND (AND computer.computer (JOIN
computer.computer.processor m.0620f)) (JOIN
computer.computer.includes_models m.0f0w2)))
```

Figure 3: An illustration of the inference process of KB-Coder.

their semantic information, we map all entity identifiers to their surface name to help the LLM obtain the semantic information of these entities. Finally, the demo example corresponding to Figure 2 is reformulated as the code from line 28 to line 35 in Figure 4. To obtain D, we sample K (question, S-Expression) in the training set in total, convert each pair to the above form, and split them using line breaks.

New Query Q Compared to the form of demo examples, the new query is an incomplete piece of code and only contains the part of the variable `question`, For example, the question "how many religious texts does syncretism have" will be reformulated the code of line 42 in Figure 4. For each Q, the LLM is expected to complement the remaining function call sequence corresponding to the question with the capability of in-context learning.

In practice, I remains constant. For D, we can provide a consistent D for each Q in the dataset to follow the few-shot setting, or we can select a different D for each Q based on the similarity for better performance. Both two settings will be studied in subsequent experiments.

Reasoning

In this section, we describe how to post-process the function call sequence generated by the LLM so that they can be converted into the query, obtaining the predicted answer.

Entity Linking & Relation Matching Recall that the LLM can not have a direct view of the information in KB, making it difficult to generate completely correct entity names and relationship names. However, We believe that the names generated have a high level of semantic similarity with the correct ones, so the names generated can be treated as mentions for entity linking and relation matching. Benefiting from our strict definition of the domain of meta-functions, both entity mentions and relation mentions can be parsed easily. Specifically, for entity linking, we first convert all surface names of all entities in the KB into representations by the off-the-shelf embedding model, SimCSE (Gao, Yao, and Chen 2021), and build the entity index with Faiss (Johnson, Douze, and Jégou 2019). When linking for an entity mention, we first use the HARD MATCH strategy to obtain all entities that have the same surface name as the mention as the candidate set. If the size of the candidate set is larger than a hyper-parameter M_e, we will only retain the most popular M_e entities referring to FACC1 (Gabrilovich, Ringgaard, and Subramanya 2013). In contrast, if the size is less than M_e, we will search the most similar entities from the existing entity index to fill the candidates up to M_e. Similarly, we use the same techniques to index all relations in the KB and retrieve M_r most semantically similar relations as candidates for each generated relation name.

Answer Prediction Without loss of generality, we assume that the function call sequence contains p entities to be linked and q relations to be matched, denote the candidate set of the i-th entity mention as $\mathbb{C}_\text{e}^i$ and the candidate set of the j-th relation mention as $\mathbb{C}_\text{r}^j$. Then we can obtain the ordered tuple set $\{(c_\text{e}^1, ..., c_\text{e}^p, c_\text{r}^1, ..., c_\text{r}^q)\} = \mathbb{C}_\text{e}^1 \times ... \times \mathbb{C}_\text{e}^p \times \mathbb{C}_\text{r}^1 \times ... \times \mathbb{C}_\text{r}^q$, where $\times$ represents the Cartesian product. For each ordered tuple in the set, let each element in the ordered tuple replace the corresponding generated names in the function call sequence one by one, we can obtain a candidate for the whole function call sequence (Figure 3). Finally, there will be $(M_e)^p.(M_r)^q$ candidate items for each function call sequence, which can be a huge number for some special cases. Therefore, we will execute the sequence of function calls one by one to get the S-Expression, and then convert the S-Expression to SPARQL to execute it. Once the queried result is not null, we will just terminate the process of trying one by one and consider the result of the queried result as the answer to the question.

```
 1 '''
 2 Please use the functions defined below to generate the
 3 expression corresponding to the question step by step.
 4 '''
 5 def START(entity: str):
 6     return entity
 7
 8 def JOIN(relation:str, expression:str):
 9     return '(JOIN {} {})'.format(relation, expression)
10
11 def AND(expression:str, sub_expression:str):
12     return '(AND {} {})'.format(expression, sub_expression)
13
14 def ARG(op:str, expression:str, relation:str):
15     assert op in ['ARGMAX', 'ARGMIN']
16     return '({} {} {})'.format(op, expression, relation)
17
18 def CMP(op:str, relation:str, expression:str):
19     assert operator in ['<', '<=', '>', '>=']
20     return '({} {} {})'.format(op, relation, expression)
21
22 def COUNT(expression:str):
23     return '(COUNT {})'.format(expression)
24
25 def STOP(expression:str):
26     return expression
27
28 question = 'how many american presenters in total'
29 expression = START('presenter')
30 expression = JOIN('profession', expression)
31 expression1 = START('united States of America')
32 expression1 = JOIN('nationality', expression)
33 expression = AND(expression, expression1)
34 expression = COUNT(expression)
35 expression = STOP(expression)
36
37 '''
38 Some relations for reference are as follows:
39 location.country.languages_spoken
40 '''
41
42 question = 'what does jamaican people speak'
```

Figure 4: An example of contextual learning of a code style. The part of demo examples is shown only one example due to space constraints.

One Related Relation for Zero-shot Generalization

Code-Style in-context learning allows the LLM to be more adaptable to the task form, resulting in a lower format error rate. However, when the queried domain is not involved in the demo examples, which is called the zero-shot generalization scenario by (Gu et al. 2021), the performance of the LLM is still poor. This is not difficult to understand, as the change in the prompt form of the task does not bring new additional knowledge of the queried domain to the LLM. After analyzing the bad cases from the preliminary experiments, we found that the biggest problem with the error was relation matching, where the relation mentions generated by the LLM usually cannot hit the correct relation under the zero-shot generalization scenario. Subsequently, we mitigate this problem by providing an additional relation name for LLM. Specifically, we use the entire test question as a query to retrieve the similar relation from the relation index, and the relation with the highest similarity is inserted between demo examples D and the new question Q with the comment format like the code from line 37 to line 40 in Figure 4. Based on preliminary experiments, it is observed that one relation brings the best performance and more relations have little improvement on the performance. We will analyze the effect of the number of relations on the results in detail in the **Experiment** section.

Experiment

Experiment Setup

Dataset We use three mainstream datasets in KBQA, WebQSP (Yih et al. 2016), GraphQ (Su et al. 2016), and GrailQA (Gu et al. 2021), which represent the three generalization capabilities of i.i.d, compositional, and zero-shot, respectively, to evaluate the effect of KB-Coder.

LLM Due to the deprecation of the Codex family of models, we select `gpt-3.5-turbo` from OpenAI for our experiments. In all experiments, we used the official API [1] to obtain model results, where `temperature` is set to 0.7, `max_tokens` is set to 300, and other parameters are kept at default values.

Baseline We mainly compare our model with KB-BINDER (Li et al. 2023b), the SOTA model on WebQSP, GrailQA, and GraphQ under the few-shot setting. The original results of KB-BINDER are obtained with the deprecated `code-davinci-002`, and we reproduce their method with `gpt-3.5-turbo` while the other setting remains consistent with us for a fair comparison. Some results obtained by training on the whole training dataset (Ye et al. 2022; Shu et al. 2022; Gu et al. 2021; Sun, Bedrax-Weiss, and Cohen 2019) are also reported for reference.

Evaluation Metric Consistent with previous works (Yu et al. 2022; Li et al. 2023c), we report F1 Score on WebQSP and GraphQ, while Exact Match (EM) and F1 Score on GrailQA as performance metrics. At the same time, we use the Format Error Rate (FER) to evaluate the proportion of logical forms generated by different methods that conform to the grammar of S-Expresssion.

Implementation Details

Without special instructions, we reported the experiment results with $M_{\mathrm{e}} = 15$ and $M_{\mathrm{r}} = 100$. SimCSE instance `sup-simcse-bert-base-uncased` is used to obtain the dense representations. Consistent with KB-BINDER (Li et al. 2023b), we conduct 100-shot for WebQSP and GraphQ, and 40-shot for GrailQA. In the following sections, we use different notations to represent different variants:

- **KB-Coder (K)**: The fixed questions sampled randomly from the training set are selected to be the demo examples. The LLM generates K candidates and uses the majority vote strategy (Wang et al. 2022) to select the final answer. The performance is expected to be further improved thanks to the self-consistency of the LLM.

[1] https://openai.com/api

Method	I.I.D		Compositional		Zero-Shot		Overall		
	EM	F1	EM	F1	EM	F1	FER↓	EM	F1
Full Supervised on the Entire Training Set									
RnG-KBQA (Ye et al. 2022)	86.7	89.0	61.7	68.9	68.8	74.7	-	69.5	76.9
DecAF (Yu et al. 2022)	88.7	92.4	71.5	79.8	65.9	77.3	-	72.5	81.4
TIARA (Shu et al. 2022)	88.4	91.2	66.4	74.8	73.3	80.7	-	75.3	81.9
In-Context Learning (Training-Free)									
KB-BINDER (1)	$40.0_{2.3}$	$43.3_{2.7}$	$33.9_{2.7}$	$36.6_{2.6}$	$40.1_{3.6}$	$44.0_{4.1}$	$20.0_{2.4}$	$38.7_{3.0}$	$42.2_{3.3}$
KB-Coder (1)	$\mathbf{40.6}_{3.3}$	$\mathbf{45.5}_{2.8}$	$\mathbf{34.5}_{3.6}$	$\mathbf{38.6}_{3.5}$	$\mathbf{42.2}_{5.9}$	$\mathbf{47.3}_{5.4}$	$\mathbf{3.0}_{0.9}$	$\mathbf{40.1}_{3.7}$	$\mathbf{44.9}_{3.4}$
KB-BINDER (6)	$43.6_{2.1}$	$48.3_{2.5}$	$\mathbf{44.5}_{2.3}$	$48.8_{2.7}$	$37.5_{2.4}$	$41.8_{2.8}$	$8.1_{0.4}$	$45.7_{2.3}$	$50.8_{2.8}$
KB-Coder (6)	$\mathbf{43.6}_{3.7}$	$\mathbf{49.3}_{3.3}$	$44.0_{2.2}$	$\mathbf{49.6}_{1.9}$	$\mathbf{37.7}_{2.6}$	$\mathbf{43.2}_{2.6}$	$\mathbf{0.6}_{0.2}$	$\mathbf{45.9}_{3.9}$	$\mathbf{51.7}_{3.3}$
KB-BINDER (1)-R	$74.7_{0.1}$	$79.7_{0.1}$	$44.6_{0.4}$	$48.5_{0.5}$	$37.1_{0.2}$	$40.8_{0.1}$	$16.4_{0.2}$	$47.6_{0.0}$	$51.7_{0.1}$
KB-Coder (1)-R	$\mathbf{76.2}_{3.0}$	$\mathbf{80.2}_{1.9}$	$\mathbf{50.4}_{0.7}$	$\mathbf{54.8}_{0.7}$	$\mathbf{45.8}_{0.4}$	$\mathbf{50.6}_{0.9}$	$\mathbf{3.1}_{0.4}$	$\mathbf{54.0}_{1.0}$	$\mathbf{58.5}_{1.0}$
KB-BINDER (6)-R	$75.8_{0.1}$	$80.9_{0.1}$	$48.3_{0.4}$	$53.6_{0.4}$	$45.4_{0.2}$	$50.7_{0.3}$	$5.2_{0.1}$	$53.2_{0.1}$	$58.5_{0.1}$
KB-Coder (6)-R	$\mathbf{76.9}_{3.1}$	$\mathbf{81.0}_{1.8}$	$\mathbf{52.7}_{0.9}$	$\mathbf{57.8}_{1.0}$	$\mathbf{48.9}_{0.2}$	$\mathbf{54.1}_{0.6}$	$\mathbf{1.5}_{0.1}$	$\mathbf{56.3}_{0.9}$	$\mathbf{61.3}_{1.0}$

Table 2: 40-shot results on the local dev set of GrailQA. The subscript is the standard deviation of the three runs.

Method	FER↓	F1
Full Supervised on the Entire Training Set		
ArcaneQA (Gu and Su 2022)	-	75.6
TIARA (Shu et al. 2022)	-	76.7
DecAF (Yu et al. 2022)	-	78.7
In-Context Learning (Training-Free)		
KB-BINDER (1)	$3.9_{1.1}$	$52.6_{1.1}$
KB-Coder (1)	$\mathbf{1.9}_{2.3}$	$\mathbf{55.7}_{1.3}$
KB-BINDER (6)	$0.3_{0.4}$	$56.6_{1.7}$
KB-Coder (6)	$\mathbf{0.1}_{0.1}$	$\mathbf{60.5}_{1.9}$
KB-BINDER (1)-R	$1.9_{0.3}$	$68.9_{0.3}$
KB-Coder (1)-R	$\mathbf{1.7}_{0.3}$	$\mathbf{72.2}_{0.2}$
KB-BINDER (6)-R	$0.7_{0.0}$	$71.1_{0.2}$
KB-Coder (6)-R	$\mathbf{0.3}_{0.2}$	$\mathbf{75.2}_{0.5}$

Table 3: 100-shot results on WebQSP. The subscript is the standard deviation of the three runs.

Method	FER↓	F1
Full Supervised on the Entire Training Set		
SPARQA (Sun et al. 2020)	-	21.5
BERT + Ranking (Gu et al. 2021)	-	25.0
ArcaneQA (Gu and Su 2022)	-	31.8
In-Context-Learning (Training-Free)		
KB-BINDER (1)	$15.4_{1.6}$	$27.1_{0.5}$
KB-Coder (1)	$\mathbf{6.3}_{2.5}$	$\mathbf{31.1}_{1.3}$
KB-BINDER (6)	$2.8_{0.6}$	$34.5_{0.8}$
KB-Coder (6)	$\mathbf{0.6}_{0.2}$	$\mathbf{35.8}_{0.6}$
KB-BINDER (1)-R	$19.1_{0.1}$	$26.7_{0.3}$
KB-Coder (1)-R	$\mathbf{10.0}_{0.2}$	$\mathbf{30.0}_{0.3}$
KB-BINDER (6)-R	$5.7_{0.5}$	$32.5_{0.5}$
KB-Coder (6)-R	$\mathbf{0.9}_{0.1}$	$\mathbf{36.6}_{0.2}$

Table 4: 100-shot results on GraphQ. The subscript is the standard deviation of the three runs.

- **KB-Coder (K)-R**: Compared to KB-Coder (K), the questions most similar to every test question are selected as their demo examples.

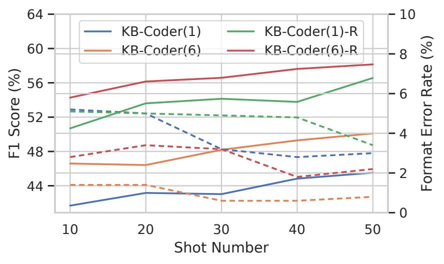
Note that **KB-Coder (K)** is strictly under the **few-shot** setting, while **KB-Coder (K)-R** is to explore the upper bound on performance when the whole training set can be accessed. We report results for K = 1 and 6 for fair comparison with KB-BINDER. For each setting, we rerun it three times and report the mean and standard deviation.

Main Results

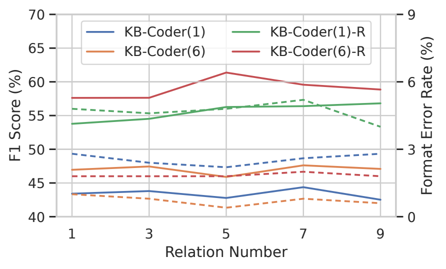
We report the performance of KB-Coder and other baselines on GrailQA, WebQSP, and GraphQ in Tables 2, 3 and 4. Next, we will analyze the format error rate (FER) and the performance metrics (EM and F1) respectively.

Recall that one of the motivations for converting logical form generation to code generation is to allow LLM to do a more familiar task to ensure the correctness of the format of the generated content. the performance on all three datasets successfully verifies the effectiveness of our method under the few-shot setting: (1) On WebQSP, where the questions are simpler, the existing method generates logical forms with a low FMR, but KB-Coder can still further reduce the FMR to even lower levels; (2) On GrailQA and GraphQ, where the questions are more complex, KB-Coder improves dramatically compared to the two methods generating logic form directly; (3) Benefiting from self-consistency in LLMs, the majority vote strategy will help alleviate the problem of KB-BINDER format error rates. While KB-Coder can work with the majority vote strategy to promote new lows in FMR.

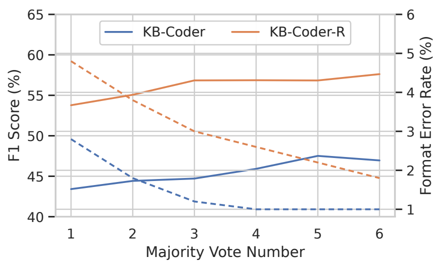
Benefiting from the lower FER, the F1 Scores (or EM) on the three datasets are both significantly improved compared to KB-BINDER under almost all settings, especially with two settings that do not introduce the majority vote strategy. (1) On WebQSP and GraphQ, the training-free KB-Coder(6)-R achieves competitive results compared to full-supervised methods, while our method further narrows down the performance with those of the fully-supervised model On GrailQA; (2) Compared to KB-BINDER, KB-Coder usually obtains a substantial lead under no dependence, reflecting the better underlying performance of our



(a) Effect of the shot number



(b) Effect of the relation number



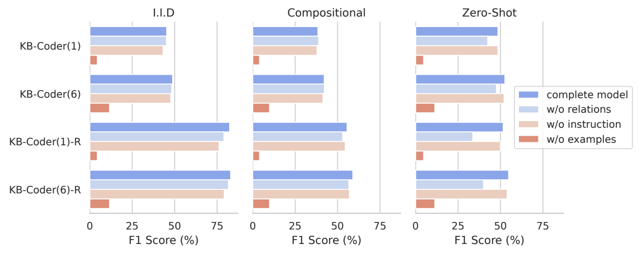
(c) Effect of the vote number

Figure 5: Effect analysis of the three factors in ICL with a subset of 500 questions from GrailQA local dev set, where the solid line is used to indicate F1 Score and the dashed line is used to indicate FER.

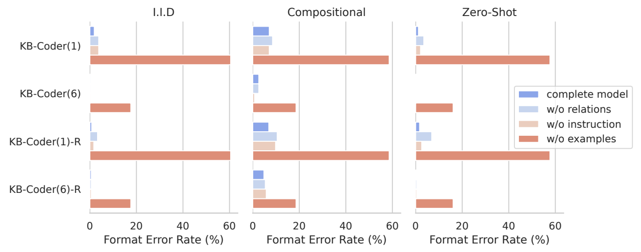
method; (3) The performance fluctuations of KB-Coder are more drastic compared to KB-BINDER, which is an issue we should focus to solve in the future work.

Ablation Study

To explore the necessity of all parts of our method, we consider three settings in the ablation experiments: (1) removing related relation (-w/o relations) (2) removing instructions (-w/o instruction); (3) removing demo examples (-w/o examples). We report the results of three generalization levels in GrailQA separately in Figure 6. From the results, it can be seen that removing relations drastically reduces the F1 performance of the questions for zero-shot generalization, while having little impact on i.i.d and compositional questions. On top of that, removing instruction would bring about a weak degeneration in the F1 and FER. And removing demo examples would render the entire paradigm nearly invalid, meaning that it would be difficult for LLMs to understand the task requirements based on instructions alone.



(a) The ablation study on F1 Score.



(b) The ablation study on Format Error Rate.

Figure 6: Ablation Study on GrailQA.

Analysis on In-Context Learning

In this section, we explore the impact of three factors in ICL: the number of demo examples, the number of related relations, and the number of participating majority votes on the performance. For cost reasons, we performed this experiment on a subset of the 500-size scale on GrailQA.

The Effects of Demo Examples We analyze the effect of the shot number on the results. Specifically, we set $K = \{10, 20, 30, 40, 50\}$ and report the performance trend with the shot number in Figure 5a. The results show that boosting the number of shot numbers has a stabilizing effect on F1, whereas FER was maintained at a low level throughout.

The Effects of Related Relation Number Similarly, we explore the effect of the number of correlations on the results. We set the related relation number as $\{1, 3, 5, 7, 9\}$ respectively and report the performance trend in Figure 5b. The results show that the introduction of more relations has little effect on both F1 and FER. Instead, too many relations make the performance degrade.

The Effects of Answer Number We explore the effect of the answer number of participating in the majority vote on the performance. Specifically, we set the answer number as $\{1, 2, 3, 4, 5, 6\}$ respectively and report the performance trend in Figure 5c. The results show that FER improves significantly as the number of results participating in the vote rises, and there was an upward trend in F1 as well.

Conclusion

In this paper, we design a training-free KBQA framework, KB-Coder, which centers on a code-style in-context learning method for reducing formatting errors in generated logic forms and a retrieval-augment method for boosting zero-shot generalization capability. Extensive experimental results demonstrate that variants of our model, KB-Coder(1) and KB-Coder(6), achieve the SOTA performance under the few-shot setting, while the other two variants, KB-Coder(1)-R and KB-Coder(6)-R, achieves competitive performance compared to fully-supervised methods in a training-free premise. In general, KB-Coder demonstrates the potential of code-style ICL in KBQA and offers a training-free but effective baseline for the community.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant 2022ZD0120200, in part by the National Natural Science Foundation of China (No. U23B2056), in part by the Fundamental Research Funds for the Central Universities, and in part by the State Key Laboratory of Software Development Environment.

References

Baek, J.; Aji, A. F.; and Saffari, A. 2023. Knowledge-Augmented Language Model Prompting for Zero-Shot Knowledge Graph Question Answering. *arXiv preprint arXiv:2306.04136*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Cheng, Z.; Xie, T.; Shi, P.; Li, C.; Nadkarni, R.; Hu, Y.; Xiong, C.; Radev, D.; Ostendorf, M.; Zettlemoyer, L.; et al. 2022. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*.

Gabrilovich, E.; Ringgaard, M.; and Subramanya, A. 2013. FACC1: Freebase annotation of ClueWeb corpora, Version 1 (Release date 2013-06-26, Format version 1, Correction level 0).

Gao, T.; Yao, X.; and Chen, D. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Gu, Y.; Deng, X.; and Su, Y. 2023. Don't Generate, Discriminate: A Proposal for Grounding Language Models to Real-World Environments. In Rogers, A.; Boyd-Graber, J.; and Okazaki, N., eds., *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 4928–4949. Toronto, Canada: Association for Computational Linguistics.

Gu, Y.; Kase, S.; Vanni, M.; Sadler, B.; Liang, P.; Yan, X.; and Su, Y. 2021. Beyond IID: three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, 3477–3488.

Gu, Y.; and Su, Y. 2022. ArcaneQA: Dynamic Program Induction and Contextualized Encoding for Knowledge Base Question Answering. In *Proceedings of the 29th International Conference on Computational Linguistics*, 1718–1731.

Izacard, G.; Lewis, P.; Lomeli, M.; Hosseini, L.; Petroni, F.; Schick, T.; Dwivedi-Yu, J.; Joulin, A.; Riedel, S.; and Grave, E. 2022. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299*.

Johnson, J.; Douze, M.; and Jégou, H. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3): 535–547.

Lan, Y.; and Jiang, J. 2020. Query Graph Generation for Answering Multi-hop Complex Questions from Knowledge Bases. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 969–974.

Li, P.; Sun, T.; Tang, Q.; Yan, H.; Wu, Y.; Huang, X.; and Qiu, X. 2023a. CodeIE: Large Code Generation Models are Better Few-Shot Information Extractors. In Rogers, A.; Boyd-Graber, J. L.; and Okazaki, N., eds., *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, 15339–15353. Association for Computational Linguistics.

Li, T.; Ma, X.; Zhuang, A.; Gu, Y.; Su, Y.; and Chen, W. 2023b. Few-shot In-context Learning on Knowledge Base Question Answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 6966–6980. Toronto, Canada: Association for Computational Linguistics.

Li, X.; Zhao, R.; Chia, Y. K.; Ding, B.; Bing, L.; Joty, S.; and Poria, S. 2023c. Chain of Knowledge: A Framework for Grounding Large Language Models with Structured Knowledge Bases. *arXiv preprint arXiv:2305.13269*.

Mallen, A.; Asai, A.; Zhong, V.; Das, R.; Khashabi, D.; and Hajishirzi, H. 2023. When Not to Trust Language Models: Investigating Effectiveness of Parametric and Non-Parametric Memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 9802–9822. Toronto, Canada: Association for Computational Linguistics.

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4): 184–195.

Shu, Y.; Yu, Z.; Li, Y.; Karlsson, B.; Ma, T.; Qu, Y.; and Lin, C.-Y. 2022. TIARA: Multi-grained Retrieval for Robust Question Answering over Large Knowledge Base. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 8108–8121.

Su, Y.; Sun, H.; Sadler, B.; Srivatsa, M.; Gür, I.; Yan, Z.; and Yan, X. 2016. On generating characteristic-rich question sets for qa evaluation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 562–572.

Sun, H.; Bedrax-Weiss, T.; and Cohen, W. 2019. PullNet: Open Domain Question Answering with Iterative Retrieval on Knowledge Bases and Text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2380–2390.

Sun, Y.; Zhang, L.; Cheng, G.; and Qu, Y. 2020. SPARQA: skeleton-based semantic parsing for complex questions over knowledge bases. In *Proceedings of the AAAI conference on artificial intelligence*, 8952–8959.

Tan, C.; Chen, Y.; Shao, W.; and Chen, W. 2023. Make a Choice! Knowledge Base Question Answering with In-Context Learning. *arXiv preprint arXiv:2305.13972*.

Wang, X.; Li, S.; and Ji, H. 2022. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810*.

Wang, X.; Wei, J.; Schuurmans, D.; Le, Q. V.; Chi, E. H.; Narang, S.; Chowdhery, A.; and Zhou, D. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837.

Ye, X.; Yavuz, S.; Hashimoto, K.; Zhou, Y.; and Xiong, C. 2022. RNG-KBQA: Generation Augmented Iterative Ranking for Knowledge Base Question Answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 6032–6043.

Yih, W.-t.; Richardson, M.; Meek, C.; Chang, M.-W.; and Suh, J. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 201–206.

Yu, D.; Zhang, S.; Ng, P.; Zhu, H.; Li, A. H.; Wang, J.; Hu, Y.; Wang, W. Y.; Wang, Z.; and Xiang, B. 2022. DecAF: Joint Decoding of Answers and Logical Forms for Question Answering over Knowledge Bases. In *The Eleventh International Conference on Learning Representations*.

Zhou, D.; Schärli, N.; Hou, L.; Wei, J.; Scales, N.; Wang, X.; Schuurmans, D.; Cui, C.; Bousquet, O.; Le, Q. V.; et al. 2022. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *The Eleventh International Conference on Learning Representations*.