Aligning Complex Knowledge Graph Question Answering as Knowledge-Aware Constrained Code Generation

Prerna Agarwal[*1,2], **Nishant Kumar**[1], **Srikanta Bedathur**[1]
[1]Indian Institute of Technology Delhi, India
[2]IBM Research India
{prerna.agarwal@scai, srikanta@cse}.iitd.ac.in, nishant.1906.kumar@gmail.com

Abstract

Generating executable logical forms (LF) using Large Language Models (LLMs) in a few-shot setting for Knowledge Graph Question Answering (KGQA) is becoming popular. However, their performance is still limited due to very little exposure to the LF during pre-training of LLMs, resulting in many syntactically incorrect LF generation. If the LF generation task can be transformed to a more familiar task for the LLM, it can potentially reduce the syntax errors and elevate the generation quality. On the other hand, there exist specialized LLMs trained/fine-tuned on code in many programming languages. They can be leveraged to generate the LF as *step-wise constrained code expression generation* using modular functions in the LF. Based on this insight, we propose CodeAlignKGQA[1]: a framework that aligns the LF generation as code generation that incorporates LF-specific constraints. We extract the question-specific subgraph information to enable Knowledge-Aware code generation. We additionally introduce a dynamic self-code-correction mechanism, to be applied as required. Our extensive experiments on Complex KGQA benchmarks such as KQA Pro demonstrate the effectiveness of our approach. CodeAlignKGQA surpasses all few-shot baselines on KQA Pro by 21%, achieving a new state-of-the-art.

1 Introduction

With recent advancements in LLMs, the research has moved towards leveraging the reasoning capability of LLMs for Knowledge Graph Question Answering (KGQA) in a few-shot setting (Gu et al., 2023; Li et al., 2023b). KGQA aims at answering a natural language question using a Knowledge Graph (KG) by producing a logical form (LF) such as SPARQL, S-Expression, etc. that is executed on the KG to retrieve the answer(s) (Shu et al., 2022; Nie et al., 2022; Neelam et al., 2022). Due to the recent surge in the complexity of questions that users pose to the systems, complex KGQA has particularly gained attention (Gu et al., 2021a; Cao et al., 2022; Huang et al., 2023).

The training-free nature of the in-context learning (ICL) paradigm is becoming popular with various state-of-the-art LLMs such as GPT-3 (Brown et al., 2020) variants to generate the LF. However, the performance of the existing ICL methods (Li et al., 2023a,b; Gu et al., 2023) is limited due to very little exposure to the LF during pre-training of LLMs, which results in high syntax errors (see Figure 1). On the other hand, if the LF generation task can be transformed to a more familiar task that a LLM has good exposure to, the generation capability can be further improved, potentially reducing the syntax errors (Wang et al., 2023; Mishra et al., 2023; Nie et al., 2024). Moreover, complex questions require joint compositional and numerical reasoning on KG to answer them. This demands the step-by-step decomposition of the reasoning process, providing transparency (Wei et al., 2023). Hence, the LF generation task needs to be transformed in such a way that supports solving the question in a step-wise manner.

For the KGQA task, KoPL (Knowledge Oriented Programming Language) (Cao et al., 2022), a symbolic LF has been introduced recently that defines various modular functions (operations) on KG catering to various complexities. This makes it suitable for complex compositional and numerical reasoning while being interpretable. There exists specialized Code LLMs (Rozière et al., 2023; Chen et al., 2021) that are trained on a wide variety of programming languages and have shown excellent logical reasoning capabilities. They support tasks such as code generation, code completion, etc. The code generation task composes different modular

[*]P. Agarwal is an employee at IBM Research. This work was carried out as part of PhD research at IIT Delhi. Email id: preragar@in.ibm.com

[1]Codebase and Data: https://github.com/data-iitd/CodeAlignKGQA

Proceedings of the 31st International Conference on Computational Linguistics, pages 3952–3978
January 19–24, 2025.

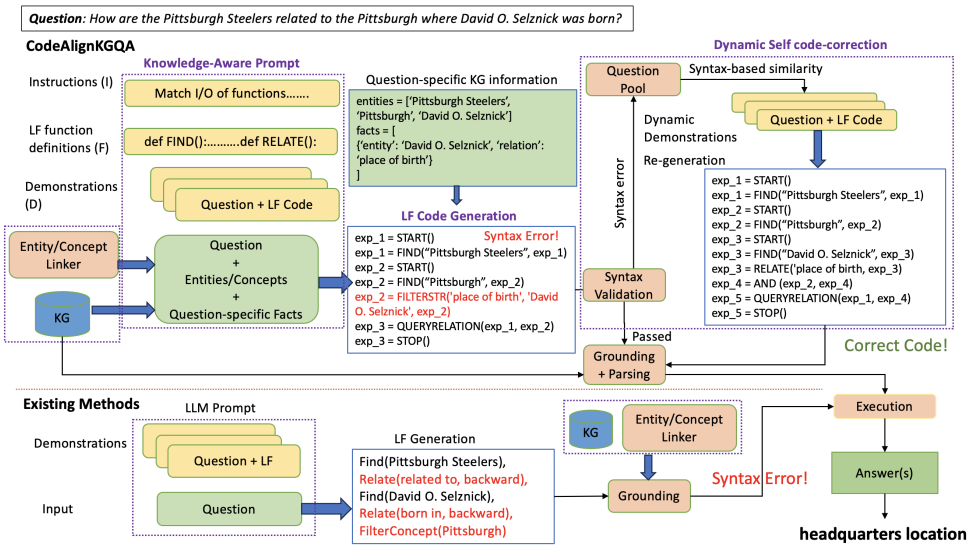


Figure 1: Comparison between the proposed CodeAlignKGQA (aligns LF generation as Knowledge-Aware Code Generation task with dynamic self code-correction mechanism) and existing few-shot LLM based methods.

functions in a step-by-step manner to solve a task.

Therefore, due to the inherent similarity between KoPL and programming languages, the task of LF generation can be transformed/aligned as a code-generation task. Specifically, in this paper, we leverage the Code LLMs to generate the KoPL as step-wise constrained and composable code expressions with modular functions supported by KoPL. We choose Python as the programming language for code-generation due to its simplicity and wide acceptance. Figure 1 shows the alignment process.

Another challenge that most of the existing methods face is, that they use LLMs to generate a draft of LF without knowing the KG structure (Nie et al., 2024; Li et al., 2023a,b; Agarwal et al., 2024b). However, the LLM may not have seen the KG on which the complex question is posed. This results in an error when the generated LF is executed directly, demanding for high amount of KG grounding. On the other hand, the need for KG grounding can be significantly reduced if the LLM is aware of the KG structure (Mallen et al., 2023; Xiong et al., 2024). Therefore, we propose a novel non-parametric method to extract the question-specific subgraph information and provide it to the LLM for Knowledge-Aware code generation (see green blocks in CodeAlignKGQA in Figure 1).

For few-shot ICL, we provide the instructions, and function definitions along with few-shot examples for LF generation. However, LLMs do not guarantee to always follow the instructions provided (Mu et al., 2023). Hence, this can still result in incorrect program generation and syntax errors. To mitigate this, we provide a self-code-correction mechanism at run-time that dynamically retrieves demonstrations with similar program syntax from a pool of questions to revise its knowledge and re-generate the code with possibly correct syntax (see Dynamic Self code-correction block in Figure 1).

Hence, we make the following contributions:

1. We propose CodeAlignKGQA that aligns the LF generation as a Knowledge-Aware Code Generation task. We use KoPL and Python as LF and programming language, respectively.
2. We propose a novel non-parametric method to extract question-specific subgraph information (facts) for Knowledge-Aware code generation.
3. We propose a novel dynamic self-code-correction mechanism that retrieves demonstrations with similar program syntax for code re-generation.
4. We experiment with different Code LLMs and heterogeneous large KGs to demonstrate the flexibility and robustness of our approach.

Experiments demonstrates that CodeAlignKGQA outperforms all other few-shot and even most of the state-of-the-art (SOTA) fully-supervised approaches by a significant margin.

2 Related Work

2.1 Few-Shot KGQA methods

Recent advancements in the development of various pre-trained LLMs, such as T5 (Raffel et al., 2023) and Codex (Chen et al., 2021), have shown SOTA performance on various downstream tasks. Some of the LLM-based frameworks have been

proposed for KGQA tasks as well. RnG-KBQA (Ye et al., 2022), TIARA (Shu et al., 2022), KB_BINDER (Li et al., 2023a) and similar works convert natural language questions to S-Expression with constrained decoding using LLMs. Pangu (Gu et al., 2023), BYOKG (Agarwal et al., 2024a) leverage an LLM-based symbolic agent to comprehend KG information through exploration. FlexKBQA (Li et al., 2023b) introduces an execution-guided self-training method and surpasses all other few-shot methods. DecAF (Yu et al., 2023) generates LF and the final answer using LLMs. ChatKBQA (Luo et al., 2024) and (Niu et al., 2023) fine-tunes LLMs for LF generation. These techniques generating SPARQL/S-expression suffer from various limitations (as discussed in Section 1).

2.2 Complex QA Reasoning using LLMs

Recent works have demonstrated that step-by-step Chain-of-Thought (CoT) decomposition enhances the reasoning capability of LLMs for Complex QA tasks (Wei et al., 2023; Zhou et al., 2023; Niu et al., 2023). SymKGQA (Agarwal et al., 2024b) generates KoPL LF using Chain-of-Symbol (CoS) prompting, however, rely completely on underlying LLM to infer KG structure and hence, suffers huge syntax errors. InteractiveKBQA (Xiong et al., 2024) generate LF by interacting with KG but their core step of interaction i.e., deconstruction of question into sub-query triples is manual and hence, not scalable. Few works have attempted the non-trivial transformation of different tasks into code generation (Wang et al., 2023). KB-Coder (Nie et al., 2024) performs code-style S-expression generation. However, suffers from the following limitations, they: (1) require the whole test set upfront to retrieve similar relation which is not available in real-world scenarios. (2) do not provide constraints of the LF, relevant instructions and knowledge in their prompt that further degrades their performance. (3) do not handle complex surface mentions of the relations i.e., backward and composite relations. (4) use a very expensive way of KG grounding i.e., takes the Cartesian product of p entities to be linked with q relations during matching. Our proposal i.e., CodeAlignKGQA addresses these major limitations in a novel way.

3 CodeAlignKGQA Framework

We introduce the details of CodeAlignKGQA framework shown in Figure 1, that consists of three stages: (1) Knowledge-Aware Prompt Generation; (2) LF Code Generation: using Few-Shot ICL; and (3) Dynamic self code-correction.

3.1 Problem Statement

The goal of the CodeAlignKGQA framework is three-fold: (1) Generate the Knowledge-Aware Prompt (Y), (2) Generate step-by-step KoPL code expression (P) for a given natural language question (Q) using a pre-trained LLM (M) by providing Y, and, (3) Execute P, on KG K, and dynamically self-correct the code (if required), to obtain the final answer(s). Here, $K \subset E \times R \times (E \cup L \cup C)$, where C is the set of concepts[2], E is the set of entities, L is the set of attributes (literals) and R is the set of binary relations.

3.2 Knowledge-Aware Prompt Generation

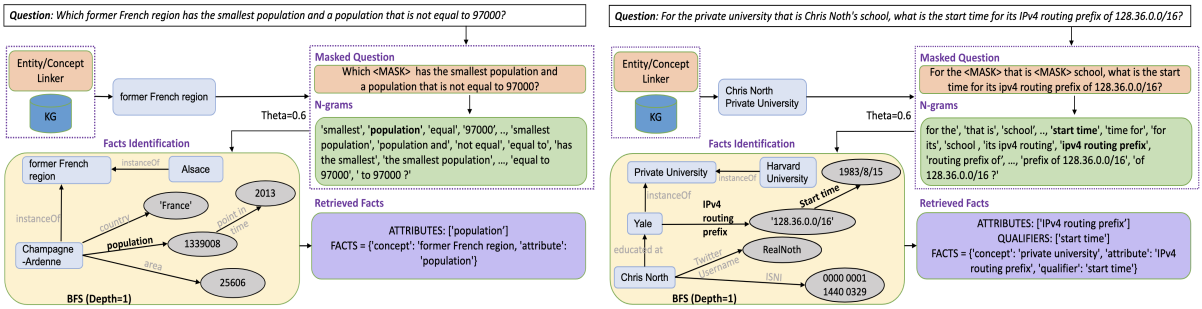
To generate step-by-step KoPL code expression, we first create a prompt $Y = (I, F, D)$ that consists of a set of Instructions I, KoPL Python Function Definitions F, and Few-Shot Demonstrations D (also shown in Figure 1) for ICL.

Instructions I: It describes the code-generation task and outlines the generation rules. A few instructions are shown below. Refer to Appendix A.1 for a full set of instructions.

```
'''
– Please use the python functions defined below to
    generate the expression corresponding to the
    question step by step.
– Make sure to validate the datatype of the input
    parameter before selecting a function using
    assert statements provided in each function.
'''
```

KoPL Python Function Definitions F: We provide the information of all 27 KoPL functions as Python function definitions for the Code LLM to understand and use them only in the generation process. Refer to Appendix A.1 to find the details of all 27 KoPL functions. Python equivalent functions of KoPL are created by ensuring the same (1) function name; and (2) description containing parameters (functional and textual inputs) and outputs as return type are used. The functional inputs come from the previous code step (expression) whereas the textual inputs come from the question. Assert statements are added to guide the Code LLM to verify the datatype of the functional input before choosing it during generation. The sample function information format is shown below.

[2]A concept is an abstraction of a set of entities



(a) Example with Numerical Constraints (b) Example with KG concepts, attributes and qualifiers

Figure 2: Question-specific Facts Extraction for Complex Questions

```
def RELATE(relation: str , expression : entities ) ->
    tuple [ entities , facts ]:
    '''
    Return entities that have the input relation with
        the given entity in the knowledge graph
    Parameters:
        relation (str): input relation name
        expression (entities): functional input from
            the expression of type entities
    Returns:
        expression (tuple [ entities , facts ]):
            evaluated expression of type
            tuple [ entities , facts ]
    '''
    assert isinstance (expression , entities ) == True
    return 'RELATE({}, {})'.format( relation ,
        expression )
```

Demonstration Selection D: The demonstration selection is a crucial part of generation. The demonstration questions should be selected such that their KoPL program steps cover all functions. We manually select $n = 10$ questions from the train set keeping in mind that their KoPL program steps, when annotated, would cover a maximum number of functions with minimal overlap to obtain a diverse yet complete set D. For all the test questions of a dataset, D will remain fixed. Refer to Appendix A.1 for the n manually chosen set D. Note that the value of n is also dependent on the prompt length supported by an LLM.

Question-specific subgraph information (Facts): Popular subgraph extraction methods (Sun et al., 2019; Zhang et al., 2022) are parametric and require a large amount of training data for learning. The non-parametric subgraph extraction methods (Das et al., 2022) are majorly dependent on kNN samples for correct retrieval of relation paths. However, for complex questions, this assumption may not hold. To illustrate one such scenario, consider the example shown in Figure 2a. Here, the answer is not present directly at k-hops, rather, it has to be computed based on certain numerical constraints ('smallest', 'not equal to'). This is because, these methods are not designed to handle KG complexities such as concepts, attributes, and qualifiers. More details are provided in Appendix A.2.

On the other hand, Q can provide the right cues of the KG elements involved that also eliminates the dependency on kNN samples. Therefore, we use the parts of the question text as the seed to identify the relevant KG elements (relations, attributes, qualifiers). Hence, we propose a Breadth First Search (BFS) based non-parametric method that use Q to identify the KG elements and form the subgraph. We demonstrate the facts extraction process using one of the complex question scenarios shown in Figure 2b using the below steps:

1. *Entity and Concept Identification*: The set of entities E' and set of concepts C' for Q are first identified. They can be annotated using any off-the-shelf entity linker such as (Gu et al., 2023; Mohammed et al., 2018) popularly used in KGQA, Named Entity Recognition (NER) methods, POS tags, or can be explicitly provided. We use NER as the entity linker. As shown in Figure 2b, $E' = \{Chris\ North\}$ and $C' = \{private\ university\}$.

2. *Text Extraction*: We then mask E' and C' in Q (see Masked Question in Figure 2b). Subsequently, we derive n-grams from this masked question, utilizing *n = [1, 2, 3,.., p]* to encompass potential facts within the KG that may pertain to the context of Q (see N-grams in Figure 2b). Here, p is the maximum element length in the KG. We further refine the generated n-grams by eliminating the stop-words using nltk[3] python library.

3. *Facts Identification*: We perform BFS in both the directions i.e., forward and backward on the KG to retrieve the facts. The initiation of the BFS involves designating E' and C' as the starting nodes (shown in Figure 2b). Relations or attributes associ-

[3] https://www.nltk.org/

ated with the starting node are then chosen only if the cosine similarity of the BERT embeddings with any n-gram obtained exceeds a specified threshold θ. BERT embeddings ensures to capture the semantic similarity even when their linguistics appear differently. Qualifiers are selected using the same approach, if they exist.

Our approach provides additional novel advantages as compared to existing subgraph extraction methods when: (1) backward relation inference is required; (2) the surface mention of the relation is a composition of a set of relations in the KG. The list of E', C', and facts thus obtained for each demonstration $d \in D$ is provided as Python variables in the below template.

```
question = ``For the private university that is Chris
    Noth's school, what is the start time for its
    IPv4 routing prefix of 128.36.0.0/16?"
entities = ['Chris North']
concepts = [' private university ']
facts = [ {'concept': ' private university ',
    ' attribute ': 'IPv4 routing prefix ', ' qualifie r':
    ' start time'}]
Make sure to validate the datatype of the input
    parameter before selecting a function using
    assert statements. Given the facts provided, the
    steps to solve this question are:
expression_1 = START()
```

After assembling the prompt Y using I, F, and D, the LLM is now supposed to generate the step-by-step KoPL code expression for each test question conditioned on Y and Q. *The extracted facts thus capture the KG constraints among entities, concepts, relations, etc. in a unified way, making the prompt knowledge-aware.*

3.3 Dynamic self code-correction

The KoPL code steps thus generated, can still have syntax errors due to hallucinations in LLMs. Therefore, we design the below self code-correction mechanism to mitigate this.

Syntax Validation: The syntax of the generated program is first validated as follows: (1) The input-output datatype match is validated for each consecutive pair of KoPL function sequences. (2) The parameter datatype match is validated for each input of every KoPL function in the sequence.

For the cases where the syntax validation fails, we perform dynamic self code-correction by selecting the dynamic demonstrations (described below).

Dynamic demonstration selection (D'): The generated KoPL code steps provide a sketch of the reasoning steps that an LLM is choosing to solve Q. Hence, a new demonstration set D' is obtained by retrieving the question-program pairs whose KoPL sequence syntax is similar to the generated program steps. This will dynamically revise the context of LLM for Q and increase the likelihood of correct re-generation of the code expression with possibly correct syntax.

To obtain D', we first create a function coverage-based pool of N (100) questions from the train set inspired from "bottom-up matching" (Drozdov et al., 2022), as described below:

1. *Pool Creation (N)*: Since, `Find` and `FindAll` functions appear with almost all other remaining 25 functions, we randomly select $k = N/25$ questions that specifically demonstrate each of the 25 functions (except `Find` and `FindAll`). This ensures diversity and coverage of all functions. This pool will remain constant for all questions of a dataset.

2. *Obtaining D' (n)*: We build a retriever using this pool to retrieve top $n = 10$ nearest neighbor questions having similar sequence syntax of functions to the generated sequence. We serialize the tree-structured program steps by post-order traversal into a sequence of steps and encode it using BERT to create the vector-based index using ChromaDB[4]. This technique would encode both sequence and tree-based dependency features (Zhao et al., 2023). We chose ChromaDB due to its superior performance ($5-7\%$) as compared to popular BM25-based retrievers (Robertson and Zaragoza, 2009) after experimentation.

The code is then re-generated after replacing D with D' in Y. Existing works retrieve dynamic demonstrations based only on the question similarity (results in $\downarrow 8\%$ performance). However, similar questions do not necessarily possess a similar program structure. Retrieving D' with similar KoPL sequence will provide syntax information more explicitly and elevate the quality of generation.

3.4 KG Grounding and Execution

The KoPL code expression thus generated by LLM can still require grounding of textual inputs due to hallucinations. Therefore, based on the datatype of the input (entities, relations, attributes, etc.), we ground the generated KoPL as follows:

We retrieve top-10 similar KG element to the generated input based on its datatype. We then ask the LLM to choose the most relevant element among the top-10 given question Q as the context.

[4] https://docs.trychroma.com/

Dataset	KoPL	Train	Val	Test
KQA Pro	✓	94,376	11,797	11,797
MetaQA 1-hop	✓	96,106	9,992	9,947
MetaQA 2-hop	✓	118,948	14,872	14,872
MetaQA 3-hop	✓	114,196	14,274	14,274
WebQSP	×	2,998	100	1,639

Table 1: Statistics of the Datasets used

This makes the KG grounding process question-aware. Refer to Appendix A.3 for more details on the grounding process and the prompt used.

Parsing & Execution: Once the generated code undergoes grounding, the code is parsed based on the expression dependency to convert it into its standard KoPL format. The KoPL program steps are then executed on the KG as per the default KoPL executor provided by (Cao et al., 2022).

4 Experiments

Our experiments answers the following research questions: (1) How does CodeAlignKGQA compare against existing KGQA frameworks in the few-shot setting? (2) How does the performance of CodeAlignKGQA vary with different Code LLMs? (3) What are the contributions of each stage of CodeAlignKGQA? (4) How much gain is observed with the proposed alignment strategy?

4.1 Datasets

We experiment with 3 datasets based on 2 largest KGs i.e., Wikidata and Freebase having different: (a) question complexities; (b) KG domain; and (c) number of inference hops required. The statistics of these datasets are shown in Table 1.

1. **KQA Pro** (Cao et al., 2022): This recent dataset contains much **harder and complex QA pairs** with numerical quantities, concepts, and entities for multi-hop compositional and numerical reasoning unlike other datasets (Gu et al., 2021b; Dutt et al., 2023). It contains questions that require upto 10-hops answerable through Wikidata KG.

2. **MetaQA** (Zhang et al., 2018): It is the largest multi-hop dataset widely used (Choi et al., 2023; Agarwal et al., 2024a,b) for **domain-specific** (movie) KGQA. It provides upto 3-hop QA pairs.

3. **WebQSP** (Yih et al., 2016): This dataset is based on Freebase KG that contains questions upto 2-hops from **Google query logs**. It exhibits more complex structures i.e., Isomorphisms (ISO-4) as compared to other datasets (Gu et al., 2021b).

4.2 Models

We experiment with below SOTA Code LLMs:

1. **CodeLlama Instruct (34B)** (Rozière et al., 2023): A code-focused LLM, built upon Llama 2. It can follow programming instructions without prior training for various programming tasks with a prompt limit of $4K$ tokens.

2. **DeepSeek-Coder Instruct (33B)** (DSC Ins.) (Guo et al., 2024): Pre-trained on a high-quality project-level code corpus to enhance code generation and infilling with prompt limit of $16K$ tokens.

3. **Gemini Pro 1.0** (Akter et al., 2023): It possesses language abilities including reasoning, answering knowledge-based questions, code generation, and following instructions. We use `gemini-pro-001` version of this model with a prompt limit of $8K$ tokens.

4. **GPT-4** (OpenAI et al., 2023): Due to the deprecation of the Codex models, we use GPT-4 which has demonstrated SOTA performance on code-related tasks. Because of the costs associated with the model, we randomly sample 5% questions from the test set of each of the three datasets and compare the performance with other models. We use `gpt-4-0125-preview` version of this model.

4.3 Baselines

We select the following 3 categories of baselines:

• **Fully Supervised**: We compare with SOTA fully supervised techniques for each dataset. It includes KVMemNet (Miller et al., 2016), EmbedKGQA (Saxena et al., 2020) for all datasets and:

1. MetaQA: SRN (Qiu et al., 2020), PullNet (Sun et al., 2019), NSM (He et al., 2021) and TransferNet (Shi et al., 2021).
2. KQA Pro: SRN, RGCN (Schlichtkrull et al., 2018), Subgraph Retrieval (Zhang et al., 2022), BART + KoPL (Cao et al., 2022), GraphQ IR (Nie et al., 2022).
3. WebQSP: DecAF (Yu et al., 2023), Subgraph Retrieval and ChatKBQA (Luo et al., 2024).

Note: Among the above - SRN, PullNet, NSM, Subgraph Retrieval and DecAF are SOTA subgraph extraction based (SE) baselines.

• **Few-Shot**[5]: We compare with few-shot techniques that have shown SOTA for each dataset:

[5]Benchmarking InteractiveKBQA is not feasible due to its interactive nature with human for action correction whereas CodeAlignKGQA works in a non-interactive setting.

Model	Overall	Multi-hop	Qualifier	Comparison	Logical	Count	Verify	Zero-Shot
CodeLlama Ins	70.56	65.68	44.16	89.68	53.49	46.66	85.42	76.03
DeepSeek-Coder Ins.	63.94	62.22	41.45	85.87	48.98	43.85	76.92	75.41
Gemini Pro 1.0	**72.70**	**67.98**	**47.40**	**92.47**	**58.59**	**50.08**	**87.77**	**77.40**
SymKGQA	51.11	44.29	32.50	49.02	37.28	36.95	54.32	56.99

Table 2: Category-wise performance of different models on KQA Pro dataset and comparison with SOTA SymKGQA in a few-shot setting.

Method	Models	Hits@1	SER%
Fully Supervised	KVMemNet	6.90	-
	EmbedKGQA	20.27	-
	SRN (SE)	11.84	-
	RGCN	29.12	-
	Subgraph Retrieval (SE)	22.82	-
	BART + KoPL	83.28	8.2
	GraphQ IR	79.13	-
Few-Shot (100 Shots)	FlexKBQA	42.68	-
	LLM-ICL (SPARQL)	27.75	-
	SymKGQA (KoPL)	51.10	29.2
CodeAlignKGQA Few-Shot (100 Shots)	CodeLlama Ins.	<u>70.56</u>	**2.84**
	DSC Ins.	63.94	7.50
	Gemini Pro 1.0	**72.70**	4.95

Table 3: CodeAlignKGQA Results for KQA Pro* on dev set (*Bold and underline denote best and second best performance **among few-shot setting**)

1. KQA Pro: FlexKBQA (Li et al., 2023b), LLM-ICL (SPARQL)[6] and SymKGQA (Agarwal et al., 2024b).
2. WebQSP: FlexKBQA, KB_BINDER (Li et al., 2023a), KB-Coder (Nie et al., 2024) and Pangu (Gu et al., 2023) all generating S-Expression and SymKGQA that generates KoPL. We use KB_BINDER (1) and KB-Coder (1) setting for fair comparison.
3. MetaQA: No few-shot baselines has benchmarked MetaQA except SymKGQA.

Refer to Appendix A.7 for more details on the techniques adopted by each of the baseline.

4.4 Implementation and Hyper-parameters

PyTorch Python framework is used to implement CodeAlignKGQA with greedy LLM decoding (for reproducibility). `all-distilroberta-v1` BERT model is used for dynamic demonstration selection and facts extraction. Hits@1 for KQA Pro[7] and MetaQA, and F1 score for WebQSP are the evaluation metrics used. We additionally report the Syntax Error Rate (SER) metric. It is calculated as the percentage of the questions for which the generated code does not conform to the syntax of the KoPL. Appendix 5.8 provides the compute resources utilized.

[6]Alternate to Pangu for evaluation (as used in FlexKBQA)

[7]Each question in KQA Pro has only one answer, hence, Hits@1 and Accuracy values will be the same.

5 Results and Analysis

The results on each dataset are discussed below in detail. Qualitative analysis examples are provided in Appendix A.6.

5.1 Results: KQA Pro

The results of CodeAlignKGQA compared with baselines are shown in Table 3. Detailed category-wise performance is provided in Table 2. As shown, CodeAlignKGQA with Gemini Pro 1.0 model outperforms all few-shot baselines and *achieves a new SOTA in the few-shot setting for KQA Pro*. It beats FlexKBQA and LLM-ICL (SPARQL) by 30% and 45% respectively (both generate SPARQL using GPT-3 variants). It also beats SymKGQA by 21.5%. The performance with DeepSeek-Coder Instruct is inferior to CodeLlama Instruct and Gemini Pro 1.0 by 7% and 9% respectively. Programs generated without providing KG Facts have lower syntactic accuracy and contain less relevant textual input. Hence, the number of questions for which atleast 1 function requires KG grounding increase by 5.2% when KG facts are not provided. We compare the SER of CodeAlignKGQA only with the baselines where KoPL is used to demonstrate the effectiveness of our alignment approach keeping the LF as same. The SER of CodeAlignKGQA is remarkably less than all other baselines. It outperforms all fully supervised baselines except BART + KoPL and GraphQ IR. For the sampled test set (shown in Table 6), its performance with GPT-4 beats other LLMs and is nearly the same as Gemini Pro 1.0.

5.2 Results: WebQSP

The results of CodeAlignKGQA on WebQSP are shown in Table 5. *CodeAlignKGQA achieves a new SOTA for WebQSP* outperforming all few-shot and fully-supervised baselines. It beats FlexKBQA

Method	Models	Hits@1			SER%
		1-hop	2-hop	3-hop	
Fully Supervised	KVMemNet	96.2	82.7	48.9	-
	EmbedKGQA	97.5	98.8	94.8	-
	SRN (SE)	97.0	95.1	75.2	-
	PullNet (SE)	97.0	99.9	91.4	-
	NSM (SE)	97.2	99.9	98.9	-
	TransferNet	97.5	100	100	-
Few-Shot (100 Shots)	SymKGQA (KoPL)	99.1	99.7	99.7	0.2
CodeAlignKGQA Few-Shot (100 Shots)	CodeLlama Ins.	**99.6**	**99.8**	99.7	0.0
	DSC Ins.	96.5	98.0	94.9	0.0
	Gemini Pro 1.0	99.2	99.7	**99.8**	**0.0**

Table 4: CodeAlignKGQA Results on MetaQA*

Method	Models	F1	SER%
Fully Supervised	Program Transfer	74.6	-
	EmbedKGQA	66.6	-
	Subgraph Retrieval (SE)	66.7	-
	DecAF (SE)	78.8	-
	ChatKBQA	79.8	-
Few-Shot (100 Shots)	FlexKBQA	60.6	-
	KB_BINDER (1)	52.5	3.9
	KB-Coder (1)	55.7	1.9
	Pangu	54.5	-
	SymKGQA (KoPL)	70.6	1.7
CodeAlignKGQA Few-Shot (100 Shots)	CodeLlama Ins.	**81.6**	**0.0**
	DSC Ins.	79.2	0.0
	Gemini Pro 1.0	81.5	0.0

Table 5: CodeAlignKGQA Results on WebQSP*

Models	KQA Pro	MetaQA	WebQSP
CodeLlama Ins.	65.70	**100.00**	**76.54**
DSC Ins.	60.78	96.97	72.84
Gemini Pro 1.0	68.93	**100.00**	72.84
GPT-4	**74.70**	97.98	75.31

Table 6: Comparison with GPT-4 (Sampled Test Set)

Dataset	Precision	Recall	θ
KQA Pro	0.964	0.758	0.8
MetaQA (combined)	0.984	0.613	0.5
WebQSP	0.823	0.369	0.5

Table 7: Question-specific Facts extraction performance

and Pangu by 21% and 27% respectively. It also beats SymKGQA by 10% and KB-Coder (1) by 26%. It even beats the SOTA ChatKBQA method. The performance with DeepSeek-Coder Instruct is inferior to CodeLlama Instruct and Gemini Pro 1.0 by 2%. It achieves no SER whereas the baselines observe approx. 2% SER. The number of questions for which atleast 1 function requires KG grounding increase by 4.62% when KG facts are not provided. For the sampled test set (see Table 6), the performance with GPT-4 is higher than other LLMs except CodeLlama Instruct.

5.3 Results: MetaQA

The results of CodeAlignKGQA on MetaQA are shown in Table 4. *CodeAlignKGQA achieves a new SOTA for 1-hop*. For 2-hop and 3-hop, the performance is nearly the same as SOTA fully supervised baselines. The performance with DeepSeek-Coder Instruct is inferior to CodeLlama Instruct and Gemini Pro 1.0 by 3%. It achieves no SER for this dataset. The number of questions for which atleast 1 function requires KG grounding increase by 3.66% when KG facts are not provided. For the sampled test set (shown in Table 6), the performance with GPT-4 is slightly less than CodeLlama Instruct and Gemini Pro 1.0.

5.4 Results: Facts Extraction

The precision and recall of Question-specific subgraph (facts) extraction with similarity threshold θ for each dataset are reported in Table 7. We experiment with different θ and choose the one with the highest precision. The θ value for MetaQA and WebQSP is lower due to high dissimilarity between the surface names in the question and the KG. A detailed discussion along with qualitative examples is provided in Appendix A.2 (Table 10).

5.5 Observations

Overall, we observe the following:

1. GPT-4 has the better reasoning ability on complex questions as compared to other models, whereas, all models possess similar ability to handle simpler questions.

2. The performance gap with fully-supervised baselines is attributed to the use of a large amount of annotated LF data during learning, whereas, CodeAlignKGQA addresses a more challenging

Config	KQA Pro	MetaQA	WebQSP
- Code Correction	55.16 (↓ 13.9%)	96.83 (↓ 1.75%)	68.12 (↓ 12.6%)
- KG Grounding	32.84 (↓ 36.2%)	85.54 (↓ 13%)	14.63 (↓ 66.13%)

Table 8: Ablation Study

few-shot setting.

3. CodeAlignKGQA outperforming KB-Coder by 26% signifies the importance of each of the proposed module in LF code generation that KB-Coder lacks.

4. CodeAlignKGQA is applicable with other popular LFs such as SPARQL by transforming them into KoPL[8].

5.6 Ablation Study

Table 8 shows the performance when the following modules are eliminated to obtain the answers:

1. **Dynamic self code-correction**: Here, programs are generated using only the manually selected demonstrations while obtaining the answers. On this elimination, the performance dropped by 13.9% for KQA Pro dataset and 12.6% for WebQSP. However, not much variation is observed in the generation and hence, in performance for MetaQA due to the simplicity of the programs for this dataset.

2. **KG Grounding**: Here, the generated programs are directly executed to obtain the answers. On this elimination, the performance dropped by 36.2% for KQA Pro dataset and a whooping 66.13% for WebQSP. For MetaQA, the performance dropped by 13%. Hence, removal of the KG Grounding shows a huge drop in the performance of the model. However, KG Grounding is able to ground most of the generated function inputs.

The performance drop quantifies the contribution of each of the eliminated module.

5.7 Error Analysis

We analyze the following sources of errors in CodeAlignKGQA (avg. across LLMs):

• *Entity Linking*: Fraction of questions that generated the correct program, but did not retrieve the correct entities/concepts is observed only in KQA Pro, for 0.69% of questions. No such issues are observed in MetaQA and WebQSP.

• *Syntax Error*: Fraction of questions where mismatch in Input-Output types of the generated steps are observed. Errors in program execution due to mismatch in input-output types of the steps generated are seen only for KQA Pro, for 5% of questions. No such errors are observed in MetaQA and WebQSP.

• *Incorrect KG Grounding by Resolver:* The wrong answers obtained due to incorrect grounding of KG components (other than entity and concepts) by the QUACK resolver is observed in KQA Pro, for 17.33% and WebQSP, for 22.52% of questions. No such issue is observed for MetaQA.

5.8 Inference Latency

We make API calls to generate KoPL programs with average latency of 5s/call that runs on CPU. We use NVIDIA V100 GPU with 32 GB RAM for execution and resolution. On average it consumes following GPU hours:

- 3 Hrs: Execution and Resolution for 11,797 test questions of KQA Pro.
- 2 Hrs: Execution and Resolution for 1,639 test questions of WebQSP.
- 3 Hrs: Execution and Resolution for 13,031 test questions of MetaQA (average on all hops).

6 Conclusion

We propose CodeAlignKGQA, which generates the LF as step-wise constrained code in a few-shot setting. To the best of our knowledge, it is the first one to generate KoPL LF by transforming it as a code-generation task for complex KGQA. We provide a method for Knowledge-Aware generation of LLM prompt for the aligned task as python code expression. We then perform LF code generation using few-shot ICL and perform self-code-correction at run-time using the dynamic demonstration selection technique. Our extensive experiments show that CodeAlignKGQA surpasses all few-shot and most of the fully supervised SOTA baselines, setting a new benchmark for various Complex KGQA datasets. Our approach being independent of underlying LLM knowledge, makes it applicable for any other symbolic LF while being interpretable.

[8] https://github.com/Flitternie/GraphQ_Trans

7 Limitations

Despite the strong reasoning capabilities of SOTA Code LLMs, obtaining step-wise code expressions is still prone to errors, which can impact the performance of CodeAlignKGQA, as discussed in Section 5.7. Most of the errors stem from a mismatch between the input-output type of the subsequent generated steps. Fortunately, these errors are generally interpretable and can be rectified with human intervention.

8 Risks

Our work does not have any obvious risks that we are aware of.

Acknowledgements

Srikanta Bedathur was partially supported by a DS Chair of AI fellowship and by a IITD-IBM AIHN collaborative project.

References

Dhruv Agarwal, Rajarshi Das, Sopan Khosla, and Rashmi Gangadharaiah. 2024a. Bring your own KG: Self-supervised program synthesis for zero-shot KGQA. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 896–919, Mexico City, Mexico. Association for Computational Linguistics.

Prerna Agarwal, Nishant Kumar, and Srikanta Bedathur. 2024b. SymKGQA: Few-shot knowledge graph question answering via symbolic program generation and execution. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10119–10140, Bangkok, Thailand. Association for Computational Linguistics.

Syeda Nahida Akter, Zichun Yu, Aashiq Muhamed, Tianyue Ou, Alex Bäuerle, Ángel Alexander Cabrera, Krish Dholakia, Chenyan Xiong, and Graham Neubig. 2023. An in-depth look at gemini's language abilities.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Shulin Cao, Jiaxin Shi, Liangming Pan, Lunyiu Nie, Yutong Xiang, Lei Hou, Juanzi Li, Bin He, and Hanwang Zhang. 2022. KQA pro: A dataset with explicit compositional programs for complex question answering over knowledge base. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6101–6119, Dublin, Ireland. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Hyeong Kyu Choi, Seunghun Lee, Jaewon Chu, and Hyunwoo J Kim. 2023. Nutrea: Neural tree search for context-guided multi-hop kgqa. In *Advances in Neural Information Processing Systems*, volume 36, pages 35954–35965. Curran Associates, Inc.

Rajarshi Das, Ameya Godbole, Ankita Naik, Elliot Tower, Robin Jia, Manzil Zaheer, Hannaneh Hajishirzi, and Andrew McCallum. 2022. Knowledge base question answering by case-based reasoning over subgraphs. In *ICML*.

Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. 2022. Compositional semantic parsing with large language models.

Ritam Dutt, Sopan Khosla, Vinayshekhar Bannihatti Kumar, and Rashmi Gangadharaiah. 2023. GrailQA++: A challenging zero-shot benchmark for knowledge base question answering. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 897–909, Nusa Dua, Bali. Association for Computational Linguistics.

Yu Gu, Xiang Deng, and Yu Su. 2023. Don't generate, discriminate: A proposal for grounding language models to real-world environments. In *Proceedings*

of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 4928–4949, Toronto, Canada. Association for Computational Linguistics.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. 2021a. Beyond i.i.d.: Three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, WWW '21, page 3477–3488, New York, NY, USA. Association for Computing Machinery.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. 2021b. Beyond iid: three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, pages 3477–3488. ACM.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Gaole He, Yunshi Lan, Jing Jiang, Wayne Xin Zhao, and Ji-Rong Wen. 2021. Improving multi-hop knowledge base question answering by learning intermediate supervision signals. WSDM '21, page 553–561, New York, NY, USA. Association for Computing Machinery.

Xiang Huang, Sitao Cheng, Yuheng Bao, Shanshan Huang, and Yuzhong Qu. 2023. MarkQA: A large scale KBQA dataset with numerical reasoning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10241–10259, Singapore. Association for Computational Linguistics.

Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. 2023a. 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)*, pages 6966–6980, Toronto, Canada. Association for Computational Linguistics.

Zhenyu Li, Sunqi Fan, Yu Gu, Xiuxing Li, Zhichao Duan, Bowen Dong, Ning Liu, and Jianyong Wang. 2023b. Flexkbqa: A flexible llm-powered framework for few-shot knowledge base question answering.

Haoran Luo, Haihong E, Zichen Tang, Shiyao Peng, Yikai Guo, Wentai Zhang, Chenghao Ma, Guanting Dong, Meina Song, and Wei Lin. 2023. Chatkbqa: A generate-then-retrieve framework for knowledge base question answering with fine-tuned large language models.

Haoran Luo, Haihong E, Zichen Tang, Shiyao Peng, Yikai Guo, Wentai Zhang, Chenghao Ma, Guanting Dong, Meina Song, Wei Lin, Yifan Zhu, and Anh Tuan Luu. 2024. ChatKBQA: A generate-then-retrieve framework for knowledge base question answering with fine-tuned large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 2039–2056, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. 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)*, pages 9802–9822, Toronto, Canada. Association for Computational Linguistics.

Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-value memory networks for directly reading documents. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1400–1409, Austin, Texas. Association for Computational Linguistics.

Mayank Mishra, Prince Kumar, Riyaz Bhat, Rudra Murthy, Danish Contractor, and Srikanth Tamilselvam. 2023. Prompting with pseudo-code instructions. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 15178–15197, Singapore. Association for Computational Linguistics.

Salman Mohammed, Peng Shi, and Jimmy Lin. 2018. Strong baselines for simple question answering over knowledge graphs with and without neural networks. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 291–296, New Orleans, Louisiana. Association for Computational Linguistics.

Norman Mu, Sarah Chen, Zifan Wang, Sizhe Chen, David Karamardian, Lulwa Aljeraisy, Dan Hendrycks, and David Wagner. 2023. Can llms follow simple rules?

Sumit Neelam, Udit Sharma, Hima Karanam, Shajith Ikbal, Pavan Kapanipathi, Ibrahim Abdelaziz, Nandana Mihindukulasooriya, Young-Suk Lee, Santosh Srivastava, Cezar Pendus, Saswati Dana, Dinesh Garg, Achille Fokoue, G P Shrivatsa Bhargav, Dinesh Khandelwal, Srinivas Ravishankar, Sairam Gurajada, Maria Chang, Rosario Uceda-Sosa, Salim Roukos, Alexander Gray, Guilherme Lima, Ryan Riegel, Francois Luus, and L V Subramaniam. 2022. SYGMA: A system for generalizable and modular question answering over knowledge bases. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 3866–3879, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Lunyiu Nie, Shulin Cao, Jiaxin Shi, Jiuding Sun, Qi Tian, Lei Hou, Juanzi Li, and Jidong Zhai. 2022.

GraphQ IR: Unifying the semantic parsing of graph query languages with one intermediate representation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5848–5865, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zhijie Nie, Richong Zhang, Zhongyuan Wang, and Xudong Liu. 2024. Code-style in-context learning for knowledge-based question answering.

Yilin Niu, Fei Huang, Wei Liu, Jianwei Cui, Bin Wang, and Minlie Huang. 2023. Bridging the gap between synthetic and natural questions via sentence decomposition for semantic parsing. *Transactions of the Association for Computational Linguistics*, 11:367–383.

OpenAI, :, Josh Achiam, and Steven Adler. 2023. Gpt-4 technical report.

Yunqi Qiu, Yuanzhuo Wang, Xiaolong Jin, and Kun Zhang. 2020. Stepwise reasoning for multi-relation question answering over knowledge graph with weak supervision. WSDM '20, page 474–482, New York, NY, USA. Association for Computing Machinery.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the limits of transfer learning with a unified text-to-text transformer.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Apoorv Saxena, Aditay Tripathi, and Partha Talukdar. 2020. Improving multi-hop question answering over knowledge graphs using knowledge base embeddings. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4498–4507, Online. Association for Computational Linguistics.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web*, pages 593–607, Cham. Springer International Publishing.

Jiaxin Shi, Shulin Cao, Lei Hou, Juanzi Li, and Hanwang Zhang. 2021. TransferNet: An effective and transparent framework for multi-hop question answering over relation graph. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4149–4158, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yiheng Shu, Zhiwei Yu, Yuhan Li, Börje Karlsson, Tingting Ma, Yuzhong Qu, and Chin-Yew Lin. 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*, pages 8108–8121, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Haitian Sun, Tania Bedrax-Weiss, and William Cohen. 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)*, pages 2380–2390, Hong Kong, China. Association for Computational Linguistics.

Xingyao Wang, Sha Li, and Heng Ji. 2023. Code4Struct: Code generation for few-shot event structure prediction. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3640–3663, Toronto, Canada. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Guanming Xiong, Junwei Bao, and Wen Zhao. 2024. Interactive-KBQA: Multi-turn interactions for knowledge base question answering with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10561–10582, Bangkok, Thailand. Association for Computational Linguistics.

Xi Ye, Semih Yavuz, Kazuma Hashimoto, Yingbo Zhou, and Caiming Xiong. 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)*, pages 6032–6043, Dublin, Ireland. Association for Computational Linguistics.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 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)*, pages 201–206, Berlin, Germany. Association for Computational Linguistics.

Donghan Yu, Sheng Zhang, Patrick Ng, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Yiqun Hu, William Wang, Zhiguo Wang, and Bing Xiang. 2023. Decaf: Joint decoding of answers and logical forms for question answering over knowledge bases.

Jing Zhang, Xiaokang Zhang, Jifan Yu, Jian Tang, Jie Tang, Cuiping Li, and Hong Chen. 2022. Subgraph retrieval enhanced model for multi-hop knowledge base question answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5773–5784, Dublin, Ireland. Association for Computational Linguistics.

Yuyu Zhang, Hanjun Dai, Zornitsa Kozareva, Alexander J. Smola, and Le Song. 2018. Variational reasoning for question answering with knowledge graph. AAAI'18/IAAI'18/EAAI'18. AAAI Press.

Yunfei Zhao, Yihong Dong, and Ge Li. 2023. Seq2seq or seq2tree: Generating code using both paradigms via mutual learning. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, Internetware '23, page 238–248, New York, NY, USA. Association for Computing Machinery.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-most prompting enables complex reasoning in large language models.

A Appendix

A.1 CodeAlignKGQA Prompt

Full prompt for a sample test question of KQA Pro is shown in Figure 3.

A.2 Question-specific Facts Extraction

As described in Section 3.1, for the following scenarios, the existing subgraph extraction techniques will not work whereas our approach will work in a seamless manner:

- In case of disconnected graphs, when entities for a given question are disconnected with each other in the *KG*. *e.g. Does Dwayne Johnson or Kate Moss have the lower personal net worth?*
- In case where the *KG* is complex, containing entities, concepts, relations, attributes, and qualifiers, and the given question includes details regarding concepts, attributes, and qualifiers. *e.g. What is the number of episodes in TV series with Twitter username Thomas-Friends (the subscription number of this statement is 15947)?*

We experiment with different θ values and pick the one with highest precision. We also experimented by choosing the θ values with highest recall but observed that the additional spurious facts extracted due to high recall leads to deviation in the program generation and leads to lower performance than providing precise facts. On the other hand, LLM is able to generate better programs given a question and its entities/concepts information even if few relevant facts are missing. Hence, we choose to provide highly precise facts to LLM during generation.

The θ value for MetaQA and WebQSP is lower due to high dissimilarity between the relation mentions in the question and the relation name present in the KG. This leads to low cosine similarity and hence lower θ value. However, this is not the case for KQA Pro.

A.3 KG Grounding

The full set of each KG element i.e., entities, concepts, relations, attributes, qualifiers, etc. are indexed separately in ChromaDB. The KG element generated at each step is used as the input query to retrieve the top-10 closest corresponding KG element. The retrieved list is then passed to the LLM using the prompt shown below. This approach of encoding each KG element set separately provides the following advantages over indexing triples for retrieval:

For the grounding task, the grounding of each generated KG element has to be done w.r.t that element only. For example: if we consider the grounding of the generated relations, it has to be picked from the set of relations in the KG only. It will never be picked up from the set of other KG elements i.e., entities, concepts, etc. Hence, indexing each KG element set separately provides constraint retrieval.

Given that the input query will be the generated relation, on the other hand, if the triples of the KG are indexed, then the retrieved triples will not be constrained to the relations. It will contain the corresponding entities as well that could be irrelevant for a given question and hence introduce some noise in the retriever output.

We experiment with both these approaches and found out that indexing KG items separately provides provides superior performance $(8 - 10\%)$ than indexing KG triples.

```
From below relation list, select only top
    relation that is most similar to the
    relation extracted from question.

question = ''What is the higher education
    institution is headquartered in the city
    whose postal code is 20157?"
relation_list = ['headquarters location', 'work
```

```
    location', 'located in the administrative
    territorial entity', 'capital of', 'located
    in time zone', 'residence', 'capital',
    'country', 'filming location', 'place of
    birth']
relation_extracted_from_question =
    ['headquartered in']
answer =
```

A.4 CodeAlignKGQA v/s KB-Coder

Comparison of our approach CodeAlignKGQA v/s KB-Coder (Nie et al., 2024) to showcase the novelty of our approach is provided below:

1. KB-Coder rely on underlying LLM for KG structure understanding and suffer while grounding the generated code for a KG. On the other hand, CodeAlignKGQA explicitly provide facts and do not rely on underlying LLM to infer KG structure.

2. CodeAlign offers a novel question-specific subgraph extraction technique that handles (1) semantically similar (2) backward (3) composite relations that KB-Coder does not handle.

3. KB-Coder do not provide constraints and instructions related to the LF to LLM which further degrades the performance of their framework. On the other hand, CodeAlignKGQA provide LF constraints (input-output matching with explicit data types) and explicit instructions to generate python expressions.

4. KB-Coder takes the cartesian product of p entities to be linked and q relations to be matched for answer prediction which is a very expensive way of grounding. On the other hand, CodeAlignKGQA offer a simple yet effective way of grounding the LF.

5. KB-Coder use the entire test question as a query to retrieve the similar relation, and the relation with the highest similarity is provided in the prompt. This makes their method dependent on the availability of the whole test set upfront but in real-world scenarios the questions are available one at a time. Hence, their method suffers significantly while grounding KG relations. On the other hand, CodeAlignKGQA is not dependent on the test set and retrieves similar relations from the KG which is a more realistic approach.

Hence, CodeAlignKGQA not only address the above mentioned limitations but offers many advantages. This leads to a 30% gain in the performance of CodeAlignKGQA as compared to KB-Coder.

A.5 List of Numerical Operators

The list of the numerical operators supported are as follows: `[`$<, \leq, >, \geq, =, \neq$`, argmin, argmax]`

A.6 Qualitative Analysis

The examples of the generated KoPL steps that are not directly executable before dynamic code-correction and executes correctly after the code-correction for each dataset are shown in Table 9. The examples of the generated KoPL steps before and after adding Knowledge-Aware facts are shown in Table 10.

A.7 Baselines

- **Pangu** (Gu et al., 2023) is a recent SOTA KGQA model. It uses LLMs for discrimination rather than generation for grounding the generated draft. It incrementally constructs plans in a step-wise fashion to handle large search spaces.

- **KB_BINDER** (Li et al., 2023a) enables few-shot learning for KBQA using LLMs through two key stages: Draft Generation, where given a question, an LLM generates a preliminary "draft" logical form using few-shot examples; and Knowledge Base Binding, where entities and relations in the draft are grounded to the target KG using string matching and similarity search.

- **LLM-ICL (SPARQL)** is an in-context learning-based baseline we implement for KQA Pro. As there are no experimental results of Pangu and KB_BINDER on KQA Pro, we use LLM-ICL as an alternative. Since KQA Pro models do not include an entity linking stage, LLM-ICL directly generates SPARQL queries without further grounding stage, ensuring a fair comparison.

- **KVMemNet** (Miller et al., 2016) performs QA by first storing facts in a key-value structured memory before reasoning on them in order to predict an answer. At each reasoning step, the collected information from the memory is cumulatively added to the original query to build context for the next reasoning iteration.

- **SRN** (Qiu et al., 2020) model starts from the question entity and uses a path search tech-

nique to predict the relation path sequence to reach the target entity.

- **RGCN** (Schlichtkrull et al., 2018) uses a graph convolution network-based technique to encode the KG into graph form and perform QA.
- **EmbedKGQA** (Saxena et al., 2020) uses KG embeddings to perform multi-hop reasoning using a RoBERTa-based question encoder.
- **Subgraph Retrieval** (Zhang et al., 2022) use a dual-encoder that provides better retrieval as compared to the existing retrieval methods.
- **PullNet** (Sun et al., 2019) extracts a question specific subgraph from the entire relation graph using a graph CNN instead of heuristics and then retrieves the answer.
- **NSM** (He et al., 2021) propose a teacher-student approach. The student network aims to find the correct answer to the query, while the teacher network tries to learn intermediate supervision signals for improving the reasoning capacity of the student network. They utilize both forward and backward reasoning to enhance the learning of intermediate entity distributions.
- **BART + KoPL** (Cao et al., 2022) is an end-to-end generation model that directly produces the corresponding KoPL program steps given a question. It is worth noting that the pre-trained BART model is forced to have the capability to memorize the relations and entities present in the KG.
- **GraphQ IR** (Nie et al., 2022) proposes a unified intermediate representation for graph query languages, named GraphQ IR. It has a natural-language-like expression that bridges the semantic gap and formally defined syntax that maintains the graph structure. A neural semantic parser is used to convert user queries into GraphQ IR, which can be later losslessly compiled into various downstream graph query languages such as SPARQL, Lambda DCS, etc.
- **TransferNet** (Shi et al., 2021) answers multi-hop questions by attending to different parts of the question at each step. It then computes activated scores for relations, and then transfers the previous entity scores along activated relations in a differentiable way.
- **FlexKBQA** (Li et al., 2023b) utilizing Large Language Models (LLMs) as program translators. It leverages automated algorithms to sample diverse programs, such as SPARQL queries, from the knowledge base, which are subsequently converted into natural language questions via LLMs. They use this synthetic dataset to facilitate training of a specialized lightweight model for a KG.
- **DecAF** (Yu et al., 2023) jointly generates both logical forms and direct answers and then combines the merits of them to get the final answers. They treat logical forms as regular text strings, reducing efforts of hand-crafted engineering. DecAF linearizes KG into text documents and leverages free-text retrieval methods to locate relevant sub-graphs.
- **ChatKBQA** (Luo et al., 2023) propose generate-then-retrieve KBQA framework built on fine-tuning open-source LLMs such as Llama-2, ChatGLM2 and Baichuan2. It generates the logical form with fine-tuned LLMs first, then retrieve and replace entities and relations through an unsupervised retrieval method.
- **SymKGQA** (Agarwal et al., 2024b) propose a Chain of Symbol (CoS) based promting method with KoPL logical form. They use open-source LLMs such as CodeLlama Instruct, PaLM 2 and Llama 2 models to achieve state-of-the-art performance in a few-shot setting. They rely on underlying LLM for KG structure inference for KoPL draft generation and hence, suffers from high syntax error rate.

Figure 3: CodeAlignKGQA Prompt for a Sample Test Question in KQA Pro

```
'''
You are a helpful and faithful python code generator that always follows the below specified rules:
    - Please use the functions defined below to generate the expression corresponding to the
        question step by step.
    - Use the training examples to understand the step generation process and stick only to the
        output format provided in the training examples. Do not generate any explanation text.
    - Do not use entities and concepts outside of the list provided in each test question. If None
        is mentioned in concept in question then it means that their is no concept present in the
        test question and you can't generate any concept related function.
    - Use Verify Functions as the last step of the program before STOP function.
    - The datatypes are as follows:
        - entities: list of entity type
        - value: value of an attribute
        - qvalue: value of a qualifier
        - boolean: True or False
        - relation: relation name
        - facts: knowledge graph fact of the form (entity, predicate, object)
'''

def START():
    '''
    Initialize the expression
    Parameters: None
    Returns:
        expression (any): initialize expression
    '''
    return 'START()'

def FIND(entity: str, expression: any) -> entities:
    '''
    Return all entities having the input entity as name in the knowledge graph
    Parameters:
        entity (str): input entity name
        expression (any): the expression on which it will be executed
    Returns:
        expression (entities): evaluated expression of type entities
    '''
    assert isinstance(expression, any) == True
    return 'FIND({}, {})'.format(entity, expression)

def FINDALL(expression: any) -> entities:
    '''
    Return all entities in the knowledge graph
    Parameters:
        expression (any): the expression on which it will be executed
    Returns:
        expression (entities): evaluated expression of type entities
    '''
    assert isinstance(expression, any) == True
    return 'FINDALL({})'.format(expression)

def FILTERCONCEPT(concept: str, expression: entities) -> entities:
    '''
    Return entities that belongs to the input concept in the knowledge graph
    Parameters:
        concept (str): input concept name
        expression (entities): functional input from the expression
    Returns:
        expression (entities): evaluated expression of type entities
    '''
    assert isinstance(expression, entities) == True
    return 'FILTERCONCEPT({}, {})'.format(concept, expression)
```

```
def FILTERSTR(attribute: str, value: str, expression: entities) -> tuple[entities, facts]:
    '''
    Return entities with the input attribute and value of string type in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        value (str): input attribute value of type string
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, entities) == True
    return 'FILTERSTR({}, {}, {})'.format(attribute, value, expression)

def FILTERNUM(attribute: str, value: int, op: str, expression: entities) -> tuple[entities, facts]:
    '''
    Return entities with the input attribute and value of integer type and op in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        value (int): input attribute value of type integer
        op (str): operator to be applied
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, entities) == True
    return 'FILTERNUM({}, {}, {}, {})'.format(attribute, value, op, expression)

def FILTERYEAR(attribute: str, value: year, op: str, expression: entities) -> tuple[entities, facts]:
    '''
    Return entities with the input attribute and value of year and op in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        value (year): input attribute value of type year
        op (str): operator to be applied
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, entities) == True
    return 'FILTERYEAR({}, {}, {}, {})'.format(attribute, value, op, expression)

def FILTERDATE(attribute: str, value: date, op: str, expression: entities) -> tuple[entities, facts]:
    '''
    Return entities with the input attribute and value of date and op in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        value (date): input attribute value of type date
        op (str): operator to be applied
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, entities) == True
    return 'FILTERDATE({}, {}, {}, {})'.format(attribute, value, op, expression)
```

```
def QFILTERSTR(qualifier: str, qvalue: str, expression: tuple[entities, facts]) -> tuple[entities,
    facts]:
    '''
    Return entities with the input qualifier and qualifier value of string type in the knowledge
        graph
    Parameters:
        qualifier (str): input qualifier name
        qvalue (str): input qualifier value of type string
        expression (tuple[entities, facts]): functional input from the expression of type
            tuple[entities, facts]
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, tuple[entities, facts]) == True
    return 'QFILTERSTR({}, {}, {})'.format(qualifier, qvalue, expression)

def QFILTERNUM(qualifier: str, qvalue: int, op: str, expression: tuple[entities, facts]) ->
    tuple[entities, facts]:
    '''
    Return entities with the input qualifier and qualifier value of integer type and op in the
        knowledge graph
    Parameters:
        qualifier (str): input qualifier name
        qvalue (int): input qualifier value of type integer
        op (str): operator to be applied
        expression (tuple[entities, facts]): functional input from the expression of type
            tuple[entities, facts]
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, tuple[entities, facts]) == True
    return `QFILTERNUM({}, {}, {}, {})'.format(qualifier, qvalue, op, expression)

def QFILTERYEAR(qualifier: str, qvalue: year, op: str, expression: tuple[entities, facts]) ->
    tuple[entities, facts]:
    '''
    Return entities with the input qualifier and qualifier value of year and op in the knowledge
        graph
    Parameters:
        qualifier (str): input qualifier name
        qvalue (int): input qualifier value of type year
        op (str): operator to be applied
        expression (tuple[entities, facts]): functional input from the expression of type
            tuple[entities, facts]
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, tuple[entities, facts]) == True
    return 'QFILTERYEAR({}, {}, {}, {})'.format(qualifier, qvalue, op, expression), entities

def QFILTERDATE(qualifier: str, qvalue: date, op: str, expression: tuple[entities, facts]) ->
    tuple[entities, facts]:
    '''
    Return entities with the input qualifier and qualifier value of date and op in the knowledge
        graph
    Parameters:
        qualifier (str): input qualifier name
        qvalue (int): input qualifier value of date
        op (str): operator to be applied
        expression (tuple[entities, facts]): functional input from the expression of type
            tuple[entities, facts]
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, tuple[entities, facts]) == True
    return 'QFILTERDATE({}, {}, {}, {})'.format(qualifier, qvalue, op, expression)
```

```
def RELATE(relation: str, expression: entities) -> tuple[entities, facts]:
    '''
    Return entities that have the input relation with the given entity in the knowledge graph
    Parameters:
        relation (str): input relation name
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (tuple[entities, facts]): evaluated expression of type tuple[entities, facts]
    '''
    assert isinstance(expression, entities) == True
    return 'RELATE({}, {})'.format(relation, expression)

def QUERYATTR(attribute: str, expression: entities) -> value:
    '''
    Return the attribute value of the entity in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (value): evaluated expression of type value
    '''
    assert isinstance(expression, entities) == True
    return 'QUERYATTR({}, {})'.format(attribute, expression)

def QUERYATTRQUALIFIER(attribute: str, value: str, key: str, expression: entities) -> qvalue:
    '''
    Return the qualifier value of the fact (Entity, Key, Value) in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        value (str): input value of type string
        key (str): input key of type string
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (qvalue): evaluated expression of type qvalue
    '''
    assert isinstance(expression, entities) == True
    return 'QUERYATTRQUALIFIER({}, {}, {}, {})'.format(attribute, value, key, expression)

def QUERYRELATION(expression_1: entities, expression_2: entities) -> relation:
    '''
    Return the relation between two entities in the knowledge graph
    Parameters:
        expression_1 (entities): functional input from the expression of type entities
        expression_2 (entities): functional input from another expression of type entities.
            expression_1 and expression_2 should be different.
    Returns:
        expression (relation): evaluated expression of type relation
    '''
    assert isinstance(expression_1, entities) == True
    assert isinstance(expression_2, entities) == True
    return 'QUERYRELATION({}, {})'.format(expression_1, expression_2)

def QUERYRELATIONQUALIFIER(relation: str, qualifier: str, expression_1: entities, expression_2:
     entities) -> qvalue:
    '''
    Return the qualifier value of the fact in expressions from the knowledge graph
    Parameters:
        relation (str): input relation name
        qualifier (str): input qualifier name
        expression_1 (entities): functional input from the expression of type entities
        expression_2 (entities): functional input from another expression of type entities.
            expression_1 and expression_2 should be different.
    Returns:
        expression (qvalue): evaluated expression of type qvalue
    '''
    assert isinstance(expression_1, entities) == True
    assert isinstance(expression_2, entities) == True
    return 'QUERYRELATIONQUALIFIER({}, {}, {}, {})'.format(relation, qualifier, expression_1,
         expression_2)
```

```
def SELECTBETWEEN(attribute: str, op: str, expression_1: entities, expression_2: entities) -> str:
    '''
    From the two entities, find the one whose attribute value is greater or less and return its name
        in the knowledge graph
    Parameters:
        attribute (str): input attribute name
        op (str): operator to be applied
        expression_1 (entities): functional input from the expression of type entities
        expression_2 (entities): functional input from another expression of type entities.
            expression_1 and expression_2 should be different.
    Returns:
        expression (str): evaluated expression of type string
    '''
    assert isinstance(expression_1, entities) == True
    assert isinstance(expression_2, entities) == True
    return 'SELECTBETWEEN({}, {}, {}, {})'.format(attribute, op, expression_1, expression_2)

def SELECTAMONG(attribute: str, op: str, expression: entities) -> str:
    '''
    From the entity set, find the one whose attribute value is the largest or smallest in the
        knowledge graph
    Parameters:
        attribute (str): input attribute name
        op (str): operator to be applied
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (str): evaluated expression of type string
    '''
    assert isinstance(expression, entities) == True
    return 'SELECTAMONG({}, {}, {}, {})'.format(attribute, op, expression)

def QUERYATTRUNDERCONDITION(attribute: str, qualifier: str, value: str, expression: entities) ->
    value:
    '''
    Return the attribute value whose corresponding fact should satisfy the qualifier key in the
        knowledge graph
    Parameters:
        attribute (str): input attribute name
        qualifier (str): input qualifier name
        value (str): input value of type string
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (value): evaluated expression of type value
    '''
    assert isinstance(expression, entities) == True
    return 'QUERYATTRUNDERCONDITION({}, {}, {}, {})'.format(attribute, qualifier, value, expression)

def VERIFYSTR(value: str, expression: value) -> boolean:
    '''
    Return whether the value is equal as string with the expression
    Parameters:
        value (str): input value of type string
        expression (value): functional input from the expression of type value
    Returns:
        expression (boolean): evaluated expression of type boolean
    '''
    assert isinstance(expression, value) == True
    return 'VERIFYSTR({}, {})'.format(value, expression)
```

```
def VERIFYNUM(value: int, op: str, expression: value) -> boolean:
    '''
    Return whether the value satisfies the op condtion as integer with the expression
    Parameters:
        value (str): input value of type integer
        op (str): operator to be applied
        expression (value): functional input from the expression of type value
    Returns:
        expression (boolean): evaluated expression of type boolean
    '''
    assert isinstance(expression, value) == True
    return 'VERIFYNUM({}, {}, {})'.format(value, op, expression)

def VERIFYYEAR(value: year, op: str, expression: value) -> boolean:
    '''
    Return whether the value satisfies the op condtion as year with the expression
    Parameters:
        value (str): input value of type year
        op (str): operator to be applied
        expression (value): functional input from the expression of type value
    Returns:
        expression (boolean): evaluated expression of type boolean
    '''
    assert isinstance(expression, value) == True
    return 'VERIFYYEAR({}, {}, {})'.format(value, op, expression)

def VERIFYDATE(value: date, op: str, expression: value) -> boolean:
    '''
    Return whether the value satisfy the op condition as date with the expression
    Parameters:
        value (str): input value of type integer
        op (str): operator to be applied
        expression (value): functional input from the expression of type value
    Returns:
        expression (boolean): evaluated expression of type boolean
    '''
    assert isinstance(expression, value) == True
    return 'VERIFYYEAR({}, {}, {})'.format(value, op, expression)

def AND(expression_1: entities, expression_2: entities) -> entities:
    '''
    Return the intersection of the input expressions
    Parameters:
        expression_1 (entities): functional input from the expression of type entities
        expression_2 (entities): functional input from another expression of type entities.
            expression_1 and expression_2 should be different.
    Returns:
        expression (entities): evaluated expression of type entities
    '''
    assert isinstance(expression_1, entities) == True
    assert isinstance(expression_2, entities) == True
    return '(AND {}, {})'.format(expression_1, expression_2)

def OR(expression_1: entities, expression_2: entities) -> entities:
    '''
    Return the union of the input expressions
    Parameters:
        expression_1 (entities): functional input from the expression of type entities
        expression_2 (entities): functional input from another expression of type entities.
            expression_1 and expression_2 should be different.
    Returns:
        expression (entities): evaluated expression of type entities
    '''
    assert isinstance(expression_1, entities) == True
    assert isinstance(expression_2, entities) == True
    return '(OR {}, {})'.format(expression_1, expression_2)
```

```
def COUNT(expression: entities) -> int:
    '''
    Return the count of elements
    Parameters:
        expression (entities): functional input from the expression of type entities
    Returns:
        expression (int): evaluated expression of type integer
    '''
    assert isinstance(expression, entities) == True
    return '(COUNT {})'.format(expression)

def STOP(expression: any):
    '''
    Stop and return the expression
    '''
    return expression

Training Examples:

Training Example 1:
question = "What is the connection between A Serious Man to Ireland (the one whose nominal GDP is
    239389340720.488 United States dollar)?"
entities = ['A Serious Man', 'Ireland']
concepts = None
facts = [
{'entity': 'Ireland', 'attribute': 'PPP GDP per capita'},
{'entity': 'Ireland', 'attribute': 'GDP (PPP)'},
{'entity': 'Ireland', 'attribute': 'nominal GDP per capita'},
{'entity': 'Ireland', 'attribute': 'nominal GDP'},
{'entity': 'Ireland', 'relation': 'currency'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('A Serious Man', expression_1)
expression_2 = START()
expression_2 = FIND('Ireland', expression_2)
expression_2 = FILTERNUM('nominal GDP', '239389340720.488 United States dollar', '=', expression_2)
expression_3 = QUERYRELATION(expression_1, expression_2)
expression_3 = STOP(expression_3)

Training Example 2:
question = "Which first-level administrative country subdivision established post-1829 covers the
    biggest area?"
entities = None
concepts = ['first-level administrative country subdivision']
facts = [
{'concept': 'first-level administrative country subdivision', 'relation': 'operating area'},
{'concept': 'first-level administrative country subdivision', 'attribute': 'area'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FINDALL(expression_1)
expression_1 = FILTERYEAR('inception', 1829, '>', expression_1)
expression_1 = FILTERCONCEPT('first-level administrative country subdivision', expression_1)
expression_1 = SELECTAMONG('area', 'largest', expression_1)
expression_1 = STOP(expression_1)
```

```
Training Example 3:
question = "What is the ISNI of John Broome (the one born in 1738-01-01)?"
entities = ['John Broome']
concepts = None
facts = [
{'entity': 'John Broome', 'attribute': 'ISNI'},
{'entity': 'John Broome', 'relation': 'place of birth'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('John Broome', expression_1)
expression_1 = FILTERDATE('date of birth', '1738-01-01', '=', expression_1)
expression_1 = QUERYATTR('ISNI', expression_1)
expression_1 = STOP(expression_1)

Training Example 4:
question = "Does the sovereign state that has a diplomatic relation with Malaysia (the subject of
    this statement is East TimorMalaysia relations), have the CIVICUS Monitor country entry of
    saint-lucia?"
entities = ['Malaysia']
concepts = ['sovereign state']
facts = [
{'entity': 'Malaysia', 'attribute': 'CIVICUS Monitor country entry'},
{'entity': 'Malaysia', 'relation': 'diplomatic relation', 'qualifier': 'statement is subject of'},
{'entity': 'Malaysia', 'relation': 'country'},
{'entity': 'Malaysia', 'relation': 'country for sport'},
{'concept': 'sovereign state', 'relation': 'country', 'qualifier': 'statement disputed by'},
{'concept': 'sovereign state', 'relation': 'diplomatic relation', 'qualifier': 'statement is subject
    of'},
{'concept': 'sovereign state', 'relation': 'country of origin'},
{'concept': 'sovereign state', 'relation': 'country for sport'},
{'concept': 'sovereign state', 'relation': 'main subject'},
{'concept': 'sovereign state', 'attribute': 'CIVICUS Monitor country entry'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('Malaysia', expression_1)
expression_1 = RELATE('diplomatic relation', expression_1)
expression_1 = QFILTERSTR('statement is subject of', 'East TimorMalaysia relations', expression_1)
expression_1 = FILTERCONCEPT('sovereign state', expression_1)
expression_1 = QUERYATTR('CIVICUS Monitor country entry', expression_1)
expression_1 = VERIFYSTR('saint-lucia', expression_1)
expression_1 = STOP(expression_1)

Training Example 5:
question = "What is the umber of episodes in TV series with Twitter username ThomasFriends (the
    subscription number of this statement is 15947)?"
entities = None
concepts = ['television series']
facts = [
{'concept': 'television series', 'relation': 'part of the series'},
{'concept': 'television series', 'relation': 'series spin-off'},
{'concept': 'television series', 'attribute': 'number of episodes'},
{'concept': 'television series', 'attribute': 'Twitter username', 'qualifier': 'number of
    subscribers'},
{'concept': 'television series', 'attribute': 'Instagram username'},
{'concept': 'television series', 'attribute': 'Twitter hashtag'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FINDALL(expression_1)
expression_1 = FILTERSTR('Twitter username', 'ThomasFriends', expression_1)
expression_1 = QFILTERNUM('number of subscribers', 15947, '=', expression_1)
expression_1 = FILTERCONCEPT('television series', expression_1)
expression_1 = QUERYATTR('number of episodes', expression_1)
expression_1 = STOP(expression_1)
```

```
Training Example 6:
question = "When was born the person that was nominated for Tony Award for Best Actor in a Musical
    in 1967?"
entities = ['Tony Award for Best Actor in a Musical']
concepts = ['human']

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('Tony Award for Best Actor in a Musical', expression_1)
expression_1 = RELATE('nominated for', expression_1)
expression_1 = QFILTERYEAR('point in time', 1967, '=', expression_1)
expression_1 = FILTERCONCEPT('human', expression_1)
expression_1 = QUERYATTR('date of birth', expression_1)
expression_1 = STOP(expression_1)

Training Example 7:
question = "Does Pierce County that is located in Washington or Grays Harbor County have less area?"
entities = ['Washington', 'Pierce County', 'Grays Harbor County']
concepts = None
facts = [
{'entity': 'Pierce County', 'attribute': 'area'},
{'entity': 'Washington', 'attribute': 'area'},
{'entity': 'Washington', 'relation': 'headquarters location'},
{'entity': 'Grays Harbor County', 'attribute': 'area'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('Washington', expression_1)
expression_1 = RELATE('located in the administrative territorial entity', expression_1)
expression_2 = START()
expression_2 = FIND('Pierce County', expression_2)
expression_3 = AND(expression_1, expression_2)
expression_4 = START()
expression_4 = FIND('Grays Harbor County', expression_4)
expression_5 = SELECTBETWEEN('area', 'less', expression_3, expression_4)
expression_5 = STOP(expression_5)

Training Example 8:
question = "How many researchers are the occupation of Aristotle or practice motivational speaking?"
entities = ['Aristotle', 'motivational speaking']
concepts = ['researcher']
facts = [
{'entity': 'Aristotle', 'relation': 'occupation'},
{'concept': 'researcher', 'relation': 'occupation'},
{'concept': 'researcher', 'relation': 'field of this occupation'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('Aristotle', expression_1)
expression_1 = RELATE('occupation', expression_1)
expression_1 = FILTERCONCEPT('researcher', expression_1)
expression_2 = START()
expression_2 = FIND('motivational speaking', expression_2)
expression_2 = RELATE('practiced by', expression_2)
expression_2 = FILTERCONCEPT('researcher', expression_2)
expression_3 = OR(expression_1, expression_2)
expression_3 = COUNT(expression_3)
expression_3 = STOP(expression_3)
```

```
Training Example 9:
question = "Is the nominal GDP of Guinea-Bissau over 69000000 United States dollars on the date
    1996-01-01?"
entities = ['Guinea-Bissau']
concepts = None
facts = [
{'entity': 'Guinea-Bissau', 'attribute': 'PPP GDP per capita'},
{'entity': 'Guinea-Bissau', 'attribute': 'GDP (PPP)'},
{'entity': 'Guinea-Bissau', 'attribute': 'nominal GDP per capita'},
{'entity': 'Guinea-Bissau', 'attribute': 'nominal GDP'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('Guinea-Bissau', expression_1)
expression_1 = QUERYATTRUNDERCONDITION('nominal GDP', 'point in time', '1996-01-01', expression_1)
expression_1 = VERIFYNUM('69000000 United States dollar', '>', expression_1)
expression_1 = STOP(expression_1)

Training Example 10:
question = "Which university has fewer students, George Washington University or University of
    Hamburg?"
entities = ['George Washington University', 'University of Hamburg']
concepts = None
facts = [
{'entity': 'George Washington University', 'attribute': 'students count'},
{'entity': 'University of Hamburg', 'attribute': 'students count'}
]

Make sure to validate the datatype of the parameter before selecting a function using assert
    statements provided in each function. Generate Find function for each entity in the entities
    list. Given the facts provided, the steps to solve this question are:
expression_1 = START()
expression_1 = FIND('George Washington University', expression_1)
expression_2 = START()
expression_2 = FIND('University of Hamburg', expression_2)
expression_3 = SELECTBETWEEN('students count', 'less', expression_1, expression_2)
expression_3 = STOP(expression_3)

Test Question:
```

KQA Pro	Question: ***What was the cost of The Maltese Falcon (the one that received National Film Registry)?***	
Gold KoPL	Step 1: FIND(National Film Registry), Step 2: RELATE(award received), Step 3: FIND(The Maltese Falcon), Step 4: AND(), Step 5: QUERYATTR(cost)	
Gold Answer	300000 United States dollar	
Generated KoPL	Step 1: FIND(National Film Registry), Step 2: FIND(The Maltese Falcon), Step 3: RELATE(award received), Step 4: QUERYRELATION(), Step 5: QUERYATTR(cost)	×
After dynamic code-correction	Step 1: FIND(National Film Registry), Step 2: RELATE(award received), Step 3: FIND(The Maltese Falcon), Step 4: AND(), Step 5: QUERYATTR(cost)	✓
MetaQA (1-hop)	Question: ***What kind of film is Road to Morocco?***	
Gold KoPL	Step 1: FIND(Road to Morocco), Step 2: RELATE(has genre), Step 3: WHAT()	
Gold Answer	Comedy	
Generated KoPL	Step 1: FIND(Road to Morocco), Step 2: RELATE(has tags), Step 3: WHAT()	×
After dynamic code-correction	Step 1: FIND(Road to Morocco), Step 2: RELATE(has genre), Step 3: WHAT()	✓
MetaQA (2-hop)	Question: ***Which movies share the screenwriter with Ugly?***	
Gold KoPL	Step 1: FIND(Ugly), Step 2: RELATE(written by), Step 3: RELATE(written by), Step 4: WHAT()	
Gold Answer	Udaan, Water	
Generated KoPL	Step 1: FIND(Ugly), Step 2: RELATE(screenwriter), Step 3: WHAT()	×
After dynamic code-correction	Step 1: FIND(Ugly), Step 2: RELATE(written by), Step 3: RELATE(written by), Step 4: WHAT()	✓
MetaQA (3-hop)	Question: ***Which person wrote the movies starred by the actors in The Gift?***	
Gold KoPL	Step 1: FIND(The Gift), Step 2: RELATE(starred actors), Step 3: RELATE(starred actors), Step 4: RELATE(written by), Step 5: WHAT()	
Gold Answer	Harvey Weitzman, Neal Jimenez, Robert L. Freedman, Mark Miller, Jacquelin Perske, Katja von Garnier, Joseph Kanon, Vittorio de Benedetti, Robert Mark Kamen, George Lucas, Ben Younger, Woody Allen, George Gallo, Cesare Zavattini, Anthony Minghella, Harry Bates, Jeff Nathanson, Noah Baumbach, Piero Tellini, Robert M. Edsel, Garth Ennis, Marc Norman, John Logan, Kate Kondell, Nancy Meyers, Dean DeBlois, Charles Perrault, Sebastian Faulks, Robin Swicord, F. Scott Fitzgerald, James Ellroy, Herman Raucher, Garry Marshall, George Clooney, Peter Carey, Neal Cassady, Patrick Marber, Scott Caan, Gus Van Sant, Kevin Brodbin, William Gibson, Andy Weiss, Patricia Highsmith, Oliver Parker, Ernest K. Gann, Edmund H. North, Jerry O'Connell, Stephen Gaghan, Scott Silver, Grant Heslov, David Auburn, Christopher Nolan, Kon Ichikawa, Michael Hirst, Philip K. Dick, John August, Eric Roth, Matthew Robbins, Cressida Cowell, Jessica Bendinger, Steven Baigelman, Douglas Fairbanks, Richard Linklater, David S. Goyer, Derek Kolstad, David Koepp, Bernardo Bertolucci, Guillermo del Toro, Oscar Wilde, Wes Anderson, Kevin Williamson, Annie Proulx, Chris Weitz, Bruce Beresford	
Generated KoPL	Step 1: FIND(The Gift), Step 2: RELATE(written by), Step 3: RELATE(written by), Step 4: RELATE(starred actors), Step 5: WHAT()	×
After dynamic code-correction	Step 1: FIND(The Gift), Step 2: RELATE(starred actors), Step 3: RELATE(starred actors), Step 4: RELATE(written by), Step 5: WHAT()	✓
WebQSP	Question: ***Who was vice president after kennedy died?***	
Gold KoPL	Step 1: FIND(John F. Kennedy), Step 2: RELATE(vice president)	
Gold Answer	Lyndon B. Johnson	
Generated KoPL	Step 1: FIND(John F. Kennedy), Step 2: RELATE(successor) Step 3: RELATE(vice president)	×
After dynamic code-correction	Step 1: FIND(John F. Kennedy), Step 2: RELATE(vice president)	✓

Table 9: KoPL Program Steps before and after dynamic code-correction

KQA Pro	Question: ***How are the Atlanta Silverbacks related to association football?***	
Gold KoPL	Step 1: FIND(Atlanta Silverbacks), Step 2: FIND(association football), Step 3: QUERYRELATION()	
Gold Answer	sport	
Generated KoPL without facts	Step 1: FIND(Atlanta Silverbacks), Step 2: RELATE(member of sports team), Step 3: FIND(association football), Step 4: AND(), Step 5: QUERYRELATION(member of sports team)	×
Generated KoPL with facts	Step 1: FIND(Atlanta Silverbacks), Step 2: FIND(association football), Step 3: QUERYRELATION(member of sports team)	✓
MetaQA (1-hop)	Question: ***What does Dexter Fletcher appear in?***	
Gold KoPL	Step 1: FIND(Dexter Fletcher), Step 2: RELATE(starred actors), Step 3: WHAT()	
Gold Answer	Below, The Rachel Papers	
Generated KoPL without facts	Step 1: FIND(Dexter Fletcher), Step 2: RELATE(appeared in), Step 3: WHAT()	×
Generated KoPL with fact	Step 1: FIND(Dexter Fletcher), Step 2: RELATE(starred actors), Step 3: WHAT()	✓
MetaQA (2-hop)	Question: ***What are the movies that have the same screenwriter of Rubin and Ed?***	
Gold KoPL	Step 1: FIND(Rubin and Ed), Step 2: RELATE(written by), Step 3: RELATE(written by), Step 4: WHAT()	
Gold Answer	The Beaver Trilogy	
Generated KoPL without facts	Step 1: FIND(Rubin and Ed), Step 2: RELATE(written by), Step 3: WHAT()	×
Generated KoPL with fact	Step 1: FIND(Rubin and Ed), Step 2: RELATE(written by), Step 3: RELATE(written by), Step 4: WHAT()	✓
MetaQA (3-hop)	Question: ***Which person wrote the films acted by the actors in The Slingshot?***	
Gold KoPL	Step 1: FIND(The Slingshot), Step 2: RELATE(starred actors), Step 3: RELATE(starred actors), Step 4: RELATE(written by), Step 5: WHAT()	
Gold Answer	Matt Damon, Ben Affleck, J.D. Zeik, Dashiell Hammett, Eric Lomax, Ronald Harwood, David Mamet, Hans Petter Moland, Aimee Lagos, Catherine Johnson, Steve Barancik, Hans Alfredson, Wesley Strick, Zafar Hai, Kjell Grede	
Generated KoPL without facts	Step 1: FIND(The Slingshot), Step 2: RELATE(starred actors), Step 3: RELATE(written by), Step 4: WHAT()	×
Generated KoPL with fact	Step 1: FIND(The Slingshot), Step 2: RELATE(starred actors), Step 3: RELATE(starred actors), Step 4: RELATE(written by), Step 5: WHAT()	✓
WebQSP	Question: ***Who was vice president after kennedy died?***	
Gold KoPL	FIND(John F. Kennedy), Step 2: RELATE(vice president)	
Gold Answer	Lyndon B. Johnson	
Generated KoPL without facts	Step 1: FIND(Franklin D. Roosevelt), Step 2: RELATE(successor), Step 3: RELATE(vice president)	×
Generated KoPL with facts	Step 1: FIND(Franklin D. Roosevelt), Step 2: RELATE(vice president)	✓

Table 10: KoPL Program Steps with and without Knowledge-Aware Facts Extraction