SymBa: Symbolic Backward Chaining for Structured Natural Language Reasoning

Anonymous ACL submission

Abstract

While Large Language Models (LLMs) have demonstrated remarkable reasoning ability lately, providing a structured, explainable proof to ensure explainability, *i.e.* structured reasoning, still remains challenging. Among two directions of structured reasoning, we specifically focus on backward chaining, where the query is recursively decomposed to subgoals by applying inference rules. We point out that current popular backward chaining implementations (Least-to-most prompting and LAMBADA) fail to implement the necessary features of backward chaining, such as arbitrary-depth recursion and binding propagation. To this end, we propose a novel backward chaining framework, SymBa (Symbolic Backward Chaining). In SymBa, a symbolic solver controls the whole proof process, and an LLM searches for the relevant natural language premises and translates them into a symbolic form for the solver. By this LLM-solver integration, while producing a completely structured proof that is symbolically verified, SymBa achieves significant improvement in performance, proof accuracy, and efficiency in diverse structured reasoning benchmarks compared to baselines.

1 Introduction

Recently, large language models (LLMs) trained with massive amounts of natural language text have shown remarkable reasoning ability (Wei et al., 2022; Kojima et al., 2022, *inter alia.*). However, LLMs might generate inaccurate and ungrounded reasoning paths as the number of reasoning steps increases (Saparov and He, 2023). To simultaneously enhance the accuracy and explainability of generated proofs against complex problems, *structured reasoning*, where the model provides an explicit, well-structured reasoning path instead of rationales in free-form text, has been frequently explored as a solution (Creswell et al., 2023; Kazemi et al., 2023).

In general, strategies for reasoning can be typically divided into two categories, *forward chaining* and *backward chaining* (Poole and Mackworth, 2010). Forward chaining reasoners first collect the base facts and repeatedly derive a new fact using logical rules until it finally proves the user's query. In contrast, backward chaining reasoners start from the query and apply rules that decompose the query into a set of subgoals. These subgoals are recursively decomposed until they can be directly proved or refuted using the base facts.

In terms of structured reasoning, forward chaining methods require a tailored *planner* module that selects the most likely next reasoning step to prevent proof divergence (Sprague et al., 2023; Creswell et al., 2023; Yang et al., 2022). Consequently, these approaches suffer from severe performance drop at longer reasoning paths due to planning failure (Kazemi et al., 2023). In contrast, backward chaining methods are guaranteed to terminate, which removes the necessity for a planner.

However, we claim that current LLM-based backward chaining implementations do not fully implement the backward chaining algorithm, by omitting features like arbitrary-depth recursion and binding propagation (Section 3.1). These features, necessary for performing sound and accurate backward chaining in diverse settings, are well-defined and can be effectively handled with symbolic solvers.

To this end, we propose a novel framework, **SymBa** (Symbolic Backward Chaining), a modular backward chaining approach that integrates a symbolic solver with an LLM. In SymBa, the solver controls the entire reasoning process, and the LLM is instructed to generate a single reasoning step only when the solver fails to prove a subgoal. By interleaving the natural language sentences and corresponding symbolic representations, SymBa can leverage the natural language reasoning abilities of LLMs and the logical soundness provided

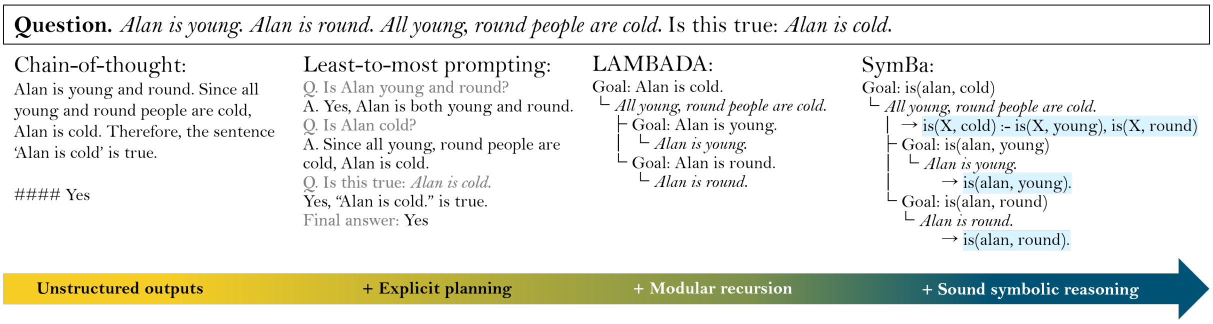

Figure 1: Brief comparison between natural language-based structured backward chaining methods and SymBa.

by the symbolic solver.

We directly compare the proposed method with LLM-based backward chaining baselines, Least-to-most prompting (Zhou et al., 2023) and LAMBADA (Kazemi et al., 2023), in seven diverse benchmarks that span over deductive, relational, and arithmetic reasoning. SymBa outperforms previous methods in terms of task performance, proof accuracy, and efficiency, while being able to provide a strictly structured proof in both symbolic and natural language forms[1].

2 Background

2.1 Logic programming

Logic programming is a programming paradigm based on formal logic. Generally, each statement of a logic program is expressed as a *rule*, which describes an implication relation between *terms* that have boolean truth values.

$$h \text{ :- } p_1, ..., p_n, \texttt{not } q_1, ..., \texttt{not } q_m. \qquad (1)$$

This rule denotes that when every *subgoal* terms p_i and $\texttt{not } q_j$ are true, the *head* term h is also proven true. A rule with an empty body, a *fact*, expresses that the head term h is unconditionally true.

For instance, consider the logic program in Equation 2. The terms $\texttt{dad}(\texttt{alan}, \texttt{carl})$ and $\texttt{dad}(\texttt{carl}, \texttt{bill})$ are true by the corresponding facts. When we substitute variables of Rule1 (*i.e.* *bind*) using the binding $\{A/\texttt{alan}, B/\texttt{bill}, C/\texttt{carl}\}$, all subgoals become identical to already proved terms, so the respective bound head $\texttt{granddad}(\texttt{alan}, \texttt{bill})$ is also deduced as true.

$$\begin{aligned} &\text{Rule1. } \texttt{granddad}(A, B) \text{ :- } \texttt{dad}(A, C), \texttt{dad}(C, B). \\ &\text{Fact1. } \texttt{dad}(\texttt{alan}, \texttt{carl}) \text{ :-.} \qquad (2) \\ &\text{Fact2. } \texttt{dad}(\texttt{carl}, \texttt{bill}) \text{ :-.} \end{aligned}$$

2.2 Backward chaining solver

Backward chaining solvers (top-down solvers) are logic program interpreters that start from the query term and recursively apply rules until the proof is complete. When a user provides a query term, the solver searches through the *database* for symbolic rules and facts that might prove the query. A rule or a fact can prove the query only if there exists a binding that can make the query and the head identical, *i.e.* the query and the head *unify*. If a rule that unifies with the query is found, the solver recursively proves each subgoal. When all subgoals are successfully proven true, the query is also proved.

Consider the logic program in Equation 2. If the query is given as $\texttt{granddad}(\texttt{alan}, \texttt{bill})$, the only statement that has a unifying head is Rule1. To make the rule head and query identical, we apply the binding $\{A/\texttt{alan}, B/\texttt{bill}\}$ to Rule1, obtaining two subgoals $\texttt{dad}(\texttt{alan}, C)$ and $\texttt{dad}(C, \texttt{bill})$. The first subgoal can be proved by binding $C/\texttt{carl}$. Subsequently, the binding is dynamically propagated to the following subgoals (*i.e. binding propagation*), in this case updating the second subgoal to $\texttt{dad}(\texttt{carl}, \texttt{bill})$. As this is also true, it can be concluded that the original query is proven.

3 Methods

3.1 Baselines

We select two popular natural language-based backward chaining methods as our baseline, namely **Least-to-most prompting** (Zhou et al., 2023) and **LAMBADA** (Kazemi et al., 2023).

[1]We publicly disclose our implementation of baselines and SymBa, test data, prompts, and anything necessary to reproduce this study in the following repository.

Least-to-most prompting is a two-stage task decomposition method. In the initial *Decompose* stage, the LLM is instructed to decompose the given question into sub-questions and order them from least complicated to most. The questions are passed to the *Solution* stage, where each question is answered in an incremental order. This process can be seen as explicitly planning the proof's structure first and executing the plan during the actual reasoning later.

While having more structure in its proof compared to Chain-of-thought reasoning, as Least-to-most prompting performs decomposition only once, it is required to predict the total ordering of sub-questions in a single run, which is challenging especially when there exist multiple potential reasoning paths (Patel et al., 2022). We further examine the proof accuracy problem of Least-to-most prompting in Section 5.2.

LAMBADA implements a modular backward chaining approach that operates on pure natural language. When given a query, it tests all facts and rules against the query to find out which might apply[2]. If a matching fact is retrieved, it stops recursion. If any rules are retrieved, they are then bound and decomposed into subgoals. Finally, it is ensured that the rule and the query have the same negation status.

While LAMBADA overcomes the limitation of Least-to-most prompting by allowing an arbitrary decomposition depth, LAMBADA's capability is severely limited due to the lack of binding propagation. As binding propagation is necessary for operations like coreferencing between subgoals (illustrated in Equation 2) or returning a value, LAMBADA is inherently incapable of various types of reasoning including relational reasoning with bridging entities (Sinha et al., 2019; Yang et al., 2018) and arithmetic reasoning (Cobbe et al., 2021). Besides the binding propagation problem, we find LAMBADA to be highly inefficient compared to other methods (Section 5.3).

3.2 Proposed method

3.2.1 Symbolic Backward Chaining

To overcome the limitations of previously proposed methods, we propose **SymBa** (Symbolic Backward Chaining), which integrates a symbolic backward chaining solver and an LLM for natural language reasoning.

The workflow of SymBa is briefly illustrated in Figure 2. A symbolic solver is capable of deducing a query if the solver's database includes every necessary statement. However, when the relevant context is only given in natural language, the database is initially empty, automatically failing to prove the query. To make progress, the solver calls the LLM to check if the failed query can be entailed from the natural language context. The LLM then generates a statement that unifies with the subgoal, and the solver retries proving the failed subgoal with the updated database. The process is continued until the original query is proved, or every possible reasoning path fails. Appendix A includes a formal, detailed description of SymBa's mechanism.

Delegating proof control to a symbolic solver has numerous benefits. Most importantly, symbolic solvers algorithmically produce sound and formally verified proofs. We compare the proof accuracy to baselines in Section 5.2. Furthermore, SymBa can handle tasks like relational reasoning and mathematical reasoning that LAMBADA fails to address by leveraging the solver's in-built binding propagation. Finally, solver operations are computationally efficient compared to neural network inferences. By performing operations like goal decomposition and binding propagation with symbols, SymBa is significantly efficient compared to natural language-based backward chaining methods (Section 5.3).

3.2.2 Single-step statement generation

In SymBa, the LLM is instructed to generate a logic program statement from the context that might prove the current subgoal. Similar to previous works on structured reasoning that adopt modular strategy (Creswell et al., 2023; Kazemi et al., 2023), we divide the single-step statement generation process into five modules: Fact/Rule Search, Fact/Rule Translation, and Symbolic Validation (Figure 3).

Fact/Rule Search In the first stage, the LLM is prompted with the symbolic query and the context, and is instructed to generate a description of a reasoning step that might prove the query in natural language.

Fact/Rule Translation Subsequently, the LLM is given the query and the description of the backward chaining step (obtained from the Search module) and generates a symbolic statement. Complet-

[2] While the original paper requires classification of each sentence as either fact or rule before the actual reasoning, we do not follow their implementation to ensure a fair comparison.

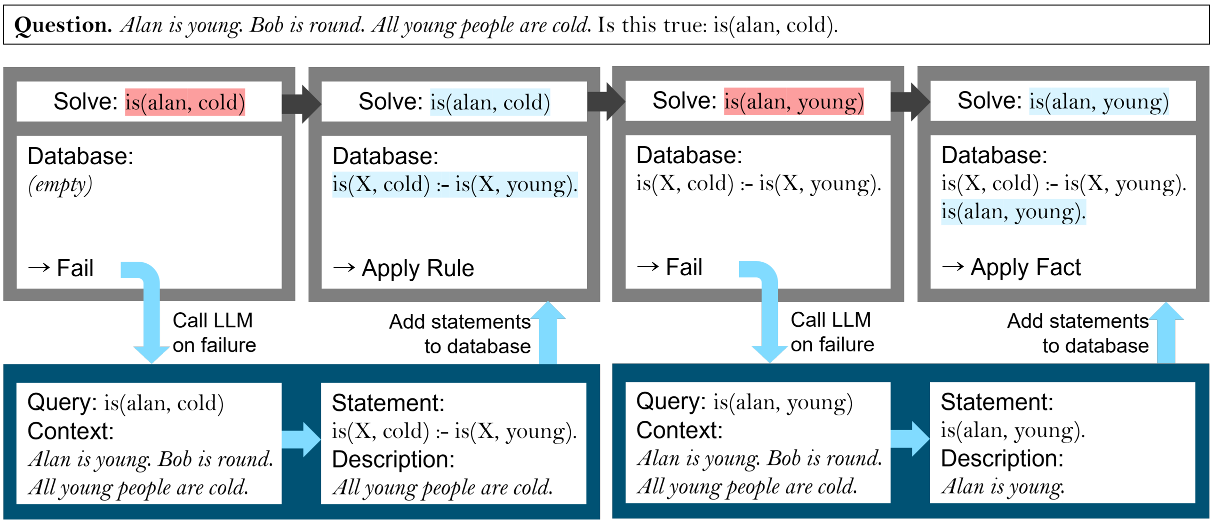

Figure 2: Overview of SymBa. The proof process is mainly controlled by a symbolic backward chaining solver (gray). When a goal is not provable by the solver alone, an LLM (navy) is called and generates a single reasoning step which is added to the symbolic solver's database.

ing both the Search and the Translation step yields the symbolic representation of the logical rule/fact that proves the given query term.

Symbolic validation We verify the generated logic program statement by checking if the statement is syntactically correct, and if the head of the statement unifies to the given query. Note that this step is purely symbolic and does not require any LLM inference.

4 Experimental settings

4.1 Benchmarks

Deductive reasoning We make use of four representative benchmarks for deductive reasoning, namely the ProofWriter family (ProofWriter, Birds-Electricity, ParaRules) (Tafjord et al., 2021; Clark et al., 2020) and PrOntoQA (Saparov and He, 2023). Each instance is formulated as a binary classification task, deciding whether the given query can be proved according to the given rules and facts. For ProofWriter, we leverage the most challenging subset that contains problems with reasoning depth up to 5. For PrOntoQA, we sample examples with fictional entities (hardest) and reasoning depth 4.

Relational reasoning CLUTRR (Sinha et al., 2019) is a relational reasoning benchmark based on human-written stories about family relations. For our experiments, we reformulate the task into true-or-false form, where two entities and a relation are presented and one should predict if the given relation is true or false. We sample from the hardest subset where there are up to 9 bridging entities.

Arithmetic reasoning To evaluate arithmetic reasoning performance, we leverage two benchmarks, namely MAWPS (Koncel-Kedziorski et al., 2016) and GSM8k (Cobbe et al., 2021). The goal of these two tasks is to predict the numeric answer to a given question. MAWPS includes synthetic arithmetic problems that can be solved within 1-3 elementary operations. In contrast, GSM8k contains human-written questions with diverse vocabulary and complex solutions.

More information regarding data statistics, few-shot example construction, logic program representation, and evaluation of each benchmark can be found in Appendix B.

4.2 LLM and Few-shot examples

To reproduce baselines and implement SymBa, we use three open- and closed-sourced state-of-the-art LLMs: GPT-4 Turbo, Claude 3 Sonnet, and LLaMa 3 70B Instruct. A brief comparison of these models is shown in Table 1.

Model	Provider	Open?	Release date
GPT-4 Turbo	OpenAI	N	11/04/2023
Claude-3 Sonnet	Anthropic	N	02/29/2024
LLaMa 3 70B	Meta	Y	04/18/2024

Table 1: Brief information of LLMs applied in this study. *Release date* column refers to the version of the specific checkpoints or API endpoints used for the experiments.

We sample few-shot demonstrations from each training split and manually reformat them as defined by each baseline (Appendix B). For SymBa,

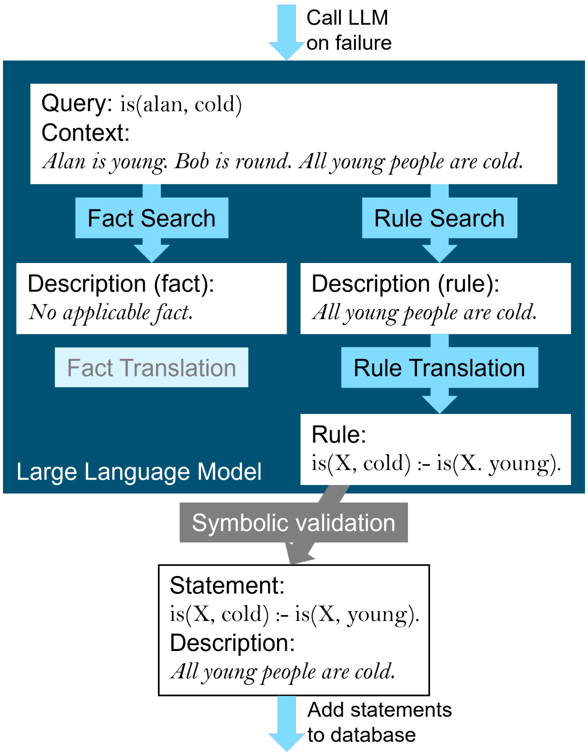

Figure 3: Brief illustration of the modules in SymBa's single statement generation procedure. When the solver fails to prove a term (as illustrated in Figure 2), the single-step statement generation procedure is initiated. Search modules retrieve plausible reasoning steps from the context, which is translated to symbolic form by Translation modules. Statements that passed Symbolic Validation module are added to the solver's database.

Search
Context: *Alan is young. All young people are cold.*
Pos is(alan, cold) → *All young people are cold.*
Neg is(alan, red) → *No applicable rules.*

Translation
Description: *All young people are cold.*
Pos is(alan, cold) → is(X, cold) :- is(X, young).
Neg is(alan, red) → is(X, cold) :- is(X, young).

Figure 4: Examples of Positive/Negative demonstrations included in the prompts for the Search/Translation module of SymBa.

we combine the Positive and Negative examples to reduce hallucination in the Search/Translation modules (Figure 4); effects of these Negative examples are presented in Section 6.2.

4.3 Solver

To implement the algorithm described in Section 2.2, we develop a custom backward chaining solver in Python that is able to process logic programs with arithmetic operations. We formally define the solver's algorithm in Appendix A.

5 Results

5.1 Task performance

The main results are presented in Table 2. Among the three backward chaining methods compared (Least-to-most prompting, LAMBADA, and SymBa), SymBa demonstrates strong performance robust to the type of reasoning (deductive, relational, and arithmetic) and the base language model.

As the benchmarks incorporate multiple plausible reasoning paths with significant depth, the limited planning ability of Least-to-most prompting hinders performance in large-depth benchmarks, such as ProofWriter, ParaRules, CLUTRR, and GSM8k. While it achieves task performance comparable to SymBa in some settings, we further show that the proof might not be accurate and faithful due to the propagation of *Decomposition* errors (Section 6.1).

The accuracy LAMBADA achieves in deductive reasoning is also lower than SymBa. As LAMBADA implements a fully recursive proof generation process, the task performance is less affected by the accuracy of the speculative planning. However, the large performance gap in ParaRules, where the model must extract the underlying reasoning statement despite the syntactic distortion, demonstrates the effectiveness of intermediate symbolic representations that capture the intended logical meaning. Furthermore, as previously mentioned, LAMBADA cannot reason through relational and arithmetic reasoning benchmarks (CLUTRR, MAWPS, and GSM8k) due to the missing backward propagation.

We present complete results including standard deviations in Appendix C.

5.2 Proof accuracy

One of the key benefits of structured reasoning is that it generates more inspectable outputs (Ribeiro et al., 2023). In this section, we analyze the proof accuracy of three backward chaining methods and Chain-of-Thought prompting in four benchmarks. Following Kazemi et al. (2023), the first 30 correct proofs for positive (non-negated) queries are sampled and examined if they include any false intermediate statements or exclude necessary reasoning steps.

Model	Method	Deductive				Relational	Arithmetic	
		ProofWriter	BirdsElec	ParaRules	PrOntoQA	CLUTRR	MAWPS	GSM8k
GPT-4	Least-to-most	71.5	88.2	71.8	87.5	81.5	84.3	60.6
	LAMBADA	69.7	83.4	59.7	96.0	X	X	X
	SymBa	**79.8**	**94.4**	**79.2**	**96.3**	**84.3**	**86.7**	**63.8**
Claude-3	Least-to-most	60.3	75.7	54.0	86.0	77.0	**94.2**	59.3
	LAMBADA	69.3	62.7	57.7	67.0	X	X	X
	SymBa	**77.6**	**77.3**	**69.0**	**91.0**	**85.0**	**94.1**	**67.4**
LLaMa-3	Least-to-most	61.4	71.0	66.7	**95.0**	72.0	**89.0**	61.5
	LAMBADA	64.0	82.3	62.1	90.8	X	X	X
	SymBa	**70.4**	**92.9**	**71.7**	93.3	**90.5**	87.9	**67.0**

Table 2: Average accuracy (%) on four runs per each benchmark, LLM model, and reasoning method. Boldface indicates that the score is significantly higher than others (confidence 95%). LAMBADA is incapable of handling relational and arithmetic benchmarks.

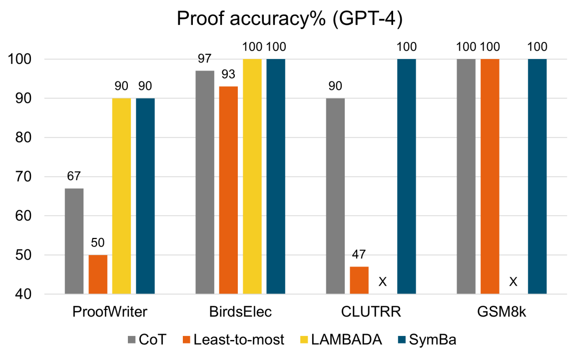

Figure 5: Proof accuracy on four reasoning benchmarks. In the first 30 examples that each method got correct, SymBa and LAMBADA achieved the highest proof accuracy, while Least-to-most achieved the lowest.

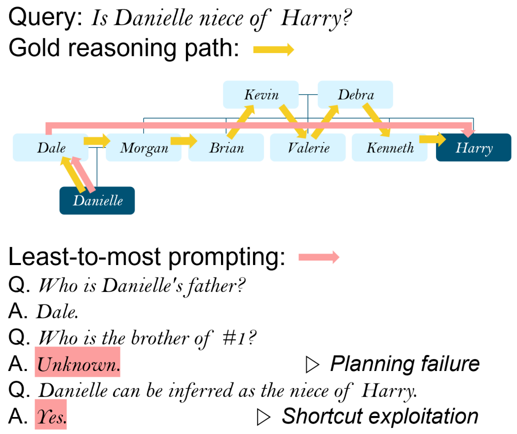

Figure 6: Example of shortcuts by Least-to-most prompting, sampled from CLUTRR. Even though the proof planning is completely inaccurate.

Results are presented in Figure 5. It is shown that two modular methods (LAMBADA and SymBa) generate the most accurate proofs, where Least-to-most prompting demonstrates significantly degraded proof accuracy. Such behavior can be attributed to shortcuts, where it has failed to predict the decomposition order but reached the correct conclusion. Figure 6 illustrates the case where Least-to-most produces incorrect reasoning paths.

In summary, we show that the modular approach can significantly contribute to the proof accuracy as previously claimed in Creswell et al. (2023) and Kazemi et al. (2023).

5.3 Efficiency

To compare the efficiency, we report the token usage, API cost, and execution time for completing 300 examples in ProofWriter following Kazemi et al. (2023).

The results are presented in Table 3. SymBa achieves 9x token/cost efficiency and 22x speed compared to LAMBADA. While LAMBADA uses

	Tokens	Cost($)	Time(h)
CoT	202,420	8.02	0.62
Least-to-most	1,485,989	47.14	1.18
LAMBADA	6,625,623	221.72	23.96
SymBa	**880,106**	**27.22**	**1.15**

Table 3: Token/cost/time consumption (lower the better) for 300 examples in ProofWriter benchmark in GPT-4 Turbo. Regarding the cost, the OpenAI API used in this study charges $0.03 per 1,000 input tokens and $0.05 per 1,000 output tokens.

an LLM to perform unification checks and subgoal decomposition, these processes are delegated to the symbolic solver in SymBa, which results in significantly reduced LLM inference costs.

Despite that SymBa requires multiple LLM inferences per each reasoning step, SymBa is even more efficient than Least-to-most prompting, a non-modular approach. While Least-to-most prompting can be optimized by dynamically appending the questions to intermediate sequences during the inference, currently available commercial LLM APIs do not support such functionality.

6 Analysis

6.1 Error analysis

We manually classify the errors observed from SymBa into three categories: Search-Hallucination, Search-Miss, and Translation. Definitions of the error types are shown in Table 4.

Error Type	Definition
Search-Hallucination	The generated description is not in the context, or unrelated to the query.
Search-Miss	A relevant description stated in the context was not retrieved.
Translation	Symbolic statement is unfaithfully translated from the description (*i.e.* syntax error, misleading symbol names).

Table 4: Description of three error classes observed from SymBa. If multiple errors occur simultaneously in one example, we select the error that appears first.

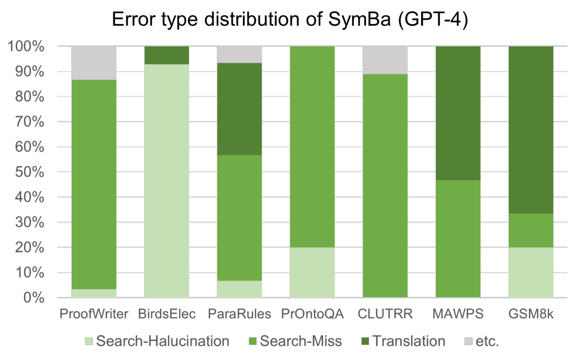

Figure 7: Error analysis results for SymBa. We sampled 30 proofs that resulted in wrong answers and manually classified them according to Table 4.

As presented in Figure 7, the distribution of errors highly varies along the datasets. It implies that each benchmark poses unique challenges depending on numerous factors, such as reasoning type and lexical diversity.

Among the benchmarks, we focus on ProofWriter and Birds-Electricity, which are both deductive reasoning benchmarks yet display completely different error distributions. While rules in ProofWriter often contain variables (*e.g.* '*If* ***someone*** *is red then they are round*'), 99.6% of the rules from Birds-Electricity are bound (*e.g.* '*If wire is metal then wire conducts electricity*'). From this observation, we hypothesize that the higher ratio of unbound rules leads to elevated Search-miss errors.

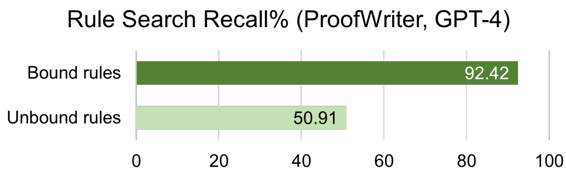

Figure 8: Recall of the Rule Search module in bound and unbound ProofWriter rules.

We compare the recall of the Rule Search module in isolation, based on whether the target rule is bound or not (Figure 8). Rule Search achieves a recall of approximately 51% when the target rule is not bound, which is significantly lower than that of bound rules (∼92%). It proves that the boundness of the provided rules seriously affects Search-Miss errors, possibly due to the low lexical overlap of unbound rules compared to bound rules (Shinoda et al., 2021; Liu et al., 2020).

6.2 Ablation study

As an ablation study, we selectively manipulate the modules or in-context demonstrations and examine the performance of four tasks.

Modules To analyze the contribution of each module, we selectively remove some and compare the performance. In the `-Search` setting, we remove Fact/Rule Search by merging it to Fact/Rule Translation, so that the symbolic statement is directly generated from the context and the query without intermediate textual representations. In the `-Unify` setting, we disable the Symbolic Validation module by not checking if the generated statement unifies to the query.

Negative in-context examples We also test the effects of the Negative in-context examples illustrated in Figure 4. In the `-SearchNeg` setting, we remove Negative examples from the Search module, while in `-TransNeg` we remove Negative examples from the Translation module.

	PW	BE	CLUTRR	GSM8k
SymBa	79.8	94.4	84.3	63.8
`-Search`	-22.7	-5.2	+2.4	+3.0
`-Unify`	-6.9	-1.6	-8.7	-0.1
`-SearchNeg`	-8.8	-29.8	+2.7	+4.1
`-TransNeg`	-2.4	-12.0	-13.8	+1.5

Table 5: Ablation results on four benchmarks using GPT-4 Turbo. All ablation results are 4-run.

As presented in Table 5, the effects of each setting highly vary along the datasets. In ProofWriter variants, the performance significantly drops for all settings. It is notable that in CLUTRR and GSM8k, some ablation settings achieve similar or even better performance compared to the original setting. However, we observe common issues related to the proof accuracy in these settings. In GSM8k, the model often directly outputs the answer instead of providing structured explanations, while in CLUTRR the model makes extreme Search-Hallucination and Translation errors (Figure 9). To summarize, the modular approach and negative in-context examples are both necessary for SymBa's robustness and accuracy in multi-step reasoning.

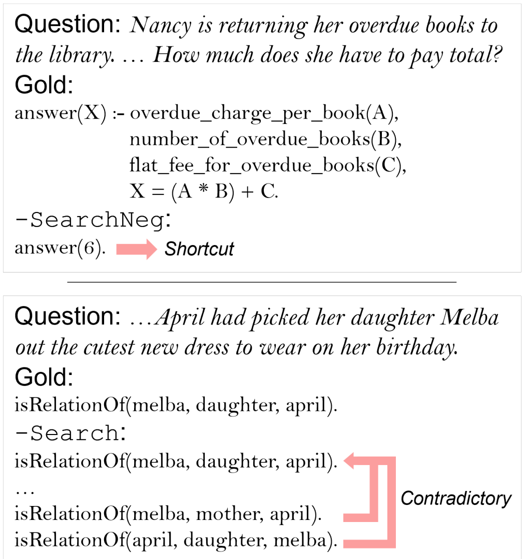

Figure 9: Examples of erroneous logic program statements, sampled from -SearchNeg in GSM8k and -Search in CLUTRR. Ablated versions often fail to produce a faithful reasoning path where SymBa generates a correct proof (denoted as Gold).

7 Related works

7.1 Backward chaining

Backward chaining has not much been explored in the era of LLM and in-context learning compared to forward chaining. At the time of writing, the only work that explicitly claims to be an LLM-based backward chaining method is LAMBADA.

Alternatively, some backward chaining methods use relatively small models directly fine-tuned with in-domain data (Tafjord et al., 2022; Bostrom et al., 2022). These methods train individual modules for rule generation and verification, achieving strong results but on behalf of the costly construction of in-domain data for training.

Furthermore, as previously described in Section 3.1, approaches based on task decomposition (Zhou et al., 2023; Khot et al., 2023; Radhakrishnan et al., 2023) can be viewed as a type of backward chaining (Huang and Chang, 2023). Nonetheless, these methods tend to demonstrate relatively low proof accuracy due to planning failure (Radhakrishnan et al., 2023, Section 5.2 of this work), while SymBa is capable of providing a fully structured proof with high accuracy.

7.2 LLM and Logic programming

Integrating logic programming and LLMs for multi-step reasoning is a recently emerging topic (Pan et al., 2023; Yang et al., 2023; Olausson et al., 2023, *inter alia.*), triggered by the improvement in reasoning and code generation ability of LLMs. The majority of these works implement a similar two-stage approach: (1) convert the problem formulated in natural language into a logic program, and (2) run an external solver to prove the query.

SymBa differs from these methods as the solver is integrated into the loop instead of operating in separate stages. It is reported that these methods often choose incompatible representations for the same concept or fail to discover information that does not surface in the premises (Olausson et al., 2023), as they generate the code without any hierarchical cues about how statements are structured. These issues can be potentially mitigated by the backward chaining of SymBa, as it ensures that all subgoals are addressed at least once and that the generated statement unifies with the query.

8 Conclusion

We introduce SymBa, a novel backward chaining method for diverse structured reasoning. While current backward chaining implementations based on LLMs either overly limit the recursion depth or cannot perform relational and arithmetic reasoning, our method integrates a symbolic solver with LLM that removes both limitations.

By the solver-LLM integration, we achieve high performance in various tasks compared to backward chaining baselines. Furthermore, SymBa provides a structured proof in both symbols and natural language with high accuracy and efficiency.

From both theoretical and empirical perspectives, we believe that SymBa significantly extends the horizon of LLM-based backward chaining.

9 Limitations

While SymBa significantly improves the performance and efficiency of LLM-based backward chaining, it still holds limitations inherited from LLMs, backward chaining, and symbolic reasoning.

To begin with, LLMs often produce counterfactual and inconsistent information, and can potentially cause risk when used in domains where high precision and factuality are required. While SymBa reduces errors by leveraging the symbolic solver and applying a modular approach, the single-step statement generation based on LLM is still subjective to producing false reasoning steps that might lead to the wrong conclusion.

Furthermore, even though backward chaining is inherently free from infinite recursion, a naively implemented backward chaining system might still require substantial computation in fact-intensive tasks such as knowledge base question answering (KBQA) (Yih et al., 2016; Gu et al., 2021). This might be mitigated by hybrid forward and backward chaining (Hong et al., 2022) or by using sophisticated planning algorithms for symbolic solvers (Lu et al., 2012; Yang et al., 2023). We leave this direction as future work.

Lastly, some reasoning problems may not be able to be formulated in logic programming notations as in this study. Most notably, solving high-order logic problems generally requires *meta-predicates* that reason over the database, such as `call/N` in Prolog (Chen et al., 1993), which cannot be handled using the current algorithm of SymBa. Besides high-order logic, some reasoning tasks (*e.g.* Dalvi et al., 2021; Zellers et al., 2019) require reasoning with complex linguistic expressions and highly pragmatic assumptions, which might not be effectively expressed using logic programming.

References

K.R. Apt and K. Doets. 1992. *A New Definition of SLDNF-resolution*. Amsterdam ILLC CT. Institute for Logic, Language and Computation.

Kaj Bostrom, Zayne Sprague, Swarat Chaudhuri, and Greg Durrett. 2022. Natural language deduction through search over statement compositions. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 4871–4883, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Weidong Chen, Michael Kifer, and David Scott Warren. 1993. HILOG: A foundation for higher-order logic programming. *J. Log. Program.*, 15(3):187–230.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2020. Transformers as soft reasoners over language. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 3882–3890. International Joint Conferences on Artificial Intelligence Organization. Main track.

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. *arXiv preprint arXiv:2110.14168*.

Antonia Creswell, Murray Shanahan, and Irina Higgins. 2023. Selection-inference: Exploiting large language models for interpretable logical reasoning. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*.

Bhavana Dalvi, Peter Jansen, Oyvind Tafjord, Zhengnan Xie, Hannah Smith, Leighanna Pipatanangkura, and Peter Clark. 2021. Explaining answers with entailment trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7358–7370, Online and Punta Cana, Dominican Republic. 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.

Ruixin Hong, Hongming Zhang, Xintong Yu, and Changshui Zhang. 2022. METGEN: A module-based entailment tree generation framework for answer explanation. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 1887–1905, Seattle, United States. Association for Computational Linguistics.

Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1049–1065, Toronto, Canada. Association for Computational Linguistics.

Mehran Kazemi, Najoung Kim, Deepti Bhatia, Xin Xu, and Deepak Ramachandran. 2023. LAMBADA: Backward chaining for automated reasoning in natural language. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6547–6568, Toronto, Canada. Association for Computational Linguistics.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. 2016. MAWPS: A math word problem repository. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1152–1157, San Diego, California. Association for Computational Linguistics.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. 2020. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 3622–3628. International Joint Conferences on Artificial Intelligence Organization. Main track.

Benjie Lu, Zhiqing Liu, and Hui Gao. 2012. An adaptive prolog programming language with machine learning. In *2nd IEEE International Conference on Cloud Computing and Intelligence Systems, CCIS 2012, Hangzhou, China, October 30 - November 1, 2012*, pages 21–24. IEEE.

Kyle Marple, Elmer Salazar, and Gopal Gupta. 2017. Computing stable models of normal logic programs without grounding. *CoRR*, abs/1709.00501.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. 2023. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5153–5176, Singapore. Association for Computational Linguistics.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.

Pruthvi Patel, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Is a question decomposition unit all we need? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4553–4569, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

David Poole and Alan K. Mackworth. 2010. *Artificial Intelligence - Foundations of Computational Agents*. Cambridge University Press.

Ansh Radhakrishnan, Karina Nguyen, Anna Chen, Carol Chen, Carson Denison, Danny Hernandez, Esin Durmus, Evan Hubinger, Jackson Kernion, Kamilė Lukošiūtė, Newton Cheng, Nicholas Joseph, Nicholas Schiefer, Oliver Rausch, Sam McCandlish, Sheer El Showk, Tamera Lanham, Tim Maxwell, Venkatesa Chandrasekaran, Zac Hatfield-Dodds, Jared Kaplan, Jan Brauner, Samuel R. Bowman, and Ethan Perez. 2023. Question decomposition improves the faithfulness of model-generated reasoning.

Danilo Neves Ribeiro, Shen Wang, Xiaofei Ma, Henghui Zhu, Rui Dong, Deguang Kong, Juliette Burger, Anjelica Ramos, zhiheng huang, William Yang Wang, George Karypis, Bing Xiang, and Dan Roth. 2023. STREET: A MULTI-TASK STRUCTURED REASONING AND EXPLANATION BENCHMARK. In *International Conference on Learning Representations*.

Abulhair Saparov and He He. 2023. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Kazutoshi Shinoda, Saku Sugawara, and Akiko Aizawa. 2021. Can question generation debias question answering models? a case study on question–context lexical overlap. In *Proceedings of the 3rd Workshop on Machine Reading for Question Answering*, pages 63–72, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. 2019. CLUTRR: A diagnostic benchmark for inductive reasoning from text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4506–4515, Hong Kong, China. Association for Computational Linguistics.

Zayne Sprague, Kaj Bostrom, Swarat Chaudhuri, and Greg Durrett. 2023. Deductive additivity for planning of natural language proofs. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 139–156, Toronto, Canada. Association for Computational Linguistics.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. 2021. ProofWriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3621–3634, Online. Association for Computational Linguistics.

Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. 2022. Entailer: Answering questions with faithful and truthful chains of reasoning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2078–2093, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96.

Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 89–105, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Sen Yang, Xin Li, Leyang Cui, Lidong Bing, and Wai Lam. 2023. Neuro-symbolic integration brings causal and reliable reasoning proofs. *arXiv preprint*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium. Association for Computational Linguistics.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206, Berlin, Germany. Association for Computational Linguistics.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*.

A Formal definition of SymBa

In this section, we provide an algorithmic description of SymBa. SymBa can be viewed as an extension of the SLDNF resolution (Selective Linear Definite resolution with Negation as Failure) algorithm (Apt and Doets, 1992) typically used in top-down solvers like SWI-Prolog (Wielemaker et al., 2012). A simplified pseudo-code for SymBa is presented in Algorithm 1. The notations used throughout this section are presented in Table 6.

Notation	Definition
h, p, q	Term (proposition)
$\mathbb{T}$	Set of all terms
B	Binding (mapping from variables to variables/constants)
$\mathcal{B}$	List of bindings.
$\mathbb{B}$	Set of all bindings.
$\mathbf{s}$	Statement (rule, fact)
$\mathbf{s}.head$	Rule head (term)
$\mathbf{s}.body$	Rule body (list of terms)
C	Context written in natural language

Table 6: Notations used in Appendix A.

Before proceeding to the algorithm, we introduce three procedures about unification and binding, namely UNIFY : $\mathbb{T} \times \mathbb{T} \rightarrow \{0, 1\}$, BINDING : $\mathbb{T} \times \mathbb{T} \rightarrow \mathbb{B}$, and BIND : $\mathbb{T} \times \mathbb{B} \rightarrow \mathbb{T}$. As described in Section 2.2, two terms are said to unify if there is a valid binding that makes the terms identical. UNIFY returns a boolean value indicating whether the two terms unify or not. BINDING returns the binding of two terms if they unify. BIND takes a term (possibly containing variables) and a binding as its argument, and returns the bound term after substituting the variables from the term to the corresponding values. By definition, for any two terms p and q that satisfy UNIFY(p, q), BIND$(p,$ BINDING$(p, q))$ $=$ BIND$(q,$ BINDING$(p, q))$ should always hold.

SOLVE is the main procedure of SymBa. It receives a query term q as a parameter and refers to the global database (set of statements) $\mathcal{D}$ to compute $\mathcal{B}_{final}$, the list of all provable bindings for q. If $\mathcal{B}_{final}$ is not empty, it implies that q can be proved on $\mathcal{D}$. Otherwise, the query cannot be proved.

Negation is handled first, in Lines 5-12. In the negation-as-failure semantics, the negation `not` q succeeds when q fails, and vice versa. Therefore, whenever the query is negated (*i.e.* `not` q_{pos}), its non-negative dual (*i.e.* q_{pos}) is proved first (Line 6). When the proof succeeds, the negated goal should be failed, therefore an empty list ($\mathcal{B}_{final}$) is returned (Line 8). When the proof fails, an empty binding is added to the $\mathcal{B}_{final}$ to indicate success of the original query.

The main loop is shown in Lines 13-31. First, the statements that have heads unifying with the query are selected from the database. The initial binding B_0 is the binding between the statement's head and the query. For each subgoal p_t, we bind the subgoal using the previous binding $B_{t-1,i}$ (Line 19). The partially bound subgoal $p_{t,i}$ is proved by recursively calling SOLVE, which returns a list of bindings for $p_{t,i}$ (Line 20). The new bindings $B_{t,i,j}$ are added to original binding $B_{t,i}$ (Line 22), which are then propagated to the next subgoal p_{t+1}. When all subgoals are proved, the query is proved, and the bindings are added to the answer set (Line 27). Note that if the query contains variables, these bindings can be used to bind the query to obtain the list of possible 'solutions', as presented in Lines 41-45.

Single-step statement generation, the novel mechanism of SymBa, is shown in Lines 32-38. The flag $isProved$ denotes whether the solver has succeeded in finding a statement that unifies with the query. If the value is **false**, the single-step statement generation (SINGLESTEPSTMTGEN) process described in Section 3.2 is called, which is expected to return a new statement $\mathbf{s}_{new}$ from the context C and the query q. If the procedure succeeds, $\mathbf{s}_{new}$ is added to $\mathcal{D}$, and the solver re-attempts to solve q with the updated database.

If the negation-as-failure succeeded (Line 10), it cannot be determined if the positive query is truly unprovable because queries that have never been previously addressed will always fail. Therefore, the $isProved$ flag remains **false** in this case, which will later invoke the single-step statement generation.

For brevity, here we do not further describe additional features, namely comparison operators, odd loop on negation (OLON) (Marple et al., 2017), goal tabling (to prevent duplicate calls and infinite recursion), and proof tree generation. Full implementation of SymBa can be found in this repository.

B Dataset details

This section describes the sampling, preprocessing, and evaluation of benchmarks. Table 7 presents brief information and statistics about the seven benchmarks used in this paper.

All datasets used in this study allow free

Algorithm 1 Algorithm of SymBa

```
global 𝒟 ← empty set
procedure SOLVE(q)                                        ▷ Input: query term, Returns: list of bindings
    ℬ_final ← empty list
    isProved ← false
    if q is negated (i.e. not q_pos) then
        ℬ_pos ← SOLVE(q_pos)
        if ℬ_pos is empty then
            return empty list                             ▷ NAF fail
        else
            Append empty binding to ℬ_final
        end if
    end if
    𝒮 ← {s ∈ 𝒟 | UNIFY(s.head, q)}                        ▷ Set of statements that have heads unifying with q
    for s ∈ 𝒮 do
        ℬ_0 ← [BINDING(s.head, q)]
        for p_t ∈ s.body = [p_1, ..., p_T] do
            ℬ_t ← empty list
            for B_{t−1,i} ∈ ℬ_{t−1} = [B_{t−1,0}, ..., B_{t−1,I}] do
                p_{t,i} ← BIND(p_t, B_{t−1,i})            ▷ Apply bindings from head & previous subgoals
                ℬ_{t,i} ← SOLVE(p_{t,i})                  ▷ Solve the partially bound subgoal
                for B_{t,i,j} ∈ ℬ_{t,i} = [B_{t−1,0}, ..., B_{t−1,J}] do
                    B_{t,i,j} ← B_{t,i,j} ∪ B_{t−1,i}     ▷ Update the binding
                end for
                Extend B_{t,i} to ℬ_t
            end for
        end for
        Extend B_T to ℬ_final
        if B_T is not empty then
            isProved ← true
        end if
    end for
    if isProved then                                      ▷ Subgoal success
        return ℬ_final
    else                                                  ▷ Subgoal failure
        s_new ← SINGLESTEPSTMTGEN(C, q)
        Add s_new to 𝒟
        return SOLVE(q)
    end if
end procedure

C ← user input
q_init ← user input
ℬ ← SOLVE(q)
for q_final ∈ {BIND(q_init, B) | B ∈ ℬ} do
    print q_final
end for
```

Dataset	Type	Test size	Avg. steps	Avg. sents	N-shot
ProofWriter (Tafjord et al., 2021)	Deductive	300	4.52	19.12	3
Birds-Electricity (*Ibid.*)	Deductive	300	2.08	13.77	3
ParaRules (Clark et al., 2020)	Deductive	300	4.37	10.56	3
PrOntoQA (Saparov and He, 2023)	Deductive	100	4.00	21.84	3
CLUTRR (Sinha et al., 2019)	Relational	100	4.86	5.20	3
MAWPS (Koncel-Kedziorski et al., 2016)	Arithmetic	300	3.06	3.20	5
GSM8k (Cobbe et al., 2021)	Arithmetic	270	9.22	4.87	5

Table 7: Statistics of each test set. *Avg. steps* denotes the average number of statements (facts and rules) required to prove the goal, and *Avg. sents* is the average number of sentences that each context contains. *N-shot* denotes the number of few-shot examples to prompt LLMs in this study.

use, modification, and redistribution for non-commercial applications.

B.1 ProofWriter family

Test split sampling From the ProofWriter family, we sample the evaluation set from the test split of the closed-world assumption subset (CWA). Specifically, for ProofWriter, we use the dep5 subset, which has a deepest maximum reasoning depth of 5. Since a single context includes multiple questions, we first sample 300 contexts and randomly sample a question from it. As a result, we obtain 300 (context, question) tuples for each dataset).

In-context demonstrations We randomly sample 3 examples from ProofWriter-dep3 and -dep2 data that contain shorter contexts to test the length generalization ability of each method. For CoT prompting and Least-to-most prompting, we provide the pre-order traversal of the golden proof tree provided for each instance, with stopwords like *since* and *so* that are known to enhance the performance in CoT prompting (Kazemi et al., 2023). For LAMBADA, we use the prompt format provided in the original paper, which is populated with the sampled in-context examples.

Logic program We consistently apply `verb(subject,object)` format to both datasets. For instance, *Bald eagle does not eat the mouse.* translates to `not eats(bald_eagle,mouse)`. Note that we apply the same format for adjective facts. For example, the corresponding symbolic form for *Alan is young.* is `is(alan,young)`, opposed to another commonly used form `young(alan)` or `young(alan,true)` (Olausson et al., 2023; Pan et al., 2023).

As a common practice for measuring the reasoning ability in out-of-distribution data (Birds-Electricity, ParaRules) using in-domain data (ProofWriter) (Tafjord et al., 2021), we use the prompts and examples sampled from ProofWriter train split for the other two benchmarks.

Evaluation We use the true/false labels provided with the original dataset without modification.

B.2 PrOntoQA

Test split sampling We sample the test set using the original script from Saparov and He (2023), using fictional entity names (*e.g. Every yumpus is a jompus.*). However, due to an unresolved issue of the script, the script only allows to generate a reasoning chain of a maximum of four steps.

In-context demonstrations Similar to the ProofWriter family, we use few-shot demonstrations with 8 premises, which is significantly lower than average (NN premises).

We use identical logic program formats and evaluation criteria for PrOntoQA with other ProofWriter variants.

B.3 CLUTRR

Test split sampling We randomly sample 100 examples from the test split of CLUTRR v1. To generate false labels, we sample half of the examples and alter the relation label of the gold triplet to a random one.

In-context demonstrations We randomly sample 3 stories from the train split that only contains 2-3 relations to test the length generalization ability of each methods. For CoT, we provide a golden chain of kinship relations that connect the two queried entities. For Least-to-most prompting, each decomposed question contains information about an entity and a relation, asking for the bridging entity. (e.g. *Who is the father of Andrea?*)

Logic program and expert system We introduce 39 manually crafted rules about family relationships. To reduce excessive recursion, we use separate predicate names for the base fact and inferred relations. For instance, '*George is the father of Andrea.*' is translated as `isRelationOf(george,father,andrea)` if it is a fact directly from the context, or `relation(george, father, andrea)` if it is inferred by more than one bridging entities. Note that the predicate name for the latter casts no effect on the performance as it is only used for the symbolic solver and not the LLM.

Examples of the expert system rules are presented as follows. Note that the semicolon(;) denotes that the rule conditions are satisfied when either of the groups is satisfied (disjunction).

```
relation(A, R, B) :-
  isRelation(A, R, B).
relation(A, son, B) :-
  isRelationOf(A, brother, C),
  relation(C, (son;daughter), B).
relation(A, daughter, B) :-
  isRelationOf(A, sister, C),
  relation(C, (son;daughter), B).
...
```

Evaluation Each model is instructed to predict if the label is correct or not (randomized).

B.4 MAWPS

Test split sampling We use the first 300 examples from the original test split.

In-context demonstrations Five few-shot examples are randomly sampled from the train split. We manually create annotations as the benchmark does not include a reasoning chain.

Logic program We denote the meaning of each numeric value with predicates of arity 1, as in `number_of_oranges(_)` or `fraction_of_trombone_section(_)`. We use `answer`(X) to express the final answer in all examples and evaluate if the variable X is successfully bound to the right numeric value (*e.g.* `answer`(5)).[3] Facts denote the base value mentioned in the text (*e.g.* `number_of_yellow_flowers`(10)), and rules express the arithmetic relations between each value (*e.g.* `fraction_of_trumpet_section(X) :- fraction_of_trombone_section(A)`, $X = A * 4$.).

Evaluation We use the numeric answer provided with the original dataset. If the answer is not a numeric string (e.g. `25,000` or `42 pages`), they are considered incorrect. While Standard prompting exceptionally suffers from this constraint, we claim that it is not unfair as each method is equally presented with 5-shot examples in the correct format.

[3]While previous approaches in logic programming-integrated LLMs use an additional step to specify which predicate corresponds to the final answer (Pan et al., 2023), we do not introduce this mechanism for universality.

B.5 GSM8k

Test split sampling We use the test split used in Yang et al. (2023), which contains 270 examples and is a subset of the original test split from Cobbe et al. (2021). We calculate the number of reasoning steps presented in Table 7 based on the semi-structured solutions included in the dataset.

In-context demonstrations We randomly sample 5 questions from the train split. For CoT prompting, we used the `answer` column from the original dataset and removed the external call snippets (equations that are wrapped in double angle brackets «...»). For Least-to-most prompting, we reformulate the `answer` column from the 'Socratic' version of the dataset that formulates the reasoning chain as consecutive sequence of questions and answers.

We use identical logic program formats and evaluation criteria for GSM8k with MAWPS.

C Complete results

Table 8 presents the complete results of the main experiment (Section 5.1). We also report the performance of Standard prompting (generating the answer without any rationales) and Chain-of-thought prompting for comparison.

Model	Method	Performance						
		ProofWriter	BirdsElec	ParaRules	PrOntoQA	CLUTRR	MAWPS	GSM8k
GPT-4	Standard	63.2±0.43	77.8±1.17	61.3±1.10	83.0±0.82	72.0±4.00	†94.2±0.58	29.4±1.81
	CoT	70.5±2.13	81.2±1.41	60.5±1.03	96.8±1.26	†84.5±1.29	†99.1±0.49	†94.2±1.00
	Least-to-most	71.5±2.10	88.2±0.76	71.8±0.71	87.5±1.29	81.5±0.58	84.3±0.56	60.6±1.96
	LAMBADA	69.7±1.18	83.4±1.20	59.7±1.30	96.0±1.41	X	X	X
	SymBa	**79.8±1.06**	**94.4±0.62**	**79.2±1.12**	**96.3±1.26**	**84.3±2.06**	**86.7±0.69**	**63.8±0.74**
Claude-3	Standard	61.3±0.00	66.0±0.00	61.3±0.00	†96.0±0.00	80.0±0.00	†96.3±0.00	17.0±0.00
	CoT	67.0±2.00	73.3±0.00	57.3±0.00	†96.0±0.00	67.0±0.00	88.0±0.00	†92.2±0.00
	Least-to-most	60.3±0.00	75.7±0.00	57.3±0.00	86.0±0.00	67.0±0.00	**94.2±0.15**	59.3±0.00
	LAMBADA	69.3±0.00	62.7±0.00	57.7±0.00	67.0±0.00	X	X	X
	SymBa	**77.6±0.00**	**77.3±0.00**	**69.0±0.00**	**91.0±0.00**	**85.0±0.00**	**94.1±0.15**	**67.4±0.00**
LLaMa-3	Standard	63.6±0.50	78.7±0.00	65.3±0.00	†99.0±0.00	75.0±0.00	†96.3±0.00	26.2±0.00
	CoT	64.8±1.26	79.0±1.29	63.0±1.67	92.5±4.12	77.0±0.00	†95.0±0.00	†89.5±1.35
	Least-to-most	61.4±0.34	71.0±0.00	66.7±0.00	**95.0±0.00**	72.0±0.00	**89.0±0.00**	61.5±0.00
	LAMBADA	64.0±1.63	82.3±0.00	62.1±1.10	90.8±0.50	X	X	X
	SymBa	**70.4±1.26**	**92.9±1.10**	**71.7±0.00**	93.3±0.50	**90.5±0.58**	87.9±0.70	**67.0±0.00**

Table 8: Average accuracy (%) and standard deviation on 4-runs per each benchmark and reasoning methods. Boldface font indicates that the score is significantly higher than other backward chaining methods, which is equivalent to the boldface in Table 2. Daggers represent that non-structured methods (Standard, Chain-of-thought) achieves significantly higher score than the best structured backward chaining results. 95% confidence applies to both notations. Note that the temperature was set to 0 for all runs, which results in zero standard deviation in some settings.