When Do Symbolic Solvers Enhance Reasoning in Large Language Models?

Zhiyuan He
University College London
nick.he.21@ucl.ac.uk

Dingmin Wang
University of Oxford
dingmin.wang@cs.ox.ac.uk

Abstract

Large Reasoning Models (LRMs) achieve strong performance on complex reasoning tasks by generating long Chains of Thought (CoTs). However, this paradigm might incur substantial token overhead, especially when models "overthink" by producing lengthy reasoning chains, which can even lead to incorrect answers. A promising direction is the symbolic-solver-integrated approach, which leverages the code generation capabilities of LLMs to translate reasoning tasks into executable code and then solve them with a symbolic solver. In this paper, we explore an open question of *when the conventional long-CoT can be enhanced by symbolic solvers*. Our experimental results show that the symbolic-solver-integrated method only helps when the problem requires limited implicit reasoning but involves an ample search space. The latest LLMs, like GPT-4o, show better performance on deductive problems with shallow reasoning depth, while the symbolic-solver-integrated method significantly improves the LLMs' performance in constraint satisfaction problems that require repeated backtracks. When a declarative exemplar is provided, even CodeLlama-13B can outperform GPT-4o in difficult Zebra puzzles.

1 Introduction

Recent Large Language Models (LLMs) [Dubey et al., 2024, Achiam et al., 2023] have demonstrated outstanding capabilities in various domains including language understanding and generation tasks [Chang et al., 2024]. Chain-of-thought (CoT) prompting [Wei et al., 2023] enables LLMs to articulate a step-by-step process to solve a problem or reach a conclusion. However, it is still an open problem whether language models can reason logically in the same way as humans do. In fact, recent works [Hassid et al., 2025, Kumar et al., 2025] show that when models "overthink" by generating unnecessarily long reasoning chains, their accuracy can actually degrade, leading to incorrect answers. LLMs continue to struggle with complex, multi-step reasoning tasks [Huang and Chang, 2022, Dziri et al., 2024], likely due to the transformer [Vaswani et al., 2017]'s inherent pattern-matching mechanism [Dziri et al., 2024, Hu et al., 2024]. This limitation significantly hampers the practical application of LLMs in fields such as education, law, and industrial code generation, where precise and reliable reasoning is critical [Venkit et al., 2024, Dahl et al., 2024].

To address this challenge, one research direction is *symbolic-solver-integrated* methods that use external symbolic solvers to enhance the reasoning capabilities of language models [Giadikiaroglou et al., 2024]. These methods use LLMs' ability of code generation to translate the problems into formal language formulations, which can be interpreted by symbolic tools [Pan et al., 2023, Ye et al., 2024, Gao et al., 2023, Chen et al., 2022]. However, these methods typically experiment on datasets comprising simple synthetic sentences that are easily translated into formal languages or involve a shallow reasoning depth. Also, with the latest models like GPT-4o, benchmarks for many popular reasoning tasks have been raised. Do symbolic-solver-integrated methods still lead in performance? If not, for what types of problems should they be used?

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Efficient Reasoning.

We investigate these critical problems in integrating LLMs with symbolic solvers. We study how to model a problem efficiently and the advantage of symbolic-solver-integrated methods over CoT prompting. We experiment on five various datasets of three types: natural language arithmetic problems (GSM8K, GSM-Reversed, and GSMHard), complex constraint satisfaction problem ZebraLogic, and logic deduction problem EntailmentBank (task 2). We propose using Prolog and Python with symbolic libraries to improve the expressiveness of logical formulations, mainly due to the advanced data structures supported and the pretraining of LLMs on these programming languages. We also design exemplars to guide the language model, generating code in a declarative style and using concise statements that accurately capture the information in the problem.

Through the empirical analysis of the experiments, we find that it is crucial to model problems concisely into logical formulations for symbolic-solver-integrated methods. Our proposed prompting effectively increases the correctness rate of the generated code. With this prompting, the symbolic-solver-integrated methods enhance the reasoning ability in LLMs on constraint satisfaction problems where repeated backtracking is required. However, CoT reasoning is still advantageous in problems involving shallow deduction depths and implicit reasoning.

2 Related Works

Analysis of LLMs' reasoning ability As transformer-based language models become more powerful and widely used, more research is increasingly focusing on the limits of these models in handling complex compositional tasks. Zhou et al. [2023] indicate that the LLMs struggle with out-of-distribution generalization for algorithm tasks beyond a certain complexity. Dziri et al. [2024] study the fundamental mechanism of LLMs solving reasoning tasks and observe that transformers generate the solutions relying on linear path matching or shortcut learning [Geirhos et al., 2020]. Hu et al. [2024] propose a congruent argument of LLMs using case-based reasoning instead of rule-based like humans. Olausson et al. [2023] compared the performances and analyzed the errors of First Order Logic (FOL) formalization and CoT methods on ProofWriter and FOLIO datasets.

Symbolic-solver-integrated methods for LLMs This type of method involves an *autoformalisation* [Wu et al., 2022] stage where the problem in natural language is translated into expressions that external tools can interpret. Gao et al. [2023] propose Program-aided Language Models (PAL) to translate problems into Python programs and solve them using a Python interpreter. Chen et al. [2022] use a similar methodology, Program of Thoughts (PoT), while it applies a different prompt that helps outperform PAL. More recently, Fleureau et al. [2024] won the first AIMO Progress Prize with their finetuned DeepSeekMath-Base 7B model by integrating the Python REPL. Another group of methods intends to translate the problems into FOL-based logical expressions to tackle the LLMs' weakness in rule-based reasoning. Olausson et al. [2023] translate the natural language premises and desired conclusion into FOL expressions. A symbolic FOL theorem prover will algorithmically determine the truth value of the conclusion. Pan et al. [2023] experiment on a larger range of formulations, including Logical Programming, Constraint Satisfaction, and Boolean Satisfaction. Ye et al. [2024] design prompts to generate declarative specifications that are closer to the problem descriptions. These methods show promising results, especially on synthetic logical deduction datasets like FOLIO [Han et al., 2022], ProofWriter [Tafjord et al., 2020], and ProntoQA Saparov and He [2022].

In this work, we aim to explore the capability of symbolic-solver-integrated methods on problems that are more naturally stated and require richer semantic understanding. Instead of FOL-based expressions, we propose to model problems using more sophisticated programming languages like Prolog and Python, with the support of libraries proficient to different types of problems. We also design the prompt to elicit the LLMs to generate declarative codes in a sentence-to-sentence manner.

3 Methodologies

3.1 Deductive Reasoning Problems

We selected our datasets to ensure coverage of diverse reasoning types, allow control over reasoning difficulty, and include non-synthetic, natural language reasoning problems.

Compared to other types of reasoning (such as abductive reasoning, inductive reasoning, and analogical reasoning), deductive reasoning is more fundamental. We address three types of deductive problems: arithmetic reasoning, entailment reasoning, and constraint satisfaction logic reasoning.

Arithmetic reasoning test sets exhibit unique characteristics that impact the performance of different methods:

- **Arithmetic calculations**: Arithmetic operations are basic, but they can become complex with large numbers.
- **Shallow reasoning depth**: Typically, these problems involve not deeply nested logical steps but relatively shallow deduction.
- **Implicit rules from word semantics**: The rules governing the problems are often hidden within the natural language semantics, making it challenging to formalize them into explicit mathematical or logical statements.

Entailment is a fundamental concept in logic. It describes the relationship between statements that hold true when one statement logically follows from one or more statements. A logical argument is valid if the conclusion is entailed by the premises. That means that the conclusion is the consequence of the premises.

Constraint satisfaction logic reasoning is related to logic problems that can be suitably formulated as Constraint Satisfaction Problems (CSP). Russell and Norvig [2016] define a CSP as a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables $x_1, x_2, \ldots, x_n$; $\mathcal{D}$ is a set of nonempty domains $D_1, D_2, \ldots, D_n$ and each variable x_i has a nonempty domain D_i of possible values; $\mathcal{C}$ is a set of constraints $c_1, c_2, \ldots, c_m$ and each constraint c_i involves some subset of the variables and specifies the allowable combinations of values for that subset. An *assignment* within a problem is represented by assigning specific values to a subset or all of the variables, for instance, $\{X_i = v_i, X_j = v_j, \ldots\}$. When every variable has been assigned a value, it is a *complete assignment*. In the context of a CSP, a solution is a complete assignment that meets all the constraints.

One classic example of CSPs is the Zebra puzzle, also known as the Einstein puzzle. In this problem, the goal is to determine specific features of a set of houses based on a series of given constraints. Each house has a unique combination of features such as color, owner, and pet. The solution space for the Zebra puzzle is given by $H! \times F$, where H denotes the number of houses, and F denotes the number of features associated with each house. The constraints in the puzzle guide the elimination of invalid permutations, thereby narrowing down the possible solutions to the unique permutation that satisfies all given constraints.

While symbolic solvers can support a backtracking mechanism for CSPs, Chain-of-Thought (CoT) reasoning in LLMs faces unique challenges. To solve CSPs efficiently, CoT must employ heuristics to determine which variable to consider next and use trial-and-error to eliminate impossible values for a variable. The model could be overwhelmed by the exponentially increasing search space.

3.2 Prompting With Declarative Exemplars

Declarative programming is a style of building the structure and elements of computer programs that expresses the logic of computation. In contrast, *imperative programming* describes the control flow of how a program operates step by step. Declarative programming focuses on high-level descriptions of its expected results. Apart from one of the most prevalent declarative programming languages, Prolog, we also use Python with symbolic libraries like *Constraint* or *SymPy* to generate code in a declarative style.

In our preliminary experiments, we observe that if we directly prompt the LLM to generate Python or Prolog code to solve the problem, it tends to generate code in an imperative way. To generate declarative style codes for better performance, we provide the LLM with a declarative exemplar in the prompt. This exemplar involves translating the problem into declarative codes in a sentence-to-sentence manner, where each natural language sentence is copied into a comment, followed by its programming language representation. Figure 1 illustrates a one-shot prompt with Python code using the SymPy library on the GSM-Hard dataset. Other prompts are illustrated in the Appendix A.

Notably, in arithmetic reasoning problems, we often need to solve for a value within an infinite, continuous range of real numbers. For Python, this is achieved using symbolic solvers from libraries

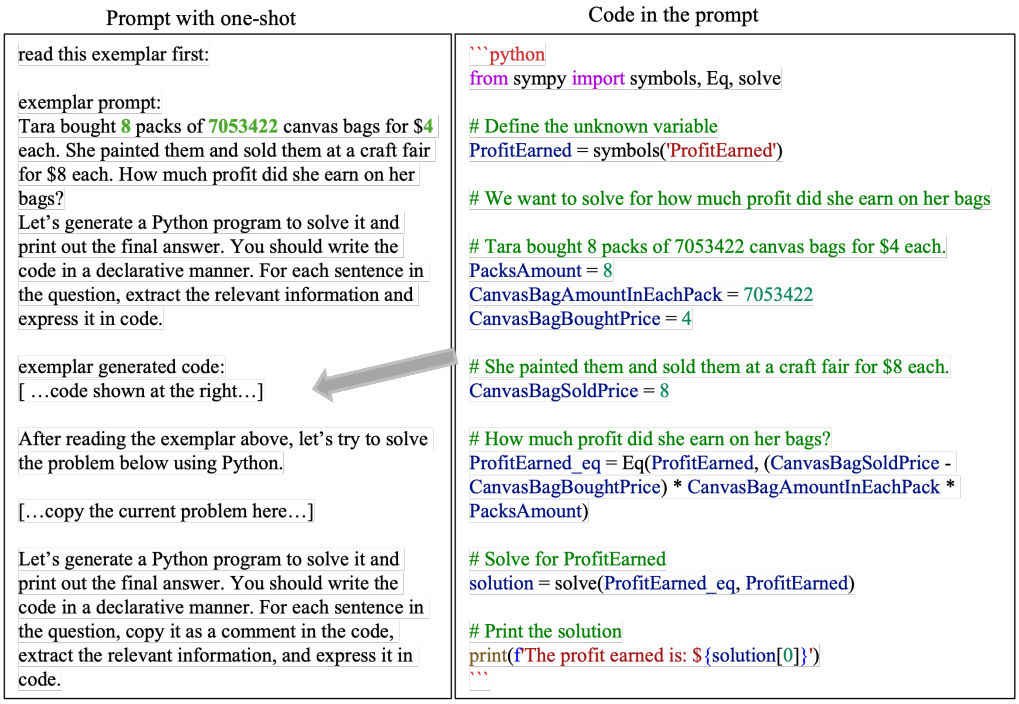


Figure 1: One-shot prompt for GSM-Hard using a declarative sentence-to-sentence translation exemplar of Python code.

such as SymPy. For Prolog, we utilize the *Constraint Logic Programming over Reals* (clpr) library [Holzbaur, 1995]. These methods rely on the symbolic solvers to find the unknown values that satisfy the given mathematical equations.

More specifically, here are the main formats that we use for problem auto-formalisation:

- **Prolog** is a declarative language designed for expressing logical statements and reasoning. It is suitable for tasks involving rule-based logic and symbolic reasoning.
- **Prolog with clpr library** extends the capabilities of the standard Prolog. By incorporating constraint logic programming for real numbers, it allows for complex problem-solving involving numerical constraints.
- **Python with Constraint library** combines the flexibility and readability of Python with powerful constraint satisfaction capabilities. This approach enables us to define and solve logical problems using constraints within the popular Python environment.
- **Python with SymPy library** applies symbolic mathematics to represent and manipulate mathematical expressions and declarative logic. This method is beneficial for problems requiring symbolic computation and algebraic manipulation.

4 Experimental Settings

In this paper, we use the following five datasets for testing.

Arithmetic reasoning problems For this category, we use GSM8K [Cobbe et al., 2021] and two of its variants, GSM-Reversed [Gao et al., 2024] and GSM-Hard [Gao et al., 2023]. GSM-Reversed tests a language model's reversed reasoning ability [Yu et al., 2023]. The final answer to the original GSM8K question is given in the prompt, but part of the input value is concealed and asked for a solution. GSM-Hard is constructed by replacing a number in the original GSM8K problem with a random larger number, up to seven digits. It tests the language model's mathematical abilities with large numbers, non-integer numbers, and negative numbers.

Dataset Name	Size	Language + Library
GSM8K	1418	
GSM-Reversed	1358	Prolog+clpr; Python+SymPy
GSM-Hard	1319	
EntailmentBank	300	Prolog
ZebraLogic	1000	Prolog; Python+Constraint

Table 1: Dataset details and the corresponding programming languages used.

Entailment reasoning We experiment on the EntailmentBank [Dalvi et al., 2021] dataset, which is a rich collection of entailment trees designed to evaluate models in natural language understanding and reasoning. Each question requires to create a tree of multi-premise entailment steps from facts that are known, through intermediate conclusions, until the desired hypothesis is reached.

Constraint satisfaction reasoning problems We evaluate CoT and symbolic-solver-integrated methods on ZebraLogic [Lin, 2024], a dataset designed to test on CSP and offers tunable difficulty levels, making it ideal for comprehensive evaluations. We extensively tested all available sizes of ZebraLogic puzzles, ranging from 2x2 to 6x6, to compare the performance of the two types of methods under different reasoning difficulties.

For each dataset, we compare the LLM's CoT reasoning against the symbolic-solver-integrated methods. We choose suitable logical formulations for each method to fit the question's purpose and formulate the question with higher accuracy. The details are presented in table 1.

We experiment with one of the state-of-the-art LLMs for code and reasoning: GPT-4o. We also test with a smaller Python-specialized LLM *CodeLlama-13b-Python-hf*[1]. We use greedy decoding for reproducibility and to reduce randomness. We use zero-shot, or one-shot if the model is required to generate code in a certain template.

For the methods that use code generation, we use the self-refinement mechanism [Pan et al., 2023], which returns the error message to the LLMs and asks for regeneration when the interpreter reports a syntax error. We set the maximum retries to be ten times for all experiments.

5 Results

Model	Prompting	Arithmetic			ZebraLogic			Entailment
		GSM8K	GSM-Rev	GSM-Hard	easy	hard	average	
GPT-4o	**CoT**	94.6	**90.3**	69.1	77.9*	8.9*	28.2*	**81.8**
Claude 3.5 Sonnet	**CoT**	95.6	-	-	87.5*	12.4*	33.4*	-
Llama-3.1-405B	**CoT**	**95.9**	-	-	87.1*	11.4*	32.6*	-
code-davinci-002	**PAL**	72.0	-	61.2	-	-	-	-
code-davinci-002	**PoT**	71.6	-	61.8	-	-	-	-
CodeLlama13B	**CoT**	22.1	-	-	-	-	-	-
GPT-4o	**PL-ZS**	85.8	76.2	71.3	64.6	15.0	28.9	27.4
GPT-4o	**decl. PL-1S**	92.6	87.8	**78.0**	**96.8**	44.0	58.8	-
GPT-4o	**PY-ZS**	87.7	87.6	75.7	69.7	26.5	38.1	-
GPT-4o	**decl. PY-1S**	90.1	87.5	72.6	96.4	**62.1**	**71.7**	-
CodeLlama13B	**decl. PL-1S**	43.0	34.1	35.8	62.5	14.6	28.5	-
CodeLlama13B	**decl. PY-1S**	38.9	29.1	34.7	74.3	20.0	35.2	-

Table 2: Performance on five datasets with different methods. The prompting methodologies are written in the following abbreviations: decl. (declarative style exemplar), 1S (one-shot), ZS (zero-shot), PL (Prolog), and PY (Python). For example, *decl. PL-1S* indicates that the LLM is prompted to generate Prolog code using one exemplar written in a declarative style.

[1]Throughout the rest of this paper, we use the abbreviations 'CodeLlama13B' or 'CodeL13B' to refer to the model *CodeLlama-13b-Python-hf*.

Experimental results in Table 2 show that GPT-4o CoT gets an accuracy greater than 90% for GSM8K and GSM-Reversed and outperforms the methods using programming languages[2]. CoT reasoning demonstrates its ability to handle shallow deductive reasoning effectively and to interpret and process the natural language semantics of the problem. These abilities allow it to navigate the implicit rules and perform the necessary calculations. In contrast, symbolic-solver-integrated methods may fail to capture the implicit rules in certain problems, as demonstrated in the failure modes discussed in Section 6.4.

The smaller LLM shows a different result. For CodeLlama13B, the methods using programming languages significantly outperform those using COT. Even their performance on GSM-Hard is better than COT's performance on the original GSM8K. It shows that symbolic-solver-integrated methods could be advantageous when the problem involves a reasoning complexity beyond the LLM's capability.

For GSM-Hard and ZebraLogic, the symbolic-solver-integrated methods with one-shot prompts significantly outperform CoT. Claude 3.5 Sonnet and Llama-3.1-405B are the top two models in the leaderboard of ZebraLogic[3]. GPT-4o with Prolog or Python using one-shot prompts outperforms them in all metrics: *easy*, *hard*, and *average*.

The results on GSM-Hard show that LLMs encounter significant challenges with large numbers. This is where symbolic solvers have a distinct advantage. These methods excel at performing precise calculations using mathematical operations and logic, regardless of the size of the numbers.

LLMs with CoT also struggle with complex CSPs. They outperform zero-shot symbolic-solver-integrated methods only on simple problems. We observe that the LLM tends to use rule-based deductions rather than a trial-and-error strategy on CSPs. As the problem size increases, the interrelationships between features become more intricate. It becomes difficult to solve through the token-by-token autoregressive generation. Frequently, unfaithful reasoning is observed, especially in the hard-level puzzles.

Symbolic formalization, on the other hand, systematically translates each natural language sentence into a corresponding mathematical equation or logical statement. This method relies on symbolic solvers (e.g., SymPy library in Python and clpr library in Prolog) to interpret the translated formal language program and find solutions that satisfy all constraints. Symbolic solvers can reason robustly and faithfully. They also employ a backtracking mechanism suitable for problems requiring extensive search trees. When a variable has multiple potential values, one should attempt a value and backtrack if it leads to conflicts. This backtracking allows the solver to explore multiple potential solutions and revert to previous states if a conflict arises.

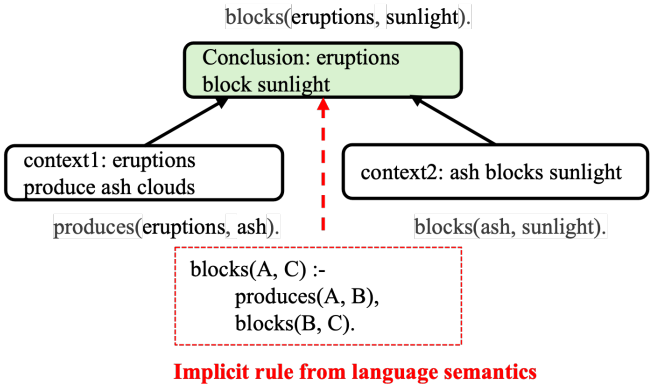
The one-shot prompting for symbolic-solver-integrated methods outperforms their corresponding zero-shot prompting over all the experiments except for solving the Reversed and Hard variations of GSM8K using Python. Prompting LLMs to generate codes in declarative style effectively improves the quality of generated code.

In zero-shot cases, Prolog code generation involves considerably more syntax errors than Python. This result is probably due to Python's widespread use and extensive repositories that provide ample training data.

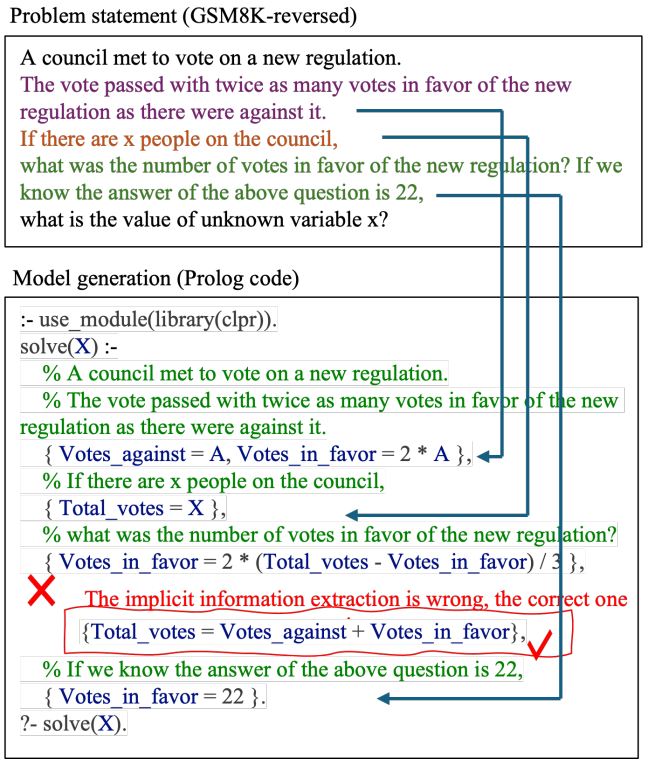
GPT-4o with Prolog gets much lower performance in the EntailmentBank dataset than GPT-4o-CoT. Modeling more natural and sophisticated problems with programs is significantly challenging. This difficulty arises from several inherent complexities in natural language (illustrated in Section 6.4) that are hard to be captured into formal languages.

[2]There is concern that LLMs perform well due to dataset contamination. In Zhang et al. [2024], GPT-4 performs better on a new test set GSM1K than GSM8K. This result validates its genuine reasoning capability.

[3]`https://huggingface.co/spaces/allenai/ZebraLogic`



(a) An implicit semantic rule omitted in EntailmentBank



(b) Incorrect implicit reasoning in GSM8K-reversed

Figure 2: Examples of LLMs failing to reason about implicit information in code generation

6 Analysis

6.1 Ablation Tests

One-shot vs Zero-shot Table 2 shows that one-shot prompting generally results in better performance. The improvement is more outstanding on the ZebraLogic dataset. The only exception is in GSM-Reversed and GSM-Hard with Python, where the accuracy decreases marginally.

Retries in code execution Furthermore, we prompt the LLMs to regenerate code when a syntax error is detected. Table 3 illustrates the impact on performance when the maximum number of attempts is set to one, two, and ten. Overall, the retry mechanism is particularly beneficial for Prolog

code generation and for generating code with smaller models, whereas the larger model can generate code with less syntax error at the first attempt, especially with Python.

Model	Prompt	GSM-Hard			ZebraLogic		
		1	2	10	1	2	10
GPT-4o	PL-ZS	49	63	71	43	43	43
GPT-4o	PL-1S	73	77	78	83	83	83
GPT-4o	PY-ZS	75	76	76	49	51	51
GPT-4o	PY-1S	71	73	73	84	84	84
CodeL13B	PL-1S	33	35	36	36	36	36
CodeL13B	PY-1S	29	33	35	32	35	41

Table 3: Accuracy in percentage with 1, 2, 10 of maximum number of attempts allowed. For ZebraLogic, we count the 160 test cases in 4 middle-level sizes: 3*3, 3*4, 4*3 and 4*4.

Figure 3 illustrates the distribution of retries required to obtain executable code using GPT-4o across Prolog and Python settings. For Prolog (Figures 3a and 3b), the zero-shot setting shows that many cases require several retries, with 80 out of 1319 problems still failing to produce executable code after ten attempts. Using a one-shot prompt significantly improves the success rate, reducing the number of cases that need multiple retries. For Python (Figures 21a and 21b), the zero-shot setting already yields high-quality executable code with few retries, and the one-shot prompt further improves the success rate by reducing the number of cases that require multiple attempts.

We also investigate the distribution of retries for ZebraLogic using GPT-4o with Prolog/Python codes, as shown in Appendix D. The results are consistent with GSM-Hard, and the same appendix includes distributions for both GSM-Hard and ZebraLogic using CodeLlama13B with Prolog/Python codes and one-shot prompts, indicating that CodeLlama13B struggles more to generate executable code than GPT-4o. This result is not surprising because of the significant difference in the model size.

6.2 Declarative Exemplar Prompting

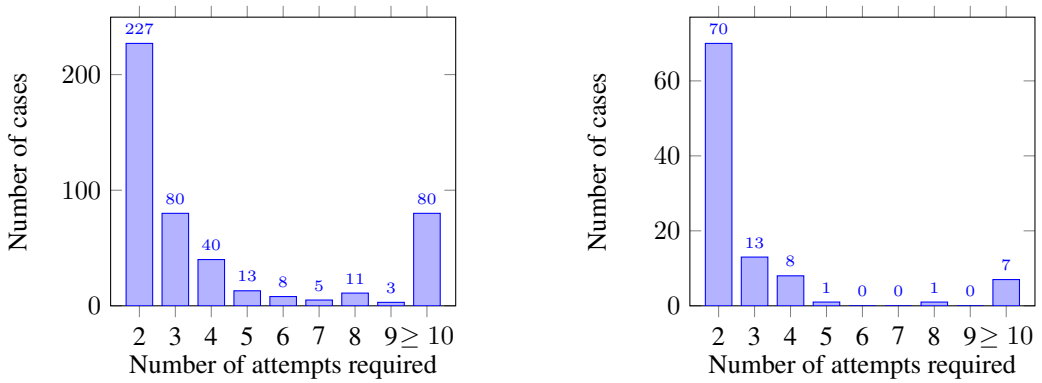
Prompting with a declarative exemplar enhances the correctness rate under almost all the experiment settings. Our proposed method addresses and resolves two types of problems shown in the zero-shot experiments.

LLMs tend to generate problematic imperative codes We observe that when directly prompting the LLM to generate Python or Prolog code to solve the problem, it tends to generate code in an imperative way. It often skips intermediate reasoning steps and leads to incorrect results. In contrast, with declarative prompting, the symbolic solver faithfully handles all intermediate reasoning rather than relying on the LLMs. The examples of comparing the two methods are illustrated in Appendix C.

Generated Prolog codes have excessive predicates LLMs tend to convert concepts into predicates in Prolog codes, generating excessive and unnecessary predicates. This result complicates the code and leads to more errors. We speculate that this issue arises because the LLM is pre-trained on Prolog code generation using logic deduction problems, where concepts are normally expressed using predicates. Our declarative exemplars help to translate the problem into concise code with a clean structure, significantly improving correctness. Appendix C illustrates this failure mode and the subsequent improvement after applying the exemplar.

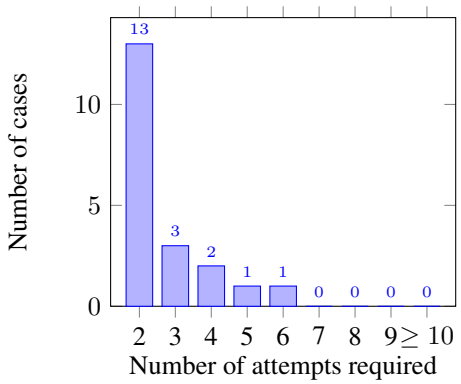
6.3 Difficulty Levels of Reasoning

We investigate the impact of the difficulty levels of reasoning on the ZebraLogic dataset. The number of houses and the number of features define the size of a ZebraLogic puzzle ranging from 2*2 to 6*6. According to the results generated and the ground truth in the ZeroEval repository, we calculated the accuracy of each puzzle size for Claude-3-5-sonnetand GPT-4o. We compare these two baselines with the accuracies using Python and Prolog codes with GPT-4o and CodeLlama13B. Part of the above results are illustrated in Table 4, and the rest are in the Appendix B.

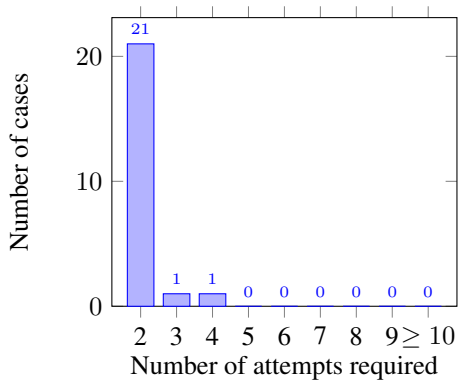


(a) Prolog zero-shot

(b) Prolog one-shot with a declarative exemplar



(c) Python zero-shot



(d) Python one-shot

Figure 3: Investigate the distribution of the number of retries needed to recover from syntax errors of the generated Prolog and Python code. We use GSM-Hard dataset, which contains 1319 problems in total, as an example.

Claude 3.5 holds the top position in the leaderboard. This model and GPT-4o perform well in easy puzzles, particularly in the first row of tables. However, the methods using code generation with GPT-4o still outperform them by accuracies larger than 97.5 in the first row.

When the size of the puzzle increases, the performance of Claude 3.5 and GPT-4o rapidly decreases. For example, the accuracy of sizes 3*3 and 4*4 from Claude 3.5 are 62.5 and 10, respectively. While the methods using code generation with GPT-4o achieve 90 and 72.5 respectively. Even CodeLlama13B outperforms the two CoT baselines on the hard-level puzzles, except for the size 4*2. These results show that code generation with declarative methods is more advantageous on some complex CSPs like ZebraLogic.

Errors made by LLMs with CoT prompting in complex reasoning We observe three representative types of errors that might lead to LLMs' failure.[4]

- •. **Partial Deduction**: At a reasoning step, the LLM could deduce only one conclusion when multiple are possible.
- •. **Unfaithful Deduction**: The reasoning process sometimes produces deductions that are not logically consistent with the given facts, leading to incorrect conclusions.
- •. **Misinterpretation of Problem Statements**: The LLM may misunderstand the problem statement, leading to incorrect formalization and flawed reasoning paths.

[4]Examples are illustrated in the Appendix C

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	100	97.5	97.5
3	92.5	87.5	90	87.5	72.5
4	100	75	75	75	65
5	90	70	62.5	42.5	30
6	87.5	50	17.5	15.0	12.5

(a) GPT-4o with Python code (one-shot)

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	100	97.5	97.5
3	92.5	90.0	90.0	77.5	65.0
4	97.5	80.0	72.5	70.0	**0**
5	85.0	75	**0**	**0**	**0**
6	80.0	**0**	**0**	**0**	**0**

(b) GPT-4o with Prolog code (one-shot)

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	97.5	85.0	75.0	67.5	77.5
3	57.5	60.0	42.5	40.0	25.0
4	40	40	22.5	25	15
5	40	22.5	7.5	5	0.15
6	7.5	12.5	0	0	0

(c) CodeLlama13B with Python code (one-shot)

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	90	95	90
3	75	62.5	37.5	30	17.5
4	55	37.5	10	2.5	2.5
5	15	10	0	0	0
6	5	0	0	0	0

(d) Claude-3.5-Sonnet with CoT

Table 4: Accuracy of ZebraLogic puzzles

6.4 Failure Modes of Symbolic-Solver-Integrated Methods

Formalizing implicit rules Natural language is rich with implicit semantic rules that guide understanding and interpretation. These rules are often not explicitly stated but are understood through context and commonsense reasoning. Formalizing these implicit rules in formal languages is challenging because it requires making all underlying assumptions and contextual knowledge explicit. Figure 2 demonstrates such failures on different datasets.

Formalizing complex sentence structures Large Language Models often struggle with complex natural sentences, particularly those containing intricate clause structures. For example, *"when carbon dioxide in the atmosphere is absorbed by plants, the amount of carbon dioxide in the atmosphere is reduced"* might lead to misinterpretation or failure to capture the nuanced relationships between different sentence parts like in conditional or dependent clauses.

Formalizing Modal Logic Modal Logic [Russell and Norvig, 2016] deals with necessity and possibility, for example statements like *It is necessary that...* or *It is possible that....* Although modal logics do not exceed the expressiveness of FOL, they need to be meticulously defined, such as using Kripke models [Kripke, 1963]. This process can pose huge challenge the code generation capabilities of a language model.

7 Conclusion

Large language models (LLMs) show good performance in shallow reasoning problems like GSM8K and its variants. Using the symbolic-solver-integrated method helps with the reasoning capability for less powerful language models, such as CodeLlama-13B, or when the problems are complex and require trial-and-error or repeated backtracks, such as hard Zebra puzzles. Our proposal to use one-shot prompting with a declarative exemplar translates the natural language description of the problem into a declarative code in a sentence-to-sentence manner. This method improves the quality of the code generated by LLMs in the symbolic-solver-integrated method, both using a traditional declarative language (Prolog) and a popular language (Python).

References

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

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

Michael Hassid, Gabriel Synnaeve, Yossi Adi, and Roy Schwartz. Don't overthink it. preferring shorter thinking chains for improved llm reasoning. *arXiv preprint arXiv:2505.17813*, 2025.

Abhinav Kumar, Jaechul Roh, Ali Naseh, Marzena Karpinska, Mohit Iyyer, Amir Houmansadr, and Eugene Bagdasarian. Overthink: Slowdown attacks on reasoning llms. *arXiv preprint arXiv:2502.02542*, 2025.

Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

Yi Hu, Xiaojuan Tang, Haotong Yang, and Muhan Zhang. Case-based or rule-based: How do transformers do the math? *arXiv preprint arXiv:2402.17709*, 2024.

Pranav Narayanan Venkit, Tatiana Chakravorti, Vipul Gupta, Heidi Biggs, Mukund Srinath, Koustava Goswami, Sarah Rajtmajer, and Shomir Wilson. " confidently nonsensical?": A critical survey on the perspectives and challenges of'hallucinations' in nlp. *arXiv preprint arXiv:2404.07461*, 2024.

Matthew Dahl, Varun Magesh, Mirac Suzgun, and Daniel E. Ho. Hallucinating law: Legal mistakes with large language models are pervasive, 2024. URL `https://hai.stanford.edu/news/hallucinating-law-legal-mistakes-large-language-models-are-pervasive`.

Panagiotis Giadikiaroglou, Maria Lymperaiou, Giorgos Filandrianos, and Giorgos Stamou. Puzzle solving using reasoning of large language models: A survey. *arXiv preprint arXiv:2402.11291*, 2024.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*, 2023.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36, 2024.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length generalization. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*, 2023.

Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. Shortcut learning in deep neural networks. *Nature Machine Intelligence*, 2(11):665–673, 2020.

Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. *arXiv preprint arXiv:2310.15164*, 2023.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

Y. Fleureau, L. Jia, E. Beeching, L. Tunstall, B. Lipkin, R. Soletskyi, S. C. Huang, and K. Rasul. How numinamath won the 1st aimo progress prize, 2024. URL `https://huggingface.co/blog/winning-aimo-progress-prize`.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, et al. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. Proofwriter: Generating implications, proofs, and abductive statements over natural language. *arXiv preprint arXiv:2012.13048*, 2020.

Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. *arXiv preprint arXiv:2210.01240*, 2022.

Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

Christian Holzbaur. Ofai clp (q, r) manual. Technical report, Citeseer, 1995.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Peizhong Gao, Ao Xie, Shaoguang Mao, Wenshan Wu, Yan Xia, Haipeng Mi, and Furu Wei. Meta reasoning for large language models. *arXiv preprint arXiv:2406.11698*, 2024.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.

Bhavana Dalvi, Peter Jansen, Oyvind Tafjord, Zhengnan Xie, Hannah Smith, Leighanna Pipatanangkura, and Peter Clark. Explaining answers with entailment trees. *arXiv preprint arXiv:2104.08661*, 2021.

Bill Yuchen Lin. ZeroEval: A Unified Framework for Evaluating Language Models, July 2024. URL `https://github.com/yuchenlin/ZeroEval`.

Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Tiffany Zhao, Pranav Raja, Dylan Slack, Qin Lyu, et al. A careful examination of large language model performance on grade school arithmetic. *arXiv preprint arXiv:2405.00332*, 2024.

Saul A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9:67–96, 1963. URL `https://api.semanticscholar.org/CorpusID:18604212`.

A Prompts And Generations

Problem statement (GSM-Hard)

A new program had 531811 downloads in the first month. The number of downloads in the second month was three times as many as the downloads in the first month, but then reduced by 30% in the third month. How many downloads did the program have total over the three months?

Model's Generation (Python code)

```python
# A new program had 531811 downloads in the first month.
downloads_first_month = 531811

# The number of downloads in the second month was three times as many as the downloads in the first month
downloads_second_month = downloads_first_month * 3

# but then reduced by 30% in the third month.
downloads_third_month = downloads_second_month * (1 - 0.30)

# How many downloads did the program have total over the three months?
total_downloads = downloads_first_month + downloads_second_month + downloads_third_month

# Print the solution
print(f'Total downloads over the three months: {total_downloads}')
```

Figure 4: GPT-4o with Python on GSM-Hard

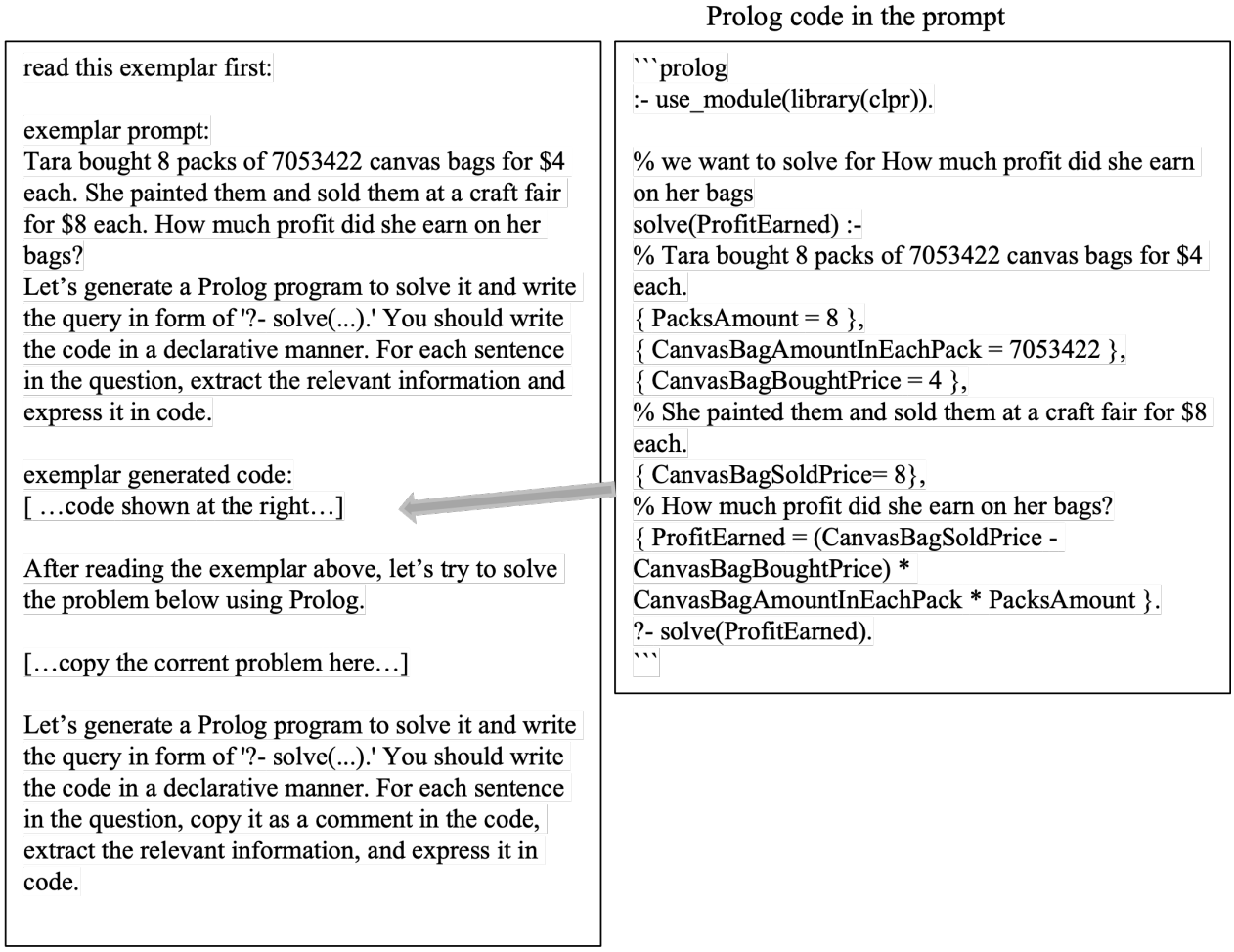


Figure 5: Prompt with a declarative Prolog exemplar for GSM-Hard

Problem statement (GSM-Hard)

A new program had 531811 downloads in the first month. The number of downloads in the second month was three times as many as the downloads in the first month, but then reduced by 30% in the third month. How many downloads did the program have total over the three months?

Model's Generation (Prolog code)

```prolog
:- use_module(library(clpr)).

% we want to solve for how many downloads did the program have total over the three months
solve(TotalDownloads) :-
% A new program had 531811 downloads in the first month.
{ FirstMonthDownloads = 531811 },
% The number of downloads in the second month was three times as many as the downloads in the first month,
{ SecondMonthDownloads = 3 * FirstMonthDownloads },
% but then reduced by 30% in the third month.
{ ThirdMonthDownloads = SecondMonthDownloads * (1 - 0.30)},
% How many downloads did the program have total over the three months?
{ TotalDownloads = FirstMonthDownloads + SecondMonthDownloads + ThirdMonthDownloads }.

?- solve(TotalDownloads).
```

Figure 6: Generation from GPT-4o with Prolog on GSM-Hard

Prompt with one-shot

read this exemplar first:

exemplar prompt:

There are 3 houses, numbered 1 to 3 from left to right, as seen from across the street. Each house is occupied by a different person. Each house has a unique attribute for each of the following characteristics:
- Each person has a unique name: `Arnold`, `Peter`, `Eric`
- Everyone has something unique for lunch: `spaghetti`, `grilled cheese`, `pizza`
- Each person has a favorite color: `yellow`, `white`, `red`

Clues:
1. The person whose favorite color is red and the person who loves yellow are next to each other.
2. The person who loves the spaghetti eater is the person who loves yellow.
3. The person who is a pizza lover and the person whose favorite color is red are next to each other.
4. Peter is directly left of the person who loves the spaghetti eater.
5. The person who loves the spaghetti eater is Eric.

Let's generate a Python program to solve it and print out the final answer.

Exemplar generated code:
[… code here …]

After reading the exemplar above, let's try to solve the problem below in Python:

[…copy the corrent problem here…]

Let's generate a Python program to solve it and print out the final answer.

Code in the prompt

```python
from constraint import Problem, AllDifferentConstraint
# Define the problem
problem = Problem()
# Define the houses
houses = range(1, 4) # House numbers are 1, 2, 3

# Define the names, lunches, and favorite colors
names = ['Arnold', 'Peter', 'Eric']
lunches = ['spaghetti', 'grilled cheese', 'pizza']
colors = ['yellow', 'white', 'red']
# Add variables
problem.addVariables(names, houses)
problem.addVariables(lunches, houses)
problem.addVariables(colors, houses)
# All values must be different
problem.addConstraint(AllDifferentConstraint(), names)
problem.addConstraint(AllDifferentConstraint(), lunches)
problem.addConstraint(AllDifferentConstraint(), colors)
# Constraints based on the problem
# 1. The person whose favorite color is red and the person who loves yellow are next to each other.
problem.addConstraint(lambda r, y: abs(r - y) == 1, ("red", "yellow"))

# 2. The person who loves the spaghetti eater is the person who loves yellow.
problem.addConstraint(lambda s, y: s == y, ("spaghetti", "yellow"))

# 3. The person who is a pizza lover and the person whose favorite color is red are next to each other.
problem.addConstraint(lambda p, r: abs(p - r) == 1, ("pizza", "red"))

# 4. Peter is directly left of the person who loves the spaghetti eater.
problem.addConstraint(lambda p, s: p == s - 1, ("Peter", "spaghetti"))

# 5. The person who loves the spaghetti eater is Eric.
problem.addConstraint(lambda s, e: s == e, ("spaghetti", "Eric"))

# Find a solution
solution = problem.getSolution()

# Print the solution
if solution:
    for i in houses:
        for name in names:
            if solution[name] == i:
                person_at_house = name
        for lunch in lunches:
            if solution[lunch] == i:
                lunch_at_house = lunch
        for color in colors:
            if solution[color] == i:
                color_at_house = color
        print(f"House {i}: Name: {person_at_house}, Lunch: {lunch_at_house}, Favorite Color: {color_at_house}")
else:
    print("No solution found")
```

Figure 7: Prompt with a declarative Python exemplar for ZebraLogic

Model's Generation (par1)

To solve this problem, we can use the `python-constraint` library, which allows us to define variables, constraints, and find solutions to constraint satisfaction problems. Below is the Python code to solve the given problem:

```python
from constraint import Problem, AllDifferentConstraint

# Define the problem
problem = Problem()

# Define the houses
houses = range(1, 5) # House numbers are 1, 2, 3, 4

# Define the names, hair colors, occupations, and favorite drinks
names = ['Arnold', 'Peter', 'Alice', 'Eric']
hair_colors = ['black', 'red', 'blonde', 'brown']
occupations = ['engineer', 'artist', 'doctor', 'teacher']
drinks = ['coffee', 'water', 'tea', 'milk']

# Add variables
problem.addVariables(names, houses)
problem.addVariables(hair_colors, houses)
problem.addVariables(occupations, houses)
problem.addVariables(drinks, houses)

# All values must be different
problem.addConstraint(AllDifferentConstraint(), names)
problem.addConstraint(AllDifferentConstraint(), hair_colors)
problem.addConstraint(AllDifferentConstraint(), occupations)
problem.addConstraint(AllDifferentConstraint(), drinks)

# Constraints based on the problem
# 1. The person who has brown hair is not in the second house.
problem.addConstraint(lambda b: b != 2, ("brown",))

# 2. The person who is a doctor is Peter.
problem.addConstraint(lambda d, p: d == p, ("doctor", "Peter"))

# 3. The person who is a doctor is the one who only drinks water.
problem.addConstraint(lambda d, w: d == w, ("doctor", "water"))

# 4. The person who is an artist is somewhere to the left of the person who is an engineer.
problem.addConstraint(lambda a, e: a < e, ("artist", "engineer"))

# 5. The person who likes milk is the person who has blonde hair.
problem.addConstraint(lambda m, b: m == b, ("milk", "blonde"))

# 6. The person who has red hair is Alice.
problem.addConstraint(lambda r, a: r == a, ("red", "Alice"))

# 7. The person who is an artist and Arnold are next to each other.
problem.addConstraint(lambda a, ar: abs(a - ar) == 1, ("artist", "Arnold"))

# 8. The person who is an artist is in the third house.
problem.addConstraint(lambda a: a == 3, ("artist",))
```

Figure 8: Generation from GPT-4o with Python on ZebraLogic (Part1)

Model's Generation (part2)

```
# 9. The person who likes milk is somewhere to the left of the tea drinker.
problem.addConstraint(lambda m, t: m < t, ("milk", "tea"))

# 10. Arnold is somewhere to the right of Eric.
problem.addConstraint(lambda ar, e: ar > e, ("Arnold", "Eric"))

# 11. The person who has black hair is the coffee drinker.
problem.addConstraint(lambda b, c: b == c, ("black", "coffee"))

# Find a solution
solution = problem.getSolution()

# Print the solution
if solution:
    for i in houses:
        for name in names:
            if solution[name] == i:
                person_at_house = name
        for hair_color in hair_colors:
            if solution[hair_color] == i:
                hair_color_at_house = hair_color
        for occupation in occupations:
            if solution[occupation] == i:
                occupation_at_house = occupation
        for drink in drinks:
            if solution[drink] == i:
                drink_at_house = drink
        print(f"House {i}: Name: {person_at_house}, Hair Color: {hair_color_at_house}, Occupation:
                                {occupation_at_house}, Favorite Drink: {drink_at_house}")
else:
    print("No solution found")
```

This code sets up the problem with the given constraints and uses the `python-constraint` library to find a solution. It then prints out the attributes of each house.

Figure 9: Generation from GPT-4o with Python on ZebraLogic (Part1)

Prompt with one-shot

read this exemplar first:

exemplar prompt:

There are 3 houses, numbered 1 to 3 from left to right, as seen from across the street. Each house is occupied by a different person. Each house has a unique attribute for each of the following characteristics:
- Each person has a unique name: `Arnold`, `Peter`, `Eric`
- Everyone has something unique for lunch: `spaghetti`, `grilled cheese`, `pizza`
- Each person has a favorite color: `yellow`, `white`, `red`

Clues:
1. The person whose favorite color is red and the person who loves yellow are next to each other.
2. The person who loves the spaghetti eater is the person who loves yellow.
3. The person who is a pizza lover and the person whose favorite color is red are next to each other.
4. Peter is directly left of the person who loves the spaghetti eater.
5. The person who loves the spaghetti eater is Eric.

Let's generate a Prolog program to solve it and write the query in form of '?- solve(...).'

Exemplar generated code:
[… code here …]

After reading the exemplar above, let's try to solve the problem below in Prolog:

[…copy the corrent problem here…]

Let's generate a Prolog program to solve it and write the query in form of '?- solve(...).'

Code in the prompt

```prolog
% Define the structure of the puzzle
solve(Solution) :-
Solution = [house(1, Name1, Lunch1, Color1),
house(2, Name2, Lunch2, Color2),
house(3, Name3, Lunch3, Color3)],

% Define unique names
member(house(_, arnold, _, _), Solution),
member(house(_, peter, _, _), Solution),
member(house(_, eric, _, _), Solution),

% Define unique lunches
member(house(_, _, spaghetti, _), Solution),
member(house(_, _, grilled_cheese, _), Solution),
member(house(_, _, pizza, _), Solution),

% Define unique colors
member(house(_, _, _, yellow), Solution),
member(house(_, _, _, white), Solution),
member(house(_, _, _, red), Solution),

% Clues from the puzzle
% 1. The person whose favorite color is red and the person who loves yellow are next to each other.
(member(house(RedPos, _, _, red), Solution),
member(house(YellowPos, _, _, yellow), Solution), abs(RedPos - YellowPos) =:= 1),

% 2. The person who loves the spaghetti eater is the person who loves yellow.
member(house(YellowPos, _, spaghetti, yellow), Solution),

% 3. The person who is a pizza lover and the person whose favorite color is red are next to each other.
(member(house(PizzaPos, _, pizza, _), Solution),
member(house(RedPos, _, _, red), Solution), abs(PizzaPos - RedPos) =:= 1),

% 4. Peter is directly left of the person who loves the spaghetti eater.
member(house(PeterPos, peter, _, _), Solution),
YellowPos is PeterPos + 1,

% 5. The person who loves the spaghetti eater is Eric.
member(house(YellowPos, eric, spaghetti, yellow), Solution).

% Query
?- solve(Solution).

```

Figure 10: Prompt with a declarative Prolog exemplar for ZebraLogic

Model's Generation

```prolog
% Define the structure of the puzzle
solve(Solution) :-
Solution = [house(1, Name1, Hair1, Occupation1, Drink1),
house(2, Name2, Hair2, Occupation2, Drink2),
house(3, Name3, Hair3, Occupation3, Drink3),
house(4, Name4, Hair4, Occupation4, Drink4)],
% Define unique names
member(house(_, arnold, _, _, _), Solution),
member(house(_, peter, _, _, _), Solution),
member(house(_, alice, _, _, _), Solution),
member(house(_, eric, _, _, _), Solution),
% Define unique hair colors
member(house(_, _, black, _, _), Solution),
member(house(_, _, red, _, _), Solution),
member(house(_, _, blonde, _, _), Solution),
member(house(_, _, brown, _, _), Solution),
% Define unique occupations
member(house(_, _, _, engineer, _), Solution),
member(house(_, _, _, artist, _), Solution),
member(house(_, _, _, doctor, _), Solution),
member(house(_, _, _, teacher, _), Solution),
% Define unique drinks
member(house(_, _, _, _, coffee), Solution),
member(house(_, _, _, _, water), Solution),
member(house(_, _, _, _, tea), Solution),
member(house(_, _, _, _, milk), Solution),
% Clues from the puzzle
% 1. The person who has brown hair is not in the second house.
\+ member(house(2, _, brown, _, _), Solution),
% 2. The person who is a doctor is Peter.
member(house(_, peter, _, doctor, _), Solution),
% 3. The person who is a doctor is the one who only drinks water.
member(house(_, peter, _, doctor, water), Solution),
% 4. The person who is an artist is somewhere to the left of the person who is an engineer.
member(house(ArtistPos, _, _, artist, _), Solution),
member(house(EngineerPos, _, _, engineer, _), Solution),
ArtistPos < EngineerPos,
% 5. The person who likes milk is the person who has blonde hair.
member(house(_, _, blonde, _, milk), Solution),
% 6. The person who has red hair is Alice.
member(house(_, alice, red, _, _), Solution),
% 7. The person who is an artist and Arnold are next to each other.
(member(house(ArtistPos, _, _, artist, _), Solution), member(house(ArnoldPos, arnold, _, _, _), Solution),
abs(ArtistPos - ArnoldPos) =:= 1),
% 8. The person who is an artist is in the third house.
member(house(3, _, _, artist, _), Solution),
% 9. The person who likes milk is somewhere to the left of the tea drinker.
member(house(MilkPos, _, _, _, milk), Solution),
member(house(TeaPos, _, _, _, tea), Solution),
MilkPos < TeaPos,
% 10. Arnold is somewhere to the right of Eric.
member(house(ArnoldPos, arnold, _, _, _), Solution),
member(house(EricPos, eric, _, _, _), Solution),
ArnoldPos > EricPos,
% 11. The person who has black hair is the coffee drinker.
member(house(_, _, black, _, coffee), Solution).
% Query
?- solve(Solution).
```

Figure 11: Generation from GPT-4o with Prolog on ZebraLogic

Problem statement (Entailment)

"context": "sent1: galaxies can be classified by shape sent2: celestial bodies / celestial objects are found in space sent3: light means light energy sent4: light is a kind of nonliving thing sent5: calculation means measurement sent6: quasars are located in galaxies sent7: the milky way galaxy appears as a glowing band across the night sky sent8: both means two sent9: measuring is used for describing an object sent10: determining often requires measuring / observing sent11: best means most positive sent12: a galaxy is a kind of structure sent13: measuring sometimes requires recording / learning an amount sent14: approximately means about sent15: seeing requires light sent16: amount is a property of something and includes ordered values of none / least / little / some / half / much / many / most / all sent17: a galaxy contains dusts / stars / planets sent18: measuring is a kind of observing sent19: the milky way is a kind of galaxy sent20: light years ly are an astronomy unit used for measuring length / distance generally used for values between 1 and 27000000000 sent21: stars are located light years apart from each other sent22: light rays means light sent23: a galaxy is made of stars sent24: a measurement is a kind of data sent25: distance moved / distance travelled is a measure of how far an object moves",
"question": "Which measurement is best expressed in light-years?"

Figure 12: A problem in EntailmentBank

Prompt (Chain of thought)

read the context: sent1: galaxies can be classified by shape sent2: celestial bodies / celestial objects are found in space sent3: light means light energy sent4: light is a kind of nonliving thing sent5: calculation means measurement sent6: quasars are located in galaxies sent7: the milky way galaxy appears as a glowing band across the night sky sent8: both means two sent9: measuring is used for describing an object sent10: determining often requires measuring / observing sent11: best means most positive sent12: a galaxy is a kind of structure sent13: measuring sometimes requires recording / learning an amount sent14: approximately means about sent15: seeing requires light sent16: amount is a property of something and includes ordered values of none / least / little / some / half / much / many / most / all sent17: a galaxy contains dusts / stars / planets sent18: measuring is a kind of observing sent19: the milky way is a kind of galaxy sent20: light years ly are an astronomy unit used for measuring length / distance generally used for values between 1 and 27000000000 sent21: stars are located light years apart from each other sent22: light rays means light sent23: a galaxy is made of stars sent24: a measurement is a kind of data sent25: distance moved / distance travelled is a measure of how far an object moves
Now answer the following question: Which measurement is best expressed in light-years?

Let's think step by step and find the answer to the question by deductive reasoning on the context sentences. Give the final answer surrounded by hashtags, for example: 'the final answer is #the rotation of Earth.#'.

Generation (GPT-4o)

To determine which measurement is best expressed in light-years, we need to consider the context sentences that mention light-years and related concepts.

1. Sent20: "light years ly are an astronomy unit used for measuring length / distance generally used for values between 1 and 27000000000."
2. Sent21: "stars are located light years apart from each other."

From these sentences, we can deduce that light-years are used to measure astronomical distances, particularly the distances between stars.

Given this information, the measurement best expressed in light-years is the distance between stars.

Therefore, the final answer is #the distance between stars.#

Figure 13: A Chain-Of-Thoughts Prompt for EntailmentBank

Prompt (Prolog)

read the context: sent1: galaxies can be classified by shape sent2: celestial bodies / celestial objects are found in space sent3: light means light energy sent4: light is a kind of nonliving thing sent5: calculation means measurement sent6: quasars are located in galaxies sent7: the milky way galaxy appears as a glowing band across the night sky sent8: both means two sent9: measuring is used for describing an object sent10: determining often requires measuring / observing sent11: best means most positive sent12: a galaxy is a kind of structure sent13: measuring sometimes requires recording / learning an amount sent14: approximately means about sent15: seeing requires light sent16: amount is a property of something and includes ordered values of none / least / little / some / half / much / many / most / all sent17: a galaxy contains dusts / stars / planets sent18: measuring is a kind of observing sent19: the milky way is a kind of galaxy sent20: light years ly are an astronomy unit used for measuring length / distance generally used for values between 1 and 27000000000 sent21: stars are located light years apart from each other sent22: light rays means light sent23: a galaxy is made of stars sent24: a measurement is a kind of data sent25: distance moved / distance travelled is a measure of how far an object moves
Now answer the following question: Which measurement is best expressed in light-years?

Let's generate a Prolog program to solve it and write the query in form of '?- solve(...).'. Include the original sentence in the comments and convert each sentence into a rule or a fact and convert the question into a query. You are allowed to define extra helper predicates.

Figure 14: Prompt with a declarative Prolog exemplar for EntailmentBank

Model's Generation (Entailment problem, using prolog, part1)

Sure, let's convert the given sentences into Prolog facts and rules, and then create a query to answer the question "Which measurement is best expressed in light-years?".

```prolog
% Sentences converted into Prolog facts and rules

% sent1: galaxies can be classified by shape
classified_by_shape(galaxies).

% sent2: celestial bodies / celestial objects are found in space
found_in_space(celestial_bodies).
found_in_space(celestial_objects).

% sent3: light means light energy
means(light, light_energy).

% sent4: light is a kind of nonliving thing
nonliving_thing(light).

% sent5: calculation means measurement
means(calculation, measurement).

% sent6: quasars are located in galaxies
located_in(quasars, galaxies).

% sent7: the milky way galaxy appears as a glowing band across the night sky
appears_as(milky_way_galaxy, glowing_band_across_night_sky).

% sent8: both means two
means(both, two).

% sent9: measuring is used for describing an object
used_for(measuring, describing_object).

% sent10: determining often requires measuring / observing
requires(determining, measuring).
requires(determining, observing).

% sent11: best means most positive
means(best, most_positive).

% sent12: a galaxy is a kind of structure
kind_of(galaxy, structure).

% sent13: measuring sometimes requires recording / learning an amount
requires(measuring, recording).
requires(measuring, learning_amount).

% sent14: approximately means about
means(approximately, about).

% sent15: seeing requires light
requires(seeing, light).
```

Figure 15: Generation from GPT-4o with Prolog on EntailmentBank (Part1)

Model's Generation (Entailment problem, using prolog, part2)

% sent16: amount is a property of something and includes ordered values of none / least / little / some / half / much / many / most / all
property(amount, [none, least, little, some, half, much, many, most, all]).

% sent17: a galaxy contains dusts / stars / planets
contains(galaxy, dusts).
contains(galaxy, stars).
contains(galaxy, planets).

% sent18: measuring is a kind of observing
kind_of(measuring, observing).

% sent19: the milky way is a kind of galaxy
kind_of(milky_way, galaxy).

% sent20: light years ly are an astronomy unit used for measuring length / distance generally used for values between 1 and 27000000000
unit(light_years, astronomy).
used_for(light_years, measuring_length).
used_for(light_years, measuring_distance).
range(light_years, 1, 27000000000).

% sent21: stars are located light years apart from each other
located_apart(stars, light_years).

% sent22: light rays means light
means(light_rays, light).

% sent23: a galaxy is made of stars
made_of(galaxy, stars).

% sent24: a measurement is a kind of data
kind_of(measurement, data).

% sent25: distance moved / distance travelled is a measure of how far an object moves
measure_of(distance_moved, how_far_object_moves).
measure_of(distance_travelled, how_far_object_moves).

% Helper predicate to determine if a measurement is best expressed in light-years
best_expressed_in_light_years(Measurement) :-
used_for(light_years, measuring_length),
used_for(light_years, measuring_distance),
measure_of(Measurement, how_far_object_moves).

% Query to answer the question
?- solve(Measurement).
solve(Measurement) :- best_expressed_in_light_years(Measurement).
```

To run the query, you would use:
```prolog
?- solve(Measurement).
```

This will determine which measurement is best expressed in light-years based on the given facts and rules.

Figure 16: Generation from GPT-4o with Prolog on EntailmentBank (Part2)
```

B ZebraLogic Results

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	100	97.5	97.5
3	92.5	87.5	90	87.5	72.5
4	100	75	75	75	65
5	90	70	62.5	42.5	30
6	87.5	50	17.5	15.0	12.5

(a) Using GPT-4o with Python code (one-shot)

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	100	97.5	97.5
3	92.5	90.0	90.0	77.5	65.0
4	97.5	80.0	72.5	70.0	**0**
5	85.0	75	**0**	**0**	**0**
6	80.0	**0**	**0**	**0**	**0**

(b) Using GPT-4o with Prolog code (one-shot). The bold zeros indicate that the executor failed to return results due to a stack space overflow.

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	97.5	85.0	75.0	67.5	77.5
3	57.5	60.0	42.5	40.0	25.0
4	40	40	22.5	25	15
5	40	22.5	7.5	5	0.15
6	7.5	12.5	0	0	0

(c) CodeLlama13B with Python code (one-shot)

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	85	75	67.5	62.5	42.5
3	50	55	32.5	25	12.5
4	42.5	32.5	22.5	17.5	**0**
5	35	12.5	**0**	**0**	**0**
6	15	**0**	**0**	**0**	**0**

(d) CodeLlama13B with Prolog code (one-shot). The bold zeros indicate that the executor failed to return results due to a stack space overflow.

Table 5: Accuracies of symbolic-solver-integrated methods on ZebraLogic puzzles

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	90	82.5	70
3	65	37.5	25	30	12.5
4	45	20	12.5	0	2.5
5	12.5	0	2.5	0	0
6	7.5	0	0	0	0

(a) Using GPT-4o

Houses	Number of Features				
	2	**3**	**4**	**5**	**6**
2	100	100	90	95	90
3	75	62.5	37.5	30	17.5
4	55	37.5	10	2.5	2.5
5	15	10	0	0	0
6	5	0	0	0	0

(b) Using Claude-3.5-Sonnet

Table 6: Accuracies of CoT on ZebraLogic puzzles

C Failure modes

Problem statement (Zebra Puzzle, size 3*3)

There are 3 houses, numbered 1 to 3 from left to right, as seen from across the street. Each house is occupied by a different person. Each house has a unique attribute for each of the following characteristics:
- Each person has a unique name: Eric, Peter, Arnold
- People have unique favorite book genres: romance, science fiction, mystery
- Everyone has something unique for lunch: grilled cheese, spaghetti, pizza

Clues:
1. Eric is directly left of the person who loves the spaghetti eater.
2. The person who loves science fiction books is in the third house.
3. The person who loves mystery books is somewhere to the right of Eric.
4. There is one house between the person who is a pizza lover and Peter.

Generation from GPT-4o using a Chain-Of-Thought prompt

Let's solve the puzzle step-by-step using the given clues:
Clues:
…
Step-by-Step Reasoning:
1.Clue 2 states that the person who loves science fiction books is in the third house.
House 3: Book Genre = Science Fiction
2.Clue 3 states that the person who loves mystery books is somewhere to the right of Eric. This means Eric cannot be in House 3.
✗ –**Partial deduction.** One conclusion missing: mystery book lover cannot be in house 3.

3.Clue 1 states that Eric is directly left of the person who loves the spaghetti eater. Therefore, Eric must be in House 1 or House 2.
✗ –**Partial deduction.** One conclusion missing: spaghetti eater cannot be in house 3.

4.Considering **Clue 4**, there is one house between the pizza lover and Peter. This means if Peter is in House 1, the pizza lover would be in House 3, or if Peter is in House 3, the pizza lover would be in House 1. **However, Peter cannot be in House 3 because the person who loves science fiction is there**.
✗ **- Misinterpretation:** Peter and fiction can be in the same house

5.From **Clue 4**, since Peter cannot be in House 3, Peter must be in House 1, and the pizza lover must be in House 3.
…

Figure 17: GPT-4o makes errors

Problem statement (Zebra Puzzle, size 3*3)

There are 3 houses, numbered 1 to 3 from left to right, as seen from across the street. Each house is occupied by a different person. Each house has a unique attribute for each of the following characteristics:
- Each person has a unique name: Arnold, Eric, Peter
- Everyone has something unique for lunch: grilled cheese, spaghetti, pizza
- People have unique hair colors: blonde, black, brown

Clues:
1. The person who has brown hair is not in the second house.
2. The person who is a pizza lover is not in the third house.
3. The person who has brown hair is not in the first house.
4. Peter is somewhere to the left of the person who has blonde hair.
5. The person who loves the spaghetti eater is Arnold.
6. Eric is the person who is a pizza lover.

Generation from GPT-4o using a Chain-Of-Thought prompt

Let's work through the clues step-by-step to solve the problem.
Clues:
...
Analysis:
1.Clue 1 and 3: The person with brown hair cannot be in house 1 or house 2. Therefore, the person with brown hair must be in house 3.
2.Clue 2 and 6: Eric loves pizza, and since the pizza lover cannot be in the third house, Eric must be in house 1 or house 2
3.Clue 4: Peter is to the left of the person with blonde hair. This means Peter cannot be in the last house.
4.Clue 5: Arnold loves spaghetti.
Based on the above analysis:
•**House 3** has the person with brown hair.
•**Eric** loves pizza and cannot be in house 3, so he is either in house 1 or house 2.
•**Peter** must be in house 1 or 2 (since he is to the left of the person with blonde hair).
•The person with blonde hair cannot be in house 1, so they must be in house 2 or 3.
Combining clues:
•Since Eric is the pizza lover and cannot be in house 3, and Peter cannot be in the last house, **Eric must be in house 1, and Peter must be in house 2**.
•Therefore, …

Unfaithful deduction: Peter can be in house1 and Eric in house 2

Figure 18: GPT-4o makes errors in ZebraLogic puzzles: unfaithful deduction

Problem statement (GSM8K-reversed)

Prompt:
A farmer is buying feed for his horses. He buys a variety of hay, oats, carrots and sugar cubes. Since sugar cubes are a rare treat, he only buys two 1-pound boxes of them for the whole stable. He only wants enough carrots to feed the horses while the vegetables are fresh, so he buys four 12-pound bags. Hay is the main diet of his horses, so he buys forty-two X-pound bales. Oats are a staple to supplement the hay, so he buys twenty 65-pound sacks. If his farm truck can carry 2250 pounds at a time, how many trips does the farmer need to transport all the feed? If we know the answer to the above question is 2, what is the maximum value of unknown variable 'X'?

Generation with Zero-shot

```
% Define the weights of the various feeds
sugar_cubes_weight(2).
carrots_weight(48). % 4 bags * 12 pounds each
oats_weight(1300). % 20 sacks * 65 pounds each

% Calculate the weight of hay given the number of bales and weight per bale (X)
hay_weight(X, HayWeight) :-
    HayWeight is 42 * X.

% Calculate total weight of all feeds
total_weight(X, TotalWeight) :-
    sugar_cubes_weight(SugarCubes),
    carrots_weight(Carrots),
    oats_weight(Oats),
    hay_weight(X, Hay),
    TotalWeight is SugarCubes + Carrots + Oats + Hay.

% Calculate the number of trips needed to transport the total weight
trips_needed(X, Trips) :-
    total_weight(X, TotalWeight),
    MaxWeight is 2250,
    Trips is ceiling(TotalWeight / MaxWeight).

% Find the maximum value of X such that the number of trips required is 2
max_X_for_two_trips(X) :-
    between(1, 1000, X), % Arbitrarily large upper limit for search space
    trips_needed(X, 2),
    !.

% Solve for X based on the problem constraints
solve(X) :-
    max_X_for_two_trips(X).

% The query to be used:
% ?- solve(X).
```

Generation with declarative exemplar

```
:- use_module(library(clpr)).

solve(X) :-
    % A farmer is buying feed for his horses.
    % He buys a variety of hay, oats, carrots, and sugar cubes.
    % Since sugar cubes are a rare treat, he only buys two ...
    { Sugar_cubes_weight = 2 * 1 },

    % He only wants enough carrots to feed the horses while the ...
    { Carrots_weight = 4 * 12 },

    % Hay is the main diet of his horses, so he buys forty-two ...
    { Hay_weight = 42 * X },

    % Oats are a staple to supplement the hay, so he buys twenty ...
    { Oats_weight = 20 * 65 },

    % If his farm truck can carry 2250 pounds at a time, how many ...
    { Total_feed_weight =
            Sugar_cubes_weight + Carrots_weight + Hay_weight + Oats_weight },
    { Trips_needed = Total_feed_weight / 2250 },

    % If we know the answer to the above question is 2, what is ...
    { Trips_needed = 2 }.

?- solve(X).
```

Using predictors creates a complicated solution and is easy to makes errors

Figure 19: GPT-4o generates excessive predicates

Problem statement (GSM8K-reversed)

Carmen has $X, Samantha has $25 more than Carmen, and Daisy has $50 more than Samantha. How much do all three girls have combined? If we know the answer to the above question is 400, what is the value of unknown variable 'X'?
Let's generate a Prolog program to solve it and write the query in form of '?- solve(...).'

Generation with Chain-Of-Thoughts prompt

Let's solve the problem step by step.
1.Define the amounts of money each girl has in terms of X:
1. Carmen has X dollars.
2. Samantha has X+25 dollars.
3. Daisy has X+25+50=X+75 dollars
2.Find the total amount of money all three girls have combined:
X+(X+25)+(X+75)
3.Simplify the expression:
X+X+25+X+75=3X+100
4.We are given that the total amount of money is 400 dollars:
3X+100=400
5.Solve for X:
•••
X=100

Generation with zero-shot prompt

```
% Calculate the values given the constraints
solve(X) :-
  Total = 400,
  % Calculate the actual value of X
  X is (Total - (25 + 50)) / 3,
% Start query
?- solve(X).
```

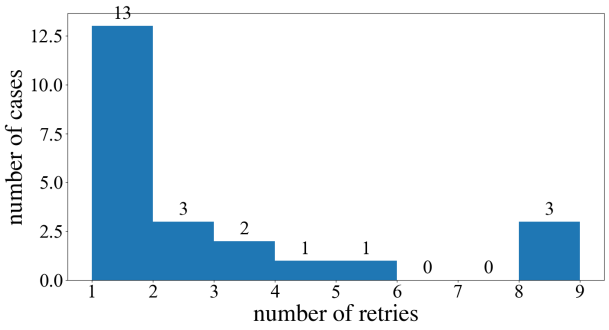
LLMs with zero shot tend to generate **imperative codes** and are easy to get wrong
The correct expression is: *X is (Total - (25 + 50 +25)) / 3*

Generation with one-shot prompt using a declarative exemplar

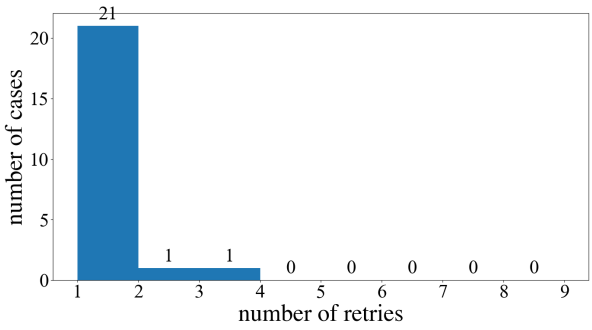
```
:- use_module(library(clpr)).
solve(X) :-
  % Carmen has $X,
  { Carmen_amount = X },
  % Samantha has $25 more than Carmen,
  { Samantha_amount = Carmen_amount + 25 },
  % and Daisy has $50 more than Samantha.
  { Daisy_amount = Samantha_amount + 50 },
  % How much do all three girls have combined?
  { Total_amount = Carmen_amount + Samantha_amount + Daisy_amount },
  % If we know the answer to the above question is 400, what is the value of unknown variable 'X'?
  { Total_amount = 400 }.
  ?- solve(X).
```

Figure 20: GPT-4o generates problematic imperative codes in GSM-Reversed

D Histogram Of Retries

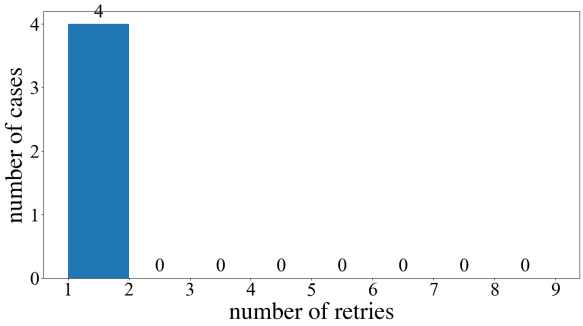


(a) Python code with zero-shot

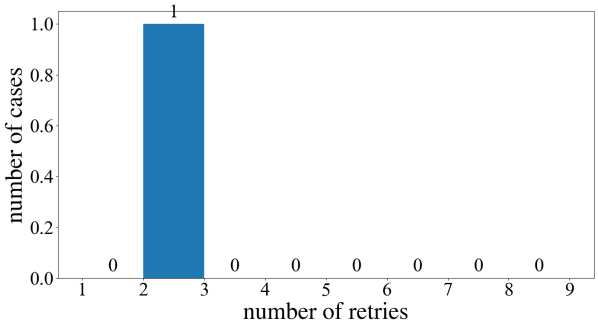


(b) Python code with one-shot

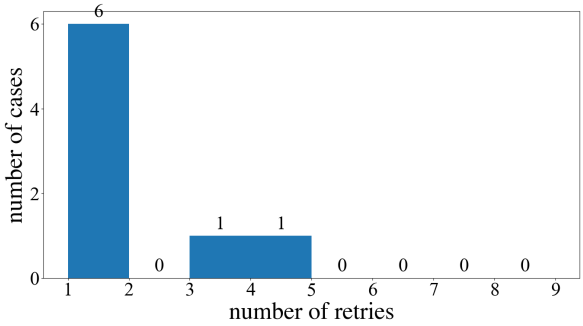
Figure 21: GSM-Hard using GPT-4o



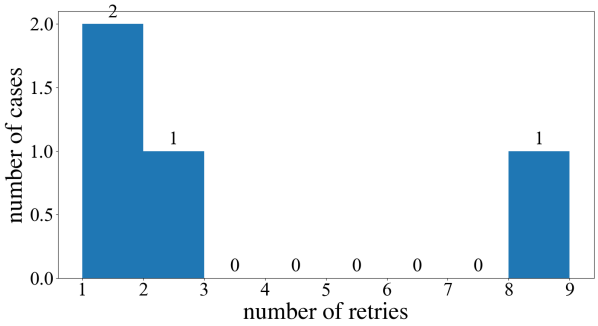
(a) Prolog code (Zero Shot)



(b) Prolog code (One Shot)

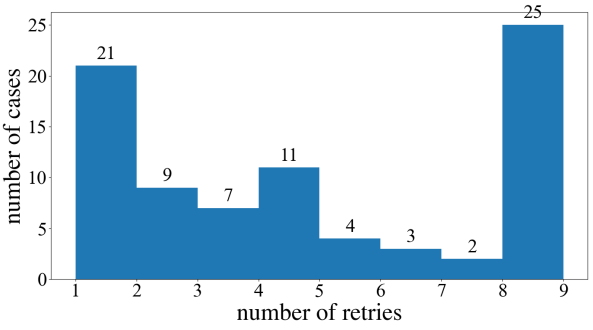


(c) Python code (Zero Shot)

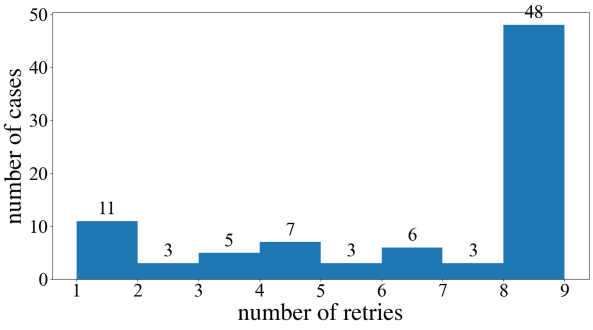


(d) Python code (One Shot)

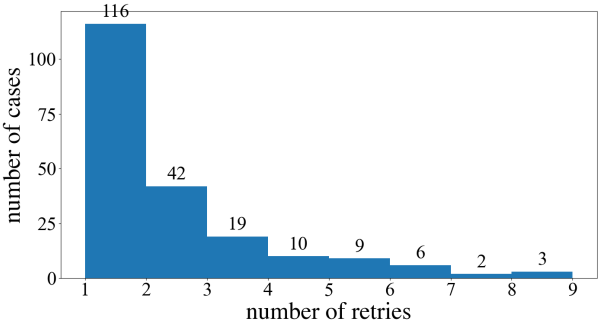
Figure 22: GPT4o for ZebraLogic puzzles



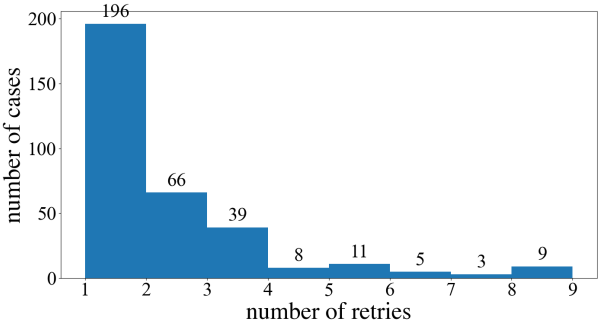
(a) Prolog code (One Shot) for ZebraLogic



(b) Python code (One Shot) for ZebraLogic



(c) Prolog code (One Shot) for GSM-Hard



(d) Python code (One Shot) for GSM-Hard

Figure 23: Investigate the retries: using CodeLlama13B for ZebraLogic puzzles and GSM-Hard