

Ultra Strong Machine Learning: Teaching Humans Active Learning Strategies via Automated AI Explanations

Lun Ai¹, Johannes Langer², Ute Schmid², Stephen Muggleton¹

¹Department of Computing, Imperial College London, UK

²Faculty Information Systems and Applied Computer Science, University of Bamberg, Germany
 {lun.ai15,s.muggleton}@imperial.ac.uk, {johannes.langer, ute.schmid}@uni-bamberg.de

Abstract

Ultra Strong Machine Learning (USML) refers to symbolic learning systems that not only improve their own performance but can also teach their acquired knowledge to quantifiably improve human performance. In this work, we present LENS (Logic Programming Explanation via Neural Summarisation), a neuro-symbolic method that combines symbolic program synthesis with large language models (LLMs) to automate the explanation of machine-learned logic programs in natural language. LENS addresses a key limitation of prior USML approaches by replacing hand-crafted explanation templates with scalable automated generation. Through systematic evaluation using multiple LLM judges and human validation, we demonstrate that LENS generates superior explanations compared to direct LLM prompting and hand-crafted templates. To investigate whether LENS can teach transferable active learning strategies, we carried out a human learning experiment across three related domains. Our results show no significant human performance improvements, suggesting that comprehensive LLM responses may overwhelm users for simpler problems rather than providing learning support. Our work provides a solid foundation for building effective USML systems to support human learning. The source code is available on: <https://github.com/lun-ai/LENS.git>.

1 Introduction

Active learning (Cohn, Atlas, and Ladner 1994) improves model performance by intelligently selecting the most informative training examples rather than treating all data equally. This strategy parallels human learning, where people naturally seek information that reduces uncertainty (Gopnik, Meltzoff, and Kuhl 1999; Cook, Goodman, and Schulz 2011), and scientific experimentation, where hypotheses are tested to gain knowledge (Popper 2005). These similarities suggest that active learning is a general strategy employed by both artificial and human learners to optimise knowledge acquisition.

The observation that humans can engage in active forms of learning across a range of contexts raises the question of whether AI can teach humans generalisable problem-solving strategies. While transfer learning is central to both human cognition and artificial intelligence (Anderson, Kushmerick,

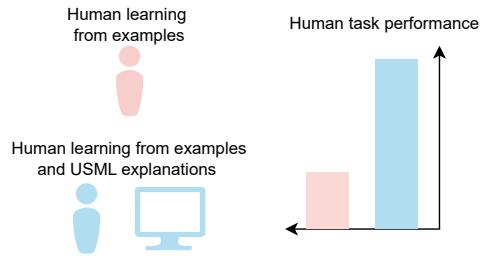


Figure 1: Ultra strong machine learning (USML) can quantifiably enhance human task performance.

and Lebiere 1993; Wiese, Konderding, and Schmid 2008; Gentner and Forbus 2011), far less is known about whether people can be supported by AI in acquiring active learning strategies that are themselves transferable. In this work, we answer the question: *Can an AI system teach humans active learning strategies that can generalise across domains?*

Ultra Strong Machine Learning (USML) (Michie 1988) distinguishes learning systems by their ability to help humans acquire applicable knowledge. *Weak* machine learning improves performance from training data without an explanation capability. *Strong* machine learning additionally outputs symbolic knowledge for human interpretation. USML builds on strong machine learning and demands that systems teach humans to achieve superhuman performance. A system achieves USML (Figure 1) when its explanations quantifiably improve human out-of-sample predictive performance compared to human learning without explanations (Schmid et al. 2017). This capability thus enables the development of dynamic learning environments and intelligent tutoring systems (Graesser, Conley, and Olney 2012).

Past work on USML demonstrated human comprehension was enhanced in learning relation (Muggleton et al. 2018), game playing (Ai et al. 2021), and algorithm discovery (Ai et al. 2023) using Inductive Logic Programming (ILP) systems. ILP (Muggleton 1995) is a machine learning approach that aims to learn rule-like logic programs from data and domain knowledge. The learned logical programs can facilitate comprehension since they explicitise reasoning steps and the concepts involved (Michalski 1983; Anderson, Kushmerick, and Lebiere 1993; Carbonell 1985). However, logic

programs were explained based on hand-crafted natural language templates of program components (Ai et al. 2021, 2023). This approach relies on human experts to provide these sensible templates, thereby limiting the applicability of the USML approach for more complex ILP problems.

We introduce LENS (Logic Programming Explanation via Neural Summarisation), as a general neuro-symbolic method for learning and explaining logic programs (Figure 2). LENS first learns logic programs with an ILP system, and then generates natural language explanations using large language models (LLMs). These explanations are scored by LLM judges, and the top-ranked explanations are manually reviewed to select those used in human trials.

We employ LENS to learn and explain an active learning strategy for selecting the most informative tests in fault diagnosis of electrical circuits. This domain represents a typical diagnostic task in troubleshooting mechanical devices, modelling physical or biological systems (De Kleer and Williams 1987). This diagnostic task requires identifying hypothesised fault locations and gathering evidence to refine hypotheses by performing tests. In this setting, locations of a single fault are hypothesised, and a circuit test should be performed to reduce plausible hypothesis candidates. Our ILP-learned active learning strategy computes an optimal test which maximises the number of hypotheses we can eliminate, following a divide-and-conquer approach.

To evaluate whether LENS explanations enable USML, we conducted a human study spanning three related domains, with LENS explanations provided exclusively during training in the first domain. Our results reveal no human performance improvements from explanations across domains, indicating that LLMs responses may overwhelm users for simpler problems. These findings provide critical insights for USML system design, establishing when automated explanations may fail to enhance human learning. Our LENS framework and empirical insights lay a solid foundation that enables systematic exploration of effective human-AI knowledge transfer.

2 Related Work

Program Synthesis. LLMs have emerged as promising tools for synthesising programs using natural language (Chen et al. 2021; Li et al. 2022; Zheng et al. 2023; Le et al. 2022; Fried et al. 2023; Hui et al. 2024; Lozhkov et al. 2024). Synthesising out-of-distribution programs given input-output examples, however, remains difficult for LLMs (Li and Ellis 2024). Traditional program synthesis searches for programs consistent with the input-output examples in the space of learnable programs (Alur et al. 2013; Gulwani et al. 2015). Recent work in Inductive Logic Programming (ILP) can search this space to learn programs that can use other programs as input (Cropper, Morel, and Muggleton 2020; Purgał, Černá, and Kaliszyk 2022; Hocquette, Dumančić, and Cropper 2024). In our work, we take the best from both worlds, where ILP is used to learn programs from examples, and LLMs can interpret programs to provide natural language explanations.

Explainable AI. A great body of work in Explainable AI (Gunning et al. 2019; Barredo Arrieta et al. 2020) aims to make learning systems more intelligible to humans by providing explanations. Often, little objective evidence (Miller 2019; Atzmueller et al. 2024) can support the benefits of explanations. It is important to take into account how well users understand, for example, from self-reported evaluation (Gunning et al. 2019; Minh et al. 2022), but this might be biased by how well the participants understand the domain (Schmid and Wrede 2022). In contrast, our human experiments focus on the objective evaluation of human understanding by comparing human predictive performance.

3 Ultra Strong Machine Learning

This section first introduces the problem setting. Then, we propose LENS (Logic Programming Explanation via Neural Summarisation) and describe how it produces explanations.

3.1 Problem Setting

Our problem is to decide if a learning system can improve human comprehension by explaining what it has learned. We consider the E_{ex} (Ai et al. 2021, 2023) framework. E_{ex} involves a function D , a human population H and a set of input-output pairs E of D . $H_1, H_2 \subset H$ are randomly sampled groups of similar and sufficient sizes, where $H_1 \cap H_2 = \emptyset$. The unaided human comprehension $\tau_h(D, H_1, E)$ is the mean performance of the human group H_1 after a brief study of E , and without further sight, can produce outputs of D from new random samples of its domain.

This contrasts with the machine-explained human comprehension, $\tau_{ex}(D, H_2, M(E))$. This is the mean performance of the human group H_2 after a brief study of the output $M(E)$ from a machine M , and without further sight, can produce outputs of D from new random samples of its domain. The effect of learning from machine explanations in population H is $E_{ex}(D, H, M(E))$ by comparing the two measurements of comprehension:

$$E_{ex}(D, H, M(E)) = \tau_{ex}(D, H_2, M(E)) - \tau_h(D, H_1, E)$$

In the context of D , $E_{ex}(D, H, M(E)) > 0$ is a necessary condition to conclude the machine M is USML with appropriate statistical tests. This problem has broad applications, as we do not limit the output format of M . For example, past work demonstrated USML when the explanations were logic programs (Muggleton et al. 2018), natural texts and images (Ai et al. 2021, 2023). We do not restrict E_{ex} to predictive accuracy since this allows us to compare continuous task performance scores.

Challenges. Past work (Ai et al. 2021, 2023) explained logic programs from ILP systems based on hand-crafted explanation templates. While this approach leverages expert knowledge, it requires manual tuning and risks creating explanations that only make sense to the experts (Atzmueller et al. 2024). In addition, crafted templates could result in less natural explanations that may hinder human comprehension. While we retain expert evaluation for ethics concerns, automated pipelines could support users in creating and selecting accessible explanations for more involved logic programs.

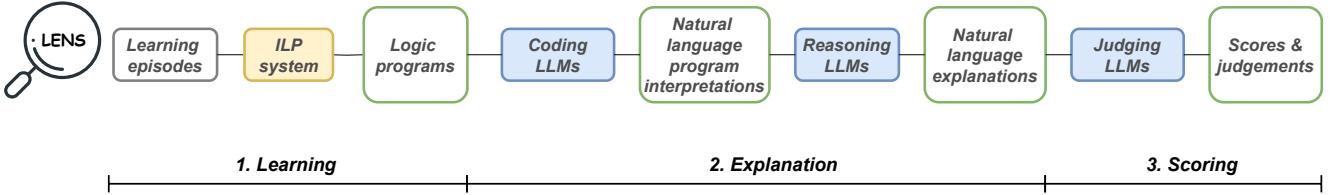


Figure 2: Logic Programming Explanation via Neural Summarisation (LENS) method.

3.2 Neuro-Symbolic Architecture

To address these challenges, we propose Logic Programming Explanation via Neural Summarisation (LENS) (Figure 2). LENS leverages ILP to learn Prolog programs and employs LLMs in-context learning to generate explanations through three pipelines: *learning*, *explanation* and *scoring*. A set of ILP learning episodes is used as input, and LENS produces a set of natural language explanations. Additional human intervention is only required during scoring after LENS has been configured with the input episodes. This avoids the need for expert-crafted explanations in comparison with the previous USML work.

In the *learning* pipeline, LENS learns logic programs from ILP learning episodes. The typical ILP learning problem involves a set of examples $E = E^+ \cup E^-$, background knowledge BK and relevant parameters for the ILP system. The objective is to find a logic program h such that $\forall e^+ \in E^+ h \cup BK \models e^+$ and $\forall e^- \in E^- h \cup BK \not\models e^-$ (“ \models ” denotes entailment). LENS uses an ILP system to solve this learning problem. The learned programs are combined into a “library” for reference by subsequent processes.

In the *explanation* pipeline, we leverage two modalities to produce a consensus explanation. We first used multiple coding LLMs to interpret the programs individually. Then, a reasoning LLM summarises code interpretations into a consensus explanation. This is inspired by the “debate” strategy (Du et al. 2024; Subramaniam et al. 2024), which was shown to improve performance in multiple factual reasoning datasets. Each coding LLM is responsible for translating the program into plain English sentences to describe what each part aims to achieve. The reasoning LLMs output natural language explanations that identify the relevant predicates and how these predicates can be applied. We use reasoning LLMs to perform one round of summarisation on the code interpretations, which aims to keep the conversation concise with lower token cost.

In the *scoring* pipeline, we use LLMs to evaluate summaries with a human-crafted reference answer based on prompts in (Zheng et al. 2023). While LLM judges do not replace human evaluations, we utilise LLM judges as supporting evidence and to score explanations objectively. We adapted their prompt templates for judging single answers with a human-crafted reference answer, and examined scores from multiple judges. We believe this mitigates the potential biases of LLMs to favour certain response ordering (position bias) or responses from certain models (self-enhancement bias) (Zheng et al. 2023).

Advantages. By leveraging ILP, LENS can facilitate a wide range of problems, such as game playing (Cropper, Evans, and Law 2020), program synthesis (Cropper et al. 2021), and scientific discovery (King et al. 2004). With LLMs, LENS takes advantage of the “wisdom of the crowd” by building a consensus from distinct modes of reasoning (code and natural language). In a, LENS uses instruction-tuned coding LLMs, which allow us to interpret under-represented programming languages such as Prolog, since coding LLMs have wider programming language coverage (Hui et al. 2024; Lozhkov et al. 2024).

3.3 ILP Learning

We refer readers to the textbook (Nienhuys-Cheng and Wolf 1997) on logic programming and ILP for background. Predicate invention (Muggleton and Buntine 1988) is the process of automatically learning new concepts in addition to the background knowledge. This technique allows ILP systems (Law, Russo, and Broda 2014; Muggleton, Lin, and Tamaddoni-Nezhad 2015; Evans and Grefenstette 2018; Dai and Muggleton 2021; Cropper and Morel 2021; Glanois et al. 2022) to create novel and usable concepts. While predicate invention results in more compact programs, a strong language bias is often required to limit the space of learnable programs (Cropper et al. 2021).

As an ILP approach, Learning from Failures (LFF) (Cropper and Morel 2021) has a minimal language bias requirement. We use an LFF ILP system Hopper (Purgał, Černá, and Kaliszyk 2022) to learn second-order programs, which can take other programs as input. Learning with second-order programs can reduce the learned program size, enhance learning performance (Cropper, Morel, and Muggleton 2020) and integrate meta-level and domain-level knowledge (Gabaldon, Langley, and Meadows 2020).

Base Domain. Our base domain involves identifying the most informative test for diagnosing faults in electrical circuits (Figure 3). All circuits are inspired by an existing network from a biological system (see Appendix A.1). Each circuit is powered by batteries and supplies energy to lightbulbs through a set of AND gates. If a gate is faulty, it does not transfer energy, and the lightbulbs connected to it would turn off. To find the faulty gate, a power source can be connected to a point of interest, which potentially makes the lightbulb turn on again, therefore giving information about the location of the fault.

Active Learning. We employ the Hopper system to learn a strategy that resembles a Bayesian active learning approach

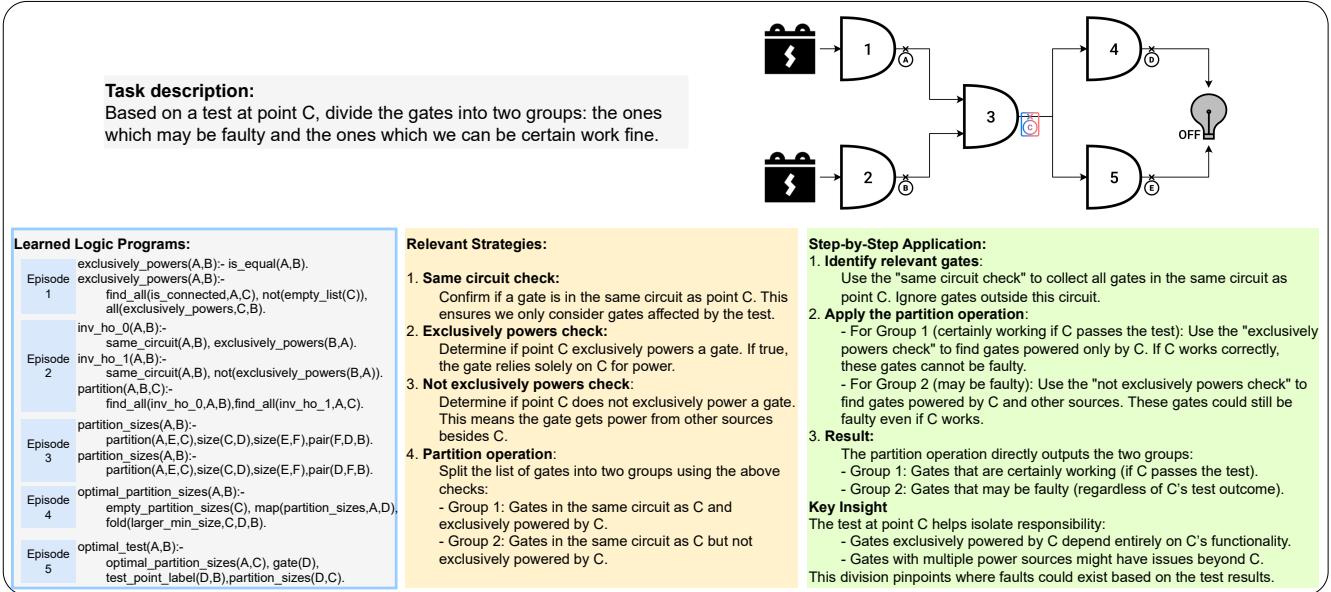


Figure 3: The left block shows ILP-learned programs, where each episode is learned from a single circuit example. The middle block summarises relevant programs identified for the task. The right block is an action strategy based on relevant programs.

(Hocquette and Muggleton 2018). The likelihood and the MAP estimate of a hypothesis h given a background knowledge BK and a label x are defined as:

$$p(x|h, BK) = \begin{cases} 1 & \text{if } h \cup BK \models x \\ 0 & \text{else} \end{cases}$$

$$h_{MAP} = \arg \max_{h \in \mathcal{H}} p(x|h, BK)$$

After querying s labels, the minimum reduction ratio of the hypothesis space V_s is the size of the minority partition in the hypothesis space for the next label:

$$V_s^+ = \{h \in V_s \mid h \cup BK \models x_{s+1}\}$$

$$V_s^- = \{h \in V_s \mid h \cup BK \not\models x_{s+1}\}$$

$$p(x_{s+1}, V_s) = \frac{\min(|V_s^+|, |V_s^-|)}{|V_s|}$$

The expected information gain following the knowledge of the $s + 1$ label becomes the binary entropy H_b of how much we reduce the hypothesis space V_s :

$$\mathbf{E}_{h \sim D_{\mathcal{H}}} [I(x_{s+1}, V_s, h)] = H_b(p(x_{s+1}, V_s)) \quad (1)$$

In our active learning setting, we consider a finite set of potential fault locations, where each location is represented as a single atomic hypothesis. All hypotheses are assigned equal prior probability due to the same textual complexity. A locally optimal test should ideally eliminate approximately half of the hypotheses since it provides the maximum information gain and could reduce the number of subsequent tests needed (Mitchell 1982). To isolate the effects of hypothesis reduction, we assume that all circuit tests have uniform cost. This cost model supports our USML setting, where we aim to help learners acquire general and transferable strategies.

ILP-Learned Programs. Hopper learns from five episodes. Each episode uses programs learned from the previous episodes and is trained using a single circuit. This is related to dependent learning in ILP (Lin et al. 2014), but with a pre-defined ordering of episodes. Each episode is assigned a name for the predicate to be invented (Figure 3). The input for Hopper is described in the Appendix A.7.

In episode 1, the system invents “exclusively_powers”, which determines whether the output from a given gate flows exclusively through another gate on its path to the light bulb. This strategy is analogous to identifying dominators in a directed graph (Prosser 1959). In a directed graph, a node B is a dominator of node A with respect to a target node T if every path from A to T passes through B . In a circuit, each test would only affect the gates that it dominates.

To maximise the information gain, the learned strategy finds the test that has half of the gates in a circuit as its dominators. In episode 2, the system utilises the learned program from episode 1 to invent “partition” for creating the two partitions of circuit gates. Given a gate of interest, one partition contains all gates that eventually lead to it. The other partition has the rest of the gates, whose output contributes elsewhere and can influence the circuit independently. Episode 3 builds on these to learn “partition_sizes” which computes the size of partitions and represents a total ordering of them.

Similarly, in episode 4, the system uses the learned programs to invent “optimal_partition_sizes”, which compares different partition sizes to obtain the most balanced partition. Finally, in episode 5, Hopper invents “optimal_test” to find a locally optimal test with the most balanced partition.

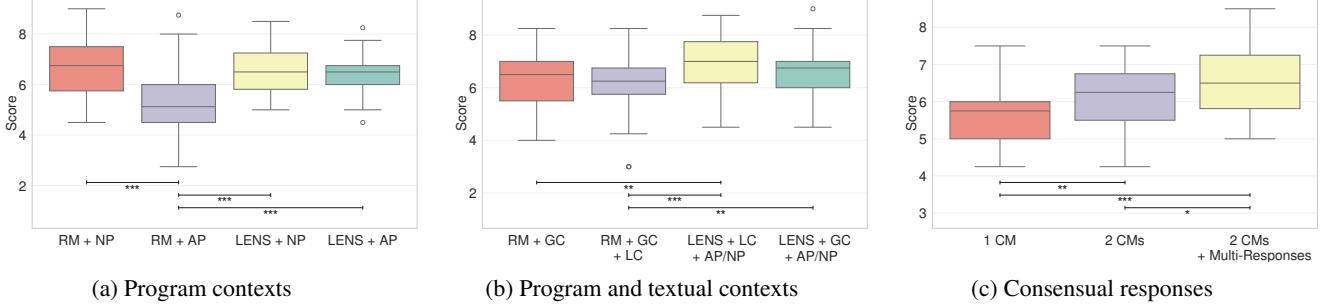


Figure 4: Distribution of LLM judged scores for electric circuit domain explanations. RMs and CMs denote reasoning and coding LLMs, respectively. The significance of results has been highlighted by: $p < 0.05$ (*), $p < 0.01$ (**), $p < 0.001$ (***)

3.4 LLM Explanation

In LENS, we employ a task-specific approach to explanation. Given all ILP-learned logic programs, each reasoning LLM in the LENS explanation pipeline identifies predicates most relevant to solving a provided task. For the USML trial, we define three tasks for teaching, each accompanied by a textual description of its objective. The first task involves identifying exclusive sources of power in a circuit. The second task focuses on partitioning the hypothesised fault locations into two groups and computing their sizes. The third task aims to identify the locally optimal test point that yields the most balanced partitions of hypotheses.

The goal of the explanations is to be understandable to individuals without a technical background. In the LENS explanation pipeline, each reasoning LLM is prompted to generate an accessible explanation by inventing intuitive names for the relevant predicates. In the LENS scoring pipeline, explanations are judged based on helpfulness, relevance, accuracy, depth, creativity, and level of detail. We used manually drafted answers as references for judgment (see Appendix A.8). The scoring models would justify their scores and penalise overly technical explanations (see Appendix A.10).

LLM Choices. For coding LLMs, we use the instruction fine-tuned versions of Qwen2.5 coder 14B (Hui et al. 2024) and StarCoder V2 15B (Lozhkov et al. 2024) due to consideration of size, performance and programming language coverage. For reasoning LLMs, we choose DeepSeek R1 and Claude 3.7 Sonnet, which are widely-studied proprietary models with comparable reasoning performance. For judging LLMs, we involve three LLM judges, DeepSeek R1, Claude 3.7 Sonnet and o3-mini, where o3-mini can provide neural evaluations as it only participates in judging. We discuss selection rationales in detail in Appendix A.2.

4 Experiments

We conducted two experiments. The first experiment evaluated the effectiveness of LENS for producing explanations. The second experiment is a human trial where we used LENS explanations to teach human active learning in three domains: electric circuits, water flow and binary search. Since USML experiments would involve humans, relying

only on automation or human evaluation could lead to ethical concerns or biases. Therefore, we consider LENS as an assistant that generates explanations and judges them. Its role is to support human evaluation: we double-check the top-ranking answers to decide the final explanations to be used in the human trial. In our experiments, we aim to answer the following questions:

- Q1.** Do coding models help frontier reasoning models generate better explanations?
- Q2.** Does using multiple coding models improve explanation over a single coding model?
- Q3.** Does LENS provide better explanations than hand-crafted templates?
- Q4.** Does explaining the ILP-learned active learning strategy help humans select more informative examples?
- Q5.** Does training with active learning strategy explanations help humans perform better at other related tasks?

4.1 Evaluating LENS Explanations

Generalisability. We examined LENS explanations across unrelated tasks. We evaluated LENS explanations in the base domain and then compared LENS against hand-crafted templates (see Section A.9) from previous USML work: game playing (Ai et al. 2021) and algorithm discovery (Ai et al. 2023).

Compared Variables and Conditions. We use this experiment to evaluate the effect of LENS in producing appropriate explanations given different domain contexts. We consider the following variables in our evaluation:

- **Named programs (NP):** Coding and reasoning LLMs are involved. The coding LLMs see the ILP-learned programs.
- **Anonymised programs (AP):** Coding and reasoning LLMs are involved. Coding LLMs see the ILP-learned programs, but all invented predicate names are anonymised.
- **Global context (GC):** LLMs receive a general description of the domain. This aims to provide the minimum information about the domain in which the tasks would be performed so LLMs can contextualise their reasoning.

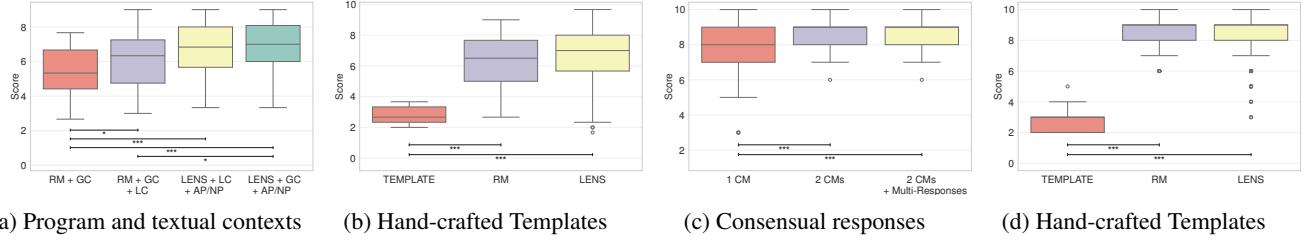


Figure 5: Distribution of LLM judged scores for explanations: (5a, 5b) game playing (Ai et al. 2021) and (5c, 5d) algorithm discovery (Ai et al. 2023). The annotations and markers are consistent with those in Figure 4.

- Local Prolog context (LC): LLMs receive the same circuit examples that would be shown to human participants. The circuits are written as Prolog programs. This aims to contextualise LLMs’ responses further to tailor explanations for these examples.

Baselines and methods. To test the necessity of coding LLMs, we set the baselines to be direct prompting of a reasoning LLM with named programs (NP), anonymised programs (AP), global context (GC), and global together with local context (GC + LC). To test the effect of consensus, we compared using a single coding LLM against using multiple coding LLMs in LENS. We generated explanations across 14 conditions, with three repetitions. We repeated the scoring process three times per explanation from each LLM judge to obtain a total of 3024, 2349 and 783 scores for the electric circuit, game playing and algorithm discovery tasks, respectively. For results, we perform one-way ANOVA followed by post-hoc Tukey’s HSD for pair-wise comparisons.

Results. In Figure 4a, LLM judges consider LENS with named or anonymised ILP-learned programs superior to direct prompting of a reasoning LLM with anonymised programs. Changing from named to anonymised programs decreases scores for reasoning LLMs, suggesting a dependency on sensible program names or even an inability to interpret program logic. In contrast, LENS exhibits robustness in maintaining reasonable scores with anonymised programs. In addition, in Figure 4b and 5a, when local or global contexts are present, LENS outperforms reasoning LLMs with named or anonymised programs. To answer **Q1**, coding LLMs help frontier reasoning LLMs generate better explanations, particularly when programs have less meaningful names. This result suggests LENS can interpret programs better, especially for real-world learning problems where invented program names are not always easily defined.

In Figure 4c and 5c, LLM judges consider that using multiple coding LLMs in LENS produces better explanations than using just a single coding LLM. Compared to generating consensus explanations by sampling one response from each coding LLM, LENS produces even higher scores when coding LLMs generate multiple responses. To answer **Q2**, both multiple coding models and multiple responses from several models improve explanation quality over a single coding LLM. This result aligns with past work (Du et al. 2024; Subramaniam et al. 2024) where consensus between

multiple LLMs results in higher quality responses.

Hand-crafted templates from previous USML work translate ILP-learned logic programs into natural language using predefined patterns. While these explanations are more compact and preserve the structure of the programs, LLM judges rated them as lacking accessibility and educational value. The insufficient context and depth are reflected in the lower scores of hand-crafted templates compared to LENS explanations (Figure 5b and 5d). To answer **Q3**, LENS provides better explanations than hand-crafted templates.

Human Validation. We leveraged LLM scoring to filter explanations and guide the final explanation selection by human experts. The top-scoring 25% percentile of explanations is manually evaluated for each task. In the final explanation selection, we value technical correctness over LLM judged scores, as teaching humans incorrect strategies would undermine the learning objectives. We show a selected explanation in Figure 3 with the rest in Appendix A.5 and A.6. For task 1, we selected the highest-scoring explanation (LENS + AP + GC, DeepSeek R1) after confirming no technical errors. For task 2, we selected an explanation (LENS + NP, DeepSeek R1) that correctly accounts for the learned strategy. Other high-scored explanations contain the critical error of ignoring circuit structure. For task 3, we chose an explanation (LENS + AP + GC, Claude 3.7 Sonnet) that correctly frames partitioning based on the learned strategy.

4.2 Teaching Active Learning to Humans

Target Domains In the study, we frame the active learning problem described in Section 3.3 in the following domains. The domain descriptions and visual presentations in the experiment are included in Appendix A.4.

Waterflow Domain. A well-known analogy to the electric circuit base domain (Section 3.3) is the water flow problem (Wiese, Konderding, and Schmid 2008; Gentner and Gentner 2014). The two domains are isomorphic - water flow circuits and the electric circuits have both component correspondences and functional analogies. The task here is to find exactly one clog in the system by measuring water pressure after circuit nodes.

List Binary Search Domain. The task requires finding a number in an ordered list. This domain is isomorphic to the waterflow and circuit domains for ordered lists, since binary search resembles binary partitioning of linear circuits.

Compared Conditions and Design. Our study involves a curriculum of three phases (see Appendix A.4), which correspond to the three tasks described in Section 3.4.

After the learning phases, participants see a total of 15 trial items equally distributed among the domains. We start with the circuits domain but randomise the other two to avoid sequence effects. In each item, participants see a visual presentation of the problem according to the domain. Participants must select the first test to perform out of a list of single-choice options, with an “I don’t know” alternative option. The domains are introduced extensively before the learning phases and each set of trial items.

We map each response to the entropy reduction of a test according to Equation (1). Since not every problem contains a test that achieves the maximum entropy reduction of 1, we normalise the maximum achievable score in each trial to 1 while keeping the minimum at 0. This way, a zero score always corresponds to a split with no information gain at all.

Using the symbols introduced in Section 3.1, we refer to self-learning as the control group (H_1) and machine-explained learning as the experimental group (H_2). Both groups receive visual feedback on their responses to the items in the learning phases. In addition to the LENS explanations, H_2 also sees additional highlights in the visual feedback, which aligns with intermediary program outputs. For the third phase, only H_2 sees the sizes for each split in the sequences.

Results and Discussion. We recruited 100 participants via the platform Prolific¹. We recorded a mean age of 34 with a standard deviation of 12 years. 50 participants were male, 49 female, and 1 non-binary. We exclude 17 participants due to spending a mean time of more than 60s per trial (population $\mu = 29s$, std = $12s$). Out of the remaining 83, 58 were in H_2 , and 25 were in H_1 . Single trials where excluded when participants selected the escape option.

To answer **Q4** and **Q5**, we use the non-parametric Mann-Whitney U-test (MWU, Mann and Whitney (1947)) due to non-normally distributed data for the difference in performance between the two conditions for each of the domains. Non-significant test results do not indicate an effect of explanations on trial performance in the tested conditions (circuits: $U = 18514.5$, $p = .13$, water flow: $U = 16847.5$, $p = .47$, lists: $U = 16621.0$, $p = .15$). Figure 6 shows the per-trial metrics for both conditions and the three domains. The answer to both **Q4** and **Q5** is *no*: explanations of the ILP learned strategy do not help humans select more informative examples, and training with these explanations does not help humans perform better at related tasks.

To further examine the performance of two human groups, we simulated random trial answers by uniformly sampling from the answer options. The MWU test shows that across all domains, random information gain is significantly lower than the information gain from participants ($U = 264574.0$, $p < .001$, CLES = .30). This indicates that the presented task is either too trivial or could be learned based on the curriculum alone, without the need for explanations.

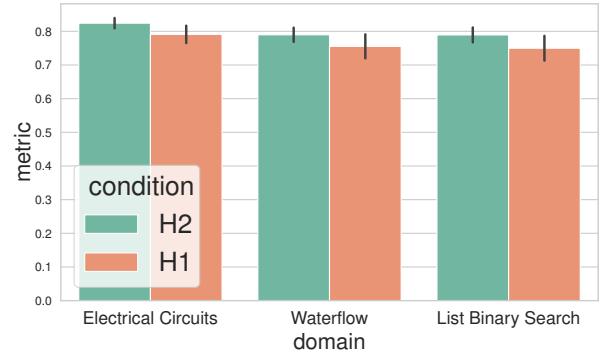


Figure 6: Mean and standard error per domain and condition.

The successful learning without explanations suggests a critical mismatch between task and explanation complexity. Our results align with predictions from prior work that explanations can become counterproductive when their informational complexity exceeds the cognitive cost of solving the underlying problem (Ai et al. 2021). This exposes a fundamental challenge where LLMs’ comprehensive explanations may overwhelm rather than assist users in solving simple tasks. Two complementary factors for our null findings are that either a) the tasks in combination with the learning curriculum are too straightforward or b) the presented explanations may have introduced unnecessary cognitive load. Future studies should systematically vary both task and explanation complexity to identify optimal combinations.

5 Limitation and Future Work

While our human experiments show no USML effect, the better-than-random human performance suggests that the explanations might be too complex for the low-stakes decisions participants faced. While LLMs can generate comprehensive explanations, they may overwhelm users on simpler problems rather than providing learning support. This aligns with prior work showing that explanation benefits depend critically on balancing task and explanation complexity (Ai et al. 2021). Future USML research could investigate this balance by modelling human strategy learning through a Bayesian approach to Theory of Mind (Tenenbaum, Griffiths, and Kemp 2006), predicting which strategies are likely to be generalised by humans under self-learning and machine-explained conditions.

Our findings also highlight opportunities for improving LENS itself: incorporating recent advances in LLM-based (Li and Ellis 2024; Novikov et al. 2025) and second-order (Hocquette, Dumančić, and Cropper 2024) program synthesis could generate explanations for more complex learning tasks that involve meta-level knowledge. The computational framework we established provides a foundation for such investigations, enabling systematic exploration of when and how automated explanation systems can effectively enhance human learning.

¹<https://www.prolific.com>

Ethical Statement. An ethics approval for the empirical study on human participants was issued by the home university's ethics board.

Acknowledgement. The third author acknowledges support from the project "Learning from Learners (VoLL-KI), funded by the German Federal Ministry of Education and Research (BMBF). The authors appreciate suggestions from Céline Hocquette and David Cerna on adapting Hopper.

References

- Ai, L.; Langer, J.; Muggleton, S. H.; and Schmid, U. 2023. Explanatory machine learning for sequential human teaching. *Machine Learning*, 112: 3591–3632.
- Ai, L.; Muggleton, S. H.; Hocquette, C.; Gromowski, M.; and Schmid, U. 2021. Beneficial and harmful explanatory machine learning. *Machine Learning*, 110: 695–721.
- Alur, R.; et al. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 1–8.
- Anderson, J. R.; Kushmerick, N.; and Lebiere, C. 1993. The Tower of Hanoi and goal structures. In *Rules of the Mind*, 121–142. Psychology Press.
- Atzmueller, M.; Fürnkranz, J.; Kliegr, T.; and Schmid, U. 2024. Explainable and interpretable machine learning and data mining. *Data Min. Knowl. Discov.*, 38(5): 2571–2595.
- Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. ArXiv:2108.07732.
- Barredo Arrieta, A.; et al. 2020. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58: 82–115.
- Carbonell, J. 1985. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. *Machine Learning*, 11: 26.
- Chen, M.; et al. 2021. Evaluating Large Language Models Trained on Code. ArXiv:2107.03374.
- Cohn, D.; Atlas, L.; and Ladner, R. 1994. Improving generalization with active learning. *Machine Learning*, 15(2): 201–221.
- Cook, C.; Goodman, N. D.; and Schulz, L. E. 2011. Where science starts: Spontaneous experiments in preschoolers' exploratory play. 120(3): 341–349.
- Cropper, A. 2021. ILP Experimentation Framework. <https://github.com/logic-and-learning-lab/ilp-experiments.git>.
- Cropper, A.; Dumancic, S.; Evans, R.; and Muggleton, S. H. 2021. Inductive Logic Programming at 30. *Machine Learning*, 111: 147–172.
- Cropper, A.; Evans, R.; and Law, M. 2020. Inductive general game playing. *Machine Learning*, 109(7): 1393–1434.
- Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Machine Learning*.
- Cropper, A.; Morel, R.; and Muggleton, S. 2020. Learning higher-order logic programs. *Machine Learning*, 109(7): 1289–1322.
- Dai, W.-Z.; and Muggleton, S. H. 2021. Abductive Knowledge Induction from Raw Data. In Zhou, Z.-H., ed., *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, 1845–1851. International Joint Conferences on Artificial Intelligence Organization.
- De Kleer, J.; and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence*, 32(1): 97–130.
- Du, Y.; Li, S.; Torralba, A.; Tenenbaum, J. B.; and Mordatch, I. 2024. Improving Factuality and Reasoning in Language Models through Multiagent Debate. In *Proceedings of the 41st International Conference on Machine Learning*, 11733–11763. PMLR. ISSN: 2640-3498.
- Evans, R.; and Grefenstette, E. 2018. Learning explanatory rules from noisy data. *J. Artif. Int. Res.*, 61(1): 1–64.
- Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, S.; Zettlemoyer, L.; and Lewis, M. 2023. InCoder: A Generative Model for Code Infilling and Synthesis.
- Gabaldon, A.; Langley, P.; and Meadows, B. 2020. Meta-Level and Domain-Level Processing in Task-Oriented Dialogue. *Common Model of Cognition Bulletin*, 1(1): 25–31.
- Gentner, D.; and Forbus, K. D. 2011. Computational models of analogy. *WIREs Cognitive Science*, 2(3): 266–276.
- Gentner, D.; and Gentner, D. R. 2014. Flowing Waters or Teeming Crowds: Mental Models of Electricity. In Gentner, D.; and Stevens, A. L., eds., *Mental Models*, 107–138. Psychology Press. ISBN 978-1-315-80272-5.
- Glanois, C.; Jiang, Z.; Feng, X.; Weng, P.; Zimmer, M.; Li, D.; Liu, W.; and Hao, J. 2022. Neuro-Symbolic Hierarchical Rule Induction. In *Proceedings of the 39th International Conference on Machine Learning*, 7583–7615. PMLR. ISSN: 2640-3498.
- Gopnik, A.; Meltzoff, A. N.; and Kuhl, P. K. 1999. *The scientist in the crib : minds, brains, and how children learn*. William Morrow & Co.
- Graesser, A. C.; Conley, M. W.; and Olney, A. 2012. Intelligent tutoring systems. In Harris, K. R.; Graham, S.; Urden, T.; Bus, A. G.; Major, S.; and Swanson, H. L., eds., *APA educational psychology handbook, Vol. 3, Application to learning and teaching*, 451–473. American Psychological Association.
- Gu, A.; Rozière, B.; Leather, H.; Solar-Lezama, A.; Synnaeve, G.; and Wang, S. I. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. arXiv:2401.03065.
- Gulwani, S.; Hernández-Orallo, J.; Kitzelmann, E.; Muggleton, S. H.; Schmid, U.; and Zorn, B. 2015. Inductive programming meets the real world. *Commun. ACM*, 58(11): 90–99.
- Gunning, D.; Stefik, M.; Choi, J.; Miller, T.; Stumpf, S.; and Yang, G.-Z. 2019. XAI—Explainable artificial intelligence. *Science Robotics*, 4(37): eaay7120. Publisher: American Association for the Advancement of Science.
- Hocquette, C.; Dumančić, S.; and Cropper, A. 2024. Learning logic programs by discovering higher-order abstractions. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 3421–3429.

- Hocquette, C.; and Muggleton, S. 2018. How Much Can Experimental Cost Be Reduced in Active Learning of Agent Strategies? In *Inductive Logic Programming*, volume 11105, 38–53. Cham: Springer International Publishing.
- Hui, B.; et al. 2024. Qwen2.5-Coder Technical Report. ArXiv:2409.12186.
- King, R. D.; Whelan, K. E.; Jones, F. M.; Reiser, P. G. K.; Bryant, C. H.; Muggleton, S. H.; Kell, D. B.; and Oliver, S. G. 2004. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427: 247–252.
- Law, M.; Russo, A.; and Broda, K. 2014. Inductive Learning of Answer Set Programs. In Fermé, E.; and Leite, J., eds., *Logics in Artificial Intelligence*, Lecture Notes in Computer Science, 311–325. Cham: Springer International Publishing. ISBN 978-3-319-11558-0.
- Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *Advances in Neural Information Processing Systems*, 35: 21314–21328.
- Li, W.-D.; and Ellis, K. 2024. Is Programming by Example Solved by LLMs? In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Li, Y.; et al. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097. Publisher: American Association for the Advancement of Science.
- Lin, D.; Dechter, E.; Ellis, K.; Tenenbaum, J.; and Muggleton, S. 2014. Bias reformulation for one-shot function induction. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, 525–530. NLD.
- Lozhkov, A.; et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. ArXiv:2402.19173.
- Mann, H. B.; and Whitney, D. R. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.
- Michalski, R. 1983. A Theory and Methodology of Inductive Learning. *Machine Learning: Symbolic Computation*.
- Michie, D. 1988. Machine learning in the next five years. In *Proceedings of the 3rd European Working Session on Learning*, 107–122.
- Miller, T. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267: 1–38.
- Minh, D.; Wang, H.; Li, Y.; and Nguyen, T. N. 2022. Explainable artificial intelligence: a comprehensive review. *Artificial Intelligence Review*.
- Mitchell, T. M. 1982. Generalization as search. *Artificial Intelligence*, 18: 203–226.
- Moore, L. R.; Caspi, R.; Boyd, D.; Berkmen, M.; Mackie, A.; Paley, S.; and Karp, P. D. 2024. Revisiting the y-ome of Escherichia coli. *Nucleic Acids Research*, 52(20): 12201–12207.
- Muggleton, S. H. 1995. Inverse Entailment and Progol. *New Generation Computing*, 13: 245–286. Place: JPN Publisher: Ohmsha.
- Muggleton, S. H.; and Buntine, W. 1988. Machine Invention of First-order Predicates by Inverting Resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, 339–352.
- Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100(1): 49–73.
- Muggleton, S. H.; Schmid, U.; Zeller, C.; Tamaddoni-Nezhad, A.; and Besold, T. 2018. Ultra-Strong Machine Learning: Comprehensibility of programs learned with ILP. *Machine Learning*, 107: 1119–1140.
- Nienhuys-Cheng, S.; and Wolf, R. 1997. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc.
- Novikov, A.; et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. ArXiv:2506.13131.
- Popper, K. 2005. *The Logic of Scientific Discovery*. Routledge.
- Prosser, R. T. 1959. Applications of Boolean matrices to the analysis of flow diagrams. In *Eastern joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '59 (Eastern), 133–138. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-7868-0.
- Purgał, S. J.; Cerna, D. M.; and Kaliszyk, C. 2022. Learning Higher-Order Logic Programs From Failures. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, 2726–2733.
- Schmid, U.; and Wrede, B. 2022. What is Missing in XAI So Far? *KI - Künstliche Intelligenz*, 36(3): 303–315.
- Schmid, U.; Zeller, C.; Besold, T.; Tamaddoni-Nezhad, A.; and Muggleton, S. H. 2017. How does predicate invention affect human comprehensibility? In *Proceedings of the 26th International Conference on Inductive Logic Programming*, 52–67.
- Subramaniam, V.; Du, Y.; Tenenbaum, J. B.; Torralba, A.; Li, S.; and Mordatch, I. 2024. Multiagent Finetuning: Self Improvement with Diverse Reasoning Chains. In *The Thirteenth International Conference on Learning Representations*.
- Tenenbaum, J. B.; Griffiths, T. L.; and Kemp, C. 2006. Theory-based Bayesian models of inductive learning and reasoning. *Trends in Cognitive Sciences*, 10(7): 309–318. Publisher: Elsevier.
- Wiese, E.; Konderding, U.; and Schmid, U. 2008. Mapping and Inference in Analogical Problem Solving — As Much as Needed or as Much as Possible? *Proceedings of the Annual Meeting of the Cognitive Science Society*, 30(30).
- Zheng, L.; et al. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.
- Zhuo, T. Y.; et al. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. arXiv:2406.15877.

A Appendix

A.1 Circuit Design

All circuits were created from an existing metabolic network in biology. A metabolic network is a network of chemical processes that determine the physiological and biochemical properties of a biological system. We used the metabolic network from EcoCyc (Moore et al. 2024), which is an online database containing the state-of-the-art biological information about a model organism, *Escherichia coli* (*E.coli*). We used a collection of key pathways in the metabolic network and their major chemical compounds as connections and gates for building our electric circuits.

A.2 LLM Selections

Coding LLM Choices. Regarding coding LLM choices, we consider language coverage important for selecting coder models. We use performance in code generation and reasoning (Austin et al. 2021; Zhuo et al. 2025; Gu et al. 2024) as references for LLMs’ ability to understand and explain programs. In addition, we selected models of similar sizes since this would not cause a significant difference in model capabilities. Qwen2.5 coder 14B (Hui et al. 2024) has high performance on the benchmarks for this model class and was trained on 92 programming languages. StarCoder V2 15B (Lozhkov et al. 2024) was trained over 600 programming languages and has decent performance over the benchmarks. We used the instruction fine-tuned versions of Qwen2.5 coder 14B and StarCoder V2 15B.

Reasoning LLM Choices. DeepSeek R1 is a reasoning-focused LLM with open-sourced weights, which is competitive to proprietary models in tasks involving reasoning and coding. Claude 3.7 Sonnet is an assistant-focused LLM that has good performance on instruction-following, coding, and reasoning benchmarks, with a good balance between inference speed, cost, and reasoning power.

Judging LLM Choices. We employed three LLM judges, DeepSeek R1, Claude 3.7 Sonnet and o3-mini, with each judge evaluating each explanation independently. We use o3-mini as a third-party judge for neural evaluations. Unlike DeepSeek R1 and Claude 3.7 Sonnet, which both generate and assess answers (self and peer assessments), o3-mini only participates in judging. Judgments from models DeepSeek R1 and Claude 3.7 Sonnet are combined with o3-mini’s assessments to mitigate potential self-enhancement bias in LLM self-judgment.

A.3 LLM Experimental details

For interpreting logic programs, we used Ollama with the LangChain Python package to prompt the coding LLMs. The temperature was the default value of 0.8 from the LangChain API. The hardware specification is Google Colab (Ubuntu)/Intel(R) Xeon(R) CPU @ 2.20GHz with 84GB RAM/NVIDIA A100 GPU with 40GB memory. To prompt the reasoning and judging models, we used a temperature of 0.0 to lower the variability of explanation generation and set the reasoning level to medium for o3-mini. The explanations

might not be fully reproducible with the same temperatures or future model versions.

A.4 Empirical Study Design Details

Paragraphs in *italics* are direct quotations from the experimental interface. An exported version of the study interface, as well as all images (those shown here and those not) are included in the supplementary material.

Exemplary Domain Description. *Your task is to perform tests to detect faults at connections in electrical circuits. Each circuit has one or more sources of energy (battery), lightbulbs (which should always be **on**), and gates (which are connections between cables). If a gate is faulty, it does not transfer energy, and the lightbulbs connected to it turn off. To find the faulty gate, you can add an additional power source at a test point, which potentially makes the lightbulb turn on again, giving you information about the location of the fault in the circuit. Here is a simple illustration:*

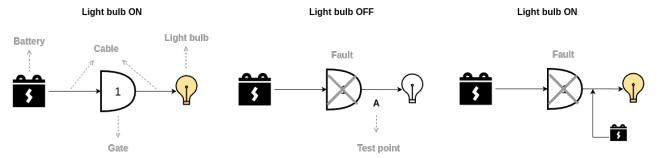


Figure 7: Visual domain introduction used in the study.

These are simplified circuits in that we can always assume that the circuit is closed after the lightbulb and the fault lies “on the path to the lightbulb”. Here is a summary of the components of the circuits:

- **Battery:** The battery is the “source” of the current going through the circuit.
- **Cable:** The arrows are cables which transfer current. Note that current is only transferred in the direction of the arrows!
- **Gate:** The shapes labeled with numbers (in the example above “1”) are gates which connect the current from the incoming arrow to the outgoing arrow. They may be faulty, in that they do not output the current, although they receive current from all input cables.
- **Lightbulb:** The lightbulb turns on if it receives power from all input cables.
- **Test point:** Test points are cables labeled with letters (in the example above “A”). You can perform a test by supplying power to such a cable, effectively ignoring all prior gates.

Both the gate and the lightbulb only work if they receive current from all connected input cables (similar to an AND gate). Consider these examples:

Gate 1 in the left example is only active if it receives current from both input cables. Similarly, the lightbulb in the right example must receive current from both input cables to turn on.

Each circuit you will see in this study has exactly one faulty gate. Your task is to choose the first test to perform

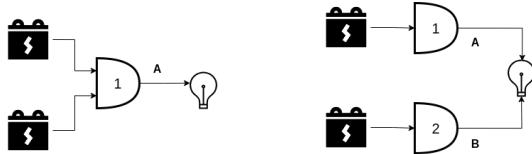


Figure 8: Demonstration of AND gates used in the study.

so that the fault can be found with the least tests possible, regardless of the result of the test. Here is a worked out example:

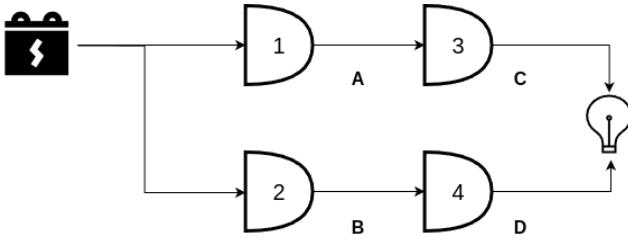


Figure 9: Worked out example used for the introduction of the circuits domain in the study.

In this scenario, two tests would be equally viable: **C** and **D**. Consider this for test **C**: If the lightbulb turns on, the fault must be in the top part, otherwise the bottom part. In either case, the next test will certainly reveal the fault, as there are only two options left. For test **D** the same principle applies, but with the top and bottom switched.

Note that the worked example is only included for the re-introduction of the domain after the learning phases. Before the learning phases, these paragraphs are omitted. The other domains are introduced in a similar way.

First Learning Phase. The first learning phase aims to teach participants to identify dominators in a directed graph. Participants see the graph in Figure 10 without any highlights and the following task description:

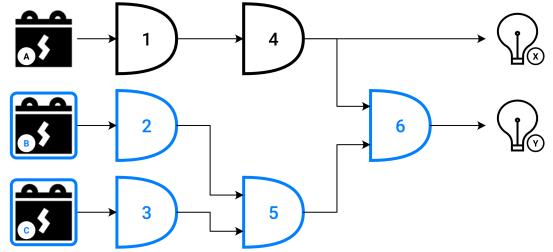
Find all gates and batteries which are exclusively a source of power for lightbulb Y, that is they don't supply current to any other lightbulb than Y.

Participants are offered the numbered gates 1 through 6, the batteries A, B, and C as well as the lightbulbs X and Y as multiple choice options.

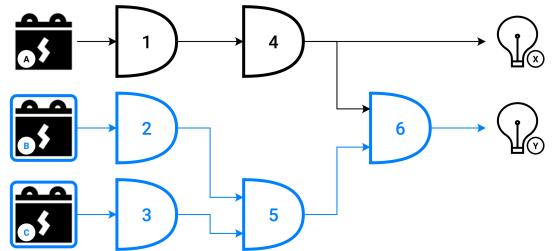
Second Learning Phase. The second learning phase aims to teach participants to partition the graph. Participants see the graphs in Figure 11 without any highlights and the following task description:

The goal of performing a test is to divide the gates in the network into two groups: The ones which may be faulty, and the ones which we can be certain work fine. Remember: Test Points are cables labeled with capital letters. To perform a test, consider the following:

*When you perform a test, you add a power supply to the selected cable. If the lightbulb turns on after supplying power, the fault must be somewhere **before** the test cable, as the network seems to work fine with power supplied at*



(a) Visual feedback given to H_1 .



(b) Visual feedback given to H_2

Figure 10: Participants see this circuit without any highlighted nodes or edges during learning phase 1.

a point in the middle. If the lightbulb still doesn't turn on after supplying power, the fault must be **after** the test point. Since we also know that there is only one fault, this means all previous nodes work fine.

We offer two different circuits to illustrate the edge-case of a bypass around a test point. For each numbered gate in the circuit, participants are given a single choice option between *certainly working, potentially faulty* and *don't know*.

Third Learning Phase. The third learning phase aims to teach participants to select the locally optimal test based on partition sizes. Participants see the two solution traces shown in Figure 12 with the following task description:

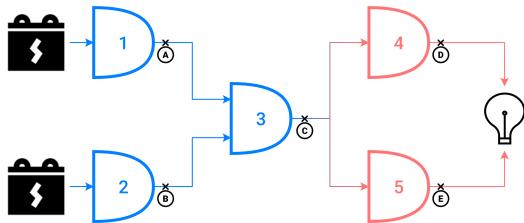
Consider the following approaches to finding the fault in the same simple circuit. Familiarise yourself with the two strategies and think about which of the two is better suited for finding faults quickly and reliably.

Participants are then given a single choice between the two options displayed. Other than the first two learning phases, participants don't receive the LENS explanations with the feedback, but directly after the two traces.

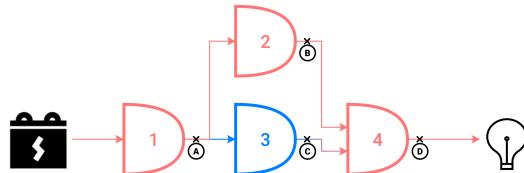
Trial Items. Figure 13 shows exemplary trial graphs shown to participants for each domain. All used graphs are included in the supplementary material.

A.5 Selected Explanations for Empirical Study

Here we include explanations selected from LENS for the empirical study. Each explanation is labelled by task, con-



(a) The first circuit shown to participants.



(b) The second circuit shown to participants with the goal of illustrating bypasses.

Figure 11: Participants see these circuits without any highlighted nodes or edges during learning phase 2. These circuits contain the visual feedback for H_2 . H_1 sees no highlighted edges just as it is demonstrated in Figure 10.

dition and reasoning model used by LENS. In these conditions, LENS use the same two coding models, Qwen2.5 coder 14B and StarCoder V2 15B.

Explanation for Task 1 (LENS + AP + GC, DeepSeek R1)

To solve this task, we use two main strategies:

1. Path Checker: This checks if a gate can send power to a specific lightbulb's connection point. It works by:

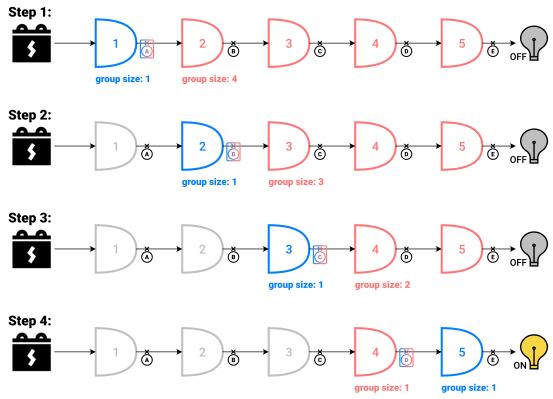
- Directly matching if the gate is the lightbulb's connection point itself
- Following each outgoing arrow from the gate to see if any path leads to the lightbulb's connection point

2. Exclusive Source Verifier: This confirms a gate only powers one specific lightbulb (Y) by:

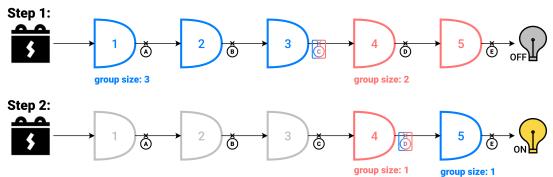
- Using the Path Checker to verify it powers lightbulb Y
- Checking it doesn't power any other lightbulb (using the Path Checker for all other lightbulbs)

The solution involves:

1. Finding all gates that power lightbulb Y using the Path Checker
2. For each candidate gate, using the Exclusive



(a) Option 1.



(b) Option 2.

Figure 12: Two traces for solving a simple linear circuit as seen by H_2 . H_1 does not see the group size annotations. Apart from that, the visual presentation is identical.

Source Verifier to confirm it doesn't power any other lightbulb

3. Keeping only gates that pass both checks

This ensures we find gates that exclusively send power to lightbulb Y and nowhere else.

Explanation for Task 2 (LENS + NP, DeepSeek R1)

To solve the task of dividing gates into two groups based on a test at point C (one group that may be faulty and one that is certainly working), follow these steps:

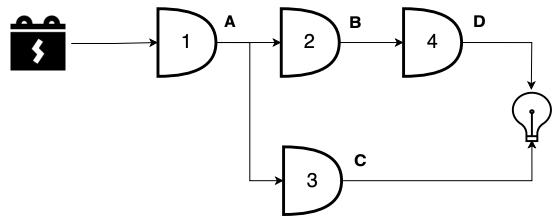
Relevant Strategies

1. Same circuit check:

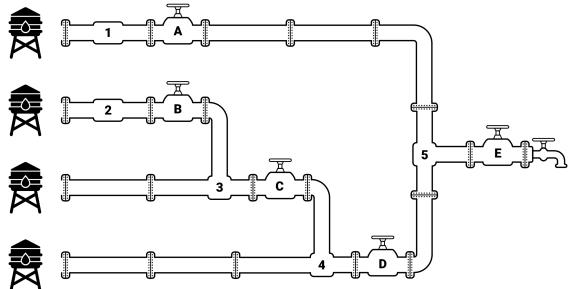
Confirm if a gate is in the same circuit as point C. This ensures we only consider gates affected by the test.

2. Exclusively powers check:

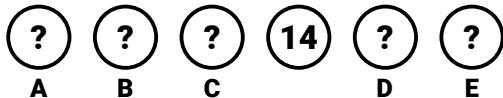
Determine if point C exclusively powers a gate. If true, the gate relies solely on C for power.



(a) Circuit domain.



(b) Water flow domain.



(c) List binary search domain.

Figure 13: Exemplary graphs used during the test phase.

3. Not exclusively powers check:

Determine if point C does not exclusively power a gate. This means the gate gets power from other sources besides C.

4. Partition operation:

Split the list of gates into two groups using the above checks:

- Group 1: Gates in the same circuit as C and exclusively powered by C.
- Group 2: Gates in the same circuit as C but not exclusively powered by C.

Step-by-Step Application

1. Identify relevant gates:

Use the "same circuit check" to collect all gates in the same circuit as point C. Ignore gates outside this circuit.

2. Apply the partition operation:

- For Group 1 (certainly working if C passes the test):

Use the "exclusively powers check" to find gates powered only by C. If C works correctly, these gates cannot be faulty.

- For Group 2 (may be faulty):

Use the "not exclusively powers check" to find gates powered by C and other sources. These gates could still be faulty even if C works.

3. Result:

The partition operation directly outputs the two groups:

- Group 1: Gates that are certainly working (if C passes the test).
- Group 2: Gates that may be faulty (regardless of C's test outcome).

Key Insight

The test at point C helps isolate responsibility:

- Gates exclusively powered by C depend entirely on C's functionality.
- Gates with multiple power sources might have issues beyond C.

This division pinpoints where faults could exist based on the test results.

Explanation for Task 3 (LENS + AP + GC, Claude 3.7 Sonnet)

Choosing the Most Appropriate Test to Find a Fault

To decide which test is most appropriate for finding a fault quickly and reliably in an electrical circuit, we need to use several strategies:

Relevant Strategies

1. **Test Point Evaluator:** Assesses different test points to determine which one provides the most useful information.
2. **Circuit Divider:** Splits the circuit into sections based on where test points are located.
3. **Group Size Calculator:** Counts how many gates fall into each group when a particular test is applied.
4. **Optimal Test Finder:** Identifies which test creates the most balanced division of gates into groups.

How to Apply These Strategies

1. First, identify all possible test points in the circuit (these are usually labeled with letters).
2. For each test point, use the Circuit Divider to determine how it would split the gates into two groups:
 - Gates that might be faulty if this test point is used
 - Gates that would be confirmed as working properly

3. Use the Group Size Calculator to count how many gates would be in each group for each potential test point.

4. Apply the Optimal Test Finder to compare all possible test points and find the one that:

- Creates the most balanced division (where the smaller group is as large as possible)
- Minimizes the number of gates you would need to check after the test
- Provides the clearest information about where the fault might be

5. The test point that creates the most balanced division is usually the best choice because:

- It eliminates the maximum number of possibilities with a single test
- It divides the remaining possibilities into manageable groups
- It helps you find the fault with the fewest additional tests

This approach is similar to a binary search strategy, where you try to eliminate half of the possibilities with each test. By choosing the test point that creates the most balanced division, you can find the fault in the circuit with the minimum number of tests, saving time and effort.

A.6 Case Study LLM Responses

Owing to the large volume of generated explanations, we present case studies highlighting typical errors identified in LLM responses. These errors highlight the importance of combining human evaluation and automated LLM evaluations. The most common error type involves proposing fundamentally different approaches from both the reference answer and the ILP-learned logic programs, which we attribute to LLM hallucination. Hallucination is a well-documented tendency for language models to generate plausible-sounding but factually incorrect content.

For example, the following response from direct prompting of DeepSeek R1 provides a specific algorithm for identifying the groups by starting from the test point and working forwards or backwards (highlighted in yellow). This proposed algorithm incorrectly assumes that any test point serves as the primary dominator, implying that exactly half the circuit gates would eventually pass through it. However, this assumption does not hold in general circuit topologies, so the strategy is not technically correct.

Explanation for Task 3 (RM + GC, DeepSeek R1)

To solve this task of choosing the best test point to find the fault quickly and reliably, use these strategies:

Strategy 1: Focus on the Lightbulb Path

Identify the specific path from the battery to the lightbulb that isn't working. Only consider test points along this path, as they directly affect the faulty lightbulb. Ignore test points not on this path.

Strategy 2: Count Gates Before and After

For each test point on the path:

- Count the gates between the battery and the test point ("gates before").
- Count the gates between the test point and the lightbulb ("gates after").

Strategy 3: Choose the Most Balanced Test

For each test point, compare its "gates before" and "gates after" counts:

- Take the **larger** of these two numbers for each test point.
- Select the test point where this larger number is the **smallest** among all options.

How to Apply:

1. Use **Focus on the Lightbulb Path** to list all test points between the battery and the faulty lightbulb.

2. For each test point, apply Count Gates Before

and After to get two numbers.

3. Apply Choose the Most Balanced Test:

- Compare the larger number from each test point.

- Pick the test point with the smallest larger number.

Why This Works:

This test point splits the path into segments as evenly as possible. After testing:

- If the lightbulb turns on, the fault is in the "gates before" segment.
- If it stays off, the fault is in the "gates after" segment.

By choosing the test point where the larger segment is as small as possible, you minimize the worst-case number of gates to check next. This saves time and ensures reliability.

Despite this fundamental error, the explanation received mixed ratings: 8 from DeepSeek R1, 6 from Claude 3.7 Sonnet, and 9 from o3-mini, illustrating how superficially coherent but incorrect explanations can still receive favourable scores from some judges.

The second common error type involves overgeneralised responses that lack sufficient specificity. While the example below correctly references some key components from the ILP-learned logic programs, it fails to identify the central concept of how to construct proper partitions in detail. This type of error demonstrates how LLMs can produce responses that appear knowledgeable by incorporating some relevant terminology, yet remain unactionable due to missing the core algorithmic concepts.

Explanation for Task 3 (LENS + NP + GC, Claude 3.7 Sonnet)

Choosing the Best Test Point to Find a Fault

To find a fault quickly and reliably, we need to choose the most appropriate test point in the circuit.

Relevant Tools

- **Circuit Group Divider:** Separates gates into groups based on specific criteria.
- **Group Size Calculator:** Determines how many gates are in each group.
- **Optimal Division Finder:** Identifies the best way to divide gates into groups to efficiently locate faults.
- **Best Test Selector:** Determines which test point will be most effective for finding faults.

Strategy to Solve This Task

1. **Evaluate All Possible Test Points:** We need to consider each available test point in the circuit.

2. For Each Test Point:

- Use the Circuit Group Divider to separate gates into two groups:
 - * Gates that might be faulty if the test is successful
 - * Gates that must be working properly if the test is successful

3. **Calculate Group Sizes:** For each potential test point, determine how many gates fall into each group using the Group Size Calculator.

4. **Find the Optimal Division:** The best test point will divide the gates in a balanced way, so that:

- Neither group is too large
- The difference between group sizes is minimized
- This makes it easier to narrow down the location of the fault

5. Select the Best Test: Use the Best Test Selector to choose the test point that creates the most balanced division.

Applying This Strategy

To apply this strategy:

1. Identify all available test points in the circuit.
2. For each test point, use the Circuit Group Divider

to separate the gates into two groups.

3. Use the Group Size Calculator to determine how many gates are in each group for each test point.
4. Use the Optimal Division Finder to compare all the different divisions and identify which one is most balanced.
5. The Best Test Selector will then choose the test point that creates this optimal division.
6. Perform the test at this selected point first, as it will give you the most information about where the fault might be, allowing you to find the problem more quickly and with fewer tests.

This approach ensures you're using your testing resources efficiently to locate the fault with minimal effort.

A.7 Hopper Input

Hopper receives Prolog programs representing example, background knowledge and language bias as input. As part of the background knowledge for Hopper, each circuit is represented as relations representing the circuit components and their connectivity.

Circuit Example

```
gate(1). gate(2). gate(3). gate(4). gate(5).  
  
test_point_label(1, output_a).  
test_point_label(2, output_b).  
test_point_label(3, output_c).  
test_point_label(4, output_d).  
test_point_label(5, output_e).  
  
is_connected(0, 1). is_connected(0, 2).  
is_connected(1, 3). is_connected(2, 3).  
is_connected(3, 4). is_connected(3, 5).  
is_connected(4, lightbulb).  
is_connected(5, lightbulb).
```

We declare the predicates that would appear in the program and define the types, as well as which program arguments are inputs and outputs. We refer to these as language bias and use language bias similar to that from (Cropper and Morel 2021; Cropper 2021). Our second-order background knowledge was built from (Purgał, Cerna, and Kaliszyk 2022).

Background Knowledge

```
find_all(P, A, L):-  
    findall(H, call(P, A, H), L).  
  
all(P, [H|T], C) :-  
    call(P, H, C), !,
```

```

all(P, T, C).
all(_, [], _).

empty_list([]).

is_equal(A, A).
same_circuit(A, B) :- gate(A), gate(B),
    N is A // 100, M is B // 100, N == M.
size(L, S) :- is_list(L), length(L, S).

not_list(A) :- not(is_list(A)).

pair(A,B,[A,B]) :- not_list(A), not_list(B).
larger_min_size(A, B, A) :-
    is_list(A),
    is_list(B),
    length(A, 2),
    length(B, 2),
    min_list(A,M1),
    min_list(B,M2),
    max(M1, M2, M1).
larger_min_size(A, B, B) :-
    is_list(A),
    is_list(B),
    length(A, 2),
    length(B, 2),
    min_list(A,M1),
    min_list(B,M2),
    max(M1, M2, M2).

min(A, B, C) :- min_list([A,B],C).
max(A, B, C) :- max_list([A,B],C).

fold(P,Acc,[H|T],Out) :-
    call(P,Acc,H,Inter),!,
    fold(P,Inter,T,Out).
fold(_P,Acc,[],Acc).

map(P,[H|T],[H1|T1]) :-
    call(P,H,H1),!,
    map(P,T,T1).
map(_P,[],[]).

empty_partition_sizes([0, 0]).
```

Language Bias: (exclusively_powers)

```

enable_recursion.

% Predicate declarations
head_pred(exclusively_powers,2).
body_pred(is_equal,2).
body_pred(not_empty_list,1).
body_pred(is_connected,2).
body_pred(find_all,3,ho).
body_pred(all,3,ho).

% Types
type(exclusively_powers,(element,element)).
type(is_equal,(element,element)).
type(not_empty_list,(list,)).
type(is_connected,(element,element)).
```

```

type(find_all,((element,element),element,list)).
type(all,((element,element),list,element)).

% Input-output signatures
direction(exclusively_powers,(in,out)).
direction(is_equal,(in,out)).
direction(not_empty_list,(in,)).
direction(is_connected,(in,out)).
direction(find_all,((in,out),in,out)).
direction(all,((in,out),in,out)).

% Constraint the occurrence of predicates
occurFO(is_equal, 1).
occurFO(not_empty_list, 1).
occurFO(is_connected, 1).
occurHO(find_all,1).
occurHO(all, 1).
:- body_pred(X,_),
#count{C,X,V: body_literal(C,X,_,V)} > Z,
occurFO(X,Z).
:- body_pred(X,_,ho),
#count{C,Y,V: body_literal(C,Y,_,V)},
X=@honameparse(Y) } >Z,
occurHO(X,Z).

% Turn off invented predicates in HO
:- invented_ho_used(_,__).
```

Language Bias: (partition)

```

% Predicate declarations
head_pred(partition,3).
body_pred(exclusively_powers,2).
body_pred(not_exclusively_powers,2).
body_pred(same_circuit,2).
body_pred(find_all,3,ho).

% Types
type(partition,(element,list,list)).
type(exclusively_powers,(element,element)).
type(not_exclusively_powers,(element,element)).
type(same_circuit,(element,element)).
type(find_all,((element,element),element,list)).

% Input-output signatures
direction(partition,(in,out,out)).
direction(exclusively_powers,(in,out)).
direction(not_exclusively_powers,(in,out)).
direction(same_circuit,(in,out)).
direction(find_all,((in,out),in,out)).

% Must use the following predicates
:- not body_literal(_,same_circuit,_,(0,1)).
:- not body_literal(_,exclusively_powers,_,(1,0)).
:- not body_literal(_,not_exclusively_powers,
_,(1,0)).
```

Language Bias: (partition_sizes)

```
% Predicate declarations
head_pred(partition_sizes,2).
body_pred(partition,3).
body_pred(size,2).
body_pred(pair,3).

% Types
type(partition_sizes,(element,list)).
type(partition,(element,list,list)).
type(size,(list,element)).
type(pair,(element,element,list)).

% Input-output signatures
direction(partition_sizes,(in,out)).
direction(partition,(in,out,out)).
direction(size,(in,out)).
direction(pair,(in,in,out)).
```

Language Bias: (optimal_partition_sizes)

```
% Predicate declarations
head_pred(optimal_partition_sizes,2).
body_pred(partition_sizes,2).
body_pred(empty_partition_sizes,1).
body_pred(larger_min_size,3).
body_pred(map,3,ho).
body_pred(fold,4,ho).

% Types
type(optimal_partition_sizes,(list,list)).
type(partition_sizes,(element,list)).
type(empty_partition_sizes,(list,)).
type(larger_min_size,(list,list,list)).
type(map,((element,list),list,list)).
type(fold,((list,list,list),list,list)).

% Input-output signatures
direction(optimal_partition_sizes,(in,out)).
direction(partition_sizes,(in,out)).
direction(empty_partition_sizes,(out,)).
direction(larger_min_size,(in,in,out)).
direction(map,((in,out),in,out)).
direction(fold,((in,in,out),in,in,out)).

% Turn off invented predicates in HO
:- invented_ho_used(_,_).
```

Language Bias (optimal_test)

```
% Predicate declarations
head_pred(optimal_test,2).
body_pred(optimal_partition_sizes,2).
body_pred(partition_sizes,2).
body_pred(gate,1).
body_pred(test_point_label,2).

% Types
type(optimal_test,(list,element)).
type(optimal_partition_sizes,(list,list)).
type(partition_sizes,(element,list)).
```

```
type(gate,(element,)).
type(test_point_label,(element,element)).

% Input-output signatures
direction(optimal_test,(in,out)).
direction(optimal_partition_sizes,(in,out)).
direction(partition_sizes,(in,out)).
direction(gate,(out,)).
direction(test_point_label,(in,out)).

% Turn off invented predicates in HO
:- invented_ho_used(_,_).
```

A.8 Human Reference Answers

LLM judges received the following manually composed reference answers when scoring explanations. For the electric circuit domain, each of the three below references corresponds to a task discussed in Section 3.4 and 4.2.

Reference 1 (Electric Circuit)

Find each gate, for which every outgoing cable eventually leads to a point in the circuit (either a gate or a lightbulb). Start from the point in question. For each incoming cable, check if the origin gate has cables to anything other than that point. If not, mark that gate. Now, check for each marked gate whether each outgoing cable leads to either the first point or another marked gate. If so, mark that gate as well. Repeat this, until there are no new gates to add.

Reference 2 (Electric Circuit)

The circuit must be divided in two groups: The first group contains all gates from which each outgoing cable eventually leads to the test point. The second group contains all other gates. To find the first group, start from the test point. For each incoming cable, check if the origin gate has cables to anything other than the test point. If not, add that gate to the group. Now, check for each gate added to the group whether each outgoing cable leads to either the test point or another gate in the group. If so, add that gate to the group as well. Repeat this until there are no more gates to add to the group. Finally, add all gates not in group 1 to group 2.

Reference 3 (Electric Circuit)

Choose the test for which the sizes of the two groups are the most balanced. For each possible test, note the sizes of the two resulting groups. Finally, choose the test, for which the smaller group is largest.

The past USML work (Ai et al. 2021) features the Island

Game, a specially designed two-player game isomorphic to Noughts and Crosses. The Island Game contains three islands, where each island has three territories, and every territory has one or more resources. A player wins when he or she controls either all territories on one island or three instances of the same resource. The nine territories resemble the nine cells in the Noughts and Crosses board. The game is played similarly to Noughts and Crosses through players' turns until one player wins or no player can ever win.

Reference 1 (Game Playing, Island Game)

Identify the move from the board, that can immediately create a triplet of the same resource type. Generally, you should locate resource types from your territories which you have two of the three. Taking the third resource would lead to the WINNING condition.

Reference 2 (Game Playing, Island Game)

Find a move that creates two pairs of different resources and make sure your opponent does not have any pairs. The idea is that you can build two threats instead of one, where one pair can easily be blocked if your opponent is playing optimally. In this case, your opponent could only block one of the two pairs and cannot win immediately, so you taking the third resource would lead to the WINNING condition.

Reference 3 (Game Playing, Island Game)

Choose a move that gives you one pair of resources and advantage over the next two moves. Your opponent would block your attempt to create a triplet of resources. You can follow up by creating two pairs of different types in your next turn. Your opponent could only block one of the two pairs, so you can reach the WINNING condition by taking the third resource.

The past USML work (Ai et al. 2023) explored using ILP-learned logic programs to teach humans computational algorithms. In the paper, the authors focused on helping humans rediscover the concept of merge sort by explaining how to merge. Merge sort is a recursive sorting algorithm based on a divide-and-conquer approach. However, a bottom-up variant of the merge sort algorithm was learned for sorting positive integers. An isomorphic problem was designed to represent integer sequences by the weights of fruits in piles. The task requires arranging fruits from two boxes into a correct sequence on a conveyor belt while operating a balance scale to compare the weights of two fruits.

Reference 1 (Algorithm Discovery, Merge Sort)

You should compare the two items from the boxes to find the lighter item. Then, append the lighter item to the conveyor belt. Afterwards, you should check if all remaining items are from one of the two boxes. In that case, you can safely add the rest of the items to the conveyor belt.

A.9 Explanations Generated from Hand-Crafted Templates

We adopted the textual explanation templates from (Ai et al. 2021, 2023). The parts in angle brackets correspond to where the visualisation would be. The placeholders in curly brackets would be replaced by computing the learned logic programs.

Explanation 1 (Game Playing, Island Game)

<Positive Example Move {move}>
This is a right move. You select this territory and obtain 1 triplet ({resource}).

<Negative Example Other Moves>
This is a wrong move. Contrast: No triplet.

Explanation 2 (Game Playing, Island Game)

<Positive Example Move {move}>
You select this territory and obtain 2 pairs ({resource1}, {resource2}).

<Positive Example Opponent's Move>
Opponent has no pair.

<Negative Example Other Moves Case 1>
This is a wrong move. Contrast: Not enough pair(s).

<Negative Example Other Moves Case 2>
This is a wrong move. Contrast: opponent has 1 pair.

Explanation 3 (Game Playing, Island Game)

<Positive Example Move {move}>
You select this territory and obtain 1 pair ({resource}).

<Positive Example Opponent's Move>
Opponent conquers and prevents you from getting a triplet ({resource}).

<Positive Example Next Move {move}>
You obtain 2 pairs ({resource1}, {resource2}) and

opponent has no pair.

<Negative Example Other Moves>

This is a wrong move. Contrast: Not enough pair(s).

<Negative Example Next Moves>

This is a wrong move. Contrast: Not enough pair(s).

Explanation 4 (Algorithm Discovery, Merge Sort)

Remaining Items are {item1} and {item2}.

<Positive Example Move {item1} then {item2}>

Item {item1} is lighter than item {item2}; append item {item1}. Append remaining item(s): {item2}.

<Negative Example Other Moves>

Item {item1} is lighter than item {item2} SO item {item1} should be appended.

A.10 Prompt Templates for Evaluating LENS Explanations

We used the following prompt templates for the experiment in Section 4.1.

Prompt Templates for coding models

System

You are an expert at explaining Prolog code to people with no programming background. Your job is to write clear, simple, and friendly explanations that help a complete beginner understand what a given Prolog program does. Assume the reader has no technical background.

Follow these steps:

1. Examine the structure and components of the Prolog program. (Note: invented programs are prefixed with inv, and higher-order programs use the ho identifier)
2. Translate the logic into plain English sentences that describe what each part does.
3. Explain the overall purpose of the program and what it's designed to achieve.
4. Avoid jargon.
5. Keep sentences short and accessible.

User

Below is a Prolog program. Explain it to someone with no programming background. Avoid jargon and use simple language.

“prolog
{prolog} “

Prompt Templates for Reasoning Models

System

You are a reasoning assistant tasked with summarising how to apply effective strategies to solve a sequence of related tasks, for a person with no programming background. You are now given a task domain introduction and a set of sample explanations that illustrate the general context for the tasks and how the Prolog predicates function.

Your goal:

Produce a clear and concise explanation of how the task can be solved using the available predicates. Your explanation must be understandable to someone without any knowledge of programming or logic.

Your output should:

- Create a new name for each relevant predicate in solving the task, for reusing later.
- Teach the person by summarising how relevant predicates help achieve the task, using the new names.
- Use plain, accessible language without technical detail or code-like terminology.

DO NOT:

- Use predicate names, Prolog syntax, or variables
- Use jargons, analogies, metaphors, or comparisons

Please read the following:

[Task Domain Introduction]
{domain_context}

[The End of Task Domain Introduction]

[Sample Explanations]
{samples}

[The End of Sample Explanations]

System (Direct Prompting)

You are a reasoning assistant tasked with summarising how to apply effective strategies to solve a sequence of related tasks, for a person with no technical background. You are now given a task domain introduction that illustrates the general context for the tasks.

Your goal:

Produce a clear and concise explanation of how the task can be solved using your strategies. Your explanation must be understandable to someone without any knowledge of technical background.

Your output should:

- Create a new name for each relevant strategy component in solving the task, for reusing later.
- Teach the person by summarising how relevant strategy components help achieve the task, using the new names.
- Use plain, accessible language without technical detail.

DO NOT:

- Use jargons, analogies, metaphors, or comparisons

Please read the following:

[Task Domain Introduction]

{domain_context}

[The End of Task Domain Introduction]

System (No Global Context)

You are a reasoning assistant tasked with summarising how to apply effective strategies to solve a sequence of related tasks, for a person with no programming background. You are now given a set of sample explanations that illustrate how the Prolog predicates function.

Your goal:

Produce a clear and concise explanation of how the task can be solved using the available predicates. Your explanation must be understandable to someone without any knowledge of programming or logic.

Your output should:

- Create a new name for each relevant predicate in solving the task, for reusing later.
- Teach the person by summarising how relevant predicates help achieve the task, using the new names.
- Use plain, accessible language without technical detail or code-like terminology.

DO NOT:

- Use predicate names, Prolog syntax, or variables
- Use jargons, analogies, metaphors, or comparisons

Please read the following:

[Sample Explanations]
{samples}

[The End of Sample Explanations]

User

You are given a task description and an example that illustrates the context for solving it. Identify the relevant predicates and summarise the strategies they represent. Explain how these strategies can be applied in solving the task below. Use the same names for the same predicates as established earlier. Avoid jargon and use simple language.

[Task Description]
{description}

[The End of Task Description]

[Example Type]
{example_type}

[Example]
{example}

[The End of Example]

User (Direct Prompting)

You are given a task description. Identify and summarise relevant strategies. Explain how these strategies can be applied in solving the task below. Use the same names for the same strategy components as established earlier. Avoid jargon and use simple language.

[Task Description]
{description}

[The End of Task Description]

User (Direct Prompting + Local Context)

You are given a task description and an example that illustrates the context for solving it. Identify and summarise relevant strategies. Explain how these strategies can be applied in solving the task below. Use the same names for the same strategy components as established earlier. Avoid jargon and use simple language.

[Task Description]
{description}

[The End of Task Description]

[Example Type]
{example_type}

[Example]
{example}

[The End of Example]

User (No Local Context)

You are given a task description. Identify the relevant predicates and summarise the strategies they represent. Explain how these strategies can be applied in solving the task below. Use the same names for the same predicates as established earlier. Avoid jargon and use simple language.

[Task Description]
{description}

[The End of Task Description]

[The End of Reference Answer]

[The Start of Assistant's Answer]
{answer}

[The End of Assistant's Answer]

Prompt Templates for Judging models

System

Please act as an impartial judge and evaluate the quality of the responses provided by an AI assistant to the user question displayed below. A reference human answer and the assistant's answer will be provided to you. You will review the assistant's response based on the instructions provided to you. Your evaluation should consider factors such as the helpfulness, relevance, accuracy, depth, creativity, and level of detail of the response. Begin your evaluation by providing a short explanation. Be as objective as possible. After providing your explanation, please rate the response on a scale of 1 to 10 by strictly following this format: "[[rating]]", for example: "Rating: [[5]]".

[Instructions for Assistant's Answer]
{instructions}

[The End of Instructions for Assistant's Answer]

User

[Question]
{question}

[The Start of Reference Answer]
{answer_ref}