Draft-and-Prune Improves Auto-formalization in Logical Reasoning

Anonymous ACL submission

Abstract

Auto-formalization (AF) translates natural-language reasoning problems into solver-executable programs, enabling symbolic solvers to perform sound logical deduction. In practice, AF pipelines are brittle: programs may fail to execute, or execute but encode incorrect semantics. We propose Draft-and-Prune (D&P), an inference-time framework that improves AF-based logical reasoning via diversity and verification. D&P first *drafts* a natural-language plan and conditions program generation on it. It further *prunes* executable but contradictory or ambiguous formalizations, and aggregates surviving predictions by majority vote. Across four benchmarks (AR-LSAT, ProofWriter, PrOntoQA, LogicalDeduction), D&P substantially strengthens the AF-based reasoning without extra supervision. On AR-LSAT, it achieves 77.30% accuracy in the AF-only setting and 81.48% with a CoT fallback (GPT-4o), outperforming the strongest AF baseline CLOVER by 30.5 and 18.7 points. D&P also attains near-ceiling performance on the other benchmarks, including 100% on PrOntoQA and LogicalDeduction under our setup.

1 Introduction

Large language models (LLMs) achieve strong performance on many language tasks (Achiam et al., 2023; Hurst et al., 2024), yet they remain unreliable on deductive logical reasoning (Zhong et al., 2022; Srivastava et al., 2023). Symbolic solvers provide sound deductive inference, but they require inputs written in precise formal languages (e.g., logic programs) rather than natural language. Neuro-symbolic methods (Zhang et al., 2023) aim to combine the flexibility of LLMs with the rigor of symbolic reasoning. Among them, *auto-formalization* (AF) translates a natural-language problem into an executable formalization that a solver can run, thereby offloading deduction to a sound backend (Wu et al., 2022; Pan et al., 2023).

In practice, AF pipelines are brittle and often fail in two ways: (i) *syntactic failures*, where the generated formalization does not parse or execute, and (ii) *semantic unfaithfulness*, where it executes but does not capture the original intent. Prior systems (Pan et al., 2023) largely mitigate (i) via solver feedback (e.g., error messages) to repair syntax, but (ii) remains common because mapping natural language to a faithful symbolic encoding is sophisticated. Many AF frameworks **under-utilize solver-based reasoning**: they **generate only a narrow set of candidate formalizations** (often one plus a few repairs), assume a faithful encoding will emerge within a few attempts, and **under-explored the search space of symbolic formulations**.

To address such limitations, we propose Draft-and-Prune (D&P), a lightweight inference-time framework that: (i) *drafts* multiple high-level plans to induce diverse logic programs; (ii) *prunes* executable candidates that are not well-defined (e.g., contradictory or ambiguous); and (iii) aggregates answers from the remaining candidates.

D&P substantially improves performance on AR-LSAT (Zhong et al., 2022) by over 30%, a benchmark where existing auto-formalization baselines perform poorly. On the remaining datasets, including PrOntoQA (Saparov and He, 2023) and LogicalDeduction (Srivastava et al., 2023), D&P achieves near-ceiling accuracy. Overall, these results suggest that auto-formalization-based reasoning can be a reliable approach when combined with inference-time diversity and pruning.

Contributions. (i) We analyze brittleness in AF pipelines and characterize key failure modes. (ii) We propose Draft-and-Prune (D&P), an inference-time framework that improves reasoning accuracy via draft-induced diversity and verification-based pruning. (iii) We evaluate D&P on multiple deductive reasoning benchmarks, report both end-to-end accuracy and diagnostic metrics, and provide in-depth ablations and error analysis.

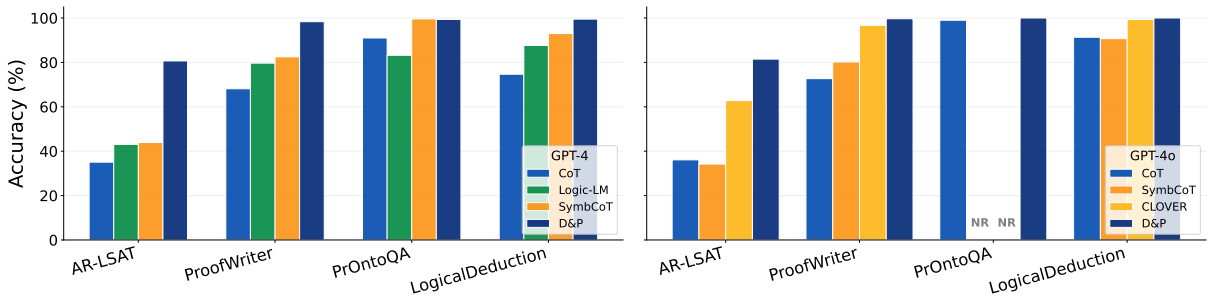


Figure 1: End-to-end accuracy (**Acc**) on four logical reasoning benchmarks. Left: GPT-4 results comparing CoT, Logic-LM, SymbCoT, and D&P. Right: GPT-4o results comparing CoT, SymbCoT, CLOVER, and D&P. "NR" denotes not reported results.

2 Background

2.1 Deductive logical reasoning

Deductive logical reasoning derives conclusions that follow necessarily from a given set of premises. Typical tasks include checking argument validity, proving theorems, and solving constraint problems. Such tasks demand both syntactic precision and semantic consistency, and differ from inductive predictions, where machine learning algorithms dominate and statistical generalization is the focus.

Formally, a set of premises P is given together with one or more candidate hypotheses Q_i. The goal is to determine the logical relationship between P and each Q_i, such as entailment ($P \models Q_i$), contradiction ($P \models \neg Q_i$), or consistency (i.e., $P \wedge Q_i$ is satisfiable).

2.2 Reasoning with LLM and Prompting

Benchmarking studies in natural-language reasoning suggest that direct generation by LLMs can produce outputs that are fluent yet logically invalid (Srivastava et al., 2023; Tafjord et al., 2021). Prompting-based methods provide procedural scaffolding to improve accuracy and reliability. Chain-of-Thought (CoT) prompting encourages intermediate reasoning steps (Wei et al., 2022), while self-ensembling (e.g., self-consistency) samples multiple reasoning traces and aggregates answers to improve accuracy and robustness (Wang et al., 2022). More recent methods explicitly structure the reasoning process with decomposition and planning, such as least-to-most prompting (Zhou et al., 2023) and plan-and-solve prompting (Wang et al., 2023). These ideas motivate our use of *drafted plans* as lightweight scaffolding to diversify candidates.

2.3 Reasoning based on symbolic solvers

Despite increasingly sophisticated prompting strategies, LLMs can be unreliable on tasks that require strict logical validity, since their training objective provides no guarantee of sound deduction. This has motivated neuro-symbolic methods that integrate neural generation with symbolic solvers (Zhang et al., 2023). A primary paradigm in this space is *auto-formalization* (AF), which translates natural-language problem statements into solver-executable formalizations (e.g., logic programs illustrated in Figure 3), that can be executed by symbolic solvers (Pan et al., 2023). Recent work has explored this direction and reported promising improvements over prompting-based baselines.

A representative system is Logic-LM (Pan et al., 2023), which performs LLM-based translation followed by iterative refinement using execution feedback from solvers to improve syntactic validity. More recent AF pipelines aim to address semantic unfaithfulness beyond syntactic correctness. For example, CLOVER (Ryu et al., 2024) decomposes the input into atomic clauses and formalizes each component before composing a final program. Overall, these approaches suggest that auto-formalization (AF) can be effective for improving logical reasoning accuracy.

In AF pipelines, failures can arise either from non-executable outputs (syntactic issues) or from executable but semantically incorrect encodings. Prior work typically reports diagnostic metrics that separate (i) whether a generated solver program is executable (*execution rate*) from (ii) whether executable programs yield the correct answer (*accuracy among executable formalizations*) to disentangle syntactic/tooling failures from semantic unfaithfulness. We follow this convention in our analysis; formal definitions are given in Section 4.3.

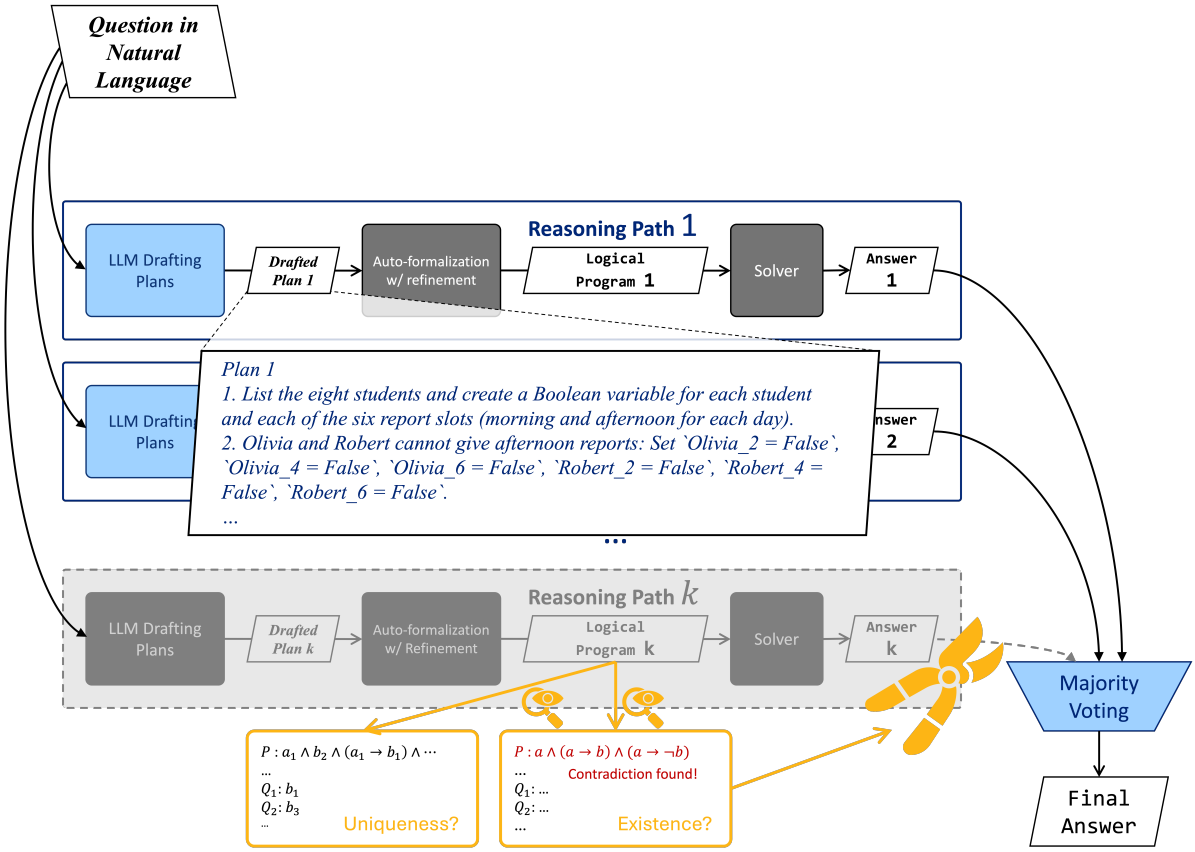


Figure 2: The algorithm pipeline of Draft-and-Prune

3 Draft-and-Prune

Overview. D&P performs *inference-time* ensembling over k independent *AF paths*. Each path i is a complete AF pipeline: *Draft* a plan p_i in natural language, *Generate* a formalization (e.g., a Z3Py program for AR-LSAT) z_i conditioned on p_i, *Repair* z_i using standard solver-feedback fixes until it executes, *Execute* z_i to derive a set of feasible answer options S_i, *Prune* ill-defined paths, and finally *Aggregate* answers from surviving paths by majority vote. Figure 2 illustrates the pipeline. All paths are independent samples; we do not perform tree search or branching within a path.

Correct formulation:

```
# Olivia and Robert cannot give afternoon reports
for student in ["Olivia", "Robert"]:
    for i in [2, 4, 6]:
        s.add(slot_map[f"{student}_{i}"] == False)
# Nina's condition affecting Helen and Irving
for i, next_day in [(1, 3), (3, 5)]:
    s.add(Implies(slot_map[f"Nina_{i}"],
            And(slot_map[f"Helen_{next_day}"],
                slot_map[f"Irving_{next_day}"])))
... # omit other codes for brevity
```

Output: [D]

Figure 3: A correct Z3Py program corresponds to an input question in AR-LSAT

Motivations. Our design is motivated by a common brittleness pattern in auto-formalization (AF) pipelines. Even when solver feedback is used to repair syntax, AF systems can fail either because the generated solver program does not execute (syntactic failures) or because it executes but encodes the wrong semantics (semantic unfaithfulness).

To disentangle these failure modes, we follow common AF reporting practice and use execution rate and accuracy among executable formalizations as complementary diagnostics that localize errors to executability versus semantics (formal definitions are provided in Section 4.3).

Prior AF baselines exhibit both types of failures on challenging inputs, suggesting that improving executability alone is insufficient and that search over semantically faithful encodings remains a key bottleneck.

This diagnosis directly motivates the two components of D&P. **Draft** encourages semantic diversity by first generating an explicit natural-language plan and then conditioning program generation on that plan, increasing the chance of producing a faithful encoding without destabilizing code generation. **Prune** improves reliability by filtering

executable but ill-defined programs (e.g., contradictory or answer-ambiguous encodings) using solver-verifiable well-definedness checks before selecting an answer. Finally, we aggregate predictions from surviving paths by majority vote.

3.1 Draft plans to induce semantic diversity

A key challenge in AF is that many distinct symbolic encodings can appear plausible from the same natural-language input, while only a small subset is semantically faithful. Directly sampling programs can significantly increase syntax/runtime errors. We therefore place stochasticity at the *plan* level: we prompt an LLM (with ICL examples) to draft n plans $\{p_i\}_{i=1}^{k}$ at temperature T_{DRAFT}.

Plan format. As illustrated in the middle of Figure 2, each plan p_i is a structured natural-language checklist that specifies: (i) entities/variables to introduce, (ii) domains and key relations, and (iii) how each textual constraint maps to solver assertions. Plans are the primary source of diversity across paths.

3.2 Prune paths to improve faithfulness

Contradictory Formulation:

```
# Formulate "Kyle and Lenore do not report"
s.add(days["Kyle"] == 0)
s.add(days["Lenore"] == 0)
# Formulate predicates: each student reports exactly once
# But forgot to exclude Kyle and Lenore here
for student in students:
    s.add(days[student] >= 1, days[student] <= 3)
    s.add(sessions[student] >= 0, sessions[student] <= 1)
```

Output: [] (Non-existence, should be pruned)

Ambiguous Formulation:

```
# only specifies there should be 2 reports each day
for d in range(len(days)):
    s.add(Sum([If(rep[i][d][s], 1, 0) for i in range(len(
        students)) for s in range(len(slots))]) == 2)
# forgets to constrain that each student reports exactly
# once, and produces a vague formulation.
```

Output: [A, B, D, E] (Non-uniqueness, should be pruned)

Figure 4: Two typical types of ill-defined logical programs which can be detected during inference

Executable programs can still encode the problem incorrectly due to semantic unfaithfulness. Since our benchmarks are closed-end and designed to have a unique answer, we prune paths whose induced answer is not well-defined.

Deriving feasible options. Executing a path program z_i yields a feasible option set $S_i \subseteq \mathcal{A}$. Operationally, S_i is obtained by running solver queries for each option $a \in \mathcal{A}$ (e.g., adding option-specific constraints or checking entailment/satisfiability, depending on the benchmark encoding) and collecting those options consistent with the program.

Well-definedness criteria. We keep path i iff:

- **Existence:** the encoding is not contradictory;
- **Uniqueness:** the encoding implies a unique answer.

If $|S_i| = 0$, the path is contradictory; if $|S_i| > 1$, the path is answer-ambiguous. Either case (Figure 4) will be pruned at inference time with verification by solver.

3.3 Majority vote over surviving paths

Each surviving path outputs a single answer a_i where $S_i = \{a_i\}$. We return the majority vote over the answers produced by surviving paths. If there is a tie (no strict majority), we choose the first-appearing answer for determinism. If all paths are pruned or unexecutable, we abstain, and the answer will be labeled as incorrect. This aggregation mirrors the spirit of self-consistency, which aggregates over multiple sampled reasoning paths, but here each path corresponds to an AF formalization-and-execution pipeline that could abstain.

LLM generation with in-context learning. Crucially, **both the drafted plans and the formalizations are generated by an LLM** during inference via fixed prompts and in-context learning (ICL) demonstrations. We use a prompt template for plan drafting, and a separate prompt template for formalization generation and syntax repair.

Hyperparameters. We draft/sample k paths with temperature $T_{\text{DRAFT}} = 1.0$ to induce diversity. We generate formalization, logic programs in our case, with greedy decoding ($T_{\text{GEN}} = 0$) for stability. Each path allows at most $R = 2$ rounds of solver-feedback repair. If no path survives pruning or there is a syntactic failure, we abstain, and the answer will be marked as incorrect.

Summary. D&P separates *where* we introduce diversity (plan drafting) from *where* we enforce stability (program generation/repair), and uses solver-based well-definedness checks to prune ill-posed paths before majority voting.

4 Experiments

4.1 Datasets

We evaluate our approach on four representative reasoning benchmarks, which jointly cover both natural and synthetic settings and assess different aspects of logical and deductive reasoning:

AR-LSAT is derived from analytical reasoning questions in the Law School Admission Test (LSAT) (Zhong et al., 2022). Each instance provides a set of natural-language premises P together with multiple candidate hypotheses $\{Q_1, \ldots, Q_m\}$. Each question specifies a target logical relation, including *"must be true"* ($P \models Q_i$), *"must be false"* ($P \models \neg Q_i$), *"could be true"* ($P \wedge Q_i$ is satisfiable), and *"could be false"* ($P \wedge \neg Q_i$ is satisfiable). This benchmark is widely regarded as challenging, with existing methods still far from perfect accuracy.

ProofWriter evaluates multi-step deductive reasoning over natural-language rules and facts. Each instance consists of premises P and a single hypothesis Q, which must be classified as entailed ($P \models Q$), contradicted ($P \models \neg Q$), or unknown (neither holds) (Tafjord et al., 2021).

PrOntoQA is a synthetic benchmark designed to test systematic deductive reasoning (Saparov and He, 2023). It follows the same single-hypothesis formulation as ProofWriter, but restricts labels to binary outcomes, where each hypothesis is either entailed or contradicted by the premises.

LogicalDeduction, from BigBench (Srivastava et al., 2023), also presents multiple candidate hypotheses under the same premises P, but each question asks for the unique hypothesis Q_i that is logically entailed by the premises, that is, the option satisfying $P \models Q_i$. Problems typically involve inferring object orderings or relations from a small set of constraints.

4.2 Baselines

To compare our method against existing LLM-based reasoning approaches, we consider a diverse set of baselines spanning prompting-based and AF-based methods: (1) **Direct** prompting and **CoT** (Wei et al., 2022); (2) **CoT-SC** (Wang et al., 2022), an enhanced decoding approach that samples multiple reasoning paths and aggregates their outputs to improve robustness and accuracy; (3) **DetermLR** (Sun et al., 2024), which transitions uncertain premises into determinate conclusions through iterative control and, potentially, external modules (e.g., memory); (4) **SymbCoT** (Xu et al., 2024), which injects symbolic reasoning operations into the chain-of-thought framework; (5) **Logic-LM** (Pan et al., 2023), which first translates the natural language problem into a symbolic form and then delegates reasoning to a symbolic solver, with syntax repair based on solver feedback over the translation process; (6) **Logic-LM++** (Kirtania et al., 2024), which extends Logic-LM by introducing multi-step refinement and pairwise comparisons; (7) **CLOVER** (Ryu et al., 2024), which introduces a compositional translation of natural language into first-order logic via logical dependency structures, coupled with verification through symbolic solvers to ensure semantic correctness.

4.3 Evaluation Metrics

We report end-to-end **accuracy** (**Acc**), defined as the fraction of instances whose *final* predicted answer matches the ground truth. Let N be the number of instances, and let $c_j \in \{0, 1\}$ indicate whether the final output for instance j is correct.

$$\mathbf{Acc} = \frac{1}{N} \sum_{j=1}^{N} c_j \tag{1}$$

To diagnose brittleness of AF-only methods (i.e., without fallback via prompting), we additionally report **execution rate** (**ExecRate**) and **accuracy among executable formalizations** (**Acc@Exec**). For an AF-only method, the system attempts to generate a solver program and execute it to obtain an answer. If no *valid* formalization is available for an instance (unexecutable or pruned), the system abstains and the instance is counted as incorrect.

Let $e_j \in \{0, 1\}$ indicate whether the generated formalization for instance j executes successfully (i.e., the solver program runs without errors/exceptions within a fixed timeout), and let $c_j^{\mathrm{AF}} \in \{0, 1\}$ indicate whether the executed formalization yields the correct answer (defined only when $e_j = 1$).

$$\mathbf{ExecRate} = \frac{1}{N} \sum_{j=1}^{N} e_j, \tag{2}$$

$$\mathbf{Acc@Exec} = \frac{\sum_{j=1}^{N} e_j c_j^{\mathrm{AF}}}{\sum_{j=1}^{N} e_j}. \tag{3}$$

For hybrid systems (e.g., Logic-LM with CoT fallback), the final answer may come from either the AF component or a fallback when AF fails; we report end-to-end accuracy without diagnostics.

4.4 Experimental Setup

Models and generation settings. We conduct our experiments using GPT-4 and GPT-4o, consistent with baselines. For all LLM API calls, the

Table 1: End-to-end accuracy (**Acc**) on four deductive reasoning benchmarks. We compare prompting-based methods and auto-formalization (AF) methods, without fallback (abstention) and with CoT fallback.

Paradigm	Method	Fallback Policy	LLM	End-to-end Accuracy (%)			
				AR-LSAT	ProofWriter	PrOntoQA	LogicalDeduction
Prompting	Direct	N/A	GPT-4	33.30	52.67	77.40	71.33
	CoT			35.06	68.11	91.00	74.67
	CoT-SC			–	69.33	93.40	74.67
	DetermLR			–	79.17	98.60	85.00
	SymbCoT			43.91	82.50	99.60	93.00
	Direct		GPT-4o	30.30	53.70	–	84.70
	CoT			36.09	72.67	99.00	91.33
	SymbCoT			34.20	80.20	–	90.70
AF	Logic-LM	Abstention (None)	GPT-4	19.56	79.00	83.00	88.00
	Logic-LM++			21.18	78.80	–	–
	D&P			78.43	98.33	99.32	99.50
	CLOVER		GPT-4o	46.8	96.5	–	99.0
	D&P			77.30	99.67	100.00	100.00
	Logic-LM	CoT	GPT-4	43.04	79.66	83.20	87.63
	Logic-LM++			46.32	79.66	–	–
	D&P			80.65	98.33	99.36	99.50
	CLOVER		GPT-4o	62.80	96.70	–	99.30
	D&P			81.48	99.67	100.00	100.00

maximum output length is set to 2,048 tokens, with nucleus sampling parameter $p = 1.0$. Unless otherwise specified, we use temperature $T = 0$ for chain-of-thought (CoT) reasoning and code generation, and temperature $T = 1.0$ for draft generation.

In-context learning (ICL) settings. On all benchmarks, draft generation and program generation are conducted under three-shot ICL with examples drawn from the training data. Syntax repair based on solver feedback is performed with zero-shot. Details can be found in the Appendix A.

Repeated experiments. All experiments involving D&P are conducted over 10 independent runs. In tables, results are reported as the mean, or mean $\pm$ standard deviation. In figures, curves show the mean performance, with shaded regions denoting ± 1 standard deviation across runs.

4.5 Main Results

Table 1 reports end-to-end accuracy (**Acc**) on four deductive reasoning benchmarks, covering prompting-based methods and auto-formalization (AF) methods. For AF, we evaluate two settings: *AF-only* (no fallback) and *AF with CoT fallback*, where CoT is used only when AF fails. This separation allows us to distinguish improvements to the AF itself from gains due to CoT fallbacks.

We first compare under the AF-only setting. On AR-LSAT, the strongest AF baseline in our table is CLOVER (GPT-4o), which achieves 46.8% accuracy, while D&P reaches 77.30% with GPT-4o, improving by 30.5 absolute points. The same pattern holds on the other benchmarks: D&P achieves 99.67% on ProofWriter and reaches 100% on PrOntoQA and LogicalDeduction under GPT-4o.

AF baselines also use CoT fallbacks for failed formalization. To make fair comparisons, we also report in the same way. With this setting, D&P remains strong: on AR-LSAT, it achieves 80.65% (GPT-4) and 81.48% (GPT-4o), and it still saturates ProofWriter, PrOntoQA, and LogicalDeduction.

Finally, we compare to prompting-based methods. Among the baselines where results are available, SymbCoT is typically the strongest. On AR-LSAT, it achieves 43.91% with GPT-4 and 34.20% with GPT-4o. Meanwhile, D&P attains 78.43% (AF-only) and 80.65% (with CoT fallback) under GPT-4, and 77.30% (AF-only) and 81.48% (with CoT fallback) under GPT-4o.

4.6 In-depth Analysis

Table 2 shows that D&P surpasses the baselines in both **ExecRate** and **Acc@Exec** when using a larger number of sampled paths (k=20). At k=1, D&P trails CLOVER in **ExecRate** because limited sampling reduces the chance of producing a

Table 2: Decomposed AF diagnostic metrics on AR-LSAT. k denotes the number of sampled AF paths.

Method	LLM	**Acc** (%)	**Exec Rate** (%)	**Acc**(%) **@Exec**
Logic-LM		19.6	32.6	60.0
D&P ($k = 1$)	GPT-4	36.4	43.3	84.1
D&P ($k = 20$)		78.4	91.7	85.5
CLOVER		46.8	59.7	78.3
D&P ($k = 1$)	GPT-4o	39.2	44.3	88.6
D&P ($k = 20$)		77.3	88.7	87.2

Table 3: Accuracy with a naive AF path w/ and w/o drafted plans (w/o pruning)

	$\mathbf{Acc}_{\mathrm{AF}}$ (%)	
	AF w/o draft	AF w/ draft
AR-LSAT	36.35 ± 1.22	41.78 ± 2.46
ProofWriter	97.69 ± 0.18	98.33 ± 0.22
PrOntoQA	94.92 ± 0.84	99.56 ± 0.18
LogicalDeduction	98.60 ± 0.58	98.23 ± 0.32

valid formalization that survives pruning.

To test our hypothesis that a few AF attempts are unlikely to yield a faithful formalization on hard problems, we examine how performance changes as we sample more candidate paths. We define a diagnostic metric, cover@k, which marks an instance as solved if *any* of the k sampled AF paths yields the correct answer. This metric characterizes the *potential* of AF under broader exploration: higher cover@k indicates a greater likelihood that at least one correct formalization appears among the k sampled paths.

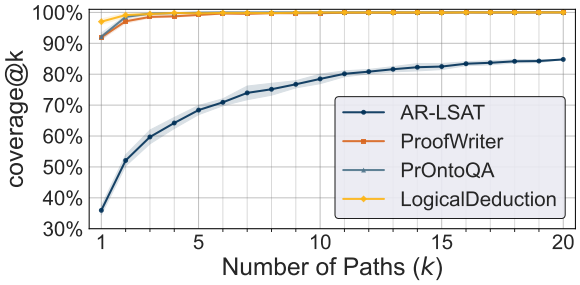


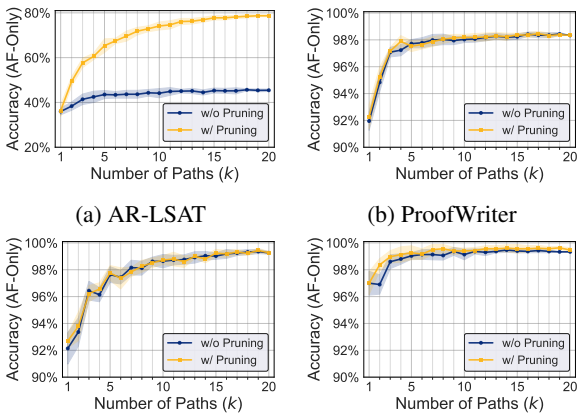
Figure 5: Coverage (cover@k) as a function of the number of sampled AF paths k across four benchmarks.

Figure 5 shows that cover@k increases monotonically with k, indicating that the chance of obtaining a correct answer grows as we explore more candidates. Moreover, the gap between cover@k and realized AF accuracy at small k suggests that many failures are due to under-exploration of the formalization space rather than the absence of faithful formalizations. In other words, many instances are solvable by AF, but not reliably within a single or a few attempts.

4.7 Effectiveness of drafting plans

Table 3 studies the effect of drafting a plan before formalization generation in a single-path AF setting without pruning. On AR-LSAT, adding drafted plans improves $\mathbf{Acc}_{\mathrm{AF}}$ by over 5%, indicating that explicit planning helps AF improve performance even without path selection or aggregation. On PrOntoQA, although plain AF already achieves high accuracy, drafted plans further reduce occasional failures and push performance closer to saturation. On ProofWriter and LogicalDeduction, both variants achieve near-ceiling performance, and drafting plans neither degrades accuracy nor introduces instability.

4.8 Sensitivity to Number of Paths



(a) AR-LSAT (b) ProofWriter

(c) PrOntoQA (d) LogicalDeduction

Figure 6: Overall accuracy significantly improves along with the number of paths (n from 1 to 20). Easier benchmarks (ProofWriter, PrOntoQA, LogicalDeduction) get saturated quickly while AR-LSAT remains unsolved. Notably, without pruning, the $\mathbf{Acc}_{\mathrm{AF}}$ plateaus around 45% quickly, while pruning enables further accuracy boost to around 80%

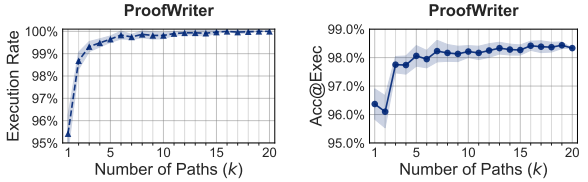
We first study the effect of sampling multiple AF paths with drafted plans on ProofWriter. As shown in Figure 7, increasing the number of paths k leads to a rapid increase in both **ExecRate** and **Acc@Exec**. This behavior indicates that, when semantic ambiguity is limited, drafted plans combined with multi-path aggregation can reliably increase AF accuracy without introducing instability.

We next examine whether this behavior holds

on more challenging benchmarks, where semantic unfaithfulness in AF is more prevalent.

Figure 6 shows that accuracy increases almost monotonically with larger k on all benchmarks. On AR-LSAT without pruning, accuracy improves by about 9% when increasing the number of paths from $k = 1$ to $k = 10$, but gains less than 1% when further increasing k to 20. In contrast, ProofWriter and PrOntoQA start from around 92–93% accuracy and rapidly approach saturation, while LogicalDeduction is already near ceiling at $k = 1$.

This plateau suggests that, on AR-LSAT, simply increasing the number of AF paths is insufficient. Many additional paths are executable but semantically ill-defined, producing contradictory or answer-ambiguous paths that cannot be resolved by aggregation alone. This observation motivates a closer examination of pruning.



(a) ExecRate (b) Acc@Exec

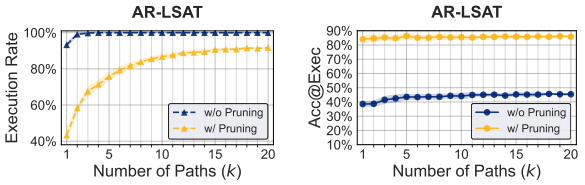
Figure 7: Diagnostic metrics on ProofWriter across different numbers of paths (k).

4.9 Effectiveness of Pruning

Table 4 shows that well-definedness pruning is critical on AR-LSAT, improving $\mathbf{Acc}_{\mathrm{AF}}$ from 45.13% to 78.43% (+33.30%). In contrast, pruning has negligible effect on ProofWriter, PrOntoQA, and LogicalDeduction, where accuracy is already near ceiling, and differences are within variation. Overall, pruning mainly helps harder problems by filtering ill-defined AF candidates before aggregation, while remaining safe on easier benchmarks.

Table 4: The $\mathbf{Acc}_{\mathrm{AF}}$ (%) of D&P w/o vs. w/ pruning on four benchmarks. There are twenty paths ($k = 20$), and every path uses drafted plans.

	$\mathbf{Acc}_{\mathrm{AF}}$(%)	
	w/o Pruning	w/ Pruning
AR-LSAT	45.13 ± 0.34	78.43 ± 0.55
ProofWriter	98.33 ± 0.00	98.33 ± 0.00
PrOntoQA	99.34 ± 0.10	99.32 ± 0.10
LogicalDeduction	99.33 ± 0.00	99.50 ± 0.18



(a) **ExecRate** (b) **Acc@Exec**

Figure 8: Diagnostic metrics on AR-LSAT w/o vs. w/ pruning across different numbers of paths (k).

Figure 8 further clarifies this effect using diagnostic metrics on AR-LSAT. As shown in Figure 8a, pruning substantially reduces **ExecRate**, since many sampled paths are discarded because they are contradictory or answer-ambiguous. However, Figure 8b shows that pruning yields a large and consistent increase in **Acc@Exec** across all values of k. This suggests that pruning removes semantically ill-defined formalizations while retaining paths that are more likely to be faithful. Consequently, although fewer paths survive, the remaining ones are substantially more reliable, which improves overall $\mathbf{Acc}_{\mathrm{AF}}$ after aggregation.

5 Conclusion

Auto-formalization (AF) enables solver-backed deductive reasoning by translating natural-language problems into executable solver programs, but existing pipelines are often brittle due to semantic unfaithfulness and limited exploration. Draft-and-Prune strengthens AF at inference time by drafting an explicit natural-language plan, pruning executable but ill-defined formalizations with solver checks, and voting over surviving predictions.

Across four benchmarks, it improves end-to-end accuracy, with the largest gains on AR-LSAT and reduced reliance on fallback prompting. Analysis suggests that, once executability is largely addressed, the main bottleneck is semantic search; future work will develop stronger verification signals, especially for settings beyond closed-end questions.

6 Limitations

Our work improves AF through inference-time ensembling, verification-guided pruning, and aggregation. We note several limitations.

Inference-time cost. Our approach samples multiple candidate formalizations and executes solver checks, which increases inference-time compute,

latency, and API cost relative to single-pass AF. While it is lightweight, the overall system cost grows with the number of candidates and solver invocations. Understanding the accuracy–cost trade-off under strict budgets remains important.

No adaptation of the base generator. We treat the foundation model as a black box and do not improve the AF generator via fine-tuning, reinforcement learning, or distillation. As a result, systematic translation errors of the base model may persist. Combining our framework with model adaptation is a promising direction.

Incomplete detection of semantic unfaithfulness. Our verification and pruning rules filter many ill-formed candidates (e.g., unexecutable or inconsistent outputs), but they cannot detect all semantically unfaithful yet executable formalizations without additional supervision. Some failures remain even after pruning.

Aggregation without equivalence awareness. We aggregate predictions at the answer level and do not cluster or de-duplicate candidates by symbolic equivalence. This may overweight redundant formalizations and miss opportunities for more principled equivalence-aware aggregation. Developing equivalence-aware or diversity-aware aggregation is an interesting direction for future work.

Scope of evaluation. We evaluate on four logical reasoning benchmarks and focus on solver-assisted AF for discrete-answer tasks. Although the results are strong on challenging benchmarks such as AR-LSAT, generalization to other AF domains (e.g., mathematics, program synthesis, or formal proof generation) and to other solver formalisms requires further study.

7 Ethical Considerations

This work studies auto-formalization for deductive reasoning on standard academic benchmarks that contain no personal, sensitive, or user-generated data, so privacy and consent issues do not arise. A key risk of auto-formalization is producing executable but semantically incorrect programs, which could be harmful if applied in high-stakes domains; our method mitigates this risk by using solver-verified checks to remove contradictory or answer-ambiguous formalizations before aggregation, though it does not guarantee full semantic correctness. The approach relies on external large language models and therefore inherits their limitations and biases, and it increases inference-time cost due to multi-path sampling and solver calls. As a result, the method should be viewed as a research system for controlled settings rather than a drop-in replacement for human-verified reasoning in real-world applications.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jon Barwise. 1977. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 5–46. Elsevier.

Melvin Fitting. 2012. *First-order logic and automated theorem proving*. Springer Science & Business Media.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Shashank Kirtania, Priyanshu Gupta, and Arjun Radhakrishna. 2024. LOGIC-LM++: Multi-step refinement for symbolic formulations. In *Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations (@ACL 2024)*, pages 56–63, Bangkok, Thailand. Association for Computational Linguistics.

William Calvert Kneale and Martha Kneale. 1984. *The development of logic*. Oxford university press.

Abhinav Lalwani, Tasha Kim, Lovish Chopra, Christopher Hahn, Zhijing Jin, and Mrinmaya Sachan. 2024. Autoformalizing natural language to first-order logic: A case study in logical fallacy detection. *arXiv preprint arXiv:2405.02318*.

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.

Hyun Ryu, Gyeongman Kim, Hyemin S Lee, and Eunho Yang. 2024. Divide and translate: Compositional first-order logic translation and verification for complex logical reasoning. *arXiv preprint arXiv:2410.08047*.

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*.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, and 1 others. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on machine learning research*.

Hongda Sun, Weikai Xu, Wei Liu, Jian Luan, Bin Wang, Shuo Shang, Ji-Rong Wen, and Rui Yan. 2024. Determlr: Augmenting llm-based logical reasoning from indeterminacy to determinacy. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9828–9862.

Oyvind Tafjord, Bhavana Dalvi 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.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

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

Jundong Xu, Hao Fei, Liangming Pan, Qian Liu, Mong-Li Lee, and Wynne Hsu. 2024. Faithful logical reasoning via symbolic chain-of-thought. *arXiv preprint arXiv:2405.18357*.

Yuan Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. 2023. Harnessing the power of large language models for natural language to first-order logic translation. *arXiv preprint arXiv:2305.15541*.

Hanlin Zhang, Jiani Huang, Ziyang Li, Mayur Naik, and Eric Xing. 2023. Improved logical reasoning of language models via differentiable symbolic programming. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 3062–3077, Toronto, Canada. Association for Computational Linguistics.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Yining Chen, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2022. Analytical reasoning of text. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2306–2319.

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)*.

A Appendix

A.1 Propositional and First-Order Logic

Propositional logic represents knowledge using *atomic statements* (propositions), which are the simplest units of truth—statements that can be either true or false, such as "it is raining" or "the light is on." These statements can be combined with logical connectives such as "and," "or," "not," and "if–then" to form more complex expressions. While powerful in its simplicity, propositional logic is limited in scope: it treats entire statements as indivisible units and cannot represent relationships between objects or quantify over them (Kneale and Kneale, 1984).

First-Order Logic (FOL) (Barwise, 1977) extends propositional logic with two key features: *predicates*, which describe properties of objects and relations among them (e.g., $Loves(x, y)$: "x loves y"), and *quantifiers*, which enable statements about all or some objects in a domain (e.g., $\forall x\, \exists y\ Loves(x, y)$: "every person loves someone"; $\exists y\, \forall x\ Loves(x, y)$: "someone is loved by everyone"). While propositional logic is best for straightforward true/false combinations, First-Order Logic is much more expressive and can model complex systems involving multiple entities and their relationships (Barwise, 1977; Fitting, 2012). FOL's clear syntax and powerful expressiveness underpin formal verification by enabling precise definitions and guaranteed logical inference (Fitting, 2012).

A.2 NL to First-Order Logic

Recent papers (Yang et al., 2023; Lalwani et al., 2024) propose using LLMs to translate natural language reasoning problems into First-Order Logic (FOL), which requires identifying the domain, entities, their properties, and the logical relationships

between them. Translation begins by specifying the domain of discourse — the set of objects under consideration (e.g., persons, numbers, graph nodes), and any basic assumptions about them. Next, one chooses a predicate vocabulary to describe properties and relations, along with symbols for specific objects and functions. These modeling choices determine which aspects of the natural language are made explicit in FOL and which are left implicit.

The process varies slightly across different benchmarks. In AR-LSAT (Zhong et al., 2022), natural language arguments are rephrased into structured predicates that capture assumptions and logical consequences. In ProofWriter (Tafjord et al., 2021), Stepwise inference is expressed with explicit predicates, quantifiers, and logical rules, allowing derivations to be precisely traced from facts. In PrOntoQA (Saparov and He, 2023), questions that involve multi-hop reasoning over short statements are mapped into predicates and quantifiers that explicitly represent each reasoning step. Finally, in LogicalDeduction (Srivastava et al., 2023), the focus is on converting abstract puzzles or conditional rules into quantified statements that ensure correct logical inference.

Across these tasks, the central challenge is faithfully capturing natural-language semantics, particularly ambiguity of reference, quantifier scope, negation, modality, and implicit background knowledge, within FOL's framework.

A.3 Prompts

A.3.1 AR-LSAT

Sketched Plan Generation (D&P)

```
\input{manuscript/prompts/ar-lsat.txt}
```

Code Generation (D&P)

```
You are an expert Python programmer specializing in the Z3
SMT solver library.
Your goal is to write executable Python Z3 code that
precisely implements the provided plan to solve the given
problem.
Remember to only output the raw Python code.

Context:
"On Tuesday Vladimir and Wendy each eat exactly four
separate meals: breakfast, lunch, dinner, and a snack. The
following is all that is known about what they eat during
that day: At no meal does Vladimir eat the same kind of food
 as Wendy. Neither of them eats the same kind of food more
than once during the day. For breakfast, each eats exactly
one of the following: hot cakes, poached eggs, or omelet.
For lunch, each eats exactly one of the following: fish, hot
 cakes, macaroni, or omelet. For dinner, each eats exactly
one of the following: fish, hot cakes, macaroni, or omelet.
For a snack, each eats exactly one of the following: fish or
 omelet. Wendy eats an omelet for lunch."
Question:
"Vladimir must eat which one of the following foods?"
Choices:
["fish", "hot cakes", "macaroni", "omelet", "poached eggs"]
Plan:
"
1. Enumerate the five possible foods and assign them integer
 codes.
2. Create integer variables for Vladimir and Wendy's meals (
breakfast, lunch, dinner, snack).
3. Add **domain constraints** for each meal variable based
on available menu options:
    - Breakfast: hot cakes, poached eggs, or omelet.
    - Lunch and dinner: fish, hot cakes, macaroni, or omelet
    .
    - Snack: fish or omelet.
4. Add **fixed constraint**: Wendy eats omelet for lunch.
5. Add **no-repeat constraints**: each person eats different
 foods at each meal.
    - Vladimir's four meals must be pairwise different.
    - Wendy's four meals must be pairwise different.
6. Add **no-same-meal constraints**: Vladimir and Wendy eat
different foods at each meal.
7. Translate the answer choices into food indices.
8. Define a answer index list 'answer_index_list' []. Use
proof by contradiction to find forced foods for Vladimir:
    - Clone the base constraints into a fresh solver.
    - Assert that Vladimir never eats a specific food at any
     meal.
    - If this assumption makes the model UNSAT, then that
    food is **forced**; append the food index to the answer
    index list.
9. Print the answer index list.
"
Z3 Program:
```
from z3 import *

Food indices
(FISH, HOT, MAC, OME, POE) = range(5)

# Meal variables: 0-breakfast, 1-lunch, 2-dinner, 3-snack
V = [Int(f"V_{m}") for m in range(4)]
W = [Int(f"W_{m}") for m in range(4)]

# Base solver
solver = Solver()

Domain constraints
Breakfast: hot cakes, poached eggs, or omelet
solver.add(Or(V[0] == HOT, V[0] == POE, V[0] == OME))
solver.add(Or(W[0] == HOT, W[0] == POE, W[0] == OME))
Lunch and dinner: fish, hot cakes, macaroni, or omelet
for meal in [1, 2]: # lunch and dinner
    solver.add(Or(V[meal] == FISH, V[meal] == HOT, V[meal]
    == MAC, V[meal] == OME))
    solver.add(Or(W[meal] == FISH, W[meal] == HOT, W[meal]
    == MAC, W[meal] == OME))
Snack: fish or omelet
solver.add(Or(V[3] == FISH, V[3] == OME))
solver.add(Or(W[3] == FISH, W[3] == OME))

# Fixed constraint
solver.add(W[1] == OME)

# No-repeat constraints
for i in range(4):
    for j in range(i + 1, 4):
        solver.add(V[i] != V[j], W[i] != W[j])

# No-same-meal constraints
for m in range(4):
    solver.add(V[m] != W[m])

# Answer choices
answer_choices = [
    FISH,
    HOT,
    MAC,
    OME,
    POE
]

# Check each answer choice using proof by contradiction
answer_index_list = []
for idx, food in enumerate(answer_choices):
    s_chk = Solver()
    s_chk.add(solver.assertions())

    # Assert Vladimir never eats this food at any meal
    s_chk.add(And(*(V[k] != idx for k in range(4))))

    # If UNSAT, this food is forced for Vladimir
    if s_chk.check() == unsat:
        answer_index_list.append(idx)

print(answer_index_list)
```

Context:
"In a repair facility there are exactly six technicians:
Stacy, Urma, Wim, Xena, Yolanda, and Zane. Each technician
repairs machines of at least one of the following three
types-radios, televisions, and VCRs-and no other types. The
following conditions apply: Xena and exactly three other
technicians repair radios. Yolanda repairs both televisions
and VCRs. Stacy does not repair any type of machine that
Yolanda repairs. Zane repairs more types of machines than
```

````
Yolanda repairs. Wim does not repair any type of machine
that Stacy repairs. Urma repairs exactly two types of
machines."
Question:
"Which one of the following pairs of technicians could
repair all and only the same types of machines as each other
?"
Choices:
["Stacy and Urma", "Urma and Yolanda", "Urma and Xena", "Wim
 and Xena", "Xena and Yolanda"]
Plan:
"
1. Enumerate the six technicians and give them integer
indices.
2. Create Boolean variables `t[i][j]` for each technician i
and machine type j (Radio, Television, VCR).
3. Add **at-least-one** constraint: each technician repairs
at least one machine type.
    - For each technician, at least one of Radio, Television
    , or VCR must be true.
4. Add **radio constraint**: Xena repairs radios, and
exactly 4 technicians total repair radios.
5. Add **Yolanda constraint**: repairs both televisions and
VCRs.
6. Add **Stacy constraint**: does not repair televisions or
VCRs (opposite of Yolanda).
7. Add **Zane constraint**: repairs more machine types than
Yolanda does.
8. Add **Wim constraint**: shares no machine types with
Stacy.
9. Add **Urma constraint**: repairs exactly 2 machine types.
10. Translate the answer choices into technician pairs.
11. Define a answer index list 'answer_index_list' []. For
every pair in the answer choices:
    - Clone the base constraints into a fresh solver.
    - Add constraint that both technicians repair identical
    machine types for all three types.
    - If the model is SAT, then this pair is possible;
    append the pair index to the answer index list.
12. Print the answer index list.
"
Z3 Program:
```
from z3 import *

# Technician indices
(S, U, W, X, Y, Z) = range(6)

# Machine type indices: 0-Radio, 1-Television, 2-VCR
t = [[Bool(f"t_{i}_{j}") for j in range(3)] for i in range
(6)]

# Base solver
solver = Solver()

# At-least-one constraint
## Each technician repairs at least one machine type
for i in range(6):
    solver.add(Or(t[i][0], t[i][1], t[i][2]))

# Radio constraint
## Xena repairs radios
solver.add(t[X][0])
## Exactly 4 technicians repair radios
solver.add(Sum([If(t[i][0], 1, 0) for i in range(6)]) == 4)

# Yolanda constraint
solver.add(t[Y][1], t[Y][2])

# Stacy constraint
solver.add(Not(t[S][1]), Not(t[S][2]))

# Zane constraint
count = lambda idx: Sum([If(t[idx][j], 1, 0) for j in range
(3)])
solver.add(count(Z) > count(Y))

# Wim constraint
for j in range(3):
    solver.add(Implies(t[S][j], Not(t[W][j])))

# Urma constraint
solver.add(count(U) == 2)

# Answer choices
answer_choices = [
    (S, U),
    (U, Y),
    (U, X),
    (W, X),
    (X, Y)
]

# Check each answer choice
answer_index_list = []
for idx, (tech1, tech2) in enumerate(answer_choices):
    s_chk = Solver()
    s_chk.add(solver.assertions())

    # Both technicians repair identical machine types
    for j in range(3):
        s_chk.add(t[tech1][j] == t[tech2][j])

    if s_chk.check() == sat:
        answer_index_list.append(idx)

print(answer_index_list)
```

Context:
"Workers at a water treatment plant open eight valves-G, H,
I, K, L, N, O, and P-to flush out a system of pipes that
needs emergency repairs. To maximize safety and efficiency,
each valve is opened exactly once, and no two valves are
opened at the same time. The valves are opened in accordance
 with the following conditions: Both K and P are opened
before H. O is opened before L but after H. L is opened
after G. N is opened before H. I is opened after K."
Question:
"Each of the following could be the fifth valve opened
EXCEPT:"
Choices:
["H", "I", "K", "N", "O"]
Plan:
"
1. Enumerate the eight valves and create integer variables
for their opening positions.
2. Add **domain constraints**: each valve position must be
between 1 and 8.
    - All positions must be distinct (each valve opened
    exactly once).
3. Add **ordering constraints** based on the given
conditions:
    - K and P are opened before H: `pos_K < pos_H` and `pos_
    P < pos_H`.
    - N is opened before H: `pos_N < pos_H`.
    - O is opened after H but before L: `pos_H < pos_O < pos
    _L`.
    - L is opened after G: `pos_G < pos_L`.
    - I is opened after K: `pos_K < pos_I`.
4. Translate the answer choices into valve indices.
5. Define a answer index list 'answer_index_list' []. For
every candidate valve in the answer choices:
    - Clone the base constraints into a fresh solver.
    - Add constraint that the valve is opened fifth: `pos_
    valve == 5`.
    - If the model is UNSAT, then this valve **cannot** be
    fifth; append the valve index to the answer index list.
6. Print the answer index list.
"
Z3 Program:
```
from z3 import *

# Valve position variables: pos_valve = opening position
(1-8)
pos = {v: Int(f"pos_{v}") for v in ["G", "H", "I", "K", "L",
 "N", "O", "P"]}

# Base solver
solver = Solver()

# Domain constraints
## Each valve position between 1 and 8
for v in pos:
    solver.add(pos[v] >= 1, pos[v] <= 8)
## All positions distinct (each valve opened exactly once)
solver.add(Distinct(*pos.values()))

# Ordering constraints
## K and P opened before H
solver.add(pos["K"] < pos["H"])
solver.add(pos["P"] < pos["H"])
## N opened before H
solver.add(pos["N"] < pos["H"])
## H opened before O, O opened before L
solver.add(pos["H"] < pos["O"])
solver.add(pos["O"] < pos["L"])
## G opened before L
solver.add(pos["G"] < pos["L"])
## K opened before I
solver.add(pos["K"] < pos["I"])

# Answer choices
answer_choices = [
    "H",
    "I",
    "K",
    "N",
    "O"
]

# Check each answer choice
answer_index_list = []
for idx, valve in enumerate(answer_choices):
    s_chk = Solver()
    s_chk.add(solver.assertions())

    # Add constraint that this valve is opened fifth
    s_chk.add(pos[valve] == 5)
````
````

```
    # If UNSAT, this valve cannot be fifth (EXCEPT answer)
    if s_chk.check() == unsat:
        answer_index_list.append(idx)

print(answer_index_list)
```

Context:
{context}
Question:
{question}
Choices:
{answers}
Plan:
{plan}
Z3 Program:
```

```

Self-Refinement (D&P)

```
You are an expert Python syntax fixer specializing in the Z3
 SMT solver library.
Your task is to fix the syntax errors in the code below
given the error message from Z3 solver.
Produce ONLY raw Python code. Do not include any other
comments or explanations.

Code with Syntax Errors:
{code}
Error message from solver:
{syntax_error}

Fixed Code:
```
from z3 import *

```
```

CoT (D&P)

```
Given a problem statement as contexts, the task is to answer
 a logical reasoning question.
Rember to output the correct option in the end: "The correct
 option is: [x]", where x is the number of the option.
The options are numbered from 0 to 4.

------
Context:
On Tuesday Vladimir and Wendy each eat exactly four separate
 meals: breakfast, lunch, dinner, and a snack. The following
 is all that is known about what they eat during that day:
At no meal does Vladimir eat the same kind of food as Wendy.
 Neither of them eats the same kind of food more than once
during the day. For breakfast, each eats exactly one of the
following: hot cakes, poached eggs, or omelet. For lunch,
each eats exactly one of the following: fish, hot cakes,
macaroni, or omelet. For dinner, each eats exactly one of
the following: fish, hot cakes, macaroni, or omelet. For a
snack, each eats exactly one of the following: fish or
omelet. Wendy eats an omelet for lunch.

Question: Vladimir must eat which one of the following foods
?

Options: ["fish", "hot cakes", "macaroni", "omelet", "
poached eggs"]

Reasoning:
Wendy eats an omelet for lunch. Because neither person eats
the same kind of food more than once, Wendy cannot eat an
omelet for any other meal. The only options for the snack
are fish or an omelet. Since Wendy cannot have an omelet,
she must eat fish for her snack. The rules state that at no
meal does Vladimir eat the same kind of food as Wendy.
Therefore, since Wendy eats fish for her snack, Vladimir
must eat the other available snack food, which is an omelet.
 Thus, Vladimir must eat an omelet, corresponding to option
[3].

The correct option is: [3]
------
Context:
In a repair facility there are exactly six technicians:
Stacy, Urma, Wim, Xena, Yolanda, and Zane. Each technician
repairs machines of at least one of the following three
types–radios, televisions, and VCRs–and no other types. The
following conditions apply: Xena and exactly three other
technicians repair radios. Yolanda repairs both televisions
and VCRs. Stacy does not repair any type of machine that
Yolanda repairs. Zane repairs more types of machines than
Yolanda repairs. Wim does not repair any type of machine
that Stacy repairs. Urma repairs exactly two types of
machines.

Question: Which one of the following pairs of technicians
could repair all and only the same types of machines as each
 other?

Options: ["Stacy and Urma", "Urma and Yolanda", "Urma and
Xena", "Wim and Xena", "Xena and Yolanda"]

Reasoning:
From the rules, we can determine the following:
Yolanda (Y) repairs televisions and VCRs, so Y = {T, V}.
Stacy (S) does not repair anything Yolanda repairs (T, V),
so Stacy must repair only radios: S = {R}.
Zane (Z) repairs more types than Yolanda (who repairs two),
so Zane must repair all three types: Z = {R, T, V}.
Wim (W) does not repair anything Stacy repairs (R), so Wim's
 repairs must be from the set {T, V}.
There are exactly four radio repairers. We have identified
Stacy and Zane as two of them. Xena is the third. Yolanda
and Wim do not repair radios, so the fourth radio repairer
must be Urma (U).
Urma (U) repairs exactly two types, one of which must be
radios. So, Urma's set is either {R, T} or {R, V}.
Xena (X) repairs radios, but there are no other restrictions
 on her. She could repair {R}, {R, T}, {R, V}, or {R, T, V}.
Now we check the options: The question asks which pair could
 have the same set of repairs.
Urma and Xena is the only possible pair. If Urma's set of
repairs is {R, T}, it is possible for Xena's set to also be
{R, T}. This scenario does not violate any rules. Similarly,
 if Urma repairs {R, V}, Xena could also repair {R, V}. All
other pairs have definite differences (e.g., Wim never
repairs radios, while Xena always does).
The corresponding option of Urma and Xena is [2].

The correct option is: [2]
------
Context:
Workers at a water treatment plant open eight valves–G, H, I
, K, L, N, O, and P–to flush out a system of pipes that
needs emergency repairs. To maximize safety and efficiency,
each valve is opened exactly once, and no two valves are
opened at the same time. The valves are opened in accordance
 with the following conditions: Both K and P are opened
before H. O is opened before L but after H. L is opened
after G. N is opened before H. I is opened after K.

Question: Each of the following could be the fifth valve
opened EXCEPT:

Options: ["H", "I", "K", "N", "O"]

Reasoning:
Let's synthesize the rules to understand the sequence of the
 valves:
    K is opened before H (K<H).
    K is also opened before I (K<I).
    The sequence H, O, L must occur in that order (H<O<L).
    By combining these rules, we can deduce that K must be
    opened before four other valves: H, I, O, and L.
The eight valves are opened in positions 1 through 8. If a
valve must be followed by at least four other valves, it
cannot be opened in position 5 or later. If K were opened in
 the fifth position, there would only be three available
positions after it (sixth, seventh, and eighth). This is not
 enough space for the four valves (H, I, O, L) that are
required to come after K. Therefore, K cannot be the fifth
valve opened, where the corresponding option is [2].

The correct option is: [2]
------
Context:
{context}

Question: {question}

Options: {answers}

Reasoning:
```

A.3.2 ProofWriter

Sketched Plan Generation (D&P)

```
You are an expert in formal verification and SMT solving
using the PyKe Solver.
Your goal is to create a DETAILED, STEP-BY-STEP PLAN to
translate the given natural language problem into PyKe
constraints.
Remember to only output the raw plan.

Problem:
The cow is blue. The cow is round. The cow likes the lion.
The cow visits the tiger. The lion is cold. The lion is nice
. The lion likes the squirrel. The squirrel is round. The
squirrel sees the lion. The squirrel visits the cow. The
tiger likes the cow. The tiger likes the squirrel. If
something is cold then it visits the tiger. If something
```

```
visits the tiger then it is nice. If something sees the
tiger and it is young then it is blue. If something is nice
then it sees the tiger. If something likes the squirrel and
it likes the cow then it visits the tiger. If something is
nice and it sees the tiger then it is young. If the cow is
cold and the cow visits the lion then the lion sees the
squirrel.
Question:
Based on the above information, is the following statement
true, false, or unknown? The tiger is not young.
Plan:
1. Identify all entities involved: cow, lion, squirrel,
tiger.
2. Define all explicit properties and relations using
ternary predicates of the form predicate(subject, object,
True).
    - Properties like is_blue, is_round, is_nice take a
    boolean as the object.
    - Relations like likes, visits, sees take another entity
     as the object.
3. Encode all directly stated facts as facts.pred(subj, obj,
 True).
4. Translate each conditional sentence from the context into
 a rule using PyKe syntax:
    $\bullet$ If something is cold $\rightarrow$ it visits
    the tiger.
    $\bullet$ If something visits the tiger $\rightarrow$ it
     is nice.
    $\bullet$ If something sees the tiger and is young $\
    rightarrow$ it is blue.
    $\bullet$ If something is nice $\rightarrow$ it sees the
     tiger.
    $\bullet$ If something likes both the squirrel and the
    cow $\rightarrow$ it visits the tiger.
    $\bullet$ If something is nice and sees the tiger $\
    rightarrow$ it is young.
    $\bullet$ If the cow is cold and visits the lion $\
    rightarrow$ the lion sees the squirrel.
5. Encode each rule using a `foreach` block for the
condition and an `assert` clause for the consequence.
6. The query tests whether the tiger is not young:
    $\rightarrow$ Use facts.is_young("tiger", False)

Problem:
Dave is big. Dave is red. Erin is smart. Fiona is kind.
Fiona is smart. Gary is rough. Gary is white. All young
people are white. If someone is kind and white then they are
 big. If someone is kind then they are young. If Fiona is
young and Fiona is rough then Fiona is red. If someone is
big then they are rough. All rough, white people are red. If
 someone is kind and not big then they are red.
Question:
Based on the above information, is the following statement
true, false, or unknown? Erin is smart.
Plan:
1. Identify the individuals: Dave, Erin, Fiona, Gary.
2. Represent all explicit facts using ternary predicates
like is_big("Dave", True), is_smart("Erin", True), etc.
3. Express all contextual implications as rules:
    $\bullet$ All young people are white.
    $\bullet$ If someone is kind and white $\rightarrow$
    they are big.
    $\bullet$ If someone is kind $\rightarrow$ they are
    young.
    $\bullet$ If Fiona is young and rough $\rightarrow$
    Fiona is red.
    $\bullet$ If someone is big $\rightarrow$ they are rough
    .
    $\bullet$ If someone is both rough and white $\
    rightarrow$ they are red.
    $\bullet$ If someone is kind and not big $\rightarrow$
    they are red.
4. For each rule, define a `foreach` clause that matches the
 precondition and an `assert` clause that encodes the
conclusion.
5. The query verifies the known fact:
    $\rightarrow$ Use facts.is_smart("Erin", True)

Problem:
The bald eagle sees the dog. The dog chases the tiger. The
dog eats the rabbit. The dog sees the bald eagle. The rabbit
 sees the dog. The tiger chases the bald eagle. The tiger
eats the rabbit. The tiger is blue. The tiger is cold. The
tiger sees the dog. If something is blue and it sees the
bald eagle then the bald eagle sees the rabbit. If something
 is cold then it eats the bald eagle. If something sees the
bald eagle then it chases the dog. If something eats the
bald eagle then it is big. If something sees the bald eagle
then the bald eagle sees the dog. If something chases the
tiger and the tiger is big then it is cold. If something
chases the tiger then it is rough.
Question:
Based on the above information, is the following statement
true, false, or unknown? The rabbit sees the tiger.
Plan:
1. Identify all involved entities: bald_eagle, dog, rabbit,
tiger.
2. Encode all directly stated facts using ternary predicates
 such as sees("dog", "bald_eagle", True), is_blue("tiger",
True), etc.
3. Translate the logical implications in the context into
rules:
    $\bullet$ If something is blue and sees the bald eagle $
    \rightarrow$ the bald eagle sees the rabbit.
    $\bullet$ If something is cold $\rightarrow$ it eats the
     bald eagle.
    $\bullet$ If something sees the bald eagle $\rightarrow$
     it chases the dog.
    $\bullet$ If something eats the bald eagle $\rightarrow$
     it is big.
    $\bullet$ If something sees the bald eagle $\rightarrow$
     the bald eagle sees the dog.
    $\bullet$ If something chases the tiger and the tiger is
     big $\rightarrow$ it is cold.
    $\bullet$ If something chases the tiger $\rightarrow$ it
     is rough.
4. Implement each rule using `foreach` for matching and `
assert` for conclusions.
5. The query checks whether the rabbit sees the tiger:
    $\rightarrow$ Use facts.sees("rabbit", "tiger", True)

Problem:
{context}
Question:
{question}
Plan:
```

Code Generation (D&P)

```
You are an expert in formal verification and SMT solving
using the PyKe Solver.

Your primary goal is to translate the given natural language
 problem into syntactically and semantically correct PyKe
program, consisting three parts: Facts, Rules, and Query.
- Facts: Declare the explicitly stated properties and
relationships of entities from the context using ternary
predicates in the form predicate(subject, object,
truth_value), where truth_value is typically True to denote
the fact holds.
- Rules: Encode conditional logic from the context (e.g., "
If... then...") as PyKe rule clauses. Each rule uses foreach
 to specify matching conditions and assert to define the
logical consequence. Each rule captures an inference pattern
.
- Query: Translate the natural language question into a
formal goal that can be proven using the defined facts and
rules. Queries usually take the form facts.predicate(subject
, expected_value) and represent the statement being tested
for truth.

Remember to only output the raw code.

Problem:
The cow is blue. The cow is round. The cow likes the lion.
The cow visits the tiger. The lion is cold. The lion is nice
. The lion likes the squirrel. The squirrel is round. The
squirrel sees the lion. The squirrel visits the cow. The
tiger likes the cow. The tiger likes the squirrel. If
something is cold then it visits the tiger. If something
visits the tiger then it is nice. If something sees the
tiger and it is young then it is blue. If something is nice
then it sees the tiger. If something likes the squirrel and
it likes the cow then it visits the tiger. If something is
nice and it sees the tiger then it is young. If the cow is
cold and the cow visits the lion then the lion sees the
squirrel.
Question:
Based on the above information, is the following statement
true, false, or unknown? The tiger is not young.
Plan:
1. Identify all entities involved: cow, lion, squirrel,
tiger.
2. Define all explicit properties and relations using
ternary predicates of the form predicate(subject, object,
True).
    - Properties like is_blue, is_round, is_nice take a
    boolean as the object.
    - Relations like likes, visits, sees take another entity
     as the object.
3. Encode all directly stated facts as facts.pred(subj, obj,
 True).
4. Translate each conditional sentence from the context into
 a rule using PyKe syntax:
    - If something is cold -> it visits the tiger.
    - If something visits the tiger -> it is nice.
    - If something sees the tiger and is young -> it is blue
    .
    - If something is nice -> it sees the tiger.
    - If something likes both the squirrel and the cow -> it
     visits the tiger.
    - If something is nice and sees the tiger -> it is young
    .
    - If the cow is cold and visits the lion -> the lion
    sees the squirrel.
5. Encode each rule using a `foreach` block for the
condition and an `assert` clause for the consequence.
6. The query tests whether the tiger is not young:
    -> Use facts.is_young("tiger", False)
```

Facts:
```facts
# --- Facts about the Cow ---
is_blue("cow", True)
is_round("cow", True)
likes("cow", "lion", True)
visits("cow", "tiger", True)

# --- Facts about the Lion ---
is_cold("lion", True)
is_nice("lion", True)
likes("lion", "squirrel", True)

# --- Facts about the Squirrel ---
is_round("squirrel", True)
sees("squirrel", "lion", True)
visits("squirrel", "cow", True)

# --- Facts about the Tiger ---
likes("tiger", "cow", True)
likes("tiger", "squirrel", True)
```
Rules:
```rules
# Rule: If something is cold then it visits the tiger.
cold_visits_tiger
    foreach
        facts.is_cold($thing, True)
    assert
        facts.visits($thing, "tiger", True)

# Rule: If something visits the tiger then it is nice.
visits_tiger_is_nice
    foreach
        facts.visits($thing, "tiger", True)
    assert
        facts.is_nice($thing, True)

# Rule: If something sees the tiger and is young then it is
blue.
sees_tiger_and_young_is_blue
    foreach
        facts.sees($thing, "tiger", True)
        facts.is_young($thing, True)
    assert
        facts.is_blue($thing, True)

# Rule: If something is nice then it sees the tiger.
nice_sees_tiger
    foreach
        facts.is_nice($thing, True)
    assert
        facts.sees($thing, "tiger", True)

# Rule: If something likes the squirrel and the cow then it
visits the tiger.
likes_both_visits_tiger
    foreach
        facts.likes($thing, "squirrel", True)
        facts.likes($thing, "cow", True)
    assert
        facts.visits($thing, "tiger", True)

# Rule: If something is nice and sees the tiger then it is
young.
nice_and_sees_tiger_is_young
    foreach
        facts.is_nice($thing, True)
        facts.sees($thing, "tiger", True)
    assert
        facts.is_young($thing, True)

# Rule: If cow is cold and visits lion then lion sees
squirrel.
cow_cold_and_visits_lion_lion_sees_squirrel
    foreach
        facts.is_cold("cow", True)
        facts.visits("cow", "lion", True)
    assert
        facts.sees("lion", "squirrel", True)
```
Query:
```query
facts.is_young("tiger", False)
```

Problem:
Dave is big. Dave is red. Erin is smart. Fiona is kind. Fiona is smart. Gary is rough. Gary is white. All young people are white. If someone is kind and white then they are big. If someone is kind then they are young. If Fiona is young and Fiona is rough then Fiona is red. If someone is big then they are rough. All rough, white people are red. If someone is kind and not big then they are red.
Question:
Based on the above information, is the following statement true, false, or unknown? Erin is smart.
Plan:
1. Identify the individuals: Dave, Erin, Fiona, Gary.
2. Represent all explicit facts using ternary predicates like is_big("Dave", True), is_smart("Erin", True), etc.
3. Express all contextual implications as rules:
    - All young people are white.
    - If someone is kind and white -> they are big.
    - If someone is kind -> they are young.
    - If Fiona is young and rough -> Fiona is red.
    - If someone is big -> they are rough.
    - If someone is both rough and white -> they are red.
    - If someone is kind and not big -> they are red.
4. For each rule, define a `foreach` clause that matches the precondition and an `assert` clause that encodes the conclusion.
5. The query verifies the known fact:
    -> Use facts.is_smart("Erin", True)
Facts:
```facts
# --- Facts about Dave ---
is_big("Dave", True)
is_red("Dave", True)

# --- Facts about Erin ---
is_smart("Erin", True)

# --- Facts about Fiona ---
is_kind("Fiona", True)
is_smart("Fiona", True)

# --- Facts about Gary ---
is_rough("Gary", True)
is_white("Gary", True)
```
Rules:
```rules
# Rule: All young people are white.
young_are_white
    foreach
        facts.is_young($person, True)
    assert
        facts.is_white($person, True)

# Rule: If someone is kind and white then they are big.
kind_and_white_are_big
    foreach
        facts.is_kind($person, True)
        facts.is_white($person, True)
    assert
        facts.is_big($person, True)

# Rule: If someone is kind then they are young.
kind_are_young
    foreach
        facts.is_kind($person, True)
    assert
        facts.is_young($person, True)

# Rule: If Fiona is young and Fiona is rough then Fiona is
red.
fiona_young_and_rough_is_red
    foreach
        facts.is_young("Fiona", True)
        facts.is_rough("Fiona", True)
    assert
        facts.is_red("Fiona", True)

# Rule: If someone is big then they are rough.
big_are_rough
    foreach
        facts.is_big($person, True)
    assert
        facts.is_rough($person, True)

# Rule: All rough, white people are red.
rough_and_white_are_red
    foreach
        facts.is_rough($person, True)
        facts.is_white($person, True)
    assert
        facts.is_red($person, True)

# Rule: If someone is kind and not big then they are red.
kind_and_not_big_are_red
    foreach
        facts.is_kind($person, True)
        facts.is_big($person, False)
    assert
        facts.is_red($person, True)
```
Query:
```query
facts.is_smart("Erin", True)
```

Problem:
The bald eagle sees the dog. The dog chases the tiger. The dog eats the rabbit. The dog sees the bald eagle. The rabbit sees the dog. The tiger chases the bald eagle. The tiger eats the rabbit. The tiger is blue. The tiger is cold. The tiger sees the dog. If something is blue and it sees the bald eagle then the bald eagle sees the rabbit. If something is cold then it eats the bald eagle. If something sees the

````
bald eagle then it chases the dog. If something eats the
bald eagle then it is big. If something sees the bald eagle
then the bald eagle sees the dog. If something chases the
tiger and the tiger is big then it is cold. If something
chases the tiger then it is rough.
Question:
Based on the above information, is the following statement
true, false, or unknown? The rabbit sees the tiger.
Plan:
1. Identify all involved entities: bald_eagle, dog, rabbit,
tiger.
2. Encode all directly stated facts using ternary predicates
 such as sees("dog", "bald_eagle", True), is_blue("tiger",
True), etc.
3. Translate the logical implications in the context into
rules:
    - If something is blue and sees the bald eagle -> the
    bald eagle sees the rabbit.
    - If something is cold -> it eats the bald eagle.
    - If something sees the bald eagle -> it chases the dog.
    - If something eats the bald eagle -> it is big.
    - If something sees the bald eagle -> the bald eagle
    sees the dog.
    - If something chases the tiger and the tiger is big ->
    it is cold.
    - If something chases the tiger -> it is rough.
4. Implement each rule using `foreach` for matching and `
assert` for conclusions.
5. The query checks whether the rabbit sees the tiger:
    -> Use facts.sees("rabbit", "tiger", True)
Facts:
```facts
# --- Facts about the Bald Eagle ---
sees("bald_eagle", "dog", True)

# --- Facts about the Dog ---
chases("dog", "tiger", True)
eats("dog", "rabbit", True)
sees("dog", "bald_eagle", True)

# --- Facts about the Rabbit ---
sees("rabbit", "dog", True)

# --- Facts about the Tiger ---
chases("tiger", "bald_eagle", True)
eats("tiger", "rabbit", True)
is_blue("tiger", True)
is_cold("tiger", True)
sees("tiger", "dog", True)
```
Rules:
```rules
# Rule: If something is blue and it sees the bald eagle then
 the bald eagle sees the rabbit.
blue_and_sees_bald_eagle_bald_eagle_sees_rabbit
    foreach
        facts.is_blue($thing, True)
        facts.sees($thing, "bald_eagle", True)
    assert
        facts.sees("bald_eagle", "rabbit", True)

# Rule: If something is cold then it eats the bald eagle.
cold_eats_bald_eagle
    foreach
        facts.is_cold($thing, True)
    assert
        facts.eats($thing, "bald_eagle", True)

# Rule: If something sees the bald eagle then it chases the
dog.
sees_bald_eagle_chases_dog
    foreach
        facts.sees($thing, "bald_eagle", True)
    assert
        facts.chases($thing, "dog", True)

# Rule: If something eats the bald eagle then it is big.
eats_bald_eagle_is_big
    foreach
        facts.eats($thing, "bald_eagle", True)
    assert
        facts.is_big($thing, True)

# Rule: If something sees the bald eagle then the bald eagle
 sees the dog.
sees_bald_eagle_bald_eagle_sees_dog
    foreach
        facts.sees($thing, "bald_eagle", True)
    assert
        facts.sees("bald_eagle", "dog", True)

# Rule: If something chases the tiger and the tiger is big
then it is cold.
chases_tiger_and_tiger_big_is_cold
    foreach
        facts.chases($thing, "tiger", True)
        facts.is_big("tiger", True)
    assert
        facts.is_cold($thing, True)

# Rule: If something chases the tiger then it is rough.
chases_tiger_is_rough
    foreach
        facts.chases($thing, "tiger", True)
    assert
        facts.is_rough($thing, True)
```
Query:
```query
facts.sees("rabbit", "tiger", True)
```

Problem:
{context}
Question:
{question}
Plan:
{plan}
````

Self-Refinement (D&P)

```
You are an expert Python syntax fixer specializing in the
PyKe solver library.

Problem:
{context}
Question:
{question}
Plan:
{plan}

## Original Code
{code}

## Error message from solver
{syntax_error}

# Instructions for Code Fixing
- Write a new version of the code that corrects the
identified syntax error.
- Ensure the new code is syntactically correct and logically
 sound.

# Output Requirements:
Translate the given natural language problem into
syntactically and semantically correct PyKe program,
consisting three parts: Facts, Rules, and Query.
- Facts: Declare the explicitly stated properties and
relationships of entities from the context using ternary
predicates in the form predicate(subject, object,
truth_value), where truth_value is typically True to denote
the fact holds.
- Rules: Encode conditional logic from the context (e.g., "
If... then...") as PyKe rule clauses. Each rule uses foreach
 to specify matching conditions and assert to define the
logical consequence. Each rule captures an inference pattern
.
- Query: Translate the natural language question into a
formal goal that can be proven using the defined facts and
rules. Queries usually take the form facts.predicate(subject
, expected_value) and represent the statement being tested
for truth.
```

CoT (D&P)

```
Given a problem statement as contexts, the task is to answer
 a logical reasoning question.
------
Problem:
The cow is blue. The cow is round. The cow likes the lion.
The cow visits the tiger. The lion is cold. The lion is nice
. The lion likes the squirrel. The squirrel is round. The
squirrel sees the lion. The squirrel visits the cow. The
tiger likes the cow. The tiger likes the squirrel. If
something is cold then it visits the tiger. If something
visits the tiger then it is nice. If something sees the
tiger and it is young then it is blue. If something is nice
then it sees the tiger. If something likes the squirrel and
it likes the cow then it visits the tiger. If something is
nice and it sees the tiger then it is young. If the cow is
cold and the cow visits the lion then the lion sees the
squirrel.
Question:
Based on the above information, is the following statement
true, false, or unknown? The tiger is not young.

Options:
A) True
B) False
C) Unknown

Reasoning:
The tiger likes the cow. The tiger likes the squirrel. If
something likes the squirrel and it likes the cow, then it
visits the tiger. So the tiger visits the tiger. If
something visits the tiger then it is nice. So the tiger is
nice. If something is nice then it sees the tiger. So the
```

```
tiger sees the tiger. If something is nice and it sees the
tiger then it is young. So the tiger is young. Therefore,
the statement "The tiger is not young" is false.

The correct option is: B
------
Problem:
Dave is big. Dave is red. Erin is smart. Fiona is kind.
Fiona is smart. Gary is rough. Gary is white. All young
people are white. If someone is kind and white then they are
 big. If someone is kind then they are young. If Fiona is
young and Fiona is rough then Fiona is red. If someone is
big then they are rough. All rough, white people are red. If
 someone is kind and not big then they are red.
Question:
Based on the above information, is the following statement
true, false, or unknown? Erin is smart.

Options:
A) True
B) False
C) Unknown

Reasoning:
The statement "Erin is smart" is given as a fact in the
context. Therefore, it is true.

The correct option is: A
------
Problem:
The bald eagle sees the dog. The dog chases the tiger. The
dog eats the rabbit. The dog sees the bald eagle. The rabbit
 sees the dog. The tiger chases the bald eagle. The tiger
eats the rabbit. The tiger is blue. The tiger is cold. The
tiger sees the dog. If something is blue and it sees the
bald eagle then the bald eagle sees the rabbit. If something
 is cold then it eats the bald eagle. If something sees the
bald eagle then it chases the dog. If something eats the
bald eagle then it is big. If something sees the bald eagle
then the bald eagle sees the dog. If something chases the
tiger and the tiger is big then it is cold. If something
chases the tiger then it is rough.
Question:
Based on the above information, is the following statement
true, false, or unknown? The rabbit sees the tiger.

Options:
A) True
B) False
C) Unknown

Reasoning:
The provided information states what the rabbit sees ("The
rabbit sees the dog") and what other animals do to the
rabbit ("The dog eats the rabbit", "The tiger eats the
rabbit"). However, there are no facts or rules that allow us
 to determine whether the rabbit sees the tiger. Therefore,
the statement is unknown.

The correct option is: C
------
Context:
{context}

Question: {question}

Options:
{options}

Reasoning:
```

A.3.3 PrOntoQA

Sketched Plan Generation (D&P)

```
You are an expert in formal verification and SMT solving
using the PyKe Solver.
Your goal is to create a DETAILED, STEP-BY-STEP PLAN to
translate the given natural language problem into PyKe
constraints.
Remember to only output the raw plan.

Problem:
Every yumpus is small. Every tumpus is hot. Each tumpus is a
 zumpus. Zumpuses are not small. Each zumpus is a rompus.
Rompuses are brown. Each rompus is a wumpus. Fae is a zumpus
.
Question:
Is the following statement true or false? Fae is not small.
Plan:
1. Identify all entities and types: Fae, yumpus, tumpus,
zumpus, rompus, wumpus.
2. Identify all properties: small, hot, brown.
3. Represent "Fae is a zumpus" as a base fact: `is_a("Fae",
 "zumpus", True)`.
4. Convert each class property and subclass relationship
into a rule:
        * Every yumpus is small.
        * Every tumpus is hot.
        * Each tumpus is a zumpus.
        * Zumpuses are not small.
        * Each zumpus is a rompus.
        * Rompuses are brown.
        * Each rompus is a wumpus.
5. Implement each rule using a `foreach` clause for the
condition and an `assert` clause for the consequence.
6. The query tests if the statement "Fae is not small" is
provable:
    -> Use `facts.is_small("Fae", False)`.

Problem:
Zumpuses are floral. Each zumpus is a jompus. Each jompus is
 brown. Jompuses are numpuses. Dumpuses are bright. Each
numpus is not opaque. Every numpus is a yumpus. Every yumpus
 is not bright. Every yumpus is a wumpus. Wumpuses are
luminous. Wumpuses are impuses. Every impus is hot. Impuses
are rompuses. Rompuses are mean. Each rompus is a vumpus.
Vumpuses are sweet. Vumpuses are tumpuses. Sam is a yumpus.
Question:
Is the following statement true or false? Sam is bright.
Plan:
1. Identify all entities and types: Sam, zumpus, jompus,
numpus, dumpus, yumpus, wumpus, impus, rompus, vumpus,
tumpus.
2. Identify all properties: floral, brown, bright, opaque,
luminous, hot, mean, sweet.
3. Represent "Sam is a yumpus" as a base fact: `is_a("Sam",
 "yumpus", True)`.
4. Translate all class properties and subclass
relationships into rules.
5. Implement each rule using a `foreach` clause for the
condition and an `assert` clause for the conclusion.
6. The query tests if the statement "Sam is bright" is
provable. Since we can prove "Sam is not bright", the
original statement is false.
    -> Use `facts.is_bright("Sam", True)`.

Problem:
Every dumpus is luminous. Each dumpus is a numpus. Numpuses
are cold. Numpuses are tumpuses. Tumpuses are bright. Each
tumpus is a yumpus. Every yumpus is opaque. Jompuses are
amenable. Yumpuses are zumpuses. Every zumpus is not
amenable. Zumpuses are rompuses. Rompuses are blue. Rompuses
 are wumpuses. Wumpuses are spicy. Every wumpus is a vumpus.
 Wren is a zumpus.
Question:
Is the following statement true or false? Wren is not
amenable.
Plan:
1. Identify all entities and types: Wren, dumpus, numpus,
tumpus, yumpus, jompus, zumpus, rompus, wumpus, vumpus.
2. Identify all properties: luminous, cold, bright, opaque,
 amenable, blue, spicy.
3. Represent "Wren is a zumpus" as a base fact: `is_a("Wren
", "zumpus", True)`.
4. Convert all class properties and subclass relationships
into rules.
5. Implement each rule using a `foreach` clause for the
condition and an `assert` clause for the consequence.
6. The query tests if the statement "Wren is not amenable"
is provable:
    -> Use `facts.is_amenable("Wren", False)`.

Problem:
{context}
Question:
{question}
Plan:
```

Code Generation (D&P)

```
You are an expert in formal verification and SMT solving
using the PyKe Solver.

Your primary goal is to translate the given natural language
 problem into syntactically and semantically correct PyKe
program, consisting three parts: Facts, Rules, and Query.
- Facts: Declare the explicitly stated properties and
relationships of entities from the context using ternary
predicates in the form predicate(subject, object,
truth_value), where truth_value is typically True to denote
the fact holds.
- Rules: Encode conditional logic from the context (e.g., "
If... then...") as PyKe rule clauses. Each rule uses foreach
 to specify matching conditions and assert to define the
logical consequence. Each rule captures an inference pattern
.
- Query: Translate the natural language question into a
formal goal that can be proven using the defined facts and
rules. Queries usually take the form facts.predicate(subject
, expected_value) and represent the statement being tested
for truth.

Remember to only output the raw code.

Problem:
```

````
Every yumpus is small. Every tumpus is hot. Each tumpus is a
 zumpus. Zumpuses are not small. Each zumpus is a rompus.
Rompuses are brown. Each rompus is a wumpus. Fae is a zumpus
.
Question:
Is the following statement true or false? Fae is not small.
Plan:
1. Identify all entities and types: Fae, yumpus, tumpus,
zumpus, rompus, wumpus.
2. Identify all properties: small, hot, brown.
3. Represent "Fae is a zumpus" as a base fact: `is_a("Fae",
 "zumpus", True)`.
4. Convert each class property and subclass relationship
into a rule:
        * Every yumpus is small.
        * Every tumpus is hot.
        * Each tumpus is a zumpus.
        * Zumpuses are not small.
        * Each zumpus is a rompus.
        * Rompuses are brown.
        * Each rompus is a wumpus.
5. Implement each rule using a `foreach` clause for the
condition and an `assert` clause for the consequence.
6. The query tests if the statement "Fae is not small" is
provable:
    -> Use `facts.is_small("Fae", False)`.
Facts:
```facts
# --- Fact about Fae ---
is_a("Fae", "zumpus", True)
```
Rules:
```rules
# Rule: Every yumpus is small.
yumpus_is_small
    foreach
        facts.is_a($thing, "yumpus", True)
    assert
        facts.is_small($thing, True)

# Rule: Every tumpus is hot.
tumpus_is_hot
    foreach
        facts.is_a($thing, "tumpus", True)
    assert
        facts.is_hot($thing, True)

# Rule: Each tumpus is a zumpus.
tumpus_is_zumpus
    foreach
        facts.is_a($thing, "tumpus", True)
    assert
        facts.is_a($thing, "zumpus", True)

# Rule: Zumpuses are not small.
zumpus_is_not_small
    foreach
        facts.is_a($thing, "zumpus", True)
    assert
        facts.is_small($thing, False)

# Rule: Each zumpus is a rompus.
zumpus_is_rompus
    foreach
        facts.is_a($thing, "zumpus", True)
    assert
        facts.is_a($thing, "rompus", True)

# Rule: Rompuses are brown.
rompus_is_brown
    foreach
        facts.is_a($thing, "rompus", True)
    assert
        facts.is_brown($thing, True)

# Rule: Each rompus is a wumpus.
rompus_is_wumpus
    foreach
        facts.is_a($thing, "rompus", True)
    assert
        facts.is_a($thing, "wumpus", True)
```
Query:
```query
facts.is_small("Fae", False)
```

Problem:
Zumpuses are floral. Each zumpus is a jompus. Each jompus is
 brown. Jompuses are numpuses. Dumpuses are bright. Each
numpus is not opaque. Every numpus is a yumpus. Every yumpus
 is not bright. Every yumpus is a wumpus. Wumpuses are
luminous. Wumpuses are impuses. Every impus is hot. Impuses
are rompuses. Rompuses are mean. Each rompus is a vumpus.
Vumpuses are sweet. Vumpuses are tumpuses. Sam is a yumpus.
Question:
Is the following statement true or false? Sam is bright.
Plan:
1. Identify all entities and types: Sam, zumpus, jompus,
numpus, dumpus, yumpus, wumpus, impus, rompus, vumpus,
tumpus.
2. Identify all properties: floral, brown, bright, opaque,
luminous, hot, mean, sweet.
3. Represent "Sam is a yumpus" as a base fact: `is_a("Sam",
 "yumpus", True)`.
4. Translate all class properties and subclass
relationships into rules.
5. Implement each rule using a `foreach` clause for the
condition and an `assert` clause for the conclusion.
6. The query tests if the statement "Sam is bright" is
provable. Since we can prove "Sam is not bright", the
original statement is false.
    -> Use `facts.is_bright("Sam", True)`.
Facts:
```facts
# --- Fact about Sam ---
is_a("Sam", "yumpus", True)
```
Rules:
```rules
# Rule: Zumpuses are floral.
zumpus_is_floral
    foreach
        facts.is_a($thing, "zumpus", True)
    assert
        facts.is_floral($thing, True)

# Rule: Each zumpus is a jompus.
zumpus_is_jompus
    foreach
        facts.is_a($thing, "zumpus", True)
    assert
        facts.is_a($thing, "jompus", True)

# Rule: Each jompus is brown.
jompus_is_brown
    foreach
        facts.is_a($thing, "jompus", True)
    assert
        facts.is_brown($thing, True)

# Rule: Jompuses are numpuses.
jompus_is_numpus
    foreach
        facts.is_a($thing, "jompus", True)
    assert
        facts.is_a($thing, "numpus", True)

# Rule: Dumpuses are bright.
dumpus_is_bright
    foreach
        facts.is_a($thing, "dumpus", True)
    assert
        facts.is_bright($thing, True)

# Rule: Each numpus is not opaque.
numpus_is_not_opaque
    foreach
        facts.is_a($thing, "numpus", True)
    assert
        facts.is_opaque($thing, False)

# Rule: Every numpus is a yumpus.
numpus_is_yumpus
    foreach
        facts.is_a($thing, "numpus", True)
    assert
        facts.is_a($thing, "yumpus", True)

# Rule: Every yumpus is not bright.
yumpus_is_not_bright
    foreach
        facts.is_a($thing, "yumpus", True)
    assert
        facts.is_bright($thing, False)

# Rule: Every yumpus is a wumpus.
yumpus_is_wumpus
    foreach
        facts.is_a($thing, "yumpus", True)
    assert
        facts.is_a($thing, "wumpus", True)

# Rule: Wumpuses are luminous.
wumpus_is_luminous
    foreach
        facts.is_a($thing, "wumpus", True)
    assert
        facts.is_luminous($thing, True)

# Rule: Wumpuses are impuses.
wumpus_is_impus
    foreach
        facts.is_a($thing, "wumpus", True)
    assert
        facts.is_a($thing, "impus", True)

# Rule: Every impus is hot.
impus_is_hot
    foreach
````
````

```
        facts.is_a($thing, "impus", True)
    assert
        facts.is_hot($thing, True)

# Rule: Impuses are rompuses.
impus_is_rompus
    foreach
        facts.is_a($thing, "impus", True)
    assert
        facts.is_a($thing, "rompus", True)

# Rule: Rompuses are mean.
rompus_is_mean
    foreach
        facts.is_a($thing, "rompus", True)
    assert
        facts.is_mean($thing, True)

# Rule: Each rompus is a vumpus.
rompus_is_vumpus
    foreach
        facts.is_a($thing, "rompus", True)
    assert
        facts.is_a($thing, "vumpus", True)

# Rule: Vumpuses are sweet.
vumpus_is_sweet
    foreach
        facts.is_a($thing, "vumpus", True)
    assert
        facts.is_sweet($thing, True)

# Rule: Vumpuses are tumpuses.
vumpus_is_tumpus
    foreach
        facts.is_a($thing, "vumpus", True)
    assert
        facts.is_a($thing, "tumpus", True)
```
Query:
```query
facts.is_bright("Sam", True)
```

Problem:
Every dumpus is luminous. Each dumpus is a numpus. Numpuses are cold. Numpuses are tumpuses. Tumpuses are bright. Each tumpus is a yumpus. Every yumpus is opaque. Jompuses are amenable. Yumpuses are zumpuses. Every zumpus is not amenable. Zumpuses are rompuses. Rompuses are blue. Rompuses are wumpuses. Wumpuses are spicy. Every wumpus is a vumpus. Wren is a zumpus.
Question:
Is the following statement true or false? Wren is not amenable.
Plan:
1. Identify all entities and types: Wren, dumpus, numpus, tumpus, yumpus, jompus, zumpus, rompus, wumpus, vumpus.
2. Identify all properties: luminous, cold, bright, opaque, amenable, blue, spicy.
3. Represent "Wren is a zumpus" as a base fact: `is_a("Wren", "zumpus", True)`.
4. Convert all class properties and subclass relationships into rules.
5. Implement each rule using a `foreach` clause for the condition and an `assert` clause for the consequence.
6. The query tests if the statement "Wren is not amenable" is provable:
    -> Use `facts.is_amenable("Wren", False)`.
Facts:
```facts
# --- Fact about Wren ---
is_a("Wren", "zumpus", True)
```
Rules:
```rules
# Rule: Every dumpus is luminous.
dumpus_is_luminous
    foreach
        facts.is_a($thing, "dumpus", True)
    assert
        facts.is_luminous($thing, True)

# Rule: Each dumpus is a numpus.
dumpus_is_numpus
    foreach
        facts.is_a($thing, "dumpus", True)
    assert
        facts.is_a($thing, "numpus", True)

# Rule: Numpuses are cold.
numpus_is_cold
    foreach
        facts.is_a($thing, "numpus", True)
    assert
        facts.is_cold($thing, True)

# Rule: Numpuses are tumpuses.
numpus_is_tumpus
    foreach
        facts.is_a($thing, "numpus", True)
    assert
        facts.is_a($thing, "tumpus", True)

# Rule: Tumpuses are bright.
tumpus_is_bright
    foreach
        facts.is_a($thing, "tumpus", True)
    assert
        facts.is_bright($thing, True)

# Rule: Each tumpus is a yumpus.
tumpus_is_yumpus
    foreach
        facts.is_a($thing, "tumpus", True)
    assert
        facts.is_a($thing, "yumpus", True)

# Rule: Every yumpus is opaque.
yumpus_is_opaque
    foreach
        facts.is_a($thing, "yumpus", True)
    assert
        facts.is_opaque($thing, True)

# Rule: Jompuses are amenable.
jompus_is_amenable
    foreach
        facts.is_a($thing, "jompus", True)
    assert
        facts.is_amenable($thing, True)

# Rule: Yumpuses are zumpuses.
yumpus_is_zumpus
    foreach
        facts.is_a($thing, "yumpus", True)
    assert
        facts.is_a($thing, "zumpus", True)

# Rule: Every zumpus is not amenable.
zumpus_is_not_amenable
    foreach
        facts.is_a($thing, "zumpus", True)
    assert
        facts.is_amenable($thing, False)

# Rule: Zumpuses are rompuses.
zumpus_is_rompus
    foreach
        facts.is_a($thing, "zumpus", True)
    assert
        facts.is_a($thing, "rompus", True)

# Rule: Rompuses are blue.
rompus_is_blue
    foreach
        facts.is_a($thing, "rompus", True)
    assert
        facts.is_blue($thing, True)

# Rule: Rompuses are wumpuses.
rompus_is_wumpus
    foreach
        facts.is_a($thing, "rompus", True)
    assert
        facts.is_a($thing, "wumpus", True)

# Rule: Wumpuses are spicy.
wumpus_is_spicy
    foreach
        facts.is_a($thing, "wumpus", True)
    assert
        facts.is_spicy($thing, True)

# Rule: Every wumpus is a vumpus.
wumpus_is_vumpus
    foreach
        facts.is_a($thing, "wumpus", True)
    assert
        facts.is_a($thing, "vumpus", True)
```
Query:
```query
facts.is_amenable("Wren", False)
```

Problem:
{context}
Question:
{question}
Plan:
{plan}

Self-Refinement (D&P)

```
You are an expert Python syntax fixer specializing in the PyKe solver library.

Problem:
```

```
{context}
Question:
{question}
Plan:
{plan}

## Original Code
{code}

## Error message from solver
{syntax_error}

# Instructions for Code Fixing
- Write a new version of the code that corrects the
identified syntax error.
- Ensure the new code is syntactically correct and logically
 sound.

# Output Requirements:
Translate the given natural language problem into
syntactically and semantically correct PyKe program,
consisting three parts: Facts, Rules, and Query.
- Facts: Declare the explicitly stated properties and
relationships of entities from the context using ternary
predicates in the form predicate(subject, object,
truth_value), where truth_value is typically True to denote
the fact holds.
- Rules: Encode conditional logic from the context (e.g., "
If... then...") as PyKe rule clauses. Each rule uses foreach
 to specify matching conditions and assert to define the
logical consequence. Each rule captures an inference pattern
.
- Query: Translate the natural language question into a
formal goal that can be proven using the defined facts and
rules. Queries usually take the form facts.predicate(subject
, expected_value) and represent the statement being tested
for truth.
```

CoT (D&P)

```
Given a problem statement as contexts, the task is to answer
 a logical reasoning question.
------
Problem:
Every yumpus is small. Every tumpus is hot. Each tumpus is a
 zumpus. Zumpuses are not small. Each zumpus is a rompus.
Rompuses are brown. Each rompus is a wumpus. Fae is a zumpus
.
Question:
Is the following statement true or false? Fae is not small.

Options:
A) True
B) False

Reasoning:
Fae is a zumpus. Zumpuses are not small. So Fae is not small
.

The correct option is: A
------
Problem:
Zumpuses are floral. Each zumpus is a jompus. Each jompus is
 brown. Jompuses are numpuses. Dumpuses are bright. Each
numpus is not opaque. Every numpus is a yumpus. Every yumpus
 is not bright. Every yumpus is a wumpus. Wumpuses are
luminous. Wumpuses are impuses. Every impus is hot. Impuses
are rompuses. Rompuses are mean. Each rompus is a vumpus.
Vumpuses are sweet. Vumpuses are tumpuses. Sam is a yumpus.
Question:
Is the following statement true or false? Sam is bright.

Options:
A) True
B) False

Reasoning:
Sam is a yumpus. Every yumpus is not bright. So Sam is not
bright.

The correct option is: B
------
Problem:
Every dumpus is luminous. Each dumpus is a numpus. Numpuses
are cold. Numpuses are tumpuses. Tumpuses are bright. Each
tumpus is a yumpus. Every yumpus is opaque. Jompuses are
amenable. Yumpuses are zumpuses. Every zumpus is not
amenable. Zumpuses are rompuses. Rompuses are blue. Rompuses
 are wumpuses. Wumpuses are spicy. Every wumpus is a vumpus.
 Wren is a zumpus.
Question:
Is the following statement true or false? Wren is not
amenable.

Options:
A) True
B) False

Reasoning:
Wren is a zumpus. Every zumpus is not amenable. So Wren is
not amenable.

The correct option is: A
------
Context:
{context}

Question: {question}

Options:
{options}

Reasoning:
```

A.3.4 LogicalDeduction

Sketched Plan Generation (D&P)

```
You are an expert in logical reasoning and python-constraint
 library.
Your goal is to create a DETAILED, STEP-BY-STEP PLAN to
translate the given natural language problem into python-
constraint code.
Remember to only output the raw plan.

Problem:
"The following paragraphs each describe a set of five
objects arranged in a fixed order. The statements are
logically consistent within each paragraph. On a branch,
there are five birds: a quail, an owl, a raven, a falcon,
and a robin. The owl is the leftmost. The robin is to the
left of the raven. The quail is the rightmost. The raven is
the third from the left."
Question:
"Which of the following is true?"
Choices:
["A) The quail is the rightmost.", "B) The owl is the
rightmost.", "C) The raven is the rightmost.", "D) The
falcon is the rightmost.", "E) The robin is the rightmost."]
Plan:
"
1.  **Initialize the Constraint Satisfaction Problem:**
    * Import the `Problem` and `AllDifferentConstraint`
    classes from the `constraint` library.
    * Create an instance of the `Problem` class.

2.  **Define Variables and Domain:**
    * Create a list of strings representing the five birds:
    `"quail"`, `"owl"`, `"raven"`, `"falcon"`, and `"robin
    "`. These will be the variables in our problem.
    * Define the domain for these variables as the integer
    positions from 1 to 5, where 1 represents the leftmost
    position and 5 represents the rightmost.
    * Add the variables and their domain to the `Problem`
    instance.

3.  **Add Constraints from the Problem Statement:**
    * Add an `AllDifferentConstraint` to ensure that each
    bird occupies a unique position.
    * Translate each statement from the problem description
    into a specific constraint using lambda functions:
        * "The owl is the leftmost" translates to the owl's
        position being equal to 1.
        * "The robin is to the left of the raven" translates
         to the robin's position value being less than the
        raven's position value.
        * "The quail is the rightmost" translates to the
        quail's position being equal to 5.
        * "The raven is the third from the left" translates
        to the raven's position being equal to 3.

4.  **Solve the Problem:**
    * Call the `getSolutions()` method on the `Problem`
    object to find all valid arrangements.
    * Since the problem has a unique solution, retrieve the
    first (and only) solution from the returned list.

5.  **Identify the Correct Choice:**
    * The question asks which bird is the rightmost (
    position 5).
    * Create a dictionary to map the choice letters ("A", "B
    ", "C", "D", "E") to the corresponding bird names from
    the choices.
    * Iterate through the choices. For each choice, check if
     the bird's position in the found solution is 5.
    * When the condition is met, print the corresponding
    choice letter as the final answer.
"

Problem:
"The following paragraphs each describe a set of five
objects arranged in a fixed order. The statements are
logically consistent within each paragraph. In an antique
car show, there are five vehicles: a hatchback, a bus, a
convertible, a tractor, and a minivan. The tractor is older
than the bus. The minivan is newer than the bus. The
```

```
hatchback is the second-newest. The minivan is older than
the convertible."
Question:
"Which of the following is true?"
Choices:
["A) The hatchback is the second-oldest.", "B) The bus is
the second-oldest.", "C) The convertible is the second-
oldest.", "D) The tractor is the second-oldest.", "E) The
minivan is the second-oldest."]
Plan:
"
1. **Initialize the Problem Environment:**
    * Import the necessary `Problem` and `
    AllDifferentConstraint` classes from the `constraint`
    library.
    * Create an instance of the `Problem` class to hold the
    variables and constraints.

2. **Define Variables and Their Domain:**
    * Establish the variables of the problem, which are the
    five vehicles: `"hatchback"`, `"bus"`, `"convertible"`,
    `"tractor"`, and `"minivan"`.
    * Define a numerical domain to represent the age ranking
    . Use integers from 1 to 5, where **1 represents the
    oldest** and **5 represents the newest** vehicle.
    * Add the vehicle variables and their rank domain to the
     `Problem` instance.

3. **Translate Rules into Constraints:**
    * Add an `AllDifferentConstraint` to ensure no two
    vehicles have the same age rank.
    * Convert each rule from the problem description into a
    formal constraint using a `lambda` function:
        * "The tractor is older than the bus" means the
        tractor's rank is a smaller number than the bus's
        rank (`tractor < bus`).
        * "The minivan is newer than the bus" means the
        minivan's rank is a larger number than the bus's
        rank (`bus < minivan`).
        * "The hatchback is the second-newest" means its
        rank is 4 (since 5 is the newest).
        * "The minivan is older than the convertible" means
        the minivan's rank is a smaller number than the
        convertible's rank (`minivan < convertible`).

4. **Solve for the Arrangement:**
    * Use the `getSolutions()` method to find all possible
    valid arrangements that satisfy every constraint.
    * Since the problem is well-defined, extract the single
    unique solution from the list of solutions.

5. **Identify the Correct Answer:**
    * Analyze the question: "Which of the following is the
    second-oldest?". Based on our ranking system (1=oldest),
     this corresponds to the vehicle with **rank 2**.
    * Create a dictionary that maps the choice letters ("A",
     "B", etc.) to the vehicle names they represent.
    * Iterate through the solved arrangement to find which
    vehicle has the rank of 2.
    * Print the letter corresponding to the vehicle that
    satisfies this condition.
"

Problem:
"The following paragraphs each describe a set of five
objects arranged in a fixed order. The statements are
logically consistent within each paragraph. On a shelf,
there are five books: a yellow book, a green book, a gray
book, a blue book, and an orange book. The gray book is to
the left of the green book. The gray book is the second from
 the right. The yellow book is to the right of the orange
book. The blue book is the second from the left."
Question:
"Which of the following is true?"
Choices:
["A) The yellow book is the leftmost.", "B) The green book
is the leftmost.", "C) The gray book is the leftmost.", "D)
The blue book is the leftmost.", "E) The orange book is the
leftmost."]
Plan:
"
1. **Set Up the Problem Environment:**
    * Import the `Problem` and `AllDifferentConstraint`
    classes from the `constraint` library.
    * Instantiate the `Problem` class to begin building the
    puzzle solver.

2. **Define Variables and the Domain:**
    * Identify the variables, which are the five books: `"
    yellow"`, `"green"`, `"gray"`, `"blue"`, and `"orange"`.
    * Define the domain for these variables. This will be a
    range of integers from 1 to 5, representing the
    positions on the shelf. Establish a clear mapping: **1 =
     leftmost** and **5 = rightmost**.
    * Add the book variables and their position domain to
    the `Problem` instance.

3. **Formulate and Add Constraints:**
    * Add an `AllDifferentConstraint` to ensure each book is
     in a different position.
    * Translate each rule from the text into a logical
    constraint using `lambda` functions:
        * "The gray book is to the left of the green book"
        implies the gray book's position number is less than
         the green book's (`gray < green`).
        * "The gray book is the second from the right"
        corresponds to **position 4** in our 1-to-5 scale.
        * "The yellow book is to the right of the orange
        book" implies the orange book's position number is
        less than the yellow book's (`orange < yellow`).
        * "The blue book is the second from the left"
        corresponds to **position 2**.

4. **Solve for the Final Arrangement:**
    * Execute the solver by calling the `getSolutions()`
    method.
    * The problem has a single, unique solution, so retrieve
     it from the list of solutions returned by the solver.

5. **Determine the Correct Answer:**
    * The question asks to identify the **leftmost** book.
    In our defined system, this is the book at **position
    1**.
    * Create a dictionary to map the multiple-choice letters
     ("A", "B", etc.) to the book names.
    * Iterate through the `solution` dictionary to find the
    book whose value (position) is 1.
    * Print the corresponding choice letter for that book as
     the final answer.
"

Problem:
{context}
Question:
{question}
Choices:
{options}
Plan:
"
"
```

Code Generation (D&P)

```
You are an expert Python programmer specializing in the
python-constraint library.
Your goal is to write executable python-constraint code that
 precisely implements the provided plan to solve the given
problem.
Remember to only output the raw Python code.

Problem:
"The following paragraphs each describe a set of five
objects arranged in a fixed order. The statements are
logically consistent within each paragraph. On a branch,
there are five birds: a quail, an owl, a raven, a falcon,
and a robin. The owl is the leftmost. The robin is to the
left of the raven. The quail is the rightmost. The raven is
the third from the left."
Question:
"Which of the following is true?"
Choices:
["A) The quail is the rightmost.", "B) The owl is the
rightmost.", "C) The raven is the rightmost.", "D) The
falcon is the rightmost.", "E) The robin is the rightmost."]
Plan:
"
1. **Initialize the Constraint Satisfaction Problem:**
    * Import the `Problem` and `AllDifferentConstraint`
    classes from the `constraint` library.
    * Create an instance of the `Problem` class.

2. **Define Variables and Domain:**
    * Create a list of strings representing the five birds:
    `"quail"`, `"owl"`, `"raven"`, `"falcon"`, and `"robin
    "`. These will be the variables in our problem.
    * Define the domain for these variables as the integer
    positions from 1 to 5, where 1 represents the leftmost
    position and 5 represents the rightmost.
    * Add the variables and their domain to the `Problem`
    instance.

3. **Add Constraints from the Problem Statement:**
    * Add an `AllDifferentConstraint` to ensure that each
    bird occupies a unique position.
    * Translate each statement from the problem description
    into a specific constraint using lambda functions:
        * "The owl is the leftmost" translates to the owl's
        position being equal to 1.
        * "The robin is to the left of the raven" translates
         to the robin's position value being less than the
        raven's position value.
        * "The quail is the rightmost" translates to the
        quail's position being equal to 5.
        * "The raven is the third from the left" translates
        to the raven's position being equal to 3.

4. **Solve the Problem:**
    * Call the `getSolutions()` method on the `Problem`
    object to find all valid arrangements.
```

```
    * Since the problem has a unique solution, retrieve the
    first (and only) solution from the returned list.

5.  **Identify the Correct Choice:**
    * The question asks which bird is the rightmost (
    position 5).
    * Create a dictionary to map the choice letters ("A", "B
    ", "C", "D", "E") to the corresponding bird names from
    the choices.
    * Iterate through the choices. For each choice, check if
     the bird's position in the found solution is 5.
    * When the condition is met, print the corresponding
    choice letter as the final answer.
"
python-constraint Program:
```
from constraint import *

"""
Solves the bird arrangement logic puzzle using the python-
constraint library.
"""
problem = Problem()

# The five birds are the variables
birds = ["quail", "owl", "raven", "falcon", "robin"]

The domain is the five positions on the branch (1=leftmost
, 5=rightmost)
positions = range(1, 6)

problem.addVariables(birds, positions)

# Add constraints based on the problem description
# 1. All birds are in different positions
problem.addConstraint(AllDifferentConstraint())

# 2. "The owl is the leftmost."
problem.addConstraint(lambda owl: owl == 1, ["owl"])

3. "The robin is to the left of the raven."
problem.addConstraint(lambda robin, raven: robin < raven, ["
robin", "raven"])

# 4. "The quail is the rightmost."
problem.addConstraint(lambda quail: quail == 5, ["quail"])

# 5. "The raven is the third from the left."
problem.addConstraint(lambda raven: raven == 3, ["raven"])

# Get the single solution
solutions = problem.getSolutions()

The choices map a letter to the bird stated as being the
rightmost.
# The rightmost position is 5.
choices = {
    "A": "quail",
    "B": "owl",
    "C": "raven",
    "D": "falcon",
    "E": "robin"
}

# Determine the correct choice
for solution in solutions:
    for letter, bird_name in choices.items():
        if solution[bird_name] == 5:
            print(letter)
```

Problem:
"The following paragraphs each describe a set of five
objects arranged in a fixed order. The statements are
logically consistent within each paragraph. In an antique
car show, there are five vehicles: a hatchback, a bus, a
convertible, a tractor, and a minivan. The tractor is older
than the bus. The minivan is newer than the bus. The
hatchback is the second-newest. The minivan is older than
the convertible."
Question:
"Which of the following is true?"
Choices:
["A) The hatchback is the second-oldest.", "B) The bus is
the second-oldest.", "C) The convertible is the second-
oldest.", "D) The tractor is the second-oldest.", "E) The
minivan is the second-oldest."]
Plan:
"
1.  **Initialize the Problem Environment:**
    * Import the necessary `Problem` and `
    AllDifferentConstraint` classes from the `constraint`
    library.
    * Create an instance of the `Problem` class to hold the
    variables and constraints.

2.  **Define Variables and Their Domain:**
    * Establish the variables of the problem, which are the
    five vehicles: `"hatchback"`, `"bus"`, `"convertible"`,
    `"tractor"`, and `"minivan"`.
    * Define a numerical domain to represent the age ranking
    . Use integers from 1 to 5, where **1 represents the
    oldest** and **5 represents the newest** vehicle.
    * Add the vehicle variables and their rank domain to the
     `Problem` instance.

3.  **Translate Rules into Constraints:**
    * Add an `AllDifferentConstraint` to ensure no two
    vehicles have the same age rank.
    * Convert each rule from the problem description into a
    formal constraint using a `lambda` function:
        * "The tractor is older than the bus" means the
        tractor's rank is a smaller number than the bus's
        rank (`tractor < bus`).
        * "The minivan is newer than the bus" means the
        minivan's rank is a larger number than the bus's
        rank (`bus < minivan`).
        * "The hatchback is the second-newest" means its
        rank is 4 (since 5 is the newest).
        * "The minivan is older than the convertible" means
        the minivan's rank is a smaller number than the
        convertible's rank (`minivan < convertible`).

4.  **Solve for the Arrangement:**
    * Use the `getSolutions()` method to find all possible
    valid arrangements that satisfy every constraint.
    * Since the problem is well-defined, extract the single
    unique solution from the list of solutions.

5.  **Identify the Correct Answer:**
    * Analyze the question: "Which of the following is the
    second-oldest?". Based on our ranking system (1=oldest),
     this corresponds to the vehicle with **rank 2**.
    * Create a dictionary that maps the choice letters ("A",
     "B", etc.) to the vehicle names they represent.
    * Iterate through the solved arrangement to find which
    vehicle has the rank of 2.
    * Print the letter corresponding to the vehicle that
    satisfies this condition.
"
python-constraint Program:
```
from constraint import *

# Initialize the constraint satisfaction problem
problem = Problem()

Define the variables (the five vehicles) and the domain (
their age rank)
Let's use ranks 1 to 5, where 1 is the oldest and 5 is the
 newest.
vehicles = ["hatchback", "bus", "convertible", "tractor", "
minivan"]
ranks = range(1, 6)
problem.addVariables(vehicles, ranks)

# Add constraints based on the puzzle's statements
# 1. All vehicles must have a different age rank.
problem.addConstraint(AllDifferentConstraint())

2. "The tractor is older than the bus." (Tractor's rank is
 a smaller number)
problem.addConstraint(lambda tractor, bus: tractor < bus, ("
tractor", "bus"))

3. "The minivan is newer than the bus." (Minivan's rank is
 a larger number)
problem.addConstraint(lambda bus, minivan: bus < minivan, ("
bus", "minivan"))

4. "The hatchback is the second-newest." (The 4th rank out
 of 5)
problem.addConstraint(lambda hatchback: hatchback == 4, ("
hatchback",))

5. "The minivan is older than the convertible."
problem.addConstraint(lambda minivan, convertible: minivan <
 convertible, ("minivan", "convertible"))

# Find the unique solution to the problem
solutions = problem.getSolutions()

The question asks which vehicle is the "second-oldest".
In our ranking system (1=oldest), the second-oldest has a
rank of 2.
choices = {
    "A": "hatchback",
    "B": "bus",
    "C": "convertible",
    "D": "tractor",
    "E": "minivan"
}

Check the solution to find the vehicle with rank 2 and
print the corresponding letter.
for solution in solutions:
    for letter, vehicle_name in choices.items():
        if solution[vehicle_name] == 2:
            print(letter)
```
```

```
Problem:
"The following paragraphs each describe a set of five
objects arranged in a fixed order. The statements are
logically consistent within each paragraph. On a shelf,
there are five books: a yellow book, a green book, a gray
book, a blue book, and an orange book. The gray book is to
the left of the green book. The gray book is the second from
 the right. The yellow book is to the right of the orange
book. The blue book is the second from the left."
Question:
"Which of the following is true?"
Choices:
["A) The yellow book is the leftmost.", "B) The green book
is the leftmost.", "C) The gray book is the leftmost.", "D)
The blue book is the leftmost.", "E) The orange book is the
leftmost."]
Plan:
"
1.  **Set Up the Problem Environment:**
    * Import the `Problem` and `AllDifferentConstraint`
    classes from the `constraint` library.
    * Instantiate the `Problem` class to begin building the
    puzzle solver.

2.  **Define Variables and the Domain:**
    * Identify the variables, which are the five books: `"
    yellow"`, `"green"`, `"gray"`, `"blue"`, and `"orange"`.
    * Define the domain for these variables. This will be a
    range of integers from 1 to 5, representing the
    positions on the shelf. Establish a clear mapping: **1 =
     leftmost** and **5 = rightmost**.
    * Add the book variables and their position domain to
    the `Problem` instance.

3.  **Formulate and Add Constraints:**
    * Add an `AllDifferentConstraint` to ensure each book is
     in a different position.
    * Translate each rule from the text into a logical
    constraint using `lambda` functions:
        * "The gray book is to the left of the green book"
        implies the gray book's position number is less than
         the green book's (`gray < green`).
        * "The gray book is the second from the right"
        corresponds to **position 4** in our 1-to-5 scale.
        * "The yellow book is to the right of the orange
        book" implies the orange book's position number is
        less than the yellow book's (`orange < yellow`).
        * "The blue book is the second from the left"
        corresponds to **position 2**.

4.  **Solve for the Final Arrangement:**
    * Execute the solver by calling the `getSolutions()`
    method.
    * The problem has a single, unique solution, so retrieve
     it from the list of solutions returned by the solver.

5.  **Determine the Correct Answer:**
    * The question asks to identify the **leftmost** book.
    In our defined system, this is the book at **position
    1**.
    * Create a dictionary to map the multiple-choice letters
     ("A", "B", etc.) to the book names.
    * Iterate through the `solution` dictionary to find the
    book whose value (position) is 1.
    * Print the corresponding choice letter for that book as
     the final answer.
"
python-constraint Program:
```
from constraint import *

# Initialize the constraint satisfaction problem.
problem = Problem()

Define the variables (the five books) and the domain (
their positions on the shelf).
We'll use positions 1 to 5, where 1 is the leftmost and 5
is the rightmost.
books = ["yellow", "green", "gray", "blue", "orange"]
positions = range(1, 6)
problem.addVariables(books, positions)

# Add constraints based on the puzzle's statements.
# 1. All books must be in a unique position.
problem.addConstraint(AllDifferentConstraint())

2. "The gray book is to the left of the green book."
(The gray book's position number is less than the green
book's).
problem.addConstraint(lambda gray, green: gray < green, ("
gray", "green"))

# 3. "The gray book is the second from the right."
# In a set of 5, this is position 4.
problem.addConstraint(lambda gray: gray == 4, ("gray",))

4. "The yellow book is to the right of the orange book."
(The orange book's position number is less than the yellow
 book's).
problem.addConstraint(lambda orange, yellow: orange < yellow
, ("orange", "yellow"))

# 5. "The blue book is the second from the left."
# This is position 2.
problem.addConstraint(lambda blue: blue == 2, ("blue",))

# Find the unique solution to the problem.
solutions = problem.getSolutions()

# The question asks which book is the "leftmost".
# The leftmost position is 1.
choices = {
    "A": "yellow",
    "B": "green",
    "C": "gray",
    "D": "blue",
    "E": "orange"
}

Check the solution to find which book is at position 1 and
 print its corresponding letter.
for solution in solutions:
    for letter, book_name in choices.items():
        if solution[book_name] == 1:
            print(letter)
```

Problem:
{context}
Question:
{question}
Choices:
{options}
Plan:
{plan}
python-constraint Program:
```

```
```

Self-Refinement (D&P)

```
You are an expert Python syntax fixer specializing in the
python-constraint library.
Your task is to fix the syntax errors in the code below
given the error message from python-constraint solver.
Produce ONLY raw Python code. Do not include any other
comments or explanations.

Code with Syntax Errors:
{code}
Error message from solver:
{syntax_error}

Fixed Code:
```
from constraint import *

```
```

CoT (D&P)

```
Given a problem statement as contexts, the task is to answer
 a logical reasoning question.
------
Context:
The following paragraphs each describe a set of five objects
 arranged in a fixed order. The statements are logically
consistent within each paragraph. On a branch, there are
five birds: a quail, an owl, a raven, a falcon, and a robin.
 The owl is the leftmost. The robin is to the left of the
raven. The quail is the rightmost. The raven is the third
from the left.

Question: Which of the following is true?

Options:
A) The quail is the rightmost.
B) The owl is the rightmost.
C) The raven is the rightmost.
D) The falcon is the rightmost.
E) The robin is the rightmost.

Reasoning:
The owl is the leftmost. This means owl is not the rightmost
. The robin is to the left of the raven. This means neither
robin nor raven is the rightmost. The quail is the rightmost
. This directly answers the question, so option A) The quail
 is the rightmost is true. The raven is the third from the
left. This means raven is also not the rightmost. So, the
answer is: A) The quail is the rightmost.

The correct option is: A) The quail is the rightmost.
------
Context:
The following paragraphs each describe a set of five objects
 arranged in a fixed order. The statements are logically
```

```
consistent within each paragraph. In an antique car show,
there are five vehicles: a hatchback, a bus, a convertible,
a tractor, and a minivan. The tractor is older than the bus.
 The minivan is newer than the bus. The hatchback is the
second-newest. The minivan is older than the convertible.

Question: Which of the following is true?

Options:
A) The hatchback is the second-oldest.
B) The bus is the second-oldest.
C) The convertible is the second-oldest.
D) The tractor is the second-oldest.
E) The minivan is the second-oldest.

Reasoning:
The hatchback is the second-newest, which means it is the
fourth-oldest, so option A is incorrect. The tractor is
older than the bus, and the minivan is newer than the bus.
This establishes the order: tractor, bus, minivan. The
minivan is older than the convertible, extending the order
to: tractor, bus, minivan, convertible. These four cars must
 occupy the oldest, second-oldest, third-oldest, and newest
positions. Based on this order, the tractor is the oldest,
and the bus is the second-oldest. So, the answer is: B) The
bus is the second-oldest.

The correct option is: B) The bus is the second-oldest.
------
Context:
The following paragraphs each describe a set of five objects
 arranged in a fixed order. The statements are logically
consistent within each paragraph. On a shelf, there are five
 books: a yellow book, a green book, a gray book, a blue
book, and an orange book. The gray book is to the left of
the green book. The gray book is the second from the right.
The yellow book is to the right of the orange book. The blue
 book is the second from the left.

Question: Which of the following is true?

Options:
A) The yellow book is the leftmost.
B) The green book is the leftmost.
C) The gray book is the leftmost.
D) The blue book is the leftmost.
E) The orange book is the leftmost.

Reasoning:
The gray book is the second from the right (position 4) and
the blue book is the second from the left (position 2).
Neither can be the leftmost. The gray book is to the left of
 the green book, so the green book must be in the rightmost
position (position 5). The yellow book is to the right of
the orange book. The only remaining spots are the first (
leftmost) and third. Therefore, the orange book must be in
the first position and the yellow book in the third. This
makes the orange book the leftmost. So, the answer is: E)
The orange book is the leftmost.

The correct option is: E) The orange book is the leftmost.
------
Context:
{context}

Question: {question}

Options:
{options}

Reasoning:
```