# Predicate-Guided Generation for Mathematical Reasoning

**Jiajun Chen** and **Yik-Cheung Tam**
Center for Data Science
New York University Shanghai
{jc11815,yt2267}@nyu.edu

## Abstract

We present Prolog-MATH, a curated corpus designed to support mathematical reasoning in large language models (LLMs) through logic programming. Each verbal math problem in the dataset is paired with a chain-of-thought explanation to generate Prolog program via a two-stage automated pipeline. In the first stage, an LLM (e.g., Deepseek-V3) predicts a set of relevant mathematical predicates that could be useful in solving the problem. In the second stage, the LLM uses these suggested predicates along with the expected answer type to generate a complete Prolog program. To improve coverage, we fine-tune an open-source LLM using supervised fine-tuning, followed by GRPO (Group Relative Policy Optimization) training to address problems that Deepseek-V3 fails to solve. To support this training, we propose a predicate-aware reward function that evaluates how well the generated solution incorporates the suggested predicates, complementing the standard binary reward. Experimental results show that: 1) Our two-stage pipeline achieves 81.3% solution coverage on the MATH training set; 2) GRPO training with the predicate-aware reward function enables a series of base models to correctly solve additional problems missed by Deepseek-V3, further increasing solution coverage to 97.4%. Data and source code can be obtained at the Github repository[1].

## 1 Introduction

Recently, Large Language Models (LLMs) have achieved strong performance on math word problems by generating structured reasoning traces. Popular paradigms include Chain-of-Thought (CoT) (Wei et al., 2022; Shao et al., 2024), Tree-of-Thought (Yao et al., 2023), Program-of-Thought (Chen et al., 2023), and Graph-of-Thought (Besta et al., 2023). Among these, CoT remains dominant due to its simplicity and interpretability: each step is verbalized in natural language. However, such traces lack formal semantics, are not executable, and provide no reliable mechanism for verifying intermediate steps.

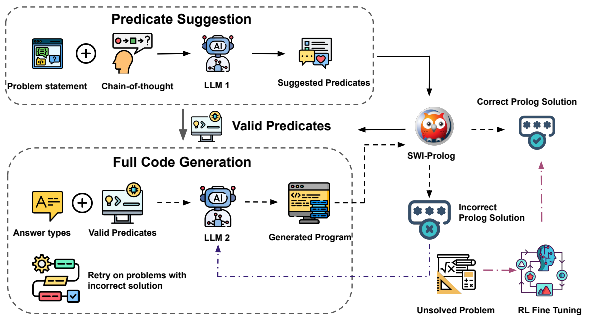

Figure 1: A two-stage predicate-guided pipeline followed by GRPO training for Prolog-MATH corpus curation.

One alternative is to generate executable programs using languages like Python and SymPy (Gou et al., 2023) to enable symbolic computation. While these systems support partial symbolic evaluation, Python's imperative and control-flow-centric nature makes it ill-suited for the declarative, compositional reasoning required in symbolic domains. This motivates the need for a representation language that natively supports logical inference, execution, and formal verification.

To address these limitations, we adopt **Prolog** as a structured representation language for mathematical reasoning, building on recent advances in symbolic problem solving (Yang et al., 2024; Chen and Tam, 2024). In this framework, solving a math problem involves constructing a Prolog program that satisfies a set of logical constraints derived from the problem statement and yields the correct final answer. Each constraint is encoded as a predicate, capturing atomic reasoning steps that can be composed into complete programs and reused across structurally similar problems. Beyond symbolic interpretability, Prolog also enables

[1]https://github.com/Tinyyhope/Prolog-MATH

effective integration with reinforcement learning objectives. We introduce a *predicate-aware* reward function that assigns feedback based on both the final correctness of a program and the structural quality of its predicates—capturing aspects such as redundancy, canonicality, and symbolic alignment. Unlike step-wise or token-level rewards (Lightman et al., 2024; Wang et al., 2024), our function operates at the program level, providing interpretable credit assignment without reliance on human preference labels. This design aligns naturally with frameworks like GRPO with rule-based rewards (Shao et al., 2024).

To scale symbolic solution construction, we design a two-stage pipeline that annotates math problems with executable Prolog programs. Stage one predicts problem-specific predicates based on mathematical structure; stage two composes full programs using these predicates and reusable background knowledge. For failures or invalid logic, we apply GRPO with a predicate-aware reward to refine or recover solutions. This yields the **Prolog-MATH** dataset—a collection of verifiable Prolog programs spanning diverse mathematical domains.

Our contributions are: 1) We curate an open-source dataset, **Prolog-MATH**, which provides symbolic solutions to math problems through a two-stage automated pipeline: predicate suggestion and program assembly using reusable predicates. 2) We extend symbolic supervision to diverse answer types, including complex numbers, equations, intervals, and matrices whereas prior datasets are limited to flat numeric outputs. 3) We integrate GRPO with a predicate-aware reward function that reflects both semantic validity and predicate structure, enabling fine-grained feedback and recovery from failed generations.

## 2 Prolog-MATH Corpus

**Problem Setting.** We use the chain-of-thought (CoT) question–answer pairs from the MATH dataset (Hendrycks et al., 2021) as our backbone, which are released under the MIT License. Let $\mathcal{X}$ be the set of math word problems and $\mathcal{T}$ the space of natural language reasoning traces (CoTs). Each input is a pair $(x, t)$ with $x \in \mathcal{X}$ and $t \in \mathcal{T}$. We aim to learn a function $f : \mathcal{X} \times \mathcal{T} \to \mathcal{C}$ that maps $(x, t)$ to an executable Prolog program $f(x, t) \in \mathcal{C}$, which yields a symbolic answer $\hat{y} \in \mathcal{A}$ upon execution.

Unlike prior work (Yang et al., 2024; Chen and Tam, 2024) that limits $\mathcal{A}$ to scalar outputs, we consider a broader space of structured forms such as vectors, intervals, equations, and matrices. This generalization introduces challenges in program synthesis and verification, requiring semantic alignment between informal CoTs and symbolic programs, as well as structural correctness in $\hat{y}$.

**Predicate Generation.** For each math problem $(x, t) \in \mathcal{X} \times \mathcal{T}$, we construct a symbolic Prolog program $f(x, t)$ in two stages. In **Stage 1**, a language model $\phi_{\text{pred}}$ (e.g. Deepseek-V3) proposes a set of predicate definitions $\mathcal{P}_{\text{gen}}(x)$ based on $(x, t)$ and a seed set $\mathcal{P}_{\text{can}}$ of predefined operators: $\mathcal{P}_{\text{gen}}(x) \leftarrow \phi_{\text{pred}}(x, t, \mathcal{P}_{\text{can}})$.

Each validated predicate is stored in a shared buffer with its definition and usage example. Predicate names are injected into subsequent prompts to encourage reuse. If a predicate is invoked across multiple problems, it is promoted to the canonical set $\mathcal{P}_{\text{can}}$. To reduce prompt length, only predicate names and listed definitions are materialized on demand.

**Program Generation.** In **Stage 2**, we synthesize a complete Prolog program $f(x, t)$ for each problem $x \in \mathcal{X}$ using a separate language model $\phi_{\text{code}}$. The model is conditioned on the problem statement $x$, its Chain-of-Thought trace $t$, the aggregated predicate set $\mathcal{P}(x)$ from Stage 1 (including both canonical and problem-specific predicates), and a list of candidate symbolic answer types $\mathcal{L}_{\mathcal{A}}$ drawn from the full type space $\mathcal{A}$. This list serves as a structural prior to guide symbolic program synthesis. The resulting program is generated as: $f(x, t) \leftarrow \phi_{\text{code}}(x, t, \mathcal{P}(x), \mathcal{L}_{\mathcal{A}})$. The program is executed using SWI-Prolog to obtain a symbolic prediction $\hat{y}$, which is compared against the gold solution $y$ using a symbolic equivalence checker adapted from MATH-VERIFY[2].

To improve symbolic coverage and correctness, we implement an iterative retry mechanism. At each iteration $t$, we allow up to $K = 3$ decoding attempts for each failed instance in $\mathcal{D}^{(t)}_{\text{fail}}$, varying decoding parameters such as temperature (See Appendix F for all answer types).We denote by $\mathcal{F}(\phi_{\text{code}}, \mathcal{D})$ the set of newly verified programs obtained by reapplying $\phi_{\text{code}}$ to a failed problem set $\mathcal{D}$, followed by execution and symbolic verification. These verified programs are then merged into the corpus: $\mathcal{D}^{(t+1)} = \mathcal{D}^{(t)} \cup \mathcal{F}(\phi_{\text{code}}, \mathcal{D}^{(t)}_{\text{fail}})$. Refer

[2]https://github.com/huggingface/Math-Verify

to Algorithm A and Figure 1 for details.

**Output.** Our two-stage pipeline produces verified Prolog programs for 6,100 out of 7,500 MATH training problems, achieving 81.3% problem-level coverage while spanning 100% of symbolic answer types in $\mathcal{A}$. In Section 3, we further enhance the coverage using GRPO on problems failed by Deepseek-V3.

## 3 Corpus Coverage Improvement

A subset of structurally complex problems such as those involving nested predicates or matrix-valued outputs remain unsolved. To address this, we introduce a reinforcement learning (RL) framework based on GRPO (Shao et al., 2024), repurposed for symbolic corpus augmentation. We use GRPO to search for correct programs over previously failed problems. We propose a **predicate-aware reward function** that evaluates each candidate program along three axes: (1) symbolic equivalence to the gold answer, (2) unused singleton predicates structural validity via static analysis from interpreter, and (3) alignment with suggested predicates $\mathcal{P}_{\text{gen}}(x)$ from Stage 1. A perfect program that produces the correct answer and uses all predicates properly receives a score of 2.0. Programs with minor structural issues, —such as unused singleton predicates, —are assigned 1.9. If the output is incorrect but the program correctly invokes predicates from $\mathcal{P}_{\text{gen}}(x)$, a partial reward of 0.5 is granted to encourage compositional reuse. All other cases receive a reward of 0.0. This program-level reward provides holistic, interpretable credit assignment without relying on token-level supervision or human preferences. The full formalization is provided in Appendix E.

## 4 Experiments

### 4.1 Setup

**Dataset** All experiments were conducted on the Prolog-MATH dataset introduced in Section 2, containing initial 6100 math word problems with symbolic programs and structured answers. We constructed experimental splits of failed problems to compare effectiveness of reward functions.

**Training** We conducted two sets of experiments: For the **GRPO experiment**, we started from a LoRA-finetuned Qwen2.5-3B-Instruct model (rank=32), and train a symbolic policy over failed problems using either a binary reward

| Configuration | Verified Accuracy |
|---|---|
| Two-stage (full) with $\mathcal{P}(x)$ and $\mathcal{L}_{\mathcal{A}}$ | 68.2% |
| Two-stage w/o $\mathcal{L}_{\mathcal{A}}$ | 26.3% |
| One-stage w/o $\mathcal{P}(x)$ | 41.6% |

Table 1: Ablation accuracy of proposed two-stage pipeline on MATH-500 testset. $\mathcal{P}(x)$ denotes the use of modular predicates, and $\mathcal{L}_{\mathcal{A}}$ denotes the answer type prompt list.

(2.0 if correct, 0.0 otherwise) or our proposed predicate-aware reward (see Appendix E). We used LoRA (Hu et al., 2021) with 4-bit quantization and rank of 32. We use Unsloth to reduce VRAM usage (Daniel Han and team, 2023). All prompts and training details are provided in Appendix B.

**Evaluation** All symbolic outputs were executed using SWI-Prolog and compared to the gold answer using a symbolic verifier adapted from MATH-VERIFY. This verifier checks for symbolic equivalence via expression normalization, LaTeX parsing, and numerical tolerance. In the **GRPO experiment**, we evaluated *coverage*: the proportion of previously failed problems for which the model successfully generated a verified Prolog program. A prediction is counted as correct if the final executed output matches the gold answer under symbolic equivalence. Coverage is reported for both the binary reward and predicate-aware reward settings.

### 4.2 Results

**Two-stage pipeline is effective.** To assess the roles of predicate abstraction and symbolic answer type prompting, we compare three pipeline configurations. Table 1 shows results on MATH-500 with symbolic verification. The two-stage system achieves the highest verified accuracy, while removing answer type prompts often makes the LLM default to numerical types, causing substantial degradation. We also tested GPT-4o: its coverage (76.3%) is slightly below DeepSeek-V3, but it produced 460 novel solutions, highlighting strong generalizability. These findings confirm the broad effectiveness of the two-stage pipeline and suggest model choice can be adapted to task requirements.

| Reward Function | Coverage |
|---|---|
| binary | +7.8% |
| predicate-aware | +9.7% |

Table 2: Coverage improvement on failed problems from Deepseek-V3 via GRPO training on Qwen2.5-3B-Instruct at 2k training steps.

| Model | Coverage |
|---|---|
| DeepSeek-V3 | 81.3% |
| Qwen2.5-3B-Instruct | +9.7% |
| Llama-3.1-8B-Instruct | +14.9% |
| Qwen2.5-Math-7B-Instruct | +15.2% |
| DeepSeek-R1-Distill-Qwen-7B | +15.7% |
| Union | +16.1% |

Table 3: Coverage improvement after GRPO training with predicate-aware reward across different models.

**GRPO improves coverage on failed problems.** Table 2 shows the coverage of Deepseek-V3 failed problems during GRPO on First, it is encouraging to see that a small 3B model is able to produce correct solutions on Deepseek-V3 failed problems. Second, our proposed predicate-aware reward function yielded relative coverage improvement by 24% compared to a binary reward function. Moreover, the series of experiments showcase that our approach generalizes well, as it performs effectively on both large models and smaller, specialized models.

To evaluate scalability, we further applied our predicate-aware GRPO pipeline to a range of models, including **Qwen2.5-3B-Instruct**, **Qwen2.5-Math-7B-Instruct**, **Llama-3.1-8B-Instruct**, and **DeepSeek-R1-Distill-Qwen-7B**. As shown in Table 3, all models benefit from consistent coverage improvements, with larger models such as Qwen2.5-Math-7B-Instruct and DeepSeek-R1-Distill-Qwen-7B achieving the largest gains (+15.2% and +15.7%, respectively). Notably, even smaller or distilled models, such as Qwen2.5-3B-Instruct, achieve substantial improvement (+9.7%), demonstrating that the method scales effectively across different backbone sizes while remaining computationally efficient.

The reward curves in Figure 3 corroborate these results: larger and specialized models converged to higher reward levels, while all tested variants showed clear upward trends. This demonstrates that our predicate-aware reward consistently drives progress across model sizes. Moreover, combining multiple models yields the **Union ensemble**, which achieves the highest coverage improvement (+16.1%), highlighting complementary strengths. Overall, these findings confirm that our approach generalizes well across model families, allowing practitioners to balance coverage gains with computational cost. Notably, all experiments were run on 24GB RTX 4090 GPUs with minimal memory consumption, highlighting the scalability and accessibility of our method under limited hardware resources.

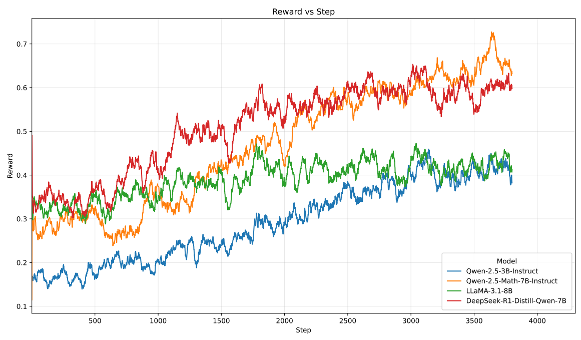

Figure 2: Reward comparison between different models using predicate-aware reward.

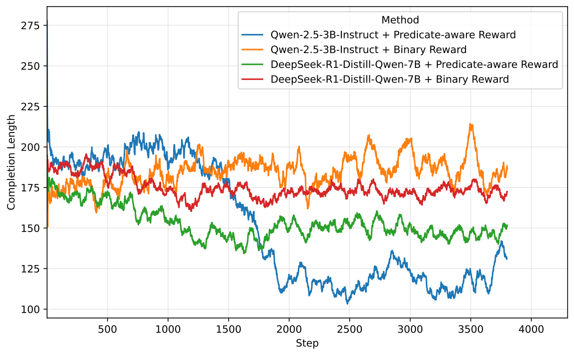

Figure 3: Completion length comparison between predicate-aware reward and binary reward.

**GRPO encourages efficient reasoning.** While improving coverage and correctness is essential, we also aim to encourage more concise solutions. As shown in Figure 2, models trained with our predicate-aware reward exhibit a clear downward trend in completion length, whereas the binary baseline remains relatively stable throughout training. For example, Qwen-3B with predicate-aware reward reduces its average completion length substantially after 1500 steps, converging to more concise outputs than its binary counterpart. A similar pattern is observed for DeepSeek. These results indicate that the predicate-aware reward not only improves coverage, but also promotes more efficient reasoning, demonstrating that conciseness can be achieved without sacrificing solution quality.

**Prolog-MATH dataset enhances supervised fine-tuning.** To further validate the effectiveness of our dataset, we compare models trained with Prolog-MATH against those trained with the original MATH dataset under the same configuration (`rank=32`, Qwen-3B). Both models were fine-tuned with SFT and evaluated on the standard `MATH-500` test set. The model trained on Prolog-MATH achieved an accuracy of 36.8%, outperforming the COT-trained counterpart at 33.2%. This demonstrates that our dataset provides more effective supervision, yielding consistent improvement under identical training conditions.

**Human Verification.** To ensure the reliability of our generated Prolog solutions, we conducted manual verification of both the two-stage pipeline and GRPO-generated solutions. For the two-stage pipeline, 96 out of 100 random problem samples from the MATH dataset produced mathematically correct and reasonable answers, with core predicates (e.g., `solve_quadratic/4`, `coprime/2`) applied appropriately in nearly all successful cases. For GRPO training, manual inspection of 100 generated solutions from Qwen2.5-3B-Instruct confirmed that 90% followed correct mathematical logic (Appendix G). These results indicate that errors are relatively rare and do not substantially affect overall dataset quality or evaluation outcomes.

## 5 Related Work

**Structured Reasoning** Chain-of-Thought (CoT) prompting (Wei et al., 2022) marked one of the earliest efforts to generate logical, step-by-step verbal reasoning in large language models. Building on this foundation, numerous advanced methods have emerged to further enhance reasoning capabilities (Zhou et al., 2023; Zhu et al., 2023; Huang et al., 2022; Liang et al., 2023). Despite these advances, most of these techniques rely heavily on natural language reasoning, which remains limited in its ability to detect and correct flaws in verbal logic. To address this limitation, researchers have explored structured approaches—such as trees and graphs—for reasoning. Graph-based representations have shown promising results, particularly in mathematical word problem solving (Zhang et al., 2020; Kipf and Welling, 2016). In addition, data augmentation techniques like forward and backward reasoning with variable and answer masking have been proposed to enrich CoT solutions (Yu et al., 2023; Jiang et al., 2024). Our work departs from natural language reasoning to embrace structured reasoning via logic programming.

**Tool-based Reasoning** Integrating external tools into large language models (LLMs) has proven effective in enhancing both their reasoning capabilities and interpretability (Cobbe et al., 2021; Mishra et al., 2023; Gou et al., 2023; Gao et al., 2023; Shao et al., 2023; Chen et al., 2023; Trinh et al., 2024). Among these tools, Prolog—a symbolic declarative language—stands out for its strength in symbolic reasoning. Its integration not only improves the logical consistency of natural language generation (Vakharia et al., 2024), but also bolsters arithmetic reasoning abilities in LLMs (Yang et al., 2024; Tan et al., 2024; Borazjanizadeh and Piantadosi, 2024). Our work employs a two-stage pipeline to suggest mathematical predicates and then generate prolog solutions for solving competition-level mathematical problems.

**Reward Design** Reward modeling plays a pivotal role in reinforcement learning. In mathematical reasoning, two common approaches have emerged: process-based and output-based reward models (Lightman et al., 2024; Wang et al., 2024). While both have demonstrated effectiveness, they typically rely on labeled datasets to train neural reward models. In contrast, rule-based reward functions offer a promising alternative, achieving strong performance without the need for labeled data (Shao et al., 2024). Building on this, our work adopts a rule-based framework and introduces a predicate-aware reward function, which enables partial reward assignment to Prolog-generated solutions aligned with suggested predicates of a given problem.

## 6 Conclusion

Our work introduces Prolog-MATH, a dataset specifically designed for tackling competition-level mathematical problems using declarative logic programming. With the proposed two-stage pipeline using the generated predicates as a guidance signal and GRPO for Prolog generation, we achieve 97.4% correct solution coverage in Prolog-MATH. GRPO training even on a small 3B LLM is proven to discover correct solutions on failed problems by Deepseek-V3. Our proposed predicate-aware reward function is effective in improving solution coverage compared to binary reward function used commonly in mathematical reasoning. We plan to apply the proposed pipeline on more mathematical corpora to enhance the presence of logic programming in mathematical reasoning and other reasoning tasks.

## Limitation

While our approach effectively curates a Prolog-MATH corpus tailored to competitive-level mathematical reasoning tasks with strong coverage, it does not yet achieve complete (100%) coverage. We also observe that large language models occasionally produce Prolog code containing syntax errors, which negatively impacts overall generation accuracy. Future work includes extending our approach to additional domains within the MATH corpus. Constrained decoding techniques (Lu et al., 2022; Geng et al., 2023) offer a promising direction to mitigate syntax errors in generated code. In particular, the VLLM decoding framework (Kwon et al., 2023), which supports grammar-constrained decoding via BNF specifications, appears especially promising. Lastly, our GRPO experiments have thus far been conducted on models of size 3B, 7B, and 8B; future work will explore the generalizability of our approach to other language models.

## Ethics Statement

This research adheres to the ACL Code of Ethics. Our work involves the automated generation and evaluation of symbolic reasoning data using large language models and a Prolog interpreter. All datasets used in this study are publicly available or derived from publicly accessible sources, and no personally identifiable or sensitive information is included. The generated Prolog-MATH corpus focuses on mathematical problem-solving and does not involve human subjects, crowdsourced annotations, or social or demographic data.

## References

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2023. Graph of thoughts: Solving elaborate problems with large language models.

Nasim Borazjanizadeh and Steven T. Piantadosi. 2024. Reliable reasoning beyond natural language.

Jiajun Chen and Yik-Cheung Tam. 2024. Enhancing mathematical reasoning in llms with background operators.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Tora: A tool-integrated reasoning agent for mathematical problem solving.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve.

Weisen Jiang, Han Shi, Longhui Yu, Zhengying Liu, Yu Zhang, Zhenguo Li, and James Kwok. 2024. Forward-backward reasoning in large language models for mathematical verification. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 6647–6661, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Thomas Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *ArXiv*, abs/1609.02907.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. 2023. Encouraging divergent thinking in large language models through multi-agent debate.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *International Conference on Learning Representations*.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. 2022. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States. Association for Computational Linguistics.

Swaroop Mishra, Matthew Finlayson, Pan Lu, Leonard Tang, Sean Welleck, Chitta Baral, Tanmay Rajpurohit, Oyvind Tafjord, Ashish Sabharwal, Peter Clark, and Ashwin Kalyan. 2023. Lila: A unified benchmark for mathematical reasoning.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Synthetic prompting: Generating chain-of-thought demonstrations for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 30706–30775. PMLR.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

Xiaoyu Tan, Yongxin Deng, Xihe Qiu, Weidi Xu, Chao Qu, Wei Chu, Yinghui Xu, and Yuan Qi. 2024. Thought-like-pro: Enhancing reasoning of large language models through self-driven prolog-based chain-of-thought.

Trieu Trinh, Yuhuai Wu, Quoc Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature*.

Priyesh Vakharia, Abigail Kufeldt, Max Meyers, Ian Lane, and Leilani H. Gilpin. 2024. Proslm: A prolog synergized language model for explainable domain specific knowledge based question answering. In *Neural-Symbolic Learning and Reasoning*, pages 291–304, Cham. Springer Nature Switzerland.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439, Bangkok, Thailand. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Xiaocheng Yang, Bingsen Chen, and Yik-Cheung Tam. 2024. Arithmetic reasoning with LLM: Prolog generation & permutation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 699–710, Mexico City, Mexico. Association for Computational Linguistics.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. In *International Conference on Learning Representations*.

Jipeng Zhang, Lei Wang, Roy Ka-Wei Lee, Yi Bin, Yan Wang, Jie Shao, and Ee-Peng Lim. 2020. Graph-to-tree learning for solving math word problems. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3928–3937, Online. Association for Computational Linguistics.

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.

Xinyu Zhu, Junjie Wang, Lin Zhang, Yuxiang Zhang, Yongfeng Huang, Ruyi Gan, Jiaxing Zhang, and Yujiu Yang. 2023. Solving math word problems via cooperative reasoning induced language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

## Appendix

## A Algorithm

**Algorithm 1:** Two-Stage Corpus Construction via Predicate Bootstrapping and Retry

```
Input  : Dataset of math problems and CoTs $\{(x_i, t_i)\}_{i=1}^{N}$, answer type space $\mathcal{A}$, LLMs
         $\phi_{\text{pred}}$, $\phi_{\text{code}}$, retry limit $K$, convergence threshold $\epsilon$
Output : Verified Prolog corpus $\mathcal{D}$
Initialize predicate buffer $\mathcal{B} \leftarrow \emptyset$          // for all generated predicate defs
Initialize canonical set $\mathcal{P}_{\text{can}} \leftarrow$ seed operators
Initialize corpus $\mathcal{D}^{(0)} \leftarrow \emptyset$
Stage 1: Predicate Extraction
foreach $(x, t) \in \mathcal{X} \times \mathcal{T}$ do
    $\mathcal{P}_{\text{gen}}(x) \leftarrow \phi_{\text{pred}}(x, t, \mathcal{P}_{\text{can}})$
    foreach $p \in \mathcal{P}_{gen}(x)$ do
        if SWI-Prolog validates $p$ then
            Add $p$ to buffer $\mathcal{B}$ with definition and usage
Stage 2: Program Generation and Retry
$t \leftarrow 0$, $\mathcal{T}^{(0)} \leftarrow \emptyset$                    // initial covered types
repeat
    $\mathcal{D}_{\text{fail}}^{(t)} \leftarrow \emptyset$
    foreach $(x, t_x) \in \mathcal{X} \times \mathcal{T}$ do
        Extract $\mathcal{P}_{\text{used}}(x)$ as names from buffer $\mathcal{B}$
        $\tau(x) \leftarrow$ predicted answer type from CoT $t_x$
        for $k \leftarrow 1$ to $K$ do
            $f(x) \leftarrow \phi_{\text{code}}(x, t_x, \mathcal{P}_{\text{used}}(x), \tau(x))$
            $\hat{y} \leftarrow \text{run}(f(x))$
            if verify$(\hat{y}, y)$ then
                $\mathcal{D}^{(t+1)} \leftarrow \mathcal{D}^{(t)} \cup \{(x, f(x), \hat{y})\}$
                Promote used predicates in $f(x)$ to $\mathcal{P}_{\text{can}}$
                break
        if no success then
            $\mathcal{D}_{\text{fail}}^{(t)} \leftarrow \mathcal{D}_{\text{fail}}^{(t)} \cup \{x\}$
    $\mathcal{T}^{(t+1)} \leftarrow$ symbolic answer types covered by $\mathcal{D}^{(t+1)}$
    $\Delta_{\text{domain}} \leftarrow |\mathcal{T}^{(t+1)} \setminus \mathcal{T}^{(t)}|$
    $t \leftarrow t + 1$
until $\Delta_{domain} < \epsilon$
return $\mathcal{D}^{(t)}$
```

## B Instruction Prompt

Below is the prompt we use for generation in Stage 1 Predicate Generation:

| Setting | Prompt Template |
|---|---|
| Predicate generation with reuse | You are a symbolic reasoning assistant. Define any new Prolog predicates needed to solve the problem below, and reuse the given background predicates when appropriate.<br>**Problem:**<br><Question><br>**Chain-of-Thought:**<br><CoT reasoning><br>**Predefined Predicates:**<br>« predicate_name(...) »<br>(repeat for multiple predefined predicates)<br>**Output:**<br>Define any new predicates in the same format:<br>predicate_name(Input1, Input2, ..., Output) :- Definition.<br>% Example: predicate_name(...).<br>Do not solve the problem. Do not include natural language. Only output the Prolog predicate definitions. |

Table 4: Prompt template for Stage 1 predicate generation with reuse. The model is encouraged to reuse provided background predicates and define additional ones if needed.

Below is the prompt for Stage 2 Prolog program generation:

| Setting | Prompt Template |
|---|---|
| Full (with answer type & suggested predicates) | Below is a Prolog generation prompt with all components.<br><br>You are a helpful Prolog programming assistant.<br><answer_type><br>Example: Provide an example problem and its complete Prolog program solution.<br>Target Problem: <problem><br>CoT Solution: <cot_solution><br>Suggested Predicates: <pred_text><br>Generate a complete executable Prolog program that solves the problem.<br>Do not include natural language.<br>`:- solve, halt.` |
| Without Answer Type | Below is the same template as above, but omitting the `<answer_type>` line. |
| Without Suggested Predicates | Below is the same template as above, but omitting the `Suggested Predicates: <pred_text>` line.<br>The model must generate all predicates from scratch, using only the problem and CoT. |

Table 5: Prompt templates for Stage 2 program generation. The full version includes both answer type hint and suggested predicates. Each ablation removes one component.

Below is the instruction prompt that we use for supervised fine-tuning in Section 4:

| Setting | Prompt Template |
|---|---|
| Prolog generation with predicates | Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.<br>### Instruction:<br>Please generate a prolog answer based on the given predicates to solve the given math problem.<br><background operators><br>### Input:<br><Question><br>### Output:<br><Prolog Code> |

Below is the prompt we used for GRPO training:

| Setting | Prompt Template |
|---|---|
| GRPO training prompt | Below is a system-user style prompt used for Prolog reward fine-tuning.<br>**System:**<br>You are a helpful Prolog programming assistant.<br>**User:**<br>&lt;problem&gt;<br>The final answer must be a valid Prolog term using one of the following formats:<br>&lt;answer type list&gt;<br>Use write(...) to output the final answer.<br>Do NOT include explanations or natural language.<br>Suggested predicates: generate necessary predicates to solve the problem. |

Table 6: Prompt template used in GRPO fine-tuning. The prompt follows a system-user format, includes answer type hints, and optionally suggested predicates.

## C Generated Predicates

Below, we provide a detailed overview of a subset of the predicates generated by our pipeline. We describe the input-output behavior of each operator, as well as the frequence of apperance.

| Operator/Predicate | Frequency |
|---|---|
| `coprime(A, B)`: Check if two numbers are coprime<br>Inputs: A (number), B (number)<br>Output: true/false | 42 |
| `number_of_diagonals(N, Diagonals)`: Calculate number of diagonals in an $N$-gon<br>Inputs: N (integer)<br>Output: Diagonals (integer) | 35 |
| `solve_quadratic(A, B, C, Roots)`: Solve $Ax^2 + Bx + C = 0$<br>Inputs: A, B, C (number)<br>Output: Roots (list) | 28 |
| `discriminant(A, B, C, D)`: Compute $D = B^2 - 4AC$<br>Inputs: A, B, C (number)<br>Output: D (number) | 27 |
| `fibonacci(N, Result)`: Compute the $N$-th Fibonacci number<br>Inputs: N (integer)<br>Output: Result (integer) | 27 |

## D Training Samples

| Question | Prolog Code |
|---|---|
| What is the sum of all values of $y$ for which the expression $\frac{y+6}{y^2-5y+4}$ is undefined? | quadratic_roots(A, B, C, Root1, Root2) :-<br>Discriminant is B*B - 4*A*C,<br>Root1 is (-B + sqrt(Discriminant)) / (2*A),<br>Root2 is (-B - sqrt(Discriminant)) / (2*A).<br><br>sum_of_roots(A, B, _, Sum) :-<br>Sum is -B / A.<br><br>solve :-<br>sum_of_roots(1, -5, 4, Sum),<br>format(" d", [Sum]).<br><br>:- solve, halt. |
| Find the distance between the points $(-5, -2)$ and $(7, 3)$. | distance((X1,Y1), (X2,Y2), D) :-<br>DX is X2 - X1,<br>DY is Y2 - Y1,<br>D is sqrt(DX * DX + DY * DY).<br><br>solve :-<br>distance((-5,-2), (7,3), D),<br>write(D), nl.<br><br>:- solve, halt. |
| For what values of $b$ is $-2$ not in the range of the function $f(x) = x^2 + bx + 2$? Express your answer in interval notation. | solve :-<br>    solve_b_range(BMin, BMax),<br>    write('The values of b are in the interval: ('),<br>    write(BMin), write(','), write(BMax), write(')'), nl.<br><br>:- solve, halt. |

## E Reward Function and GRPO Objective

### E.1 Predicate-Aware Reward Function

For each failed problem $x \in D_{\text{fail}}$, the policy $\pi_\theta$ samples $G$ candidate programs $\{f_i(x)\}_{i=1}^{G}$. Each program is executed using SWI-Prolog to produce an output $\hat{y}_i$, or fails with an error.

We define a program-level reward function $r(f_i(x)) \in [0, 2]$ that reflects both symbolic correctness and structural quality:

$$r(f_i(x)) = \begin{cases} 2.0, & \text{if } \hat{y}_i = y \text{ and } \delta_i = 0, \\ 1.9, & \text{if } \hat{y}_i = y \text{ and } \delta_i = 0.1, \\ 0.5, & \text{if } \hat{y}_i \neq y \text{ and } \psi_i = 1, \\ 0.0, & \text{otherwise.} \end{cases} \tag{1}$$

Here: $\delta_i \in \{0, 0.1\}$ is a structural penalty computed via static analysis (e.g., for unused singleton predicates); $\psi_i = 1$ indicates that all predicates from the suggested set $\mathcal{P}_{\text{gen}}(x)$ are correctly invoked.

This reward design encourages programs that are not only semantically correct but also compositionally structured and reusable.

### E.2 GRPO Objective for Symbolic Policy Optimization

We adopt a token-level GRPO objective, adapted to symbolic settings. Let $o_{i,t}$ be the $t$-th token of program $f_i(x)$, and let $\hat{A}_{i,t}$ denote the group-relative advantage at token $t$. The training objective is:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{x \sim \mathcal{X}_{\text{fail}}} \left[ \frac{1}{G} \sum_{i=1}^{G} \frac{1}{|f_i(x)|} \sum_{t=1}^{|f_i(x)|} \min\left( \rho_{i,t} \hat{A}_{i,t},\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon) \hat{A}_{i,t} \right) \right] - \beta \cdot \mathbb{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}] \tag{2}$$

where: $\rho_{i,t} = \frac{\pi_\theta(o_{i,t}|x, o_{i,<t})}{\pi_{\text{old}}(o_{i,t}|x, o_{i,<t})}$ is the importance weight; $\mathbb{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]$ denotes the KL divergence from a reference policy; $\epsilon$ is the PPO-style clipping threshold; $\beta$ is the KL regularization strength.

This formulation steers the policy toward generating structurally sound programs, while ensuring training stability via KL regularization and group-wise credit attribution.

## F Symbolic Answer Types

To enable structured verification and facilitate symbolic reasoning, we define a set of *symbolic answer types* that capture the semantics of mathematical outputs in a structured form. Each symbolic answer is expressed as a Prolog term with a clear compositional structure. These symbolic types allow us to uniformly represent numbers, expressions, intervals, equations, and other mathematical objects in a way that is both machine-interpretable and human-readable. The complete list of supported symbolic types is summarized in Table 8.

| Symbolic Type | Description and Example |
|---|---|
| `5, -3` | Plain number. Example: `5, -3` |
| `decimal(Value)` | Decimal number. Example: `decimal(2.5)` |
| `complex(Re, Im)` | Complex number. Example: `complex(1, -2)` $\rightarrow 1 - 2i$ |
| `frac(Numer, Denom)` | Fraction. Example: `frac(3, 4)` $\rightarrow \frac{3}{4}$ |
| `mixed(Whole, frac(N, D))` | Mixed number. Example: `mixed(2, frac(3,4))` $\rightarrow 2\frac{3}{4}$ |
| `eq(Variable, Value)` | Equation. Example: `eq(x, 5)` $\rightarrow x = 5$ |
| `expr(...)` | Symbolic expression. Example: `expr((1+sqrt(2))/2)` |
| `symbolic_constant` | Mathematical constant. Example: $\pi$ |
| `interval(A, B)` | Open interval. Example: `interval(1, 5)` $\rightarrow (1,\ 5)$ |
| `interval_union([...])` | Union of intervals. Example: `interval_union([interval(0,1), interval(2,3)])` |
| `solution([...])` | List of solutions. Example: `solution([1, 2])` |
| `plus_minus(A, sqrt(B))` | $\pm$ root. Example: `plus_minus(2, sqrt(3))` $\rightarrow 2 \pm \sqrt{3}$ |
| `sqrt(X)` | Square root. Example: `sqrt(5)` $\rightarrow \sqrt{5}$ |
| `base(Base, Digits)` | Base notation. Example: `base(2, 1010)` $\rightarrow 1010_2$ |
| `vector([...])` | Tuple/vector. Example: `vector([3, 4])` $\rightarrow (3,\ 4)$ |
| `matrix([[...], [...]])` | Matrix. Example: `matrix([[1,2],[3,4]])` |
| `trig(Function, Arg)` | Trig function. Example: `trig(sin, pi/6)` $\rightarrow \sin(\pi/6)$ |
| `formatted_number("...")` | Preformatted LaTeX string. |
| `unit(Value, UnitType)` | Number with unit. Example: `unit(40, percent)` $\rightarrow 40\%$ |
| `pi_expr(expr(...))` | Expression involving $\pi$. Example: `pi_expr(expr(pi/12))` |
| `signed_frac(frac(N, D))` | Signed fraction. Example: `signed_frac(frac(-3, 4))` |
| `paren_tuple(...)` | Parenthesized tuple. Example: `paren_tuple(3, 4)` $\rightarrow (3,\ 4)$ |

Table 8: Supported symbolic answer types in our framework, each representing a structured mathematical output.

## G Human Verification of Intermediate Steps

To assess the quality of the generated Prolog programs and to verify plausible but incorrect intermediate steps, we conducted a manual evaluation on two samples of 100 problems each.

### G.1 Verification of Two-stage Pipeline

We randomly sampled 100 problems from the MATH dataset, ensuring coverage across all domains: 20 algebra, 17 counting and probability, 14 geometry, 13 precalculus, 19 number theory, and 17 intermediate algebra questions.

Our analysis confirmed that **96%** of sampled problems produced mathematically correct and reasonable answers. In particular, we examined the use of core predicates (e.g., `solve_quadratic/4`, `coprime/2`), and found that 96% of successful solutions utilized these predicates appropriately.

Most errors occurred in number theory and geometry. In these cases, intermediate steps were plausible but not fully correct, often tracing back to imperfections in the CoT reference solutions. For instance, in a number theory problem asking:

> "Let $N$ be the greatest integer multiple of 8, no two of whose digits are the same. What is the remainder when $N$ is divided by 1000?"

The generated solution assumed that the last three digits must be an arrangement of {0,1,2}. Although this assumption was flawed, it coincidentally yielded the correct final result. Such issues highlight the limitations of imperfect CoT guidance. In future work, we plan to strengthen predicate design with formal logical constraints (e.g., digit constraints in number theory) to avoid invalid assumptions.

### G.2 Verification of GRPO-Generated Solutions

To further assess solution quality, we sampled 100 problems generated by `Qwen2.5-3B-Instruct` during GRPO training. Manual verification showed that **90%** of the generated programs aligned with correct mathematical logic.

The remaining 10% of cases contained characteristic errors. For example, consider the problem:

> "Compute the largest integer $k$ such that $2004^k$ divides 2004!."

The correct solution requires factorizing $2004 = 2^2 \cdot 3 \cdot 167$ and applying Legendre's formula to each prime factor, taking $k = \min(\lfloor v_2(2004!)/2 \rfloor, v_3(2004!), v_{167}(2004!)) = 12$. However, the GRPO-generated code only computed $v_{167}(2004!)$, ignoring the contributions of 2 and 3. While this coincidentally produced the correct final answer ($k = 12$), the reasoning was incomplete. This illustrates how some GRPO-generated programs exhibit plausible but logically insufficient intermediate steps.

### G.3 Conclusion

Overall, our verification confirms that plausible but incorrect intermediate steps are rare and do not significantly affect dataset quality. These evaluations also provide valuable guidance for future work, where we aim to further reduce such cases by introducing stronger logical constraints into predicate design.