PO-KGQA: Preference Optimization for Low-Resource Complex Knowledge Graph Question Answering

Prerna Agarwal[*1,2], **Ayushman Kumar Singh**[1], **Srikanta Bedathur**[1]
[1]Indian Institute of Technology Delhi, India
[2]IBM Research India
preragar@in.ibm.com, ayushman.ksingh.iitdelhi@gmail.com, srikanta@cse.iitd.ac.in

Abstract

Existing low-resource in-context learning-based knowledge graph question answering (KGQA) methods rely heavily on large language models (LLMs) to convert the natural language question into its corresponding logical form (LF), such as SPARQL, KoPL, etc. Recently, a few alignment techniques have been introduced that enable instruction-based fine-tuning of language models. They provide explicit negative signals and comparative objectives to learn how to avoid negative signals using preference optimization methods. Exploring such fine-tuning techniques with LLMs becomes very challenging due to the high computational resource requirements associated with them. Due to this, the focus has been shifted towards Small Language Models (SLMs), which offer advantages such as ease of (i) deployment for practical applications and (ii) instruction fine-tuning for specialized tasks. Motivated by this, in this work, we propose PO-KGQA[1]: An SLM-based preference optimization framework for the complex KGQA task in a low-resource setting. Our extensive experiments demonstrate how PO-KGQA outperforms other fine-tuning alignment techniques on complex benchmarks such as KQA Pro by approximately 9% (avg).

1 Introduction

Due to the dearth of annotated data and recent advancements in large language models (LLMs), research has moved towards LLM-based low-resource Knowledge Graph Question Answering (KGQA) (Gu et al., 2023; Zhang et al., 2024b). KGQA aims to answer a natural language question (NLQ) by producing a logical form such as SPARQL, KoPL, etc. that is executed on the KG to retrieve the answer(s) (Yu et al., 2023; Fang et al., 2024). Most LLM-based methods (Agarwal et al., 2024, 2025; Li et al., 2024b) take advantage of in-context learning (ICL) (Brown et al., 2020) with few-shot examples to generate the logical form.

Language Models have limited success rates on complex KGQA benchmarks. They possess limited pre-trained knowledge about the grammar and semantics of the the logical forms (SPARQL, KoPL, etc.). This leads to hallucinations and incorrect logical form generation. Hence, some works have aligned the language models to generate the desired output using supervised fine-tuning (SFT) (Luo et al., 2024; Niu et al., 2023; Fang et al., 2024; Feng and He, 2025). Alignment methods involve fine-tuning the pre-trained model using SFT or using preference alignment methods such as direct preference optimization (DPO) (Rafailov et al., 2023). Preference alignment methods enable instruction-based fine-tuning with explicit *preferred* and *dispreferred* data pairs and comparative objectives to learn how to minimize *dispreferred* output through optimization. It has proven to provide gains compared to SFT alone for various tasks (Thakkar et al., 2024; Gisserot-Boukhlef et al., 2024; Zhang et al., 2025).

LLMs have high operational costs associated with them. The high computational cost of LLMs hinders: (i) fine-tuning, and (ii) their deployment on resource-constrained devices. Due to this, the focus has shifted towards Small Language Models (SLMs)[2] that offer advantages such as ease of: (i) deployment for practical applications; (ii) fine-tuning for a specialized task that offer similar capabilities at reduced cost. This motivates us to benchmark the preference optimization to fine-tune the SLMs for the complex KGQA task.

Complex questions require joint compositional and numerical reasoning. This demands decom-

[*]P. Agarwal is an employee at IBM Research. This work was carried out as part of PhD research at IIT Delhi

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

[2]Language Models with < 10B parameters are considered as SLMs in this work

Findings of the Association for Computational Linguistics: ACL 2026, pages 41976–41997

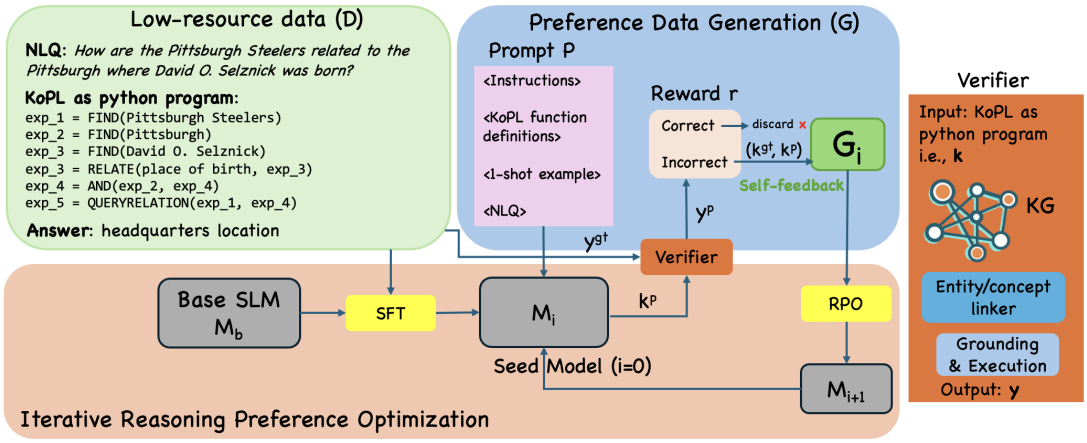

Figure 1: The **PO-KGQA framework**, consisting of 2 main steps: (a) Dynamic preference data G_i generation using self-feedback; (b) Iterative RPO (loss modified to eliminate the dependency on ground truth answer y_{gt})

position of the question and solving it in a step-by-step manner. This can be achieved with the symbolic form of chain-of-thought (CoT) (Liang et al., 2023), i.e., chain-of-symbol (CoS) prompting (Hu et al., 2024). It is one of the key factors in improving the complex reasoning capability of Language Models (Wei et al., 2023). CoS decomposes complex problems into intermediate symbolic reasoning steps, improving performance and interpretability. KoPL (Knowledge-Oriented Programming Language) (Cao et al., 2022) is a symbolic logical form for KGQA that defines various operations in KG to account for various complexities. Each KoPL step becomes a CoS reasoning step (see Figure 1). This makes it suitable for complex compositional and numerical reasoning, while being interpretable. Hence, in this work, we choose KoPL as the logical form. The transformation of the logical form generation task as code generation can further provide gains (Nie et al., 2024; Agarwal et al., 2025). Inspired by this, we generate KoPL as Python program steps.

One of the challenges in adopting preference optimization for KGQA is to obtain the preference data. To the best of our knowledge, none of the KGQA benchmarks provides preference data of any form. Most of the work uses closed-source GPT models (ope, 2024) or LLMs (with $\geq$70B parameters) (Touvron et al., 2023) to obtain the preference data (Wu et al., 2024; Ko et al., 2024; Zheng et al., 2025; Li et al., 2025) for the KG fact summarization task. None of the work has focused on generating CoS reasoning steps like preference data for KGQA. Therefore, we propose a novel method to dynamically generate preference data using low-resource data i.e., *not more than* 1000 *data points*. Preference data is generated for KoPL Python steps cost-effectively, eliminating the dependency on any teacher LLM.

Given the rapidly growing number of preference optimization methods, we choose the one that optimizes the preference between the correct vs. incorrect CoS reasoning steps. Hence, we adopt Iterative Reasoning Preference Optimization (RPO) (Pang et al., 2024) for our work. We modify the RPO loss to focus on generating correct CoS reasoning steps during the fine-tuning process (see Section 3.3).

Parameter-efficient training (PEFT) methods, such as Low-Rank Adaptation (LoRA) (Hu et al., 2022) can achieve performance comparable to that of full fine-tuning of language models at a much lower cost. Therefore, in this work, instead of full fine-tuning, we build a lightweight adapter using LoRA with SLM so that it can be used with any pre-existing deployment of an SLM. We selected LoRA as the PEFT method for our experiments because of its relatively small performance degradation over massive compute savings.

In summary, the contributions of our work are:

1. We propose PO-KGQA: An SLM-based preference optimization framework for the Complex KGQA task in a low-resource setting i.e., not more than 1000 data points.

2. We propose a novel method to dynamically generate preference data for KoPL Python steps with self-feedback.

3. We propose a modified Iterative RPO loss focusing on CoS reasoning steps during fine-tuning.

4. We experiment with different SLMs and datasets to demonstrate the robustness and flexi-

bility of our framework.

Our framework builds a light-weight adapter using the PEFT method, i.e., LoRA, that can be used with any pre-existing deployment of an SLM. *To the best of our knowledge, our work is the first to explore preference alignment of the logical form generation for the complex KGQA task.* Our extensive experiments demonstrate that PO-KGQA outperforms all other alignment techniques, setting a new benchmark.

2 Related Work

2.1 LLM-based KGQA techniques

Most of the state-of-the-art LLM-based KGQA techniques leverage the reasoning capabilities of LLM to generate the logical forms such as SPARQL and S-expression for a given NLQ. Pangu (Gu et al., 2023) and KB_BINDER (Li et al., 2023) leverage the discriminative ability of LLMs and constrained decoding, respectively, to incrementally generate S-expression. FlexKBQA (Li et al., 2024b) uses LLMs to sample SPARQL queries converted to NLQ for training. Some of the recent works have also transformed the logical form generation into a code generation task. KB-Coder (Nie et al., 2024) performs code-style S-expression generation by retrieving a similar relation and passing it to the LLM prompt. CodeAlignKGQA (Agarwal et al., 2025) performs code-style KoPL generation. They provide constraints of the logical form and relevant instructions in the LLM prompt. These methods have shown gains compared to their respective counterparts, i.e., FlexKBQA and SymKGQA (Agarwal et al., 2024). However, these techniques suffer massively when used with SLMs for practical applications. ChatKBQA (Luo et al., 2024) and (Niu et al., 2023) fine-tune SLMs with (NLQ, logical form) pairs in a fully-supervised manner and do not work in a low-resource setting.

2.2 Preference Optimization for KGQA task

EFSum (Ko et al., 2024) uses GPT-3.5 Turbo[3] as the teacher model to generate preference data for KG fact summarization. This data is then used to fine-tune a student model, Llama-7B, using DPO. KnowPAT (Zhang et al., 2024a) builds a preference dataset by retrieving relevant KG triples and generating answers using ChatGPT[4], ChatGLM-6B[5], and Vicuna-7B[6], ranked in descending order of preference. The base model, Atom-7B[7], is then fine-tuned to disfavor less preferred responses. PAQAF (Wu et al., 2024) addresses knowledge rewriting (i.e., KG fact summarization) by incorporating interleaved CoT reasoning steps. ChatGPT is first used for SFT of the knowledge rewriter, after which preference pairs are created and 7B-scale models are fine-tuned using DPO.

The goal of our work is different. Our goal is to generate CoS reasoning steps like preference data of logical form for KGQA, which, to the best of our knowledge, none of the work provides.

2.3 Self-learning in Language Models

This work focuses on fine-tuning the language models by self-generation of the training data for different tasks. STaR (Zelikman et al., 2022) generates rationales and then applies SFT using the rationales, which leads to the correct answer. Self-rewarding LLMs (Yuan et al., 2024) use iterative DPO to provide rewards on their own during training. ST-DPO (Wang et al., 2024) uses DPO to generate pseudo-labels for SFT. SPO (Li et al., 2024a) constructs a self-supervised preference degree loss combined with the alignment loss. CPO (Xu et al., 2024) uses triplet preference data, i.e., (reference, positive, negative) for machine translation.

None of the above methods is designed for the KGQA task. Motivated by this, we perform self-training with iterative RPO for the KGQA task.

3 PO-KGQA Framework

We first briefly define the problem statement and then introduce the details of the PO-KGQA framework shown in Figure 1, which consists of 2 stages:

1. Generation of dynamic preference data.
2. Alignment using modified Iterative RPO loss.

3.1 Problem Statement

The goal of PO-KGQA is twofold:

1. Generate the pairs of preference data (G_i) dynamically with *preferred* and *dispreferred* KoPL Python steps using SLM (M_i) for each iteration i.
2. Fine-tune M_{i+1} using G_i with modified iterative RPO loss to generate KoPL Python steps.

Here, KG $K \subset E \times R \times (E \cup L \cup C)$, where C, E, L and R is the set of concepts[8], entities, attributes

[3] https://platform.openai.com/docs/models/gpt-3.5-turbo

[4] https://chat.openai.com

[5] https://huggingface.co/THUDM/chatglm-6b

[6] https://ollama.com/library/vicuna:7b

[7] https://huggingface.co/FlagAlpha/Atom-7B

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

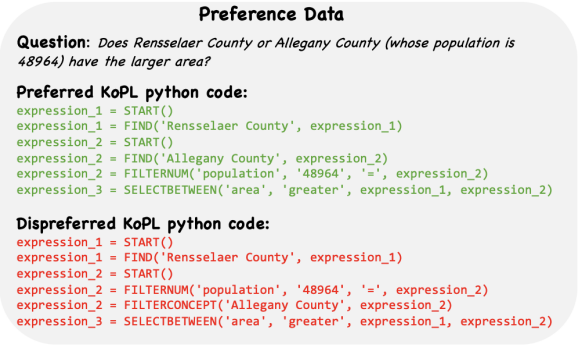

Figure 2: Example of preference data pair for a question

(literals) and binary relations, respectively.

3.2 Preference Data Generation

As shown in Figure 1, the low-resource training dataset D consists of (NLQ, KoPL Python steps, answer) tuples. We use the answer y_d for all $d \in D$ only for verification (see Verifier in Figure 1), unlike other methods (Pang et al., 2024) that use y_d for fine-tuning.

Prompt: As shown in Figure 1, we build the prompt P with instructions and KoPL function definitions to generate KoPL Python steps. We also add an example in P to demonstrate the desired output format of KoPL Python steps. This example remains constant for a dataset (see Appendix A.1 for details of P).

Generation: For each iteration $i \in [1,..,\mathrm{T}]$ of RPO, for a given training data point $d \in D$, and prompt P, the SLM (M_i) is expected to generate the KoPL Python steps. The NLQ q_d of d is appended with P to form P_d. The KoPL Python steps k_d^p are then generated using M_i using greedy decoding as follows:

$$k_d^p = \arg\max[M_i(P_d)], \quad \forall d \in D. \tag{1}$$

The binary reward r_d is then calculated based on the correctness of the final answer (y_d^p) that k_d^p produces upon its execution (by the Verifier):

$$r_d = \begin{cases} 1 & \text{if} \quad y_d^p = y_d, \\ 0 & \text{if} \quad y_d^p \neq y_d. \end{cases} \tag{2}$$

Here, y_d is the ground truth answer for d.

The preference pair dataset G_i (initialized as $G_i = \{\}$) is then constructed dynamically using data points $d \in D$ for which $r_d = 0$ (see Reward r in Figure 1) as the **self-feedback**, as follows:

$$G_i = G_i \cup \{q_d, (k_d^{gt}, k_d^p)\}, \quad \forall d \in D, \quad \text{iff } r_d = 0. \tag{3}$$

Here, k_d^{gt} is the ground truth KoPL Python steps of d (treated as ***preferred*** data) and k_d^p is treated as ***dispreferred*** data. Note that this self-feedback will change in every iteration, and therefore G_i is dynamic, while D remains constant. Thus, the diversity of G is determined by the model's ability to adapt to the failure cases over time. An example of preference data pair is shown in Figure 2.

We restrict the preference pair construction to failure cases for three main reasons:

1. Successful generations are already aligned with the gold programs. Including them would create near-duplicate positive pairs, providing limited additional gradient signal.

2. The strongest optimization signal comes from incorrect reasoning trajectories. By focusing on failure cases, we maximize the learning signal and improve discrimination between correct and incorrect programs.

3. Model-generated programs that accidentally produce the correct answer may still be logically incorrect (spurious programs). Using such programs in preference pair construction could introduce noise into training. Restricting pairs to failure cases helps mitigate this issue.

3.3 Fine-tuning using proposed RPO loss

The model is fine-tuned in an iterative manner, i.e., M_1,...., M_T where each successive iteration $i+1$ uses preference data pairs in G_i created by the i_{th} model (see Figure 1).

Initial Model M_0: As shown in Figure 1, we first perform SFT on the base model M_b to obtain the initial seed model M_0. It can then be used to generate KoPL Python steps. The SFT is performed using the following loss function:

$$\mathcal{L}_{\text{SFT}} = \mathbb{E}_{(P_d, k_d^{gt}) \sim P_d} \left[-\sum_{j=1}^{|k_d^{gt}|} \log M_0\left(k_{d_j}^{gt} \mid P_d, k_{d_{1:j-1}}^{gt}\right) \right], \quad \forall d \in D. \tag{4}$$

where, $k_{d_j}^{gt}$ is the j^{th} token in k_d^{gt} and $|k_d^{gt}|$ is the length of the KoPL Python code. Note that y_d is not used in this step.

Iterative training: The parameters of M_{i+1} are initialized from model M_i. They are updated with the RPO loss function that combines DPO and negative log-likelihood (NLL) loss for learning over the *preferred* KoPL Python steps from each pair in G_i, i.e., k_d^{gt}.

Modified RPO for Intermediate Reasoning Correctness: Unlike existing approaches that primarily optimize for the target answer y_d, our method decouples the fine-tuning process from it. This is a significant conceptual shift because y_d is inherently unstable and subject to change as KGs evolve. We shift the objective to the generation of accurate KoPL Python steps and prioritize intermediate reasoning correctness over output matching. This design inherently enhances the model's robustness and generalizability, particularly for out-of-distribution (OOD) questions where existing answer-centric models would fail. Hence, the proposed RPO loss corresponding to each preference pair in G_i is as follows:

$$\begin{aligned}\mathcal{L}_{\text{DPO+NLL}} &= \mathcal{L}_{\text{DPO}}(k_w^{gt}, k_w^p/P_w) + \alpha\mathcal{L}_{\text{NLL}}(k_w^{gt}/P_w) \\ &= -\log\sigma\Bigg(\beta\log\frac{M_{i+1}(k_w^{gt}/P_w)}{M_i(k_w^{gt}/P_w)} \\ &\quad - \beta\log\frac{M_{i+1}(k_w^{p}/P_w)}{M_i(k_w^{p}/P_w)}\Bigg) \\ &\quad - \alpha\log\frac{M_{i+1}(k_w^{gt}/P_w)}{|k_w^p|}, \quad \forall w \in G_i.\end{aligned} \tag{5}$$

Here, q_w is appended with P to form P_w. $M(x)$ denotes the probability of sequence x under the model M, and σ is the sigmoid function. M_i is used as the reference model for $\mathcal{L}_{DPO}$. NLL term is normalized by the total length of the generated KoPL Python code. The hyperparameter α balances the DPO and NLL loss. After training with $\mathcal{L}_{DPO+NLL}$, M_{i+1} is obtained, which will be used to create G_{i+1} for the subsequent iteration. The iterative training is continued till T iterations.

Zero-shot KoPL Python steps generation: The fine-tuned model M_T thus obtained is now expected to generate the KoPL Python steps for a given NLQ q in a zero-shot manner using greedy decoding as follows:

$$k_q^p = \arg\max[M_T(P_q)]. \tag{6}$$

where, q is appended with P to form P_q.

3.4 Verifier

The generated KoPL python steps k_q^p can still require grounding of textual input due to hallucinations. Therefore, based on the input datatype (entities, relations, attributes, etc.), the generated KoPL Python steps are **grounded** via the Verifier (as shown in Figure 1) as follows:

- Entities and concepts present in the generated KoPL Python steps are grounded using the **entity and concept linker**. We use Named Entity Recognition (NER) for this purpose; however, any standard linker such as (Gu et al., 2023; Mohammed et al., 2018) or POS tags can also be used.
- For relations and attributes present in each KoPL Python step:
  - For each generated input, we retrieve the top-10 most similar KG elements based on datatype. The search is performed over the KG schema (i.e., relations, attributes, and entity types), rather than the full triple store. We use a BERT-based semantic similarity retriever (`all-distilroberta-v1`[9]) to perform this retrieval.
  - The fine-tuned SLM (M_T) serves a purely discriminative role: Given a question q, it ranks the top-10 candidates and selects the most relevant one. It does not perform generation, avoiding error compounding, and ensures KG grounding is both question and model-aware. An example prompt is shown in Figure 3.

Question-aware Prompt

Instruction: From the relation_list, select one relation that aligns the most to the extracted_relation for the given 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']

extracted_relation = ['headquartered in']

Figure 3: KG Grounding Prompt

Note: Verifier grounding errors do not bias learning, since the ground truth program remains the fixed preferred reference. False negatives may reduce training efficiency, but do not promote incorrect reasoning, while false positives are excluded from training and, therefore, have no effect.

Execution: The grounded KoPL Python steps are parsed based on the generated expression dependency and converted into the standard KoPL format. The transformed KoPL steps are then executed on the KG according to the default KoPL executor provided by (Cao et al., 2022) to obtain y_q.

[9] https://huggingface.co/sentence-transformers/all-distilroberta-v1

Dataset	KG	Train	Val	Test
KQA Pro	Wikidata	94,376	11,797	11,797
WebQSP	Freebase	2,998	100	1,639

Table 1: Statistics of the Datasets used

4 Experiments

Our experiments answer the following research questions:

1. How does PO-KGQA compare with other alignment baselines in a low-resource setting?
2. How does the performance of PO-KGQA vary with different SLMs and datasets?
3. What are the contributions of each PO-KGQA component?

4.1 Datasets

We experiment with 2 datasets having different: (a) question complexities; (b) underlying KG; and (c) number of inference hops required (see Table 1).

1. **KQA Pro** (Cao et al., 2022): It contains much **harder and more complex QA pairs** with numerical quantities, concepts, and entities for multi-hop compositional and numerical reasoning. It contains questions that require up to 10-hops answerable through enriched Wikidata KG.
2. **WebQSP** (Yih et al., 2016): This dataset is based on Freebase KG and contains questions up to 2-hops from **Google query logs**. It exhibits more complex structures, i.e., isomorphisms (ISO-4), compared to other datasets (Gu et al., 2021).

We discuss why we believe that further evaluation on datasets such as GrailQA, LC-Quad 2.0, etc. would not add new scientific insights into PO-KGQA's overall utility in the Appendix A.2.

4.2 Models

We select 7B Code SLMs, as they offer a strong trade-off between performance and efficiency, being lightweight enough for single-GPU deployment while maintaining reasonable accuracy. After evaluating several SLMs including Qwen2.5-Coder (Hui et al., 2024), DeepSeek-R1-Distill (DeepSeek-AI et al., 2025), and StarCoder2 (Lozhkov et al., 2024), we selected the following top-performing models for our experiments:

1. **CodeLlama Instruct (7B)** (Rozière et al., 2023): A code-focused SLM built upon Llama 2. It can follow programming instructions for various programming tasks.
2. **DeepSeek-Coder Instruct (6.7B)** (DSC Ins.) (Guo et al., 2024): This model is re-trained on a high-quality project-level code corpus to enhance code generation capabilities.

4.3 Baselines

We select the following 3 categories of baselines:

• **Fully Supervised**: We compare with fully supervised state-of-the-art techniques for each dataset. It includes KVMemNet (Miller et al., 2016), EmbedKGQA (Saxena et al., 2020), Subgraph Retrieval (Zhang et al., 2022) for all datasets, and:

1. KQA Pro: SRN, RGCN (Schlichtkrull et al., 2018), BART + KoPL (Cao et al., 2022), GraphQ IR (Nie et al., 2022).
2. WebQSP: DecAF (Yu et al., 2023).

• **Few-Shot**: We compare with few-shot techniques that have shown state-of-the-art for each dataset. It includes FlexKBQA (Li et al., 2024b), SymKGQA (Agarwal et al., 2024), CodeAlignKGQA[10] (Agarwal et al., 2025) for all datasets, and:

1. KQA Pro: LLM-ICL (SPARQL)[11].
2. WebQSP: KB_BINDER (Li et al., 2023), KB-Coder (Nie et al., 2024) and Pangu (Gu et al., 2023). We use KB_BINDER(1) and KB-Coder(1) settings for a fair comparison.

• **Alignment**: We compare with the following state-of-the-art language model alignment techniques, i.e. preference optimization and SFT, for each dataset under the same experimental setting (including Verifier and data sampling):

1. SFT (using Eq. 4).
2. STaR† (Zelikman et al., 2022).
3. Self-rewarding LLM† (Yuan et al., 2024).
4. Self-rewarding LLM† (Yuan et al., 2024).
5. ST-DPO† (Wang et al., 2024).
6. ChatKBQA (Luo et al., 2024) (for WebQSP).

Note that SPO and CPO are not applicable in our setting; hence, they are not considered as baselines.

We also compare with the zero-shot setting of:

1. FlexKBQA (Li et al., 2024b);
2. KoPL generation with prompt P;
3. InteractiveKBQA (Xiong et al., 2024) for KQA Pro and KAPING (Baek et al., 2023) for WebQSP.

[10] w/o code-correction for a fair comparison

[11] Alternative to Pangu for evaluation

† implemented for KGQA task and eliminated y_d during fine-tuning for a fair comparison

	Model	Acc
fully supervised models		
KVMemNet	-	6.90
EmbedKGQA	-	20.27
SRN	-	11.84
RGCN	-	29.12
Subgraph Retrieval (SE)	-	22.82
BART + KoPL	-	83.28
GraphQ IR	-	79.13
few-shot models (100-shots)		
FlexKBQA	GPT-3.5 Turbo	42.68
LLM-ICL (SPARQL)	GPT-3.5 Turbo	27.75
SymKGQA (KoPL)	CodeLlama Ins. (34B)	51.10
CodeAlignKGQA (KoPL)	CodeLlama Ins. (34B)	55.16
	DSC Ins. (33B)	48.72
zero-shot models		
KoPL generation	CodeLlama Ins. (7B)	12.09
	DSC Ins. (6.7B)	11.14
FlexKBQA	GPT-3.5 Turbo	28.48
InteractiveKBQA	GPT-4 Turbo	25.25
Alignment Techniques (low-resource fine-tuning)		
SFT	CodeLlama Ins. (7B)	<u>52.5</u>
	DSC Ins. (7B)	43.85
Self-rewarding†	CodeLlama Ins. (7B)	36.07
	DSC Ins. (6.7B)	33.06
STaR†	CodeLlama Ins. (7B)	18.13
	DSC Ins. (6.7B)	10.32
ST-DPO†	CodeLlama Ins. (7B)	33.06
	DSC Ins. (6.7B)	42.28
PO-KGQA	CodeLlama Ins. (7B)	**61.07**
	DSC Ins. (6.7B)	45.4
-SFT	CodeLlama Ins. (7B)	33.76
	DSC Ins. (6.7B)	20.04
-Verifier	CodeLlama Ins. (7B)	54.64
	DSC Ins. (6.7B)	38.78

Table 2: PO-KGQA Results for KQA Pro on dev set. Bold and underline denote best and second best performance **among alignment techniques**. dagger(†) denotes re-implementation of baselines for KGQA task.

Refer to Appendix A.3 for more details on the techniques adopted by each baseline.

4.4 Implementation and Hyper-parameters

SFT is performed on pre-trained SLM using LoRA on low-resource data $|D| = 1000$. We use Structure-Aware Logarithmic Stratified Sampling (Cohen et al., 2011) to sample D from the train set. The details are provided in Appendix A.4. The model is fine-tuned over 3 epochs with a learning rate of $2e-5$. The LoRA rank is 32, the LoRA alpha is 32, and the LoRA dropout is 0.05. For RPO, the values of α and β are 1 and 0.1 respectively, fine-tuned with a learning rate of $2e-5$. The number of iterations for RPO is $T = 3$. All experiments were carried out on a server equipped with 2 V100 GPUs, each with 32GB of RAM. Pytorch[12] and Hugging Face[13] are used to implement PO-KGQA.

[12] https://pytorch.org/

[13] https://huggingface.co/

Model	Acc%
ChatGPT	24.96
Davinci-003	31.02
GPT-4	37.43
PO-KGQA	**61.07**

Table 3: Direct QA performance on KQA Pro

We use accuracy[14] for KQA Pro and the F1 score for WebQSP as evaluation metrics.

5 Results and Analysis

The results for each dataset are discussed in detail in the following. While fully supervised, few-shot, and zero-shot settings are not directly comparable due to different levels of supervision, we include them to provide an overall performance contrast.

5.1 Results on KQA Pro

The results of PO-KGQA for KQA Pro compared to different baselines are shown in Table 2. As shown, PO-KGQA with the CodeLlama Instruct model outperforms all alignment techniques and *achieves a new state-of-the-art in low-resource preference alignment techniques for KQA Pro*. Specifically, it beats Self-rewarding, STaR, and ST-DPO by 25%, 43%, 28%, respectively. PO-KGQA with CodeLlama performs 15% superior to DSC Ins. The gain observed w.r.t. the base model (see KoPL generation in zero-shot models in Table 2) is 49% and 34% for CodeLlama and DSC Ins, respectively. It outperforms all few-shot baselines that use strong LLMs as well. Specifically, it outperforms CodeAlignKGQA by 6%, SymKGQA by 10%, FlexKBQA by 18.4%, and LLM-ICL by 33.3% using CodeLlama Instruct. With DSC Ins, it outperforms FlexKBQA and LLM-ICL while it lags behind CodeAlignKGQA (DSC Ins.) and SymKGQA by approx. 3% and 6% only.

The comparison with state-of-the-art LLMs is shown in Table 3. As shown, even the state-of-the-art LLMs are incapable of handling complex natural language questions. PO-KGQA, on the other hand, enhances the reasoning capability of language models by optimizing them with CoS-like preference data for correct logical form generation. This demonstrates that SLMs that otherwise perform poorly for practical applications can now perform complex reasoning better than most existing techniques by tuning them with preferred reasoning

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

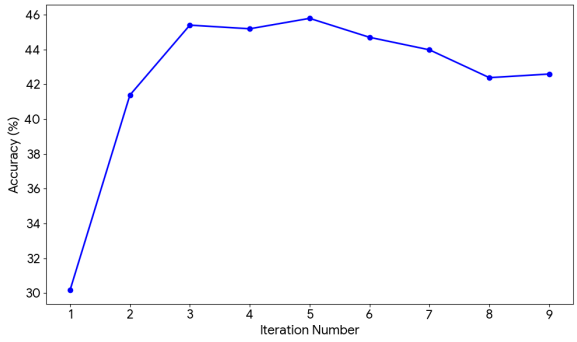

Figure 4: PO-KGQA performance over different iterations with DeepSeekCoder-Instruct on KQA Pro

steps while explicitly avoiding dispreferred reasoning steps.

We present evidence of the model's steady performance improvement across iterations. Figure 4 shows the performance of DeepSeekCoder-Instruct on KQA Pro over successive training iterations. As shown, the performance improves significantly during the first few iterations, reaches a plateau around Iterations 3–5, and then gradually decreases as the number of iterations increases further. This also indicates that the self-feedback loop converges early, and excessive iterations may lead to performance degradation.

5.2 Results on WebQSP

The results of PO-KGQA for WebQSP compared to different baselines are shown in Table 4. As shown, PO-KGQA with the CodeLlama Instruct model outperforms all alignment techniques and *achieves a new state-of-the-art in low-resource preference alignment techniques for WebQSP*. Specifically, it beats Self-rewarding, STaR, and ST-DPO by 13%, 15.6%, 15%, respectively using CodeLlama. PO-KGQA with CodeLlama performs 3.4% superior to DSC Ins.

The gain observed w.r.t the base model (see KoPL generation in zero-shot models) is 19.4% and 20.8% for CodeLlama and DSC Ins, respectively. It outperforms all few-shot baselines that use strong LLMs as well. Specifically, it outperforms CodeAlignKGQA by 6%, 2.5%; FlexKBQA by 14.5%, 10% using CodeLlama and DSC Ins, respectively. It outperforms SymKGQA using CodeLlama by 3.5% and performs nearly same using DSC Ins. PO-KGQA beats ChatKBQA under similar resource setting by 4%.

	Model	F1
fully supervised models		
KVMemNet	-	46.7
EmbedKGQA	-	66.6
DecAF	-	78.8
Subgraph Retrieval	-	66.7
few-shot models (100-shots)		
FlexKBQA	GPT-3.5 Turbo	60.6
KB_BINDER (1)	Codex	52.5
KB-Coder (1)	GPT-3.5 Turbo	55.7
Pangu	Codex	54.5
SymKGQA (KoPL)	CodeLlama Ins. (34B)	70.6
CodeAlignKGQA (KoPL)	CodeLlama Ins. (34B)	68.12
zero-shot models		
KoPL generation	CodeLlama Ins. (7B)	54.66
	DSC Ins. (6.7B)	49.84
FlexKBQA	GPT-3.5 Turbo	46.2
KAPING	T5-11B	24.91
Alignment Techniques (low-resource fine-tuning)		
ChatKBQA	Llama-2-7B	70.00
SFT	CodeLlama Ins. (7B)	69.00
	DSC Ins. (7B)	64.73
Self-rewarding†	CodeLlama Ins. (7B)	61.31
	DSC Ins. (6.7B)	38.63
STaR†	CodeLlama Ins. (7B)	58.45
	DSC Ins. (6.7B)	63.95
ST-DPO†	CodeLlama Ins. (7B)	58.23
	DSC Ins. (6.7B)	60.23
PO-KGQA	CodeLlama Ins. (7B)	**74.06**
	DSC Ins. (6.7B)	70.65
-SFT	CodeLlama Ins. (7B)	55.33
	DSC Ins. (6.7B)	59.9
-Verifier	CodeLlama Ins. (7B)	21.9
	DSC Ins. (6.7B)	15.21

Table 4: PO-KGQA Results for WebQSP

5.3 Ablation Study

We ablate the following core components of PO-KGQA: (1) SFT initialization; (2) NLL loss term in RPO; (3) KG Grounding in Verifier.

1. *SFT initialization*: We omit the 'Initial Model' step and directly fine-tune the base model M_b using the proposed RPO loss. As shown in Table 2 (**-SFT configuration**), for KQA Pro, performance decreases by 27.3% and 25.36%. For WebQSP, as shown in Table 4, the performance decreases by 18.7% and 14.16% with CodeLlama and DSC Ins, respectively.

2. *NLL loss term in RPO*: We omit the NLL loss term i.e., $\mathcal{L}_{NLL}$ in $\mathcal{L}_{RPO}$ to observe the performance of PO-KGQA. Note that this is equivalent to Self-rewarding baseline. As shown in Table 2 (**Self-rewarding baseline**), for KQA Pro, the performance decreases by 25% and 12.34%. For WebQSP, as shown in Table 4, the performance decreases by 12.75% and 32% with CodeLlama and DSC Ins, respectively. This shows the importance of $\mathcal{L}_{NLL}$ as it focuses on learning the KoPL Python steps closer to the ground truth.

3. *KG Grounding in Verifier*: We omit KG

grounding in Verifier to observe the performance of PO-KGQA. In replacement, the generated KoPL Python steps are translated directly to the standard KoPL format and executed on the KG to obtain the answers. As shown in Table 2 (**-Verifier configuration**), for KQA Pro, performance decreases by 7% and 6.6%. For WebQSP, as shown in Table 4, the performance decreases by 52.16% and 55.44% with CodeLlama and DSC Ins, respectively.
Note that, WebQSP relies on Freebase, which utilizes more semantically complex relation identifiers compared to the more intuitive and standardized naming conventions of Wikidata on which KQA Pro is based on. Therefore, a huge drop in the performance is observed for WebQSP as compared to KQA Pro as models struggle to correctly map natural language to Freebase identifiers.

5.4 Observations

• On KQA Pro, 77.58% (CodeLlama) and 84.33% (DSC Ins) of correct answers were generated by programs that differed from the ground-truth programs but were still valid. Similarly, on WebQSP, the corresponding numbers are 70.81% and 80.31%. These results highlight the importance of incorporating KoPL steps during fine-tuning, rather than relying solely on final answers as in prior work.

• Structural metrics (e.g., tree-based edit distance) measure syntactic similarity but fail to capture semantic correctness. Programs that are structurally similar can still produce incorrect results, while semantically equivalent programs may differ in structure. Since KGQA tasks often admit multiple valid reasoning programs, such metrics can be misleading. The results above further suggest that structural metrics may underestimate true correctness, and are therefore not used in evaluation.

• KQA Pro consists of: multi-hop, comparison, logical, count, verify, zero-shot, and qualifier questions. The zero-shot category is **OOD**, with PO-KGQA performing 72.38%.

• **Statistical Significance**: We conducted a paired t-test against the closest baseline (SFT) on 1000 test samples over 100 random runs. A p-value of 0.0315 (< 0.05) was obtained indicating that the improvements are statistically significant.

• PO-KGQA can be used across a spectrum of resource settings, from low to high. However, we focus on the challenging low-resource setting, where it demonstrates strong performance, highlighting its robustness and practical value.

• PO-KGQA applies to logical forms such as SPARQL, S-expression by transforming them into KoPL[15].

5.5 Error Analysis

We analyze the following sources of error:

1. *Entity Linking*: The verifier's incorrect linking of entities and concepts is observed only in KQA Pro, for 1.43% of the questions. No such issues are observed in WebQSP.

2. *Syntax Errors*: After fine-tuning the syntax errors in the KoPL python steps generated are seen only for KQA Pro, for 15% of questions. No such errors are seen for MetaQA and WebQSP

3. *Incorrect KG Grounding by Verifier*: The wrong answers obtained due to incorrect grounding of KG components (other than entities and concepts) by the Verifier are observed in KQA Pro, for 11% and WebQSP, for 13% of the questions.

5.6 Compute Resources Utilized

To reiterate, we used 2 NVIDIA V100 GPU with 32 GB RAM for training and inference. On average, it consumes the following GPU hours:

1. Training: 1 hours for each iteration; Inferencing: 4.2s for each KQA Pro question.

2. Training: 40 min for each iteration; Inferencing: 3.5s for each question of WebQSP.

6 Conclusion

We propose a novel framework, PO-KGQA: An SLM-based preference optimization framework to align preferences of logical form generation for the complex KGQA task. We generate dynamic preference data pairs of CoS-like reasoning steps in each iteration without depending on any teacher LLM. We modify the iterative RPO loss to generate preferred KoPL as Python code by eliminating dependency on the final answer. Our extensive experiments on complex benchmarks such as KQA Pro demonstrate that PO-KGQA surpasses all other alignment and few-shot techniques. It even surpasses most of the fully supervised state-of-the-art baselines, setting a new benchmark for various complex KGQA datasets. To the best of our knowledge, our work is the first to optimize SLMs to generate a preferred logical form for the KGQA task. Our approach is interpretable and applicable to any symbolic logical form, independent of underlying LLM knowledge about logical forms.

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

7 Limitations

There exist various alignment techniques, such as Identity Preference Optimization (IPO) (Azar et al., 2023) and Kahneman-Tversky Optimization (KTO) (Ethayarajh et al., 2024) that are designed for classification or ranking tasks. On the other hand, KGQA requires structured output generation (KoPL logical form), which these methods are not readily equipped to handle. One of the limitations of our work is that we did not explore how IPO and KTO can be used for KoPL logical form generation. Another limitation is that we use binary rewards while creating preference pairs. Instead, we could define a reward based on the number of correct vs. incorrect steps in a KoPL Python code and create multiple preferred data points.

8 Risks

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

References

2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Prerna Agarwal, Nishant Kumar, and Srikanta Bedathur. 2024. 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.

Prerna Agarwal, Nishant Kumar, and Srikanta Bedathur Jagannath. 2025. Aligning complex knowledge graph question answering as knowledge-aware constrained code generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 3952–3978, Abu Dhabi, UAE. Association for Computational Linguistics.

Mohammad Gheshlaghi Azar, Mark Rowland, Bilal Piot, Daniel Guo, Daniele Calandriello, Michal Valko, and Rémi Munos. 2023. A general theoretical paradigm to understand learning from human preferences. *Preprint*, arXiv:2310.12036.

Jinheon Baek, Alham Fikri Aji, and Amir Saffari. 2023. Knowledge-augmented language model prompting for zero-shot knowledge graph question answering. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 78–106, Toronto, Canada. Association for Computational Linguistics.

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.

Edith Cohen, Graham Cormode, and Nick Duffield. 2011. Structure-aware sampling on data streams. *SIGMETRICS Perform. Eval. Rev.*, 39(1):157–168.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng

Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

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.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. Kto: Model alignment as prospect theoretic optimization. *Preprint*, arXiv:2402.01306.

Haishuo Fang, Xiaodan Zhu, and Iryna Gurevych. 2024. DARA: Decomposition-alignment-reasoning autonomous language agent for question answering over knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3406–3432, Bangkok, Thailand. Association for Computational Linguistics.

Tengfei Feng and Liang He. 2025. RGR-KBQA: Generating logical forms for question answering using knowledge-graph-enhanced large language model. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 3057–3070, Abu Dhabi, UAE. Association for Computational Linguistics.

Hippolyte Gisserot-Boukhlef, Ricardo Rei, Emmanuel Malherbe, Céline Hudelot, Pierre Colombo, and Nuno M. Guerreiro. 2024. Is preference alignment always the best option to enhance LLM-based translation? an empirical analysis. In *Proceedings of the Ninth Conference on Machine Translation*, pages 1373–1392, Miami, Florida, USA. 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. 2021. 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. *Preprint*, arXiv:2401.14196.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Hanxu Hu, Hongyuan Lu, Huajian Zhang, Yun-Ze Song, Wai Lam, and Yue Zhang. 2024. Chain-of-symbol prompting elicits planning in large langauge models. *Preprint*, arXiv:2305.10276.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Sungho Ko, Hyunjin Cho, Hyungjoo Chae, Jinyoung Yeo, and Dongha Lee. 2024. Evidence-focused fact summarization for knowledge-augmented zero-shot question answering. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10636–10651, Miami, Florida, USA. Association for Computational Linguistics.

Jian Li, Haojing Huang, Yujia Zhang, Pengfei Xu, Xi Chen, Rui Song, Lida Shi, Jingwen Wang, and Hao Xu. 2024a. Self-supervised preference optimization: Enhance your language model with preference degree awareness. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 14452–14466, Miami, Florida, USA. Association for Computational Linguistics.

Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. 2023. 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.

Yuetai Li, Xiang Yue, Zhangchen Xu, Fengqing Jiang, Luyao Niu, Bill Yuchen Lin, Bhaskar Ramasubramanian, and Radha Poovendran. 2025. Small models struggle to learn from strong reasoners. *Preprint*, arXiv:2502.12143.

Zhenyu Li, Sunqi Fan, Yu Gu, Xiuxing Li, Zhichao Duan, Bowen Dong, Ning Liu, and Jianyong Wang.

2024b. Flexkbqa: A flexible llm-powered framework for few-shot knowledge base question answering. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):18608–18616.

Yuanyuan Liang, Jianing Wang, Hanlun Zhu, Lei Wang, Weining Qian, and Yunshi Lan. 2023. Prompting large language models with chain-of-thought for few-shot knowledge base question generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4329–4343, Singapore. Association for Computational Linguistics.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

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.

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.

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.

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. *Preprint*, arXiv:2309.04695.

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.

Richard Yuanzhe Pang, Weizhe Yuan, Kyunghyun Cho, He He, Sainbayar Sukhbaatar, and Jason Weston. 2024. Iterative reasoning preference optimization. *Preprint*, arXiv:2404.19733.

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.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: your language model is secretly a reward model. NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

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. *Preprint*, arXiv:2308.12950.

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.

Megh Thakkar, Quentin Fournier, Matthew Riemer, Pin-Yu Chen, Amal Zouaq, Payel Das, and Sarath Chandar. 2024. A deep dive into the trade-offs of

parameter-efficient preference alignment techniques. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5732–5745, Bangkok, Thailand. Association for Computational Linguistics.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Tianduo Wang, Shichen Li, and Wei Lu. 2024. Self-training with direct preference optimization improves chain-of-thought reasoning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11917–11928, Bangkok, Thailand. 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. *Preprint*, arXiv:2201.11903.

Yike Wu, Yi Huang, Nan Hu, Yuncheng Hua, Guilin Qi, Jiaoyan Chen, and Jeff Z. Pan. 2024. CoTKR: Chain-of-thought enhanced knowledge rewriting for complex knowledge graph question answering. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3501–3520, Miami, Florida, USA. Association for Computational Linguistics.

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.

Haoran Xu, Amr Sharaf, Yunmo Chen, Weiting Tan, Lingfeng Shen, Benjamin Van Durme, Kenton Murray, and Young Jin Kim. 2024. Contrastive preference optimization: pushing the boundaries of llm performance in machine translation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

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. *Preprint*, arXiv:2210.00063.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. 2022. Star: self-taught reasoner bootstrapping reasoning with reasoning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

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.

Ruizhe Zhang, Yongxin Xu, Yuzhen Xiao, Runchuan Zhu, Xinke Jiang, Xu Chu, Junfeng Zhao, and Yasha Wang. 2025. KnowPO: Knowledge-aware preference optimization for controllable knowledge selection in retrieval-augmented language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25895–25903. The Association for the Advancement of Artificial Intelligence.

Yichi Zhang, Zhuo Chen, Yin Fang, Yanxi Lu, Li Fangming, Wen Zhang, and Huajun Chen. 2024a. Knowledgeable preference alignment for LLMs in domain-specific question answering. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 891–904, Bangkok, Thailand. Association for Computational Linguistics.

Zhiqiang Zhang, Liqiang Wen, and Wen Zhao. 2024b. A gail fine-tuned llm enhanced framework for low-resource knowledge graph question answering. CIKM '24, page 3300–3309, New York, NY, USA. Association for Computing Machinery.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, benjamin negrevergne, and Gabriel Synnaeve. 2025. What makes large language models reason in (multi-turn) code generation? In *The Thirteenth International Conference on Learning Representations*.

A Appendix

A.1 Prompt used for PO-KGQA

The prompt used in PO-KGQA and all alignment baselines is shown below. It consists of set of instructions followed by python equivalent function definitions of KoPL functions.

A.2 Dataset Comparison

A.2.1 Comparison with GrailQA

- **Structural Analysis** reveals that KQA Pro is harder and well-rounded with questions requiring upto 10-hops of reasoning, while GrailQA is biased towards 1-hop queries. The stats (in %) in Table 5 are based on (Cao et al., 2022; Dutt et al., 2023).

Num hops	GrailQA	KQA Pro
1	77.9	26.3
2	21.5	23.8
3	0.6	38.8
4-10	0	11.2

Table 5: Structural Analysis

- **Complexity Dimensions**: The stats (in %) in Table 6 are shown on additional complexity dimensions based on (Dutt et al., 2023).

Dimension	GrailQA	KQA Pro
NRQ	16.2	45.1
Canonical logical form	4330	45563
Struct	22	135

Table 6: Complexity Dimensions

Where, NRQ - Numerical Reasoning Questions;
Canonical logical form - Number of distinct SPARQL templates;
Struct - Number of distinct skeleton SPARQL structures

- **KG analysis**: KQA Pro is based on a much larger, widely used Wikidata while GrailQA is based on Freebase. Our results on WebQSP, also Freebase-based with up to 2-hop questions, indicate similarities between GrailQA and WebQSP, except in scale. Notably, WebQSP exhibits a more complex structure, ISO-4 (Isomorphisms). Stats (in %) in Table 7 are from (Dutt et al., 2023).

Iso-Code	GrailQA	WebQSP
ISO-0	77.9	43.2
ISO-1	15.5	31.2
ISO-2	3.8	1.1
ISO-3	0.5	0.5
ISO-4	1.7	24

Table 7: KG analysis

A.2.2 Comparison with LC-Quad2.0

We chose KQA Pro over LC-Quad2.0 dataset for our experimentation because of the following reasons:

- The questions in this dataset are template-based. Whereas, the questions in other datasets such as KQA Pro and WebQSP are not templatized and resemble more real-world complex questions.
- Most of the questions in LC-Quad2.0 do not involve complexities such as KG concepts, literals, etc. Whereas most of the questions in KQA Pro involve most of the KG complexities that demand joint compositional and numerical reasoning.
- The number of questions in test/val set of LC-Quad2.0 is almost 4 times less than the number of questions in test/val set of KQA Pro.

Therefore, to demonstrate the robustness of our approach on harder and larger complex benchmarks we chose KQA Pro over LC-Quad2.0.

A.3 Baselines

- **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 technique 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.

- **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.

- **FlexKBQA** (Li et al., 2024b) 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.

- **Pangu** (Gu et al., 2023) is a recent state-of-the-art 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., 2023) 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** is an in-context learning-based baseline we implement for KQA Pro using GPT-3.5 Turbo. As there are no experimental results of Pangu and KB_BINDER on KQA Pro, we use LLM-ICL as an alternative. The original implementation is using Codex model. Codex model is now deprecated. 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.

- **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 just like answers during generation, 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., 2024) 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., 2024) 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.

- **KB-Coder** (Nie et al., 2024) performs code-style S-expression generation. It retrieves similar relation for a given question and add it in the prompt during logical form generation.

- **CodeAlignKGQA** (Agarwal et al., 2025) performs code-style KoPL generation. It retrieves abstract facts relevant to the question in an unsupervised manner. These facts are then used to generate few-shot KoPL generation. Dynamic code-correction performs code-correction based on program similarity for any incorrect generation.

- **InteractiveKBQA** (Xiong et al., 2024) generates logical forms through direct interaction with KG. They use examples to guide LLMs through the reasoning processes. They deconstruct the question into sub-query triples but, this step is manual and hence, not scalable.

- **KAPING** (Baek et al., 2023) Knowledge Augmented language model PromptING (KAPING) framework first retrieves the top-K similar triples to the question with the knowledge retriever, and then augments them as the form of the prompt.

- **SFT** performs vanilla fine-tuning using LoRA with Prompt P appended with input question and output as KoPL python steps. This baseline is implemented with same hyper-parameters as used for PO-KGQA.

- **Self-rewarding LLMs** (Yuan et al., 2024) performs Iterative DPO fine-tuning to optimize the final answer to match with the preferred answer. This baseline is implemented with same hyper-parameters as used for PO-KGQA.

- **STaR** (Zelikman et al., 2022) first perform rationale generation. Rationales are then filtered using ground truth answers. Fine-tuning is then performed on the filtered rationales using SFT. This process is repeated for T iterations. This baseline is implemented with same hyper-parameters as used for PO-KGQA.

- **ST-DPO** (Wang et al., 2024) first performs SFT on the generated preference data. Then an alternate of DPO and SFT loss is applied to update the model in an iterative manner. This baseline is implemented with same hyper-parameters as used for PO-KGQA.

A.4 Structure-Aware Logarithmic Stratified Sampling

1. **Sequence Extraction & Counting**
   Let $\mathcal{P} = \{p_i\}_{i=1}^{M}$ be the ordered KoPL function sequences for each QA pair. For each unique sequence s, compute its frequency

$$f_s \;=\; \left|\{\, i : p_i = s\}\right|.$$

2. **Logarithmic Weight Computation**
   Define a normalized weight for each sequence:

$$w_s \;=\; \frac{\ln(f_s + 1)}{\sum_{t \in \mathcal{S}} \ln(f_t + 1)}$$

   where, S is the set of all unique sequences. Allocate an initial sample count via

$$a_s \;=\; \lceil\, w_s\, N \,\rceil, \quad N = 1000.$$

   $N = 1000$ is chosen as a practical balance between the diversity of samples and the cost of the verifier. Experiments with smaller/larger values showed the same trends, confirming that performance is not tied to a single number.

3. **Total Alignment**
   Let,

$$T \;=\; \sum_{s \in \mathcal{S}} a_s.$$

   - If $T > N$, decrement the largest a_s (while maintaining $a_s \geq 1$) until $\sum_s a_s = N$.
   - If $T < N$, increment the largest a_s until $\sum_s a_s = N$.

4. **Stratified Random Sampling**
   For each sequence s, uniformly sample a_s distinct QA pairs from the set of all examples with sequence s.

Guarantees

- Coverage: Ensures $a_s \geq 1$ for all s, so every structure is represented.

- Balance: Logarithmic scaling prevents very frequent sequences from dominating.

```
'''
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
```