

Prototype-then-Refine: A Neurosymbolic Approach for Improved Logical Reasoning with LLMs

Stanford CS224N Custom Project

Hashem Elezabi
Department of Computer Science
Stanford University
hashem@stanford.edu

Bassem Akoush
Department of Mechanical Engineering
Stanford University
bakoush@stanford.edu

Abstract

Large language models (LLMs) have shown remarkable capabilities, but even the best LLMs still struggle with complex logical reasoning problems. Recent methods like Logic-LM propose to combine LLMs with symbolic solvers to improve reasoning ability. By prompting the LLM to translate a natural language problem into a symbolic formulation and then feeding the symbolic formulation to a deterministic symbolic solver, this neuro-symbolic approach capitalizes on the ability of LLMs to understand flexible natural language while relying on the faithful and guaranteed reasoning ability of symbolic solvers. However, the Logic-LM approach fails when the translation is incorrect. To enable more robust generation of correct logic programs, we propose a new Prototype-then-Refine framework (ProRef) that applies an extensible series of *prototypers* to pick a strong prototype logic program which it then refines using an extensible series of *refiners*. On the challenging AR-LSAT dataset, we find that our framework significantly improves the executable rate of generated logic programs from 20.05% to 32.47%, enabling GPT-3.5-Turbo to almost match the executable rate of GPT-4 (32.61%). However, this does not improve the accuracy (percentage of questions answered correctly). We perform a thorough analysis of the errors in the generated programs and propose further improvements in our framework that can improve the accuracy as well as the executable rate.

1 Introduction

Large language models (LLMs) have shown remarkable capabilities, including human-like reasoning abilities. However, even the best LLMs still struggle with complex logical reasoning problems. In Liu et al. (2023b), authors perform a systematic evaluation of the logical reasoning ability of GPT-4 on logical reasoning datasets like LogiQA 2.0 (Liu et al., 2023a) and AR-LSAT (Zhong et al., 2021). They find that GPT-4 achieves an accuracy of 58.49% on the LogiQA 2.0 test set (compared to 83% human average performance) and 33.48% on the AR-LSAT (Zhong et al., 2021) test set (compared to 56% human average performance). The gap is even wider when we consider the *best* human performance, which is 99% on LogiQA 2.0 and 91% on AR-LSAT.

Recent studies (e.g. Golovneva et al. (2023)) found that LLMs sometimes make *unfaithful* reasoning, where the derived conclusion does not follow the previously generated reasoning process. While Chain-of-Thought (CoT) (Wei et al., 2023) prompting tries to imitate human reasoning processes, LLMs remain black-box probabilistic models, and they lack a mechanism to guarantee the faithfulness of reasoning. In contrast, symbolic solvers are faithful and transparent because their reasoning is based on symbolically represented knowledge and follows well-defined inference rules. Inspired by this observation, Logic-LM (Pan et al., 2023) and LINC (Olausson et al., 2023) introduce frameworks that integrate LLMs with symbolic solvers. Both frameworks lead to a significant improvement over purely LLM-based reasoning. Averaged over five logical reasoning datasets, Logic-LM achieves a

performance boost of 18.4% compared to CoT using GPT-4. LINC, also using GPT-4, scores 26% higher than CoT on the ProofWriter dataset (Tafjord et al., 2021) while performing comparably on the FOLIO dataset (Han et al., 2022).

Logic-LM and LINC are both examples of *neuro-symbolic* methods that combine neural LLMs with symbolic reasoning engines. Both methods decompose the logical reasoning problem into a *Problem Formulation* stage and a *Symbolic Reasoning* stage. The Problem Formulation stage uses an LLM to convert a natural language (NL) description of the problem into a symbolic formulation, or *logic program*, which is then fed to a deterministic symbolic solver in the Symbolic Reasoning stage. If the problem is correctly converted to a logic program in the Problem Formulation stage, the symbolic solver will be able to answer it. Thus, the challenge lies in having the LLM correctly translate the NL problem into a logic program that has correct syntax (does not lead to errors) and correct semantics (accurately reflects the problem). The main limitation of Logic-LM and LINC is that LLMs sometimes fail to correctly formulate the problem for the symbolic solver. This limitation is more pronounced on the most challenging AR-LSAT dataset (Zhong et al., 2021), where the percentage of logic programs generated by GPT-4 that are *executable* is 32.9% on the development set; the others lead to either parsing or execution errors. Logic-LM introduces a self-refinement module that attempts to fix errors in generated logic programs by re-prompting the LLM with the error message, and also includes a backup strategy of direct CoT prompting of the LLM in case the logic program cannot be fixed. These strategies help slightly improve performance, but the overall accuracy of Logic-LM with GPT-4 on AR-LSAT is 43.04%. This is currently the state-of-the-art accuracy on AR-LSAT, which shows that there is significant room for improvement in the ability of LLMs to reason through complex logical problems.

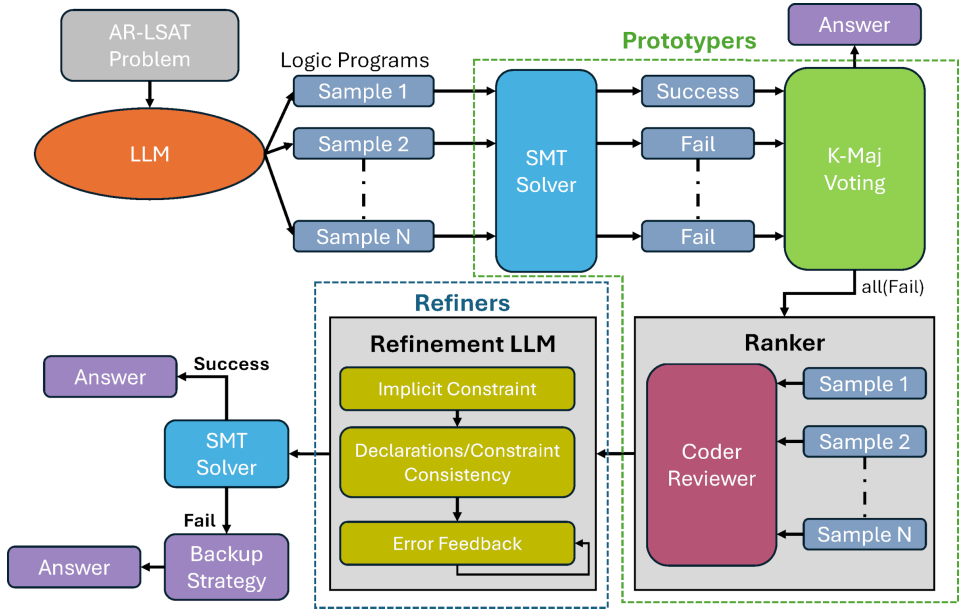


Figure 1: Our framework

We focus on the AR-LSAT dataset in our project. Our contributions are twofold. First, we perform a thorough error analysis on the failures of Logic-LM on AR-LSAT, categorizing and identifying the causes of the different errors. Second, based on observations from our error analysis, we propose a new and extensible framework, Prototype-then-Refine, that aims to improve the ability of LLMs to generate correct logic programs (i.e. improve the Problem Formulation stage of methods like Logic-LM). Figure 1 shows our framework. In the *Prototype* stage, we sample several logic programs, similar to what LINC does, and run one or more *prototypers* on the samples, with the goal of picking one final prototype logic program. While we propose a particular sequence of prototypers, the framework allows for arbitrarily more powerful prototypers to be applied. In the *Refine* stage, we run a series of error-guided *refiners* that attempt to fix the prototype logic program in a way that mimics how a human would fix an incorrect logic program.

2 Related Work

Logical Reasoning with Language Models. LLMs have shown the emergent ability to perform basic logical reasoning (Brown et al., 2020). Various prompting methods have been developed that rely on *in-context learning* and specially designed prompts to elicit improved LLM reasoning capabilities, including CoT prompting that prompts the LLM to generate step-by-step reasoning (Wei et al., 2023), self-consistency decoding that samples diverse reasoning paths and then selects the most consistent answer (Wang et al., 2023), and least-to-most prompting that breaks the problem down into simpler components that can be solved individually (Zhou et al., 2023). Other approaches rely on *fine-tuning* LLMs to improve their reasoning ability (Wei et al. (2022); Yang et al. (2022); Tafjord et al. (2022)). Despite the large variety of methods introduced in this area, LLMs still fall short of robust and faithful logical reasoning (Golovneva et al., 2023). More recently, solver-augmented LLM frameworks like Logic-LM (Pan et al., 2023) and LINC (Olausson et al., 2023) integrate LLMs with symbolic solvers to improve reasoning ability. A different approach was taken by Feng et al. (2023), whose development of LoGiPT supports the hypothesis that language models *can* be logical solvers by fine-tuning an LLM to directly emulate the reasoning processes of symbolic solvers. This approach has the benefit of not being prone to failure whenever a parsing or execution error occurs in the logic program fed to the symbolic solver. However, this approach sacrifices the deterministic and faithful reasoning offered by symbolic solvers, since it still relies on a black-box probabilistic model to perform reasoning. In our work, we follow the direction taken by Logic-LM and LINC, but we design a new Prototype-then-Refine framework that allows for more robust generation of correct logic programs.

LM-based Code Generation and Reranking. Recent pretrained LLMs have shown an impressive ability to generate code given natural language instructions in a zero-shot or few-shot manner (Chen et al. (2021); Fried et al. (2023); Chowdhery et al. (2023); Nijkamp et al. (2023)). Chen et al. (2021) observe that in many cases, among a diverse set of programs sampled from an LLM, there will be correct solutions, and they propose to find the best solutions by ranking them based on the likelihood assigned to the generated code by the LLM: $p(y|x)$, where x is the natural language instruction, y is the generated code, and $p(\cdot|x)$ is an LLM (prompted with instruction x) that serves as the *Coder* model (following the terminology in Zhang et al. (2023)). Several ranking methods have subsequently been proposed, some of which rely on information gained from executing the generated programs (Shi et al. (2022); Li et al. (2022)). Zhang et al. (2023) propose Coder-Reviewer reranking, which ranks programs using the product $p(y|x)p(x|y)$, where $p(x|y)$ is the probability given by a *Reviewer* model that is prompted to output the original instruction given the generated code. Reranking with Coder-Reviewer significantly improves over ranking with only the Coder model because it helps avoid mistakenly picking degenerate solutions such as extremely short code, repetitive or generic solutions, etc. Coder-Reviewer does not require program execution, and Zhang et al. (2023) show via experiments that it is complementary to execution-based reranking methods. In our work, we use both execution-based reranking and Coder-Reviewer reranking in the Prototype stage. While our framework aims to output logic programs, which have different characteristics from Python code or code in another popular language, we hypothesize that Coder-Reviewer reranking can help in our case as well. In designing our prototypes and refiners, we were also inspired by previous work on LLM-based program repair (Xia and Zhang, 2022, 2023).

3 Approach

We start by giving an overview of the Logic-LM framework and how we built upon it. Following the Logic-LM framework, our approach takes an input natural language problem, converts it into a symbolic logic program, executes a solver on the logic program, and outputs an answer. We focus on the AR-LSAT dataset, which consists of multiple-choice logical reasoning questions that each come with 5 options to pick the answer from. There are many different symbolic solvers, but for the analytical reasoning questions in AR-LSAT, a suitable solver is a Boolean satisfiability (SAT) solver like Z3 (de Moura and Bjørner, 2008). SAT is the problem of deciding if there is an assignment to the variables of a Boolean formula such that the formula is satisfied. Since many analytical reasoning problems can be formulated as SAT problems, the Z3 solver is suitable for AR-LSAT problems.

In the Logic-LM code for AR-LSAT, the LLM is prompted to output a logic program in an intermediate symbolic language. This intermediate output is called a *raw logic program*. This raw logic

program is then translated into a Z3 Python program, which is a Python program that implements the logic formulas given in the raw logic program and then executes the Z3 solver to determine which of the 5 given options is correct. The mapping from the raw logic program to the Z3 Python program is deterministic. Figure 2 shows an example of an AR-LSAT problem, its corresponding raw logic program, and its corresponding Z3 Python program.

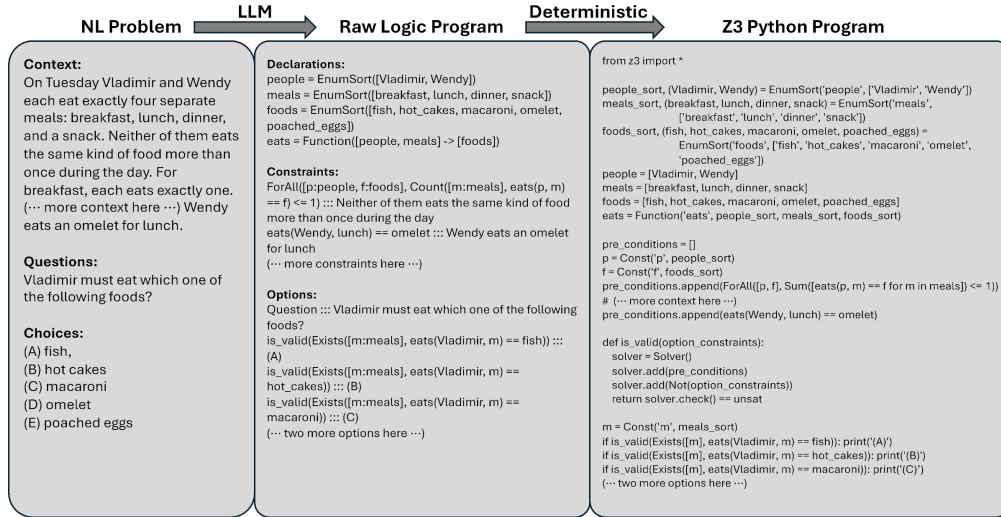


Figure 2: Translations

Most of our initial focus has been on the translation by the LLM from the NL problem description to the raw logic program. Whenever we mention "logic programs", we will be referring to raw logic programs. We adopt the same intermediate symbolic language used by Logic-LM for AR-LSAT problems, and we leverage the code translator module from the Logic-LM repository (<https://github.com/teacherpeterpan/Logic-LLM>) to convert the intermediate logic program to the Z3 Python program. There are two aspects to our approach: error analysis (Section 3.1) and the introduction of the Prototype-then-Refine framework (Section 3.2).

3.1 Error Analysis

First, we perform a thorough error analysis and manually inspect erroneous logic programs and their corresponding Z3 Python programs to identify common failure cases. While previous works Logic-LM (Pan et al., 2023) and LINC (Olausson et al., 2023) did their own error analysis, doing our own analysis enabled us to identify exactly how each error occurred, and how the relevant line in the raw logic program or Z3 Python program should be corrected. Moreover, we analyzed some cases more deeply, inspecting and modifying the automatically generated Z3 Python code and running the solver with different conditions to identify which constraints are missing or incorrect. This deeper analysis allowed us to derive concrete and useful insights that informed our design of the refiners in the Refine stage.

We focus on raw logic programs generated by GPT-4 or GPT-3.5-Turbo using the original Logic-LM prompt (shown in Appendix B). The examples we analyze are all taken from the dev set of AR-LSAT, for which the GPT-4 and GPT-3.5-Turbo output logic programs were helpfully provided by Pan et al. (2023) in their GitHub repository.

We parse the generated error messages into the following error categories: 1) Underconstrained, 2) Overconstrained, 3) TypeError, 4) SyntaxError, 5) NameError, 6) AttributeError, 7) Parsing Error, 8) Z3 Exception, and 9) No Output. Below are examples of a SyntaxError and a TypeError, with the erroneous line in **dark red** and the corrected line in **dark green**. We show examples of other error categories in the Appendix.

- SyntaxError: **invalid syntax**
`is_minimum(Count([s:seats], sits(George, s) && sits(Karl, s + 5)))`

```
is_minimum(Count([s:seats], And(sits(George, s), sits(Karl, s + 5))))
```

- TypeError: `is_sat()` takes 1 positional argument but 2 were given

```
is_exception(is_sat(serves(Guzman) == incentives_committee, serves(Hawking) ==
incentives_committee))

is_exception(is_sat(And(serves(Guzman) == incentives_committee, serves(Hawking)
== incentives_committee)))
```

Two noteworthy error categories we introduce are 1) Underconstrained and 2) Overconstrained. We explain these two error categories in Section 3.1.1. We report on the percentages of the different error categories in the Experiments section.

3.1.1 Overconstrained and Underconstrained Logic Programs

To explain overconstrained and underconstrained logic programs, we first need to describe the structure of the Z3 Python programs that serve as our symbolic solvers. Recall that every problem has five choices, so for every problem, the sampled raw logic program, as well as the generated Z3 Python program, has five queries to the Z3 solver at the end, designed so that only one of them returns True. The structure of these five queries depends on the question. Below, we give some examples of different types of questions and the solver queries they result in:

- If the question is "If Miyauchi and Wharton both serve on the finance committee, then which one of the following could be true?", and if option A says "A) Guzman and Lepp both serve on the finance committee.", then the solver query for option A would be:

```
is_sat(And(serves_on(Miyauchi) == finance_committee,
serves_on(Wharton) == finance_committee, serves_on(Guzman) ==
finance_committee, serves_on(Lepp) == finance_committee)).
```

- If the question asks if a condition CANNOT be true, then the `is_sat` call would be wrapped inside an `is_exception` call.
- If the question asks which of the five options must be true, then the five solver queries would be `is_valid` queries. Assuming option A is "A) Hamadi is appointed to the appellate court.", the corresponding solver query would be

```
is_valid(appointed(Hamadi) == appellate_court)
```

The function `is_valid` returns True if the negation of the given condition is unsatisfiable given the problem constraints, which would mean that the given condition must be true.

In our error analysis, we observed situations where the solver returns no output. This is because none of the five solver queries return True. If this occurs, the diagnosis of the problem depends on the type of solver queries.

Underconstrained. If the solver queries were `is_valid` solver queries, then if none of them return True, it means the solver does not evaluate the negation of any of them as unsatisfiable given the constraints, which means that the constraints are too loose: this is the *underconstrained* case. This case arises when some important, possibly implicit constraint is missing.

Overconstrained. On the other hand, if the solver queries are `is_sat` queries and the solver does not evaluate that any of the five choices as satisfiable, then it means that the constraints are too strict: this is the *overconstrained* case. When this occurs, it might mean that the constraints are in fact contradictory, and even just the constraints (preconditions) are not evaluated as satisfiable by the solver. If the solver *does* evaluate the preconditions as satisfiable, but does not evaluate any of the solver queries as satisfiable, then there is something wrong in the formulas of the solver queries that makes even the ground-truth correct choice unsatisfiable.

These new insights are more detailed and low-level than the error analysis done in Pan et al. (2023), and we can use these insights to design better refiners based on whether the program is underconstrained or overconstrained.

3.2 Prototype-then-Refine Framework

Our second main contribution is the development of a novel framework that we call Prototype-then-Refine, which is designed to enable more robust generation of correct logic programs. We build our approach, Prototype-then-Refine, based on the foundation given by Logic-LM. Our approach decomposes the Problem Formulation stage from Logic-LM into two stages. The purpose of the Prototype stage is to pick a strong prototype of a raw logic program, while the purpose of the Refine stage is to iteratively refine the prototype program it is given if it leads to errors, in a way that mimics how a human would fix the prototype logic program. In some cases, the prototype program would already execute in the solver without errors, in which case we don't need syntax-focused refinement, and might elect to skip further semantics-focused refinement. This design is extensible: we can introduce arbitrarily more sophisticated prototypers and refiners, each of which can make the overall pipeline more robust. Next, we discuss different examples of prototypers and refiners, and describe our main Prototype-then-Refine pipeline.

3.2.1 Prototypers

We implement two prototypers below. Note that arbitrarily more powerful prototypers can be added.

- **Execution-based prototyper with majority voting.** This prototyper samples several logic programs and executes each of them with Z3 (after translating to a Z3 Python program). Let k be the number of samples (we use $k = 10$ by default). Among k solver outputs, we take the $s \leq k$ outputs that are *not* errors. These s outputs will each be one of the five choices A, B, C, D, or E. Among these, we take the most common answer as our prototype predicted answer. We pick any of the logic programs that predicted this majority answer as the prototype program. Unless we will do heuristics-based semantic refinement in the Refine stage, this prototype predicted answer is the final answer given by our framework. Note that this prototyper is based on the LINC design (Olausson et al., 2023).
- **Coder-Reviewer prototyper.** This prototyper is applied if all k samples from the majority-voting prototyper lead to errors. In this case, we want to pick one final prototype, but we don't have any feedback from execution. Thus, we rank the programs using coder-reviewer reranking. Specifically, given a natural language instruction x , we sample k logic programs $y_{i \in 1:k}$ using a Coder model $p(\cdot|x)$ created by prompting the LLM to generate the logic program using the same prompt Logic-LM uses (see full prompt in Appendix D). We then create a Reviewer model by putting the output logic program y_i in the prompt and asking the LLM to generate the original natural language problem. Specifically, we give the following instruction to the LLM:

```
You are given a logic program consisting of three parts:  
Declarations, Constraints, and Options. This logic program is a  
formalization of an original natural language problem. Your task is  
to output the original natural language problem.
```

We also switch the inputs and outputs in the few-shot demonstrations so that the input is the logic program and the output is the NL problem (the full prompt is given in Appendix C). We then rank the program by the log of the product of the Coder and Reviewer probabilities $\log p(y_i|x)p(x_i|y_i) = \log p(y_i|x) + \log p(x_i|y_i)$, where x_i is the generated backward translation of the NL instruction given y_i . We retrieve the log-probabilities by setting `logprobs=True` in the OpenAI API call. The program y_i that ranks first under this reranking is picked as the prototype program. At this point, the prototype predicted answer is still None, since the program yields an error when the solver is executed. This program will need to be refined in the Refine stage.

3.2.2 Refiners

Below is a list of refiners we have implemented. We introduce two new refiners based on the errors we encountered during our error analysis: the *implicit constraints* refiner and the *consistency* refiner. We also include an error feedback refiner that is directly based on the self-refinement module from Logic-LM. More refiners can be added as more failure cases are discovered.

- **Implicit constraints refiner.** This refiner prompts the LLM to identify implicit constraints that are not explicitly mentioned in the problem statement. Our goal for this refiner was to add implicit constraints that are common sense to humans but need to be explicitly enforced in the logic formulas

for the solver to work together, e.g. that Fred is a boy (`gender(Fred) == boy`) or that a locker that is shared between a boy and a girl is implicitly not shared by anyone else (for any shared locker, ensure that no more than two children are assigned to that locker). Appendix A.1 shows the prompt we use and one example of our few-shot demonstrations.

- **Consistency refiner.** This refiner prompts the LLM to fix any inconsistencies between the Declarations and Constraints in the logic program. For example, in one case, the Declarations defined a list called `children` that contained four boys and three girls: `children = EnumSort([Fred, Juan, Marc, Paul, Nita, Rachel, Trisha])`. Then, to enforce that a shared locker must be shared between a boy and a girl, the LLM created a constraint that assumed that a list `boys` and a list `girls` were defined: `ForAll([l:lockers] ...<logic capturing if two children are assigned to locker l, then>... Exists([b:boys, g:girls], And(assigned(b) == 1, assigned(g) == 1))`. This refiner prompts the LLM to fix such cases, and we provide few-shot demonstrations showing manually constructed examples of incorrect and corrected programs. Appendix A.2 shows the prompt and an example demonstration.
- **Error feedback refiner.** This refiner is based on the self-refinement module of Logic-LM (Pan et al., 2023), which prompts the LLM with the error message and asks it to fix the program. This refiner is parameterized by the max number of feedback rounds, which we set to 4 in our main experiments. Appendix A.3 shows the prompt and an example correction.

4 Experiments

During our project, we ran various experiments with different goals. We omit some experiments that don't bear relevance to this final report, such as the experiment we ran to evaluate LINC on AR-LSAT, the results of which we reported in the milestone. Our main experiments aimed to evaluate three questions:

1. How do different LLMs (including open-source LLMs) perform under the baseline Logic-LM framework? The Logic-LM paper (Pan et al., 2023) only evaluates its framework using OpenAI models, so we wanted to evaluate the framework using less powerful open-source models.
2. Does our Prototype-then-Refine framework improve the Executable Rate, Executable Accuracy, and Overall Accuracy of the generated logic programs on the AR-LSAT dev set?
3. What is the incremental effect of the prototypers and refiners we implemented?

Data. We ran all experiments on the dev split of the AR-LSAT dataset (Zhong et al., 2021), which contains 231 logical reasoning questions from the Law School Admission Test between years 1991 and 2016. Given a problem, we formulate a prompt that includes instructions, the problem statement, the question, and multiple-choice answers. The task is to generate logic program and output the correct answer out of the five given options. An example prompt is shown in Appendix B.

Evaluation Method. The main evaluation method is accuracy: percentage of questions answered correctly. A more fine-grained way to evaluate the approach is to look at the percentage of executable logic programs generated by the LLM (Executable Rate) and the accuracy within executable programs (Executable Accuracy).

4.1 Experimental Details

In a given experiment, we run all 231 AR-LSAT problems from the dev set through each stage of our pipeline. After each stage (e.g. after running the prototypers, or after running the implicit constraints refiner), we run the solver using the generated logic programs and then compute the Executable Rate, Executable Accuracy, and Overall Accuracy by evaluating the outputs of the Z3 solver compared to the true answers for all 231 problems. We run our ProRef framework using GPT-3.5-Turbo. We are unable to run it with GPT-4 because it would cost over \$300.

4.2 Results

In Table 1, we benchmark our proposed framework against the Logic-LM framework with various LLMs. In Logic LM models and our framework, if the generated logic program is not executable, we

fall back to a backup strategy of directly prompting the LLM to answer the question. This is why we still see some accuracy with Mixtral even though it gets a 0% Executable Rate.

Model		Executable Rate	Executable Accuracy	Overall Accuracy
Logic-LM	GPT-4	32.61%	60%	34.78%
	GPT-3.5-Turbo	20.05%	31.82 %	25.53%
	Mixtral-8x7B-Instruct	2.60%	16.67	22.94%
	vicuna-13b-v1.5	3.48%	37.5%	24.24%
ProRef (Ours)	GPT-3.5-Turbo	32.47%	28%	23.81%

Table 1: Performance of different LLMs in Logic-LM, and the performance of GPT-3.5-Turbo in our framework. All experiments are run on the dev set of AR-LSAT, which contains 231 problems. Note that we don’t run our framework with GPT-4 because it would cost over \$300.

The first observation from these results is that our framework significantly increases the Executable Rate from 20.05% to 32.47%. More specifically, the Prototype stage (which applies k -majority voting and coder-reviewer reranking) increases the Executable Rate from 20.05% to 27.27%, while the Refine stage (which applies implicit constraints, consistency, and error feedback refiners) further increases the Executable Rate to 32.47%. However, the increase in Executable Rate does not lead to an increase in Overall Accuracy, and in fact leads to a decrease in Executable Accuracy and Overall Accuracy. We discuss these results further in the Analysis section.

	Executable Rate		Executable Accuracy		Overall Accuracy	
	w/ CR	w/o CR	w/ CR	w/o CR	w/ CR	w/o CR
k -Majority Voting	27.27%	27.27%	26.98%	26.98%	22.51%	22.51%
Implicit Constraints	27.27%	29.00%	26.98%	28.36%	22.51%	22.94%
Consistency	27.27%	29.00%	26.98%	28.36%	22.51%	22.94%
Error Feedback @1	32.03%	32.46%	27.03%	25.33%	23.38%	25.54%
Error Feedback @2	32.03%	32.47%	27.03%	25.33%	23.38%	22.51%
Error Feedback @3	32.04%	32.47%	27.03%	25.33%	23.38%	22.51%
Error Feedback @4	32.47%	32.47%	28.00%	25.33%	23.81%	22.51%

Table 2: Incremental improvement across different prototypers and refiners in our framework using GPT-3.5-Turbo. "w/ CR" and "w/o CR" refer to "with coder-reviewer" and "without coder-reviewer" respectively.

4.2.1 Coder-Reviewer Ablation

We also ran an ablation experiment where we turned off coder-reviewer (CR) reranking. Our results show that CR reranking does improve the Executable Accuracy and Overall Accuracy compared to if we simply pick a random logic program as our prototype logic program to be fed to the Refine stage. Note that the percentages are the same after k -majority voting, since we haven’t applied (or not applied) CR yet. The fact that the accuracy after different refiner rounds improves with CR compared to without it indicates that CR helps pick logic programs that are better suited to serve as prototypes. In other words, the prototype programs that CR picks are higher-quality candidates for refinement.

4.3 Error Patterns

In Table 3, we provide rigorous analysis of all the possible errors and exceptions encountered during the evaluation of LLMs within Logic-LM and our framework. The Mixtral-8x7B-Instruct model in Logic-LM struggle with higher incidence of Parsing Errors and Syntax Errors, which indicate difficulties in the syntactic interpretation and generation. In contrast, GPT-4 and GPT-3.5-turbo manifest superior adaptability across various error categories, underscoring their maturity and adeptness in generating the logic paradigms within the Logic-LM. SyntaxError, TypeError, NameError, Parsing Error, Over-Constrained and Under-Constrained are often prevalent and causes significant drop in the accuracy. By identifying these common error types, we can focus on refining them in our Refiner stage in which we can mitigate these error occurrences, thereby improving execution accuracy.

Model	Logic-LM				ProRef (Ours)
	GPT-4	GPT-3.5-Turbo	Mixtral-8x7B-Instruct	vicuna-13b-v1.5	GPT-3.5-Turbo
Underconstrained	33	12	3	5	5
Overconstrained	13	22	7	1	10
TypeError	29	41	5	22	13
SyntaxError	26	39	144	113	41
NameError	26	11	12	16	4
AttributeError	0	5	0	6	2
Parsing Error	14	13	52	22	24
Z3 Exception	11	40	2	38	29
No Output	3	4	0	0	28
Total	155	189	225	223	156

Table 3: Error analysis on the translated logic programs for various LLMs. Using our framework, we are able to almost match the error count of GPT-4 (156 vs. 155) using the 40-60x cheaper GPT-3.5-Turbo.

<pre> constraint: shared(1) == false && shared(2) == false python translation: pre_conds.append(shared(1) == false && shared(2) == false) </pre> <p style="text-align: center;">Correction</p> <pre> constraint: And(shared(1) == false, shared(2) == false) python translation: pre_conds.append(And(shared(1) == false, shared(2) == false)) </pre>
--

Figure 3: Example of a failure to handle simple syntax errors.

5 Analysis

From the Executable Rate and Accuracy numbers, we can observe that all LLMs struggle to solve AR-LSAT problems. A powerful LLM like GPT-4 is able to achieve a higher Executable Accuracy of 60%. On the other hand, Mixtral and Vicuna models mostly fail to generate correct logic programs, with an Executable Rate of 2.60% and 3.48% respectively.

Overall, our results indicate that it is not enough to optimize for an executable program. Optimizing for an executable program is different from optimizing for a *correct* program. Even though we are able to significantly increase the Executable Rate, the generated programs that became executable are still not semantically correct. Thus, we need to implement more semantics-guided refinement modules. Fortunately, our framework naturally supports the addition of such semantic refiners, which we leave for future work.

We also discovered a bug in the Logic-LM code that causes it to misclassify a solver execution as successful even though the solver returned True for all five choices. Just like an underconstrained or overconstrained logic program can lead to an empty output that would result in that logic program correctly being classified as "execution error", an underconstrained (resp. overconstrained) logic program might lead to *all* choices being judged as satisfiable (resp. unsatisfiable) by the solver.

A noteworthy observation is that, as expected, the deterministic, rule-based translation from raw logic program to Z3 Python code is rigid and lacks flexibility in detecting even simple syntax errors. Figure 3 shows an illustrative example that highlights the required flexibility of the solver for handling logical operations such as `And()` and `&&`. This presents an additional challenge for LLMs in improving their performance on AR-LSAT problems. Even though this project, like Logic-LM and LINC, is motivated by the faithfulness and guaranteed correctness of symbolic solvers, this example and similar failures make the argument for a different direction that still directly relies on LLMs and their flexibility but fine-tunes them to "function as symbolic solvers", as Feng et al. (2023) do in LoGiPT.

6 Conclusion

In this project, we developed a neuro-symbolic framework called Prototype-then-Refine (ProRef) that improves the logical reasoning ability of LLMs by enabling more robust generation of correct logic programs that are fed to symbolic solvers. We built on the foundation given by Logic-LM, which combines LLMs with symbolic solvers to solve complex logical reasoning questions. Focusing

on the AR-LSAT dataset, we started by evaluating Logic-LM, which served as our baseline, using different LLMs, including open-source LLMs which the Logic-LM authors did not evaluate with. We then benchmarked our method against Logic-LM. We found that our method significantly improves the *execution rate* of generated logic programs (from 20.05% to 32.47%), enabling GPT-3.5-Turbo to almost match the executable rate of GPT-4 (32.61%). However, we observed a decrease in the execution accuracy of the generated logic programs (and thus the overall accuracy) compared to Logic-LM.

We concluded that the goal of making the logic programs *executable* is not fully aligned with the goal of making them *semantically correct*, which motivates creating *semantics-guided refiners* that would aim to generate semantically correct logic programs. Such refiners can naturally be added to our ProRef framework, so they do not depart from the direction our proposed approach takes.

We also applied the Coder-Reviewer reranking idea and confirmed that it improves the execution accuracy of generated logic programs, especially after applying our proposed refiner modules. One idea immediately worth investigating next is to implement a *constrained decoder prototyper* that would ensure the generated logic programs adhere to the correct grammar rules to avoid syntax and type errors.

7 Key Information to include

- Mentor: Tianyi Zhang
- Team contributions: Hashem proposed the project idea and designed the Prototype-then-Refine framework. He evaluated the LINC method using GPT-4, and implemented the k -majority and coder-reviewer prototypers. Bassem benchmarked Logic-LM with open-source LLMs and implemented the implicit constraints, consistency, and error feedback refiners. He wrote the code for parsing and counting different error categories. Both contributed to error analysis, the design of the ProRef code, and the writing of the report.

References

- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.
- Jiazhan Feng, Ruochen Xu, Junheng Hao, Hiteshi Sharma, Yelong Shen, Dongyan Zhao, and Weizhu Chen. 2023. Language models can be logical solvers.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis.
- Olga Golovneva, Moya Chen, Spencer Poff, Martin Corredor, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. 2023. Roscoe: A suite of metrics for scoring step-by-step reasoning.
- Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. 2022. Folio: Natural language reasoning with first-order logic.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Hanmeng Liu, Jian Liu, Leyang Cui, Zhiyang Teng, Nan Duan, Ming Zhou, and Yue Zhang. 2023a. Logiqa 2.0—an improved dataset for logical reasoning in natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 31:2947–2962.
- Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. 2023b. Evaluating the logical reasoning ability of chatgpt and gpt-4.
- Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis.
- Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. 2023. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. *arXiv preprint arXiv:2310.15164*.
- Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*.
- Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution.
- Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. 2022. Entailer: Answering questions with faithful and truthful chains of reasoning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2078–2093, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

- Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. 2021. Proofwriter: Generating implications, proofs, and abductive statements over natural language.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models.
- Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.
- Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971.
- Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*.
- Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 89–105, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.
- Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR.
- Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2021. Ar-lsat: Investigating analytical reasoning of text. arxiv e-prints, page. *arXiv preprint arXiv:2104.06598*.
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-most prompting enables complex reasoning in large language models.

A Refiner: Input Prompt and Response

A.1 Implicit Constraint Refiner

```
Task Description: Given the initial program, update the following logic program with
additional implicit constraints to the given constraints.

Initial Program
Declarations:
people = EnumSort([Vladimir, Wendy])
meals = EnumSort([breakfast, lunch, dinner, snack])
foods = EnumSort([fish, hot_cakes, macaroni, omelet, poached_eggs])
eats = Function([people, meals] -> [foods])

Constraints:
ForAll([m:meals], eats(Vladimir, m) != eats(Wendy, m)) ::: At no meal does Vladimir eat
the same kind of food as Wendy
ForAll([p:people, f:foods], Count([m:meals], eats(p, m) == f) <= 1) ::: Neither of them
eats the same kind of food more than once during the day
ForAll([p:people], Or(eats(p, breakfast) == hot_cakes, eats(p, breakfast) == poached_eggs
, eats(p, breakfast) == omelet)) ::: For breakfast, each eats exactly one of the
following: hot cakes, poached eggs, or omelet
ForAll([p:people], Or(eats(p, lunch) == fish, eats(p, lunch) == hot_cakes, eats(p, lunch)
== macaroni, eats(p, lunch) == omelet)) ::: For lunch, each eats exactly one of the
following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, dinner) == fish, eats(p, dinner) == hot_cakes, eats(p,
dinner) == macaroni, eats(p, dinner) == omelet)) ::: For dinner, each eats exactly
one of the following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, snack) == fish, eats(p, snack) == omelet)) ::: For a snack,
each eats exactly one of the following: fish or omelet

Options:
Question ::: Vladimir must eat which one of the following foods?
is_valid(Exists([m:meals], eats(Vladimir, m) == fish)) ::: (A)
is_valid(Exists([m:meals], eats(Vladimir, m) == hot_cakes)) ::: (B)
is_valid(Exists([m:meals], eats(Vladimir, m) == macaroni)) ::: (C)
is_valid(Exists([m:meals], eats(Vladimir, m) == omelet)) ::: (D)
is_valid(Exists([m:meals], eats(Vladimir, m) == poached_eggs)) ::: (E)
-----
Updated Program
Declarations:
people = EnumSort([Vladimir, Wendy])
meals = EnumSort([breakfast, lunch, dinner, snack])
foods = EnumSort([fish, hot_cakes, macaroni, omelet, poached_eggs])
eats = Function([people, meals] -> [foods])

Constraints:
ForAll([m:meals], eats(Vladimir, m) != eats(Wendy, m)) ::: At no meal does Vladimir eat
the same kind of food as Wendy
ForAll([p:people, f:foods], Count([m:meals], eats(p, m) == f) <= 1) ::: Neither of them
eats the same kind of food more than once during the day
ForAll([p:people], Or(eats(p, breakfast) == hot_cakes, eats(p, breakfast) == poached_eggs
, eats(p, breakfast) == omelet)) ::: For breakfast, each eats exactly one of the
following: hot cakes, poached eggs, or omelet
ForAll([p:people], Or(eats(p, lunch) == fish, eats(p, lunch) == hot_cakes, eats(p, lunch)
== macaroni, eats(p, lunch) == omelet)) ::: For lunch, each eats exactly one of the
following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, dinner) == fish, eats(p, dinner) == hot_cakes, eats(p,
dinner) == macaroni, eats(p, dinner) == omelet)) ::: For dinner, each eats exactly
one of the following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, snack) == fish, eats(p, snack) == omelet)) ::: For a snack,
each eats exactly one of the following: fish or omelet
eats(Wendy, lunch) == omelet ::: Wendy eats an omelet for lunch

Options:
Question ::: Vladimir must eat which one of the following foods?
is_valid(Exists([m:meals], eats(Vladimir, m) == fish)) ::: (A)
is_valid(Exists([m:meals], eats(Vladimir, m) == hot_cakes)) ::: (B)
is_valid(Exists([m:meals], eats(Vladimir, m) == macaroni)) ::: (C)
is_valid(Exists([m:meals], eats(Vladimir, m) == omelet)) ::: (D)
is_valid(Exists([m:meals], eats(Vladimir, m) == poached_eggs)) ::: (E)
```

A.2 Consistency Refiner

Task Description: Given the initial program, check the consistency between declarations and constraints and update the following logic program to mitigate inconsistency.

Initial Program

Declarations:

```
people = EnumSort([Vladimir, Wendy])
meals = EnumSort([breakfast, lunch, dinner, snack])
foods = EnumSort([fish, hot_cakes, macaroni, omelet, poached_eggs])
eats = Function([people, meals] -> [foods])
```

Constraints:

```
ForAll([m:meals], eats(Vladimir, m) != eats(Wendy, m)) ::: At no meal does Vladimir eat
the same kind of food as Wendy
ForAll([p:folks, f:foods], Count([m:meals], eats(p, m) == f) <= 1) ::: Neither of them
eats the same kind of food more than once during the day
ForAll([p:people], Or(eats(p, breakfast) == hot_cakes, eats(p, breakfast) == poached_eggs
, eats(p, breakfast) == omelet)) ::: For breakfast, each eats exactly one of the
following: hot cakes, poached eggs, or omelet
ForAll([p:people], Or(eats(p, lunch) == fish, eats(p, lunch) == hot_cakes, eats(p, lunch)
== macaroni, eats(p, lunch) == omelet)) ::: For lunch, each eats exactly one of the
following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, dinner) == fish, eats(p, dinner) == hot_cakes, eats(p,
dinner) == macaroni, eats(p, dinner) == omelet)) ::: For dinner, each eats exactly
one of the following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, snack) == fish, eats(p, snack) == omelet)) ::: For a snack,
each eats exactly one of the following: fish or omelet
eats(Wendy, lunch) == omelet ::: Wendy eats an omelet for lunch
```

Options:

```
Question ::: Vladimir must eat which one of the following foods?
is_valid(Exists([m:meals], eats(Vladimir, m) == fish)) ::: (A)
is_valid(Exists([m:meals], eats(Vladimir, m) == hot_cakes)) ::: (B)
is_valid(Exists([m:meals], eats(Vladimir, m) == macaroni)) ::: (C)
is_valid(Exists([m:meals], eats(Vladimir, m) == omelet)) ::: (D)
is_valid(Exists([m:meals], eats(Vladimir, m) == poached_eggs)) ::: (E)
```

Updated Program

Declarations:

```
people = EnumSort([Vladimir, Wendy])
meals = EnumSort([breakfast, lunch, dinner, snack])
foods = EnumSort([fish, hot_cakes, macaroni, omelet, poached_eggs])
eats = Function([people, meals] -> [foods])
```

Constraints:

```
ForAll([m:meals], eats(Vladimir, m) != eats(Wendy, m)) ::: At no meal does Vladimir eat
the same kind of food as Wendy
ForAll([p:people, f:foods], Count([m:meals], eats(p, m) == f) <= 1) ::: Neither of them
eats the same kind of food more than once during the day
ForAll([p:people], Or(eats(p, breakfast) == hot_cakes, eats(p, breakfast) == poached_eggs
, eats(p, breakfast) == omelet)) ::: For breakfast, each eats exactly one of the
following: hot cakes, poached eggs, or omelet
ForAll([p:people], Or(eats(p, lunch) == fish, eats(p, lunch) == hot_cakes, eats(p, lunch)
== macaroni, eats(p, lunch) == omelet)) ::: For lunch, each eats exactly one of the
following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, dinner) == fish, eats(p, dinner) == hot_cakes, eats(p,
dinner) == macaroni, eats(p, dinner) == omelet)) ::: For dinner, each eats exactly
one of the following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, snack) == fish, eats(p, snack) == omelet)) ::: For a snack,
each eats exactly one of the following: fish or omelet
eats(Wendy, lunch) == omelet ::: Wendy eats an omelet for lunch
```

Options:

```
Question ::: Vladimir must eat which one of the following foods?
is_valid(Exists([m:meals], eats(Vladimir, m) == fish)) ::: (A)
is_valid(Exists([m:meals], eats(Vladimir, m) == hot_cakes)) ::: (B)
is_valid(Exists([m:meals], eats(Vladimir, m) == macaroni)) ::: (C)
is_valid(Exists([m:meals], eats(Vladimir, m) == omelet)) ::: (D)
is_valid(Exists([m:meals], eats(Vladimir, m) == poached_eggs)) ::: (E)
```

A.3 Error Feedback Refiner

Task Description: Given the initial program and the error message, debug the following logic program.

Initial Program

Declarations:

```
days = EnumSort([Monday, Tuesday, Wednesday, Thursday, Friday])
divisions = EnumSort([Operations, Production, Sales])
toured = Function([days] -> [divisions])
```

Constraints:

```
Count([d:days], toured(d) == Operations) >= 1 ::: Each division is toured at least once
Count([d:days], toured(d) == Production) >= 1 ::: Each division is toured at least once
Count([d:days], toured(d) == Sales) >= 1 ::: Each division is toured at least once
toured(Monday) != Operations ::: The Operations division is not toured on Monday
```

Error Message:

```
TypeError: '<' not supported between instances of 'DatatypeRef' and 'DatatypeRef'
```

Updated Program

Declarations:

```
days = IntSort([Monday, Tuesday, Wednesday, Thursday, Friday])
divisions = EnumSort([Operations, Production, Sales])
toured = Function([days] -> [divisions])
```

Constraints:

```
And(Monday == 1, Tuesday == 2, Wednesday == 3, Thursday == 4, Friday == 5)
Count([d:days], toured(d) == Operations) >= 1 ::: Each division is toured at least once
Count([d:days], toured(d) == Production) >= 1 ::: Each division is toured at least once
Count([d:days], toured(d) == Sales) >= 1 ::: Each division is toured at least once
toured(Monday) != Operations ::: The Operations division is not toured on Monday
```

B Prototype-Refine Logic Program Examples

In this section, we present sample prompt and the corresponding logic program outputs at each refinement stage within our framework.

Input Prompt

Task Description: Given a problem description and a question. The task is to formulate the problem as a logic program, consisting three parts: Declarations, Constraints, and Options.
Declarations: Declare the variables and functions.
Constraints: Write the constraints in the problem description as logic formulas.
Options: Write the options in the question as logic formulas.

Context: A closet contains exactly six hangers-1, 2, 3, 4, 5, and 6-hanging, in that order, from left to right. It also contains exactly six dresses-one gauze, one linen, one polyester, one rayon, one silk, and one wool-a different dress on each of the hangers, in an order satisfying the following conditions: The gauze dress is on a lower-numbered hanger than the polyester dress. The rayon dress is on hanger 1 or hanger 6. Either the wool dress or the silk dress is on hanger 3. The linen dress hangs immediately to the right of the silk dress.

Question: Which one of the following could be an accurate matching of the hangers to the fabrics of the dresses that hang on them?

Options:
 (A) 1: wool; 2: gauze; 3: silk; 4: linen; 5: polyester; 6: rayon
 (B) 1: rayon; 2: wool; 3: gauze; 4: silk; 5: linen; 6: polyester
 (C) 1: polyester; 2: gauze; 3: wool; 4: silk; 5: linen; 6: rayon
 (D) 1: linen; 2: silk; 3: wool; 4: gauze; 5: polyester; 6: rayon
 (E) 1: gauze; 2: rayon; 3: silk; 4: linen; 5: wool; 6: polyester

Reference Answer: (A)

Prototype Logic Program

Declarations:
 hangers = EnumSort([1, 2, 3, 4, 5, 6])
 dresses = EnumSort([gauze, linen, polyester, rayon, silk, wool])
 hangs = Function([hangers] -> [dresses])

Constraints:
 Distinct([h:hangers], hangs(h)) ::
 different dress on each of the hangers
 hangs(1) == rayon Or hangs(6) == rayon ::
 The rayon dress is on hanger 1 or hanger 6
 Or(hangs(3) == wool, hangs(3) == silk) ::
 Either the wool dress or the silk dress is on hanger 3
 hangs(2) == gauze And hangs(5) == polyester ::
 The gauze dress is on a lower-numbered hanger than the polyester dress
 hangs(4) == linen And hangs(4) == silk ::
 The linen dress hangs immediately to the right of the silk dress

Options:
 Question :: Which one of the following could be an accurate matching of the hangers to the fabrics of the dresses that hang on them?
 is_valid(hangs(1) == wool And hangs(2) == gauze And hangs(3) == silk And hangs(4) == linen And hangs(5) == polyester And hangs(6) == rayon) :: (A)
 is_valid(hangs(1) == rayon And hangs(2) == wool And hangs(3) == gauze And hangs(4) == silk And hangs(5) == linen And hangs(6) == polyester) :: (B)
 is_valid(hangs(1) == polyester And hangs(2) == gauze And hangs(3) == wool And hangs(4) == silk And hangs(5) == linen And hangs(6) == rayon) :: (C)

```
is_valid(hangs(1) == linen And hangs(2) ==
silk And hangs(3) == wool And hangs(4)
== gauze And hangs(5) == polyester
And hangs(6) == rayon) :: (D)
is_valid(hangs(1) == gauze And hangs(2) ==
rayon And hangs(3) == silk And hangs
(4) == linen And hangs(5) == wool And
hangs(6) == polyester) :: (E)
Solver Flag: execution error
Predicted Answer: None
```

Implicit Constraints Refiner

Declarations:
 hangers = EnumSort([1, 2, 3, 4, 5, 6])
 dresses = EnumSort([gauze, linen, polyester, rayon, silk, wool])
 hangs = Function([hangers] -> [dresses])

Constraints:
 Distinct([h:hangers], hangs(h)) ::
 different dress on each of the hangers
 hangs(1) == rayon Or hangs(6) == rayon ::
 The rayon dress is on hanger 1 or hanger 6
 Or(hangs(3) == wool, hangs(3) == silk) ::
 Either the wool dress or the silk dress is on hanger 3
 hangs(2) == gauze And hangs(5) == polyester ::
 The gauze dress is on a lower-numbered hanger than the polyester dress
 hangs(4) == linen And hangs(4) == silk ::
 The linen dress hangs immediately to the right of the silk dress
 hangs(1) == wool :: The wool dress is on hanger 1

Options:
 Question :: Which one of the following could be an accurate matching of the


```

hangers to the fabrics of the dresses
that hang on them?
is_valid(hangs(1) == wool And hangs(2) ==
gauze And hangs(3) == silk And hangs
(4) == linen And hangs(5) == polyester
And hangs(6) == rayon) ::: (A)
is_valid(hangs(1) == rayon And hangs(2) ==
wool And hangs(3) == gauze And hangs
(4) == silk And hangs(5) == linen And
hangs(6) == polyester) ::: (B)
is_valid(hangs(1) == polyester And hangs(2)
== gauze And hangs(3) == wool And
hangs(4) == silk And hangs(5) == linen
And hangs(6) == rayon) ::: (C)
is_valid(hangs(1) == linen And hangs(2) ==
silk And hangs(3) == wool And hangs(4)
== gauze And hangs(5) == polyester
And hangs(6) == rayon) ::: (D)
is_valid(hangs(1) == gauze And hangs(2) ==
rayon And hangs(3) == silk And hangs
(4) == linen And hangs(5) == wool And
hangs(6) == polyester) ::: (E)

```

Solver Flag: execution error
Predicted Answer: None

Consistency Refiner

```

Declarations:
hangers = EnumSort([1, 2, 3, 4, 5, 6])
dresses = EnumSort([gauze, linen, polyester
, rayon, silk, wool])
hangs = Function([hangers] -> [dresses])

```

```

Constraints:
Distinct([h:hangers], hangs(h)) :::
different dress on each of the hangers
hangs(1) == rayon Or hangs(6) == rayon :::
The rayon dress is on hanger 1 or
hanger 6
Or(hangs(3) == wool, hangs(3) == silk) :::
Either the wool dress or the silk
dress is on hanger 3
hangs(2) == gauze And hangs(5) == polyester
::: The gauze dress is on a lower-
numbered hanger than the polyester
dress
hangs(4) == linen And hangs(4) == silk :::
The linen dress hangs immediately to
the right of the silk dress
hangs(1) == wool ::: The wool dress is on
hanger 1

```

```

Options:
Question ::: Which one of the following
could be an accurate matching of the
hangers to the fabrics of the dresses
that hang on them?
is_valid(hangs(1) == wool And hangs(2) ==
gauze And hangs(3) == silk And hangs
(4) == linen And hangs(5) == polyester
And hangs(6) == rayon) ::: (A)
is_valid(hangs(1) == rayon And hangs(2) ==
wool And hangs(3) == gauze And hangs
(4) == silk And hangs(5) == linen And
hangs(6) == polyester) ::: (B)
is_valid(hangs(1) == polyester And hangs(2)
== gauze And hangs(3) == wool And
hangs(4) == silk And hangs(5) == linen
And hangs(6) == rayon) ::: (C)
is_valid(hangs(1) == linen And hangs(2) ==
silk And hangs(3) == wool And hangs(4)
== gauze And hangs(5) == polyester
And hangs(6) == rayon) ::: (D)
is_valid(hangs(1) == gauze And hangs(2) ==
rayon And hangs(3) == silk And hangs
(4) == linen And hangs(5) == wool And
hangs(6) == polyester) ::: (E)

```

Solver Flag: execution error
Predicted Answer: None

Error Feedback Refiner

```

Declarations:
hangers = EnumSort([1, 2, 3, 4, 5, 6])
dresses = EnumSort([gauze, linen, polyester
, rayon, silk, wool])
hangs = Function([hangers] -> [dresses])

```

```

Constraints:
Distinct([h:hangers], hangs(h)) :::
different dress on each of the hangers
Or(hangs(1) == rayon, hangs(6) == rayon) ::: The
rayon dress is on hanger 1 or hanger 6
Or(hangs(3) == wool, hangs(3) == silk) :::
Either the wool dress or the silk
dress is on hanger 3
And(hangs(2) == gauze, hangs(5) == polyester) :::
The gauze dress is on a lower-numbered
hanger than the polyester dress
And(hangs(4) == linen, hangs(4) == silk)
::: The linen dress hangs immediately
to the right of the silk dress
hangs(1) == wool ::: The wool dress is on
hanger 1

```

```

Options:
is_valid(hangs(1) == wool) ::: (A)
is_valid(hangs(1) == rayon) ::: (B)
is_valid(hangs(1) == polyester) ::: (C)
is_valid(hangs(1) == linen) ::: (D)
is_valid(hangs(1) == gauze) ::: (E)

```

Solver Flag: "Success"
Predicted Answer: (A)
Reference Answer: (A)

C Coder-Reviewer: Input Prompt

In the following code block, we provide an illustrative example showcasing the input prompt of the coder-reviewer during the process of backward translation from logic programs to natural language.

```
Task Description: You are given a logic program consisting of three parts: Declarations, Constraints, and Options. This logic program is a formalization of an original natural language problem. Your task is to output the original natural language problem.

Logic Program
Declarations:
people = EnumSort([Vladimir, Wendy])
meals = EnumSort([breakfast, lunch, dinner, snack])
foods = EnumSort([fish, hot_cakes, macaroni, omelet, poached_eggs])
eats = Function([people, meals] -> [foods])

Constraints:
ForAll([m:meals], eats(Vladimir, m) != eats(Wendy, m)) ::: At no meal does Vladimir eat the same kind of food as Wendy
ForAll([p:people, f:foods], Count([m:meals], eats(p, m) == f) <= 1) ::: Neither of them eats the same kind of food more than once during the day
ForAll([p:people], Or(eats(p, breakfast) == hot_cakes, eats(p, breakfast) == poached_eggs, eats(p, breakfast) == omelet)) ::: For breakfast, each eats exactly one of the following: hot cakes, poached eggs, or omelet
ForAll([p:people], Or(eats(p, lunch) == fish, eats(p, lunch) == hot_cakes, eats(p, lunch) == macaroni, eats(p, lunch) == omelet)) ::: For lunch, each eats exactly one of the following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, dinner) == fish, eats(p, dinner) == hot_cakes, eats(p, dinner) == macaroni, eats(p, dinner) == omelet)) ::: For dinner, each eats exactly one of the following: fish, hot cakes, macaroni, or omelet
ForAll([p:people], Or(eats(p, snack) == fish, eats(p, snack) == omelet)) ::: For a snack, each eats exactly one of the following: fish or omelet
eats(Wendy, lunch) == omelet ::: Wendy eats an omelet for lunch

Options:
Question ::: Vladimir must eat which one of the following foods?
is_valid(Exists([m:meals], eats(Vladimir, m) == fish)) ::: (A)
is_valid(Exists([m:meals], eats(Vladimir, m) == hot_cakes)) ::: (B)
is_valid(Exists([m:meals], eats(Vladimir, m) == macaroni)) ::: (C)
is_valid(Exists([m:meals], eats(Vladimir, m) == omelet)) ::: (D)
is_valid(Exists([m:meals], eats(Vladimir, m) == poached_eggs)) ::: (E)
-----
Natural Language Problem
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:
(A) fish
(B) hot cakes
(C) macaroni
(D) omelet
(E) poached eggs
```

D Prompt and Output Logic Program Examples

In this section, we present sample prompts and logic program outputs generated by the Language Models (LLMs) in Logic-LM. Specifically focusing on failure cases, the displayed LLM outputs demonstrate instances that resulted in either execution or parsing errors.

D.1 Execution Error Examples

Input Prompt

Task Description: Given a problem description and a question. The task is to formulate the problem as a logic program, consisting three parts: Declarations, Constraints, and Options.
Declarations: Declare the variables and functions.
Constraints: Write the constraints in the problem description as logic formulas.
Options: Write the options in the question as logic formulas.

Context: Four boys-Fred, Juan, Marc, and Paul-and three girls-Nita, Rachel, and Trisha-will be assigned to a row of five adjacent lockers, numbered consecutively 1 through 5, arranged along a straight wall. The following conditions govern the assignment of lockers to the seven children: Each locker must be assigned to either one or two children, and each child must be assigned to exactly one locker. Each shared locker must be assigned to one girl and one boy. Juan must share a locker, but Rachel cannot share a locker. Nita's locker cannot be adjacent to Trisha's locker. Fred must be assigned to locker 3. (... more context here ...)

Question: If Trisha is assigned to locker 3 and Marc alone is assigned to locker 1, then which one of the following must be true?

Options:
(A) Juan is assigned to locker 4.
(B) Juan is assigned to locker 5.
(C) Paul is assigned to locker 2.
(D) Rachel is assigned to locker 2.
(E) Rachel is assigned to locker 5.

Reference Answer: (B)

LLM Logic Programs

Logic-LM (GPT-4)

```
Declarations:
children = EnumSort([Fred, Juan, Marc, Paul, Nita, Rachel, Trisha])
lockers = EnumSort([1, 2, 3, 4, 5])
assigned = Function([children] -> [lockers])
shared = Function([lockers] -> [bool])

Constraints:
ForAll([c:children], Exists([l:lockers], assigned(c) == 1)) :: each child must be assigned to exactly one locker
ForAll([l:lockers], Or(Exists([c:children], assigned(c) == 1), Exists([c1:children, c2:children], And(c1 != c2, assigned(c1) == 1, assigned(c2) == 1, shared(1)))) :: Each locker must be assigned to either one or two children
ForAll([l:lockers], Implies(shared(1), Exists([b:children, g:children], And(assigned(b) == 1, assigned(g) == 1, Or(b == Fred, b == Juan, b == Marc, b == Paul), Or(g == Nita, g == Rachel, g == Trisha)))) :: Each shared locker must be assigned to one girl and one boy
Exists([l:lockers], And(assigned(Juan) == 1, shared(1))) :: Juan must share a locker
ForAll([l:lockers], Implies(assigned(Rachel) == 1, Not(shared(1)))) :: Rachel cannot share a locker

ForAll([l:lockers], Implies(Or(assigned(Nita) == 1, assigned(Trisha) == 1), Not(Or(assigned(Nita) == 1+1, assigned(Nita) == 1-1, assigned(Trisha) == 1+1, assigned(Trisha) == 1-1)))) :: Nita's locker cannot be adjacent to Trisha's locker
assigned(Fred) == 3 :: Fred must be assigned to locker 3
And(Exists([b:children], And(assigned(b) == 1, Or(b == Fred, b == Juan, b == Marc, b == Paul), Not(shared(1))))), Exists([b:children], And(assigned(b) == 2, Or(b == Fred, b == Juan, b == Marc, b == Paul), Not(shared(2)))) :: if lockers 1 and 2 are each assigned to one boy and are not shared lockers

Options:
Question :: If lockers 1 and 2 are each assigned to one boy and are not shared lockers, then locker 4 must be assigned to
is_sat(assigned(Juan) == 4) :: (A)
is_sat(assigned(Paul) == 4) :: (B)
is_sat(assigned(Rachel) == 4) :: (C)
is_sat(And(assigned(Juan) == 4, assigned(Nita) == 4, shared(4))) :: (D)
is_sat(And(assigned(Marc) == 4, assigned(Trisha) == 4, shared(4))) :: (E)

solver flag: execution error
predicted answer: (C)
```

Logic-LM (Mixtral-8x7B-Instruct)

```
Declarations:
children = EnumSort([Fred, Juan, Marc, Paul
, Nita, Rachel, Trisha])
lockers = EnumSort([1, 2, 3, 4, 5])
assigned\_to = Function([children, lockers
] -> [bool])

Constraints:
ForAll([c:children], Exists([l:lockers],
assigned\_to(c, l))) ::: Each child
must be assigned to exactly one locker
ForAll([l:lockers], (Count([c:children],
assigned\_to(c, l)) == 1) or (Count([
c:children], assigned\_to(c, l)) ==
2)) ::: Each locker must be assigned
to either one or two children
ForAll([l:lockers], Implies(assigned\_to(
Rachel, l), Count([c:children],
assigned\_to(c, l)) == 1)) ::: Rachel
cannot share a locker
ForAll([c:children, l:lockers], Implies(
assigned\_to(c, l), assigned\_to(c,
l) == assigned\_to(c, l + 1))) ::: No
child can be assigned to adjacent
lockers
assigned\_to(Fred, 3) ::: Fred must be
assigned to locker 3
assigned\_to(Marc, 1) ::: Marc alone is
assigned to locker 1
assigned\_to(Trisha, 3) ::: Trisha is
assigned to locker 3

Options:
Question ::: If Trisha is assigned to
locker 3 and Marc alone is assigned to
locker 1, then which one of the
following must be true?
is\_sat(assigned\_to(Juan, 4)) ::: (A)
is\_sat(assigned\_to(Juan, 5)) ::: (B)
is\_sat(assigned\_to(Paul, 2)) ::: (C)
is\_sat(assigned\_to(Rachel, 2)) ::: (D)
is\_sat(assigned\_to(Rachel, 5)) ::: (E)

solver flag: execution error
predicted answer: (C)
```

Logic-LM (vicuna-13b-v1.5)

```
Declarations:
people = EnumSort([Fred, Juan, Marc, Nita,
Paul, Rachel, Trisha])
lockers = EnumSort([1, 2, 3, 4, 5])
assigned = Function([people, lockers] -> [
bool])
shared = Function([people, people] -> [bool
])

Constraints:
ForAll([p:people], And(1 <= assigned(p,
lockers), assigned(p, lockers) <= 5))
::: Each locker must be assigned to
either one or two children, and each
child must be assigned to exactly one
locker
ForAll([p:people, q:people], And(assigned(p
, lockers), assigned(q, lockers), p !=
q, shared(p, q))) ::: Each shared
locker must be assigned to one girl
and one boy
And(assigned(Juan, lockers), Not(assigned(
Rachel, lockers))) ::: Juan must share
a locker, but Rachel cannot share a
locker
And(assigned(Nita, lockers), Not(assigned(
Trisha, lockers - 1))) ::: Nita's
locker cannot be adjacent to Trisha's
locker
assigned(Fred, 3) ::: Fred must be assigned
to locker 3

Options:
Question ::: If Trisha is assigned to
locker 3 and Marc alone is assigned to
locker 1, then which one of the
following must be true?
is\_sat(And(assigned(Trisha, 3), Not(
assigned(Marc, 1)), Not(assigned(Juan,
4)), Not(assigned(Juan, 5)), Not(
assigned(Paul, 2)), Not(assigned(
Rachel, 2)), Not(assigned(Rachel, 5)))
) ::: (A) Juan is assigned to locker
4.
is\_sat(And(assigned(Trisha, 3), Not(
assigned(Marc, 1)), Not(assigned(Juan,
4)), Not(assigned(Juan, 5)), Not(
assigned(Paul, 2)), Not(assigned(
Rachel, 2)), Not(assigned(Rachel, 5)),
assigned(Juan, 5))) ::: (B) Juan is
assigned to locker 5.
is\_sat(And(assigned(Trisha, 3), Not(
assigned(Marc, 1)), Not(assigned(Juan,
4)), Not(assigned(Juan, 5)), Not(
assigned(Paul, 2)), Not(assigned(
Rachel, 2)), Not(assigned(Rachel, 5)),
assigned(Juan, 4))) ::: (C) Paul is
assigned to locker 2.
is\_sat(And(assigned(Trisha, 3), Not(
assigned(Marc, 1)), Not(assigned(Juan,
4)), Not(assigned(Juan, 5)), Not(
assigned(Paul, 2)), Not(assigned(
Rachel, 2)), Not(assigned(Rachel, 5)),
assigned(Rachel, 2))) ::: (D) Rachel
is assigned to locker 2.
is\_sat(And(assigned(Trisha, 3), Not(
assigned(Marc, 1)), Not(assigned(Juan,
4)), Not(assigned(Juan, 5)), Not(
assigned(Paul, 2)), Not(assigned(
Rachel, 2)), Not(assigned(Rachel, 5)),
assigned(Rachel, 5))) ::: (E) Rachel
is assigned to locker 5.

solver flag: execution error
predicted answer: (C)
```

D.2 Parsing Error Examples

Input Prompt

Task Description: Given a problem description and a question. The task is to formulate the problem as a logic program, consisting three parts: Declarations, Constraints, and Options.
Declarations: Declare the variables and functions.
Constraints: Write the constraints in the problem description as logic formulas.
Options: Write the options in the question as logic formulas.

Context: Four boys-Fred, Juan, Marc, and Paul-and three girls-Nita, Rachel, and Trisha-will be assigned to a row of five adjacent lockers, numbered consecutively 1 through 5, arranged along a straight wall. The following conditions govern the assignment of lockers to the seven children: Each locker must be assigned to either one or two children, and each child must be assigned to exactly one locker. Each shared locker must be assigned to one girl and one boy. Juan must share a locker, but Rachel cannot share a locker. Nita's locker cannot be adjacent to Trisha's locker. Fred must be assigned to locker 3.

Question: Which one of the following is a complete and accurate list of the children who must be among those assigned to shared lockers?

Options:
(A) Fred, Juan
(B) Juan, Paul
(C) Juan, Marc, Paul
(D) Juan, Marc, Trisha
(E) Juan, Nita, Trisha

Reference Answer: (E)

LLM Logic Programs

Logic-LM (GPT-4)

Declarations:
children = EnumSort([Fred, Juan, Marc, Paul, Nita, Rachel, Trisha])
lockers = EnumSort([1, 2, 3, 4, 5])
assigned = Function([children] -> [lockers])

Constraints:
ForAll([l:lockers], Or(Count([c:children], assigned(c) == 1) == 1, And(Count([c:children], assigned(c) == 1) == 2, Exists([b:boys, g:girls], And(assigned(b) == 1, assigned(g) == 1)))))) :: Each locker must be assigned to either one or two children, and each child must be assigned to exactly one locker
Exists([c:children], assigned(Juan) == assigned(c)) :: Juan must share a locker
ForAll([c:children], assigned(Rachel) != assigned(c)) :: Rachel cannot share a locker
ForAll([l:lockers], Implies(assigned(Nita) == l, And(assigned(Trisha) != l+1, assigned(Trisha) != l-1))) :: Nita's locker cannot be adjacent to Trisha's locker
assigned(Fred) == 3 :: Fred must be assigned to locker 3

Options:
Question :: Which one of the following is a complete and accurate list of the children who must be among those assigned to shared lockers
is_accurate_list([Exists([c:children], assigned(Fred) == assigned(c)), Exists([c:children], assigned(Juan) == assigned(c))]) :: (A)
is_accurate_list([Exists([c:children], assigned(Juan) == assigned(c)), Exists([c:children], assigned(Paul) == assigned(c))]) :: (B)
is_accurate_list([Exists([c:children], assigned(Juan) == assigned(c)), Exists([c:children], assigned(Marc) == assigned(c)), Exists([c:children], assigned(Paul) == assigned(c))]) :: (C)
is_accurate_list([Exists([c:children], assigned(Juan) == assigned(c)), Exists([c:children], assigned(Marc) == assigned(c)), Exists([c:children], assigned(Trisha) == assigned(c))]) :: (D)
is_accurate_list([Exists([c:children], assigned(Juan) == assigned(c)), Exists([c:children], assigned(Nita) == assigned(c)), Exists([c:children], assigned(Trisha) == assigned(c))]) :: (E)

solver flag: parsing error
predicted answer: (B)

E Extra Results

	Model	Executable Rate	Executable Accuracy	Overall Accuracy
Logic-LM	GPT-4	32.61%	60%	31.74%
	GPT-3.5-turbo	20.05%	31.82 %	23.81 %
	Mixtral-8x7B-Instruct	0%	2.16%	24.24%
	vicuna-13b-v1.5	3.48%	37.5%	19.48%
LINC	GPT-4	70%	42.86%	30%
Our	GPT-3.5-turbo	32.47%	28%	22.51%

Table 4: Performance of different LLMs against our framework using the Random backup strategy (pick a random answer out of the five choices) instead of the LLM prompting backup strategy.

Table 5 presents the effectiveness of revising the generated logical programs iteratively using the error messages from the symbolic solver as feedback. We could not notice any improvement for the error feedback refinement while using Vicuna model in Logic-LM. Although similar results were observed in (Pan et al., 2023) on accuracy, the executable rate is improved with number of rounds by 5% on average. So this behavior can be justified due to the limited LLM’s expressiveness power.

Refinement Rounds	0	1	2	3	4
Executable Accuracy	37.5%	37.5%	37.5%	37.5%	37.5%
Executable Rate	3.48%	3.48%	3.48%	3.48%	3.48%

Table 5: Performance of Logic-LM (Vicuna) with error feedback refinement.