# Unspoken Logic: Bridging the Gap Between Free-Form and LLM-Autoformalizable Mathematical Proofs

**Chenjun Guo**
University of California, Berkeley
Berkeley, CA, USA
chenjun_guo@berkeley.edu

Manooshree Patel
University of California, Berkeley
Berkeley, CA, USA
manooshreepatel@berkeley.edu

Bjoern Hartmann
University of California, Berkeley
Berkeley, CA, USA
bjoern@eecs.berkeley.edu

J.D. Zamfirescu-Pereira
University of California, Berkeley
Berkeley, CA, USA
zamfi@berkeley.edu

Sarah Chasins
University of California, Berkeley
Berkeley, CA, USA
schasins@cs.berkeley.edu

Gireeja Ranade
University of California, Berkeley
Berkeley, CA, USA
ranade@eecs.berkeley.edu

## Abstract

Autoformalization, the translation of natural language into formal language, is challenging for many reasons. One key issue is that humans express logic in a rich and diverse set of ways, and even when a proof is rigorous from the perspective of other humans, autoformalization can fail. To have AI tools for mathematics be widely accessible, it is important that autoformalization systems be able to accept a wide range of input. In this work, we examine the misalignment between free-form human natural language and the language best suited for autoformalization. We analyze fully-correct student-written proofs to identify recurring sources of ambiguity that hinder formalization, and we develop a natural language pre-processesing system that converts free-form math proofs into a form that leads to more correct autoformalizations. We evaluate this system to identify the ambiguities the pre-processing system can and cannot resolve.

## 1 Introduction

A long-term vision for human–AI collaborative math proving is to develop an end-to-end system that maximizes accessibility and usability for a wide range of people. However, formal theorem provers such as Lean (De Moura et al., 2015), Rocq (Huet et al., 1997) and Isabelle (Paulson, 1994) can be challenging for people to use without extensive training (Shi et al., 2025). Particularly, in educational settings we aspire to have a system where a student can focus on their mathematical understanding, without being burdened by the details of the formal representation. A faithful autoformalizer[1] is a key component of building such a system.

[1]We use the term faithful autoformalizer in the sense of Murphy et al. (2024), where we want to ensure that the formal language follows the user intent, in addition to being syntactically correct.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

While some success has been achieved in autoformalization (Wu et al., 2022; Patel et al., 2025; Wang et al., 2025; Li et al., 2024; Peng et al., 2025; Murphy et al., 2024), there remain many challenges. Human reasoning — especially in casual or learning contexts — can be nonlinear, implicit, and conveyed through loosely structured natural language (NL) (Hjelte et al., 2020). Theorem proving languages are "low-resource" (Magueresse et al., 2020), making the autoformalization task difficult. Many benchmark datasets for autoformalization only contain polished, textbook-style NL and provide only a single NL proof for each problem (e.g., ProofNet (Azerbayev et al., 2023), MiniF2F (Zheng et al., 2021), Putnam-AXIOM (Gulati et al., 2024), Lean-Workbook (Ying et al., 2024)) or are generated through back-translation of formal statements, and lack true human diversity (Gao et al., 2024; Ying et al., 2024; Wang et al., 2024; Patel et al., 2025).

However, NL proofs in real-world settings, especially those of undergraduate students, can be fragmented and non-linear, posing significantly greater challenges for machine interpretation and autoformalization (Zheng et al., 2021). If we want to use LLMs to make machine-checked reasoning a reality for diverse users, we need autoformalization systems that are robust to ambiguous NL input. Which leads us to the question, **what are the ambiguities in free-form NL that prevent correct autoformalization?** In this work, we examine the "***unspoken logic***" in mathematical proofs. We think of unspoken logic as the "mathematics that is not visible on paper, but becomes visible in a computer formalized proof", defined by Riehl (2025). We classify the ambiguities that arise in formalization as a result of this unspoken logic in Section 2 and develop and evaluate methods (Section 3 and 5) to process NL human input such that it is more conducive for accurate autoformalization.

When autoformalized with the same system (LLM with the same prompt), NL proofs which have been disambiguated through our proof-of-concept processing system outperform NL proofs which have not been. Our results, grounded in formalizing real human-written proofs, make a case for processing natural language inputs before autoformalizing via deep learning methods.

## 2 Characterizing Ambiguities in NL Proofs

Figure 1 shows an example of two student written solutions to the exact same question. The colors and annotations highlight how distinct each of these completely correct solutions are. Notably these example solutions follow a similar logical path, but other examples can also show logical deviations.

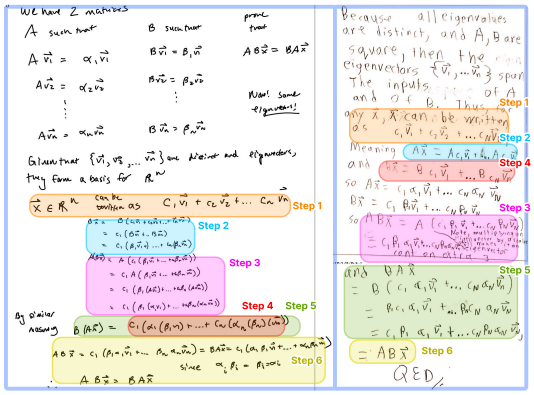


Figure 1: Two student answers of one question using same proof strategy but distinct writing styles

We examined a total of 24 proofs[2] across three undergraduate-level math exam questions (eight proofs per question) that varied in question type, proof strategy, subject matter, and difficulty. All selected student proofs received full marks. Each proof was digitalized using Optical Character Recognition, manually formalized (the student's intent and logic was preserved) by a paper author, and then autoformalized with a baseline system (details in Appendix B). We compared the autoformalized proofs to the manually formalized version of each student proof to extract syntactic and semantic errors introduced in the autoformalized proof.

Across the dataset, the baseline model failed on 46.5% of steps[3] in the first question (47/101), 66.7% on the second (46/69), and 54.2% on the third (77/142) (Full questions in Appendix B.1).

[2]The dataset is small in scale, as each student proof was manually formalized (a time-consuming task), and faithful autoformalization accuracy was manually evaluated to ensure accuracy. Even in this size sample, we were able to see convergence in ambiguity types.

[3]Each step is a NL proof step with the smallest progression unit (which consist of one antecedent, one consequent and the corresponding evidence). Each step consists of one or more Lean lines.

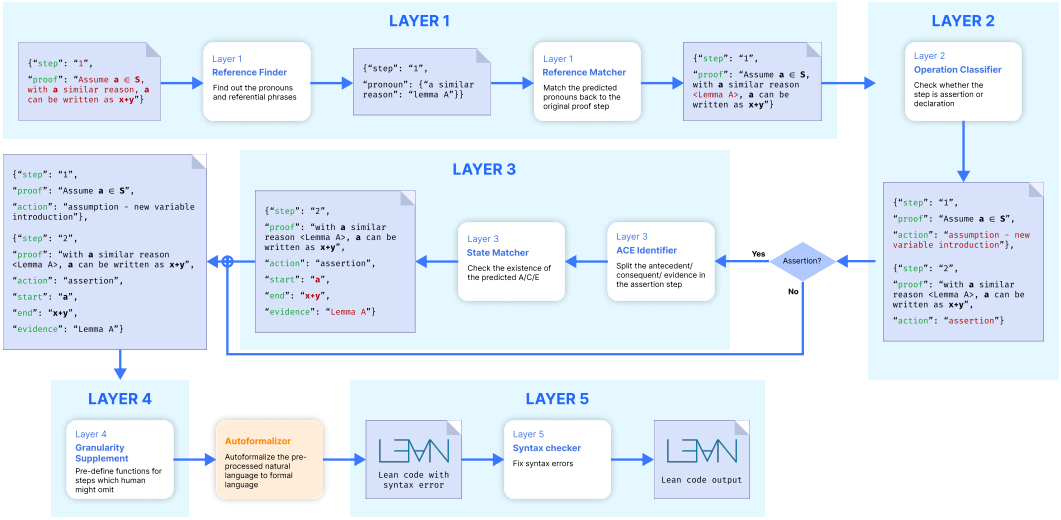


Figure 2: The pre-processing pipeline for autoformalization: We implement four pre-processing layers corresponding to each of the ambiguity categories, and a fifth post-processing layer for syntax correction after the Lean code is generated.

We grouped these failures using reflexive thematic analysis (Braun et al., 2023) (Appendix B.4) to derive a taxonomy of ambiguity types that caused baseline autoformalization failure. Errors are clustered into four ambiguity types:

**Operation ambiguity** (24 cases out of 312 student steps): This ambiguity type captures the confusion around identifying whether a student statement is a *declaration* (declaring a new fact to be true in our context) or an *assertion* (a transformation that brings the current state closer to the goal). A human reader is able to infer this information from contextual knowledge, but our baseline system struggled to do so. Examples of this ambiguity type can be found in Appendix D.2.

**Antecedent/Consequent/Evidence (ACE) ambiguity** (33 cases out of 312 student steps): ACE Ambiguity refers to the cases when a proof step lacks explicit markers for one of the three components in an assertion step. These components are: (1) the antecedent: the starting expression (unsolved goal, hypothesis, or known fact), (2) the consequent: the derived conclusion, and (3) the evidence: the justification linking them. An example can be found in D.3. Human readers rely on conventional formats (e.g., antecedent on the left, consequent on the right, joined by an equals sign) and contextual knowledge. However, the autoformalizer must infer all components based solely on the proof text as none are explicitly marked (see Section 4 and Appendix D.3 for examples.)

**Reference Ambiguity** (11 cases out of 312 student steps): Reference ambiguity captures cases where a proof step uses pronouns (e.g., it, they, them) (Ganesalingam, 2013) or referential phrases (e.g., similarly, as mentioned before). In cases where students used such expressions, instead of clearly stating the referent, the autoformalizer failed to recover the original meaning (examples in D.1).

**Granularity Ambiguity** (50 cases out of 312 student steps): Granularity ambiguity describes cases where students omitted intermediate steps that are required in a formal proof, but not expected in an informal proof. This ambiguity type has been reported on before (Shi et al., 2025) and in our study, common cases included: omitting low-level knowledge or presuppositions or using referential phrases like "*in a similar way*" to point to one of the earlier steps in the proof (examples in D.4).

An additional **Syntactic/Miscellaneous** category captured 60 non-ambiguity errors. For example, `rw [h1] at h2` rewrites `h1` into `h2`, but Lean provide multiple substitution tactics which make the correct choice theorem dependent; these errors stem from syntax, not NL ambiguity. (See Appendix C for the distribution and a more detailed explanation for each category. )

Prior research (Ionescu and Jansson, 2016) has noted that natural language ambiguity hinders novices' understanding of proofs and purposed that using domain-specific languages and controlled natural language can help eliminate such ambiguity. Our findings echoes with them. Detailed related works are discussed in Section A.2.

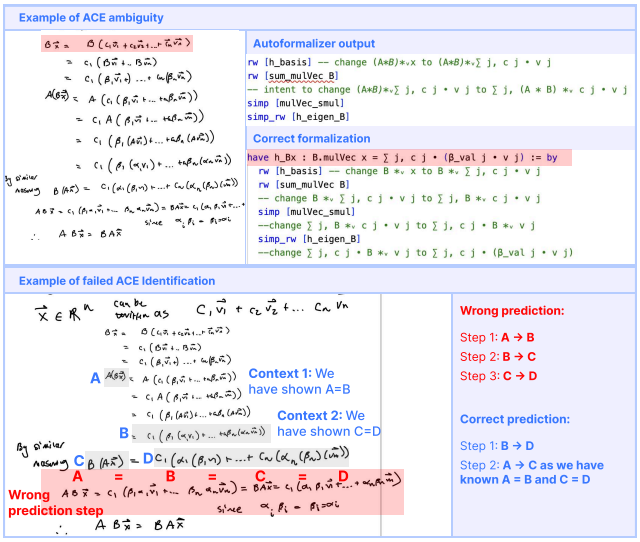


Figure 3: An example of ACE Ambiguity and an example of failed ACE identification.

## 3 Natural Language Pre-processing System

To address these ambiguities, we developed a processing pipeline to transform the student's language into natural language that would be less error-prone when autoformalizing. Our 5-layer system, shown in Figure 2, systematically tackles the aforementioned ambiguities. System input is the digitized student proof in a JSON format, where each entry corresponds to a student proof step. Layer 1 resolves pronouns or referential terms by adding their referent behind the terms. Layer 2 classifies each step as an assertion or declaration (and its sub-types). In Layer 3, assertion steps, are further decomposed into the antecedent, consequent, and evidence (if it is provided). Layer 4 addresses granularity issues in which the LLM is provided with known granularity issues. This is followed by our Autoformalizer, and finally Layer 5, in which Lean syntax errors are fixed. Due to space constraints, more details on the development and architecture of this pipeline can be found in Appendix E.1 and E.2 respectively. We measured the effectiveness of each layer at resolving the ambiguities, the results can be found in F.

## 4 ACE Ambiguity in a Student Proof

As an example of our work, we highlight an example of one ambiguity type, ACE ambiguity, as well as how the pre-processing system handles this ambiguity to improve autoformalization. Examples of other ambiguities and corresponding pre-processing can be found in the Appendix D.

In the baseline autoformalization experiment, we categorized a code segment as exhibiting "an error caused by ACE ambiguity" when its antecedent, consequent, or evidence differed from that in the student's proof. For example (Figure 3), in Question A (Figure 4), the student must show $\mathbf{AB}\vec{x} = \mathbf{BA}\vec{x}$. In the first step, the student writes: "$\mathbf{B}\vec{x} = \mathbf{B}(c_1\vec{v}_1 + c_2\vec{v}_2 + \ldots + c_n\vec{v}_n)$". Instead of choosing either left-hand side (LHS) $\mathbf{AB}\vec{x}$, or right-hand side (RHS) $\mathbf{BA}\vec{x}$ as an antecedent, the student begins with a sub-expression of the LHS: $\mathbf{B}\vec{x}$. In Lean, this shift requires a `have` statement to update the antecedent, but the model omits this. The autoformalized code applies the hypothesis `h_basis` to the wrong target $\mathbf{AB}\vec{x}$ instead of $\mathbf{B}\vec{x}$, producing syntactically valid but semantically incorrect code due to misalignment in the reasoning trajectory.

As a result, our pre-processing system includes an LLM-based layer to predict the ACE for each assertion step. While effective, the model still produces errors, mostly in sequences of continuous equations rather than text-heavy steps. For instance, in Figure 3 a student wrote the continuous equation $A = B = C = D$, where each symbol denotes a distinct term. Without context, this sentence is most readily interpreted as: "*Step 1:* $A = B$*; Step 2:*$B = C$ ...". However, the correct interpretation is: "*Step 1:* $B = D$*; Step 2: using* $A = B$ *and* $C = D$*, infer* $A = C$ *from* $B = D$."

Correct parsing here depends on the proof goal and surrounding context— We believe additional interaction with the user may be necessary to resolve such ambiguities. Appendix F provides a detailed taxonomy with examples for all errors presented in three LLM-based layers.

## 5 Evaluation

We conducted two experiments to evaluate whether the processed and disambiguated NL improved autoformalization accuracy. An autoformalized step is counted as incorrect if it does not compile (syntactic error) or diverges from student intent (semantic error).

1. **Experiment 1**: All proofs were processed from Layer 1 until the autoformalization module in our processing system. Across all proofs, we found 68 erroneous steps (out of 312). Layer 5 (Syntax checker) was applied manually, allowing an LLM one query to try and fix each erroneous step. After applying Layer 5, the error rate decreased to 35/312=11.2% steps. For reference, the baseline autoformalizer's error rate was 170/312=54.5%.
2. **Experiment 2**: We additionally implement our processing system with manual correction of the outputs from Layers 1, 2, and 3. In this much more time-intensive implementation, the error rate after the autoformalization module was 77/312, and dropped to 24/312=7% after Layer 5. This demonstrates that correct execution of the processing steps can offer a dramatic boost in autoformalization accuracy.

We additionally measure the number of successfully compiling lines in each autoformalized proof until the first error (syntactic or semantic). Here, a line is defined as one line of Lean code. We recognize that this metric is imperfect, but it is effective to evaluate the relative performance between the systems and easy to measure. The baseline system averaged 1.6 correct lines per proof, our processing system averaged 24.3 correct lines, and the processing systems with inter-layer manual corrections compiled an average of 22.7[4] lines per proof.

## 6 Conclusion and Future work

A main goal of this paper is to highlight the diversity of mathematical thinking, even for very elementary proofs. Being able to have systems that capture this diversity of expression requires not only better autoformalizers, which has been actively researched, but also proper pre-processing that can remove some input ambiguities as well as well-designed input interfaces that guide users toward clearer expression.

**Input intervention** One potential future work is input interface intervention. Manual review of the pre-processing output provides insight into which ambiguities can or cannot be predicted by LLMs, which therefore demonstrates how input interventions could enhance autoformalization accuracy. In Evaluation Experiment 2, researchers' manual edits exemplify potential interface interventions, such as using the pre-processing pipeline to detect ambiguous terms and prompting users to verify whether the predictions are accurate. Such user-in-the-loop feedback can iteratively refine input, bridging the gap between natural and LLM-autoformalizable expressions.

**Dataset generalization** The current evaluation relies on a relatively limited dataset. Future experiments should include a more diverse range of problem types, such as geometry problems, where reasoning may be presented in multi-modal representations like sketches. The future work should also includes a larger set of student responses.

**Ablation study** This paper does not yet disentangle the individual contribution of each layer in the pre-processing pipeline. Conducting an ablation study would help assess the relative importance of each stage and identify which components most significantly contribute to the improvement in autoformalization accuracy.

[4]We predict that this number decrease slightly due to (1) the manually revised disambiguated NL is cleaner, producing shorter Lean code, and (2) two proofs that compiled more steps in the first experiment encountered an early syntax error in the second.

## References


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

Anthropic. 2025. *System Card: Claude Opus 4 & Claude Sonnet 4*. System Card / Technical Report. Anthropic. `https://wwwâĂŠcdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf` PDF; 120 pages.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433* (2023).

Virginia Braun, Victoria Clarke, Nikki Hayfield, Louise Davey, and Elizabeth Jenkinson. 2023. Doing reflexive thematic analysis. In *Supporting research in counselling and psychotherapy: Qualitative, quantitative, and mixed methods research*. Springer, 19–38.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Anca D Dragan, Kenton CT Lee, and Siddhartha S Srinivasa. 2013. Legibility and predictability of robot motion. In *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 301–308.

Mohan Ganesalingam. 2013. The language of mathematics. In *The Language of Mathematics: A Linguistic and Philosophical Investigation*. Springer, 41.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. 2024. Herald: A natural language annotated lean 4 dataset. *arXiv preprint arXiv:2410.10878* (2024).

Aryan Gulati, Brando Miranda, Eric Chen, Emily Xia, Kai Fronsdal, Bruno de Moraes Dumont, and Sanmi Koyejo. 2024. Putnam-AXIOM: A Functional & Static Benchmark for Measuring Higher Level Mathematical Reasoning in LLMs. In *Forty-second International Conference on Machine Learning*.

Alexandra Hjelte, Maike Schindler, and Per Nilsson. 2020. Kinds of mathematical reasoning addressed in empirical research in mathematics education: A systematic review. *Education sciences* 10, 10 (2020), 289.

Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. 1997. The coq proof assistant a tutorial. *Rapport Technique* 178 (1997), 113.

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

Cezar Ionescu and Patrik Jansson. 2016. Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming. *arXiv preprint arXiv:1611.09475* (2016).

Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Zachary Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving steering and verification in AI-assisted data analysis with interactive task decomposition. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–19.

Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Computational linguistics* 40, 1 (2014), 121–170.

Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M Mitchell, and Brad A Myers. 2020. Multi-modal repairs of conversational breakdowns in task-oriented dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 1094–1107.

Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939* (2024).

Claire Liang, Julia Proft, Erik Andersen, and Ross A Knepper. 2019. Implicit communication of actionable information in human-ai teams. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–13.

Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.

Alexandre Magueresse, Vincent Carles, and Evan Heetderks. 2020. Low-resource languages: A review of past work and future challenges. *arXiv preprint arXiv:2006.07264* (2020).

Patrick Massot. 2024. Teaching mathematics using lean and controlled natural language. In *15th International Conference on Interactive Theorem Proving (ITP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 27–1.

Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. 2024. Autoformalizing euclidean geometry. *arXiv preprint arXiv:2405.17216* (2024).

Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How beginning programmers and code llms (mis) read each other. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–26.

OpenAI. 2025. *OpenAI o3 and o4-mini System Card*. System Card / Technical Report. OpenAI. `https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf` PDF; 33 pages.

Manooshree Patel, Rayna Bhattacharyya, Thomas Lu, Arnav Mehta, Niels Voss, Narges Norouzi, and Gireeja Ranade. 2025. LeanTutor: A Formally-Verified AI Tutor for Mathematical Proofs. *arXiv preprint arXiv:2506.08321* (2025).

Lawrence C Paulson. 1994. *Isabelle: A generic theorem prover*. Springer.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, et al. 2025. Criticlean: Critic-guided reinforcement learning for mathematical formalization. *arXiv preprint arXiv:2507.06181* (2025).

Emily Riehl. 2025. Formalizing invisible mathematics: case studies from higher category theory. *YouTube video*, Big Proof 2025.

Jessica Shi, Cassia Torczon, Harrison Goldstein, Benjamin C Pierce, and Andrew Head. 2025. QED in Context: An Observation Study of Proof Assistant Users. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 337–363.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354* (2025).

Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. 2024. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203* (2024).

Jelle Wemmenhove, Dick Arends, Thijs Beurskens, Maitreyee Bhaid, Sean McCarren, Jan Moraal, Diego Rivera Garrido, David Tuin, Malcolm Vassallo, Pieter Wils, et al. 2022. Waterproof: educational software for learning how to write mathematical proofs. *arXiv preprint arXiv:2211.13513* (2022).

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in neural information processing systems* 35 (2022), 32353–32368.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Lean workbook: A large-scale lean problem set formalized from natural language math problems. *Advances in Neural Information Processing Systems* 37 (2024), 105848–105863.

Shao Zhang, Jianing Yu, Xuhai Xu, Changchang Yin, Yuxuan Lu, Bingsheng Yao, Melanie Tory, Lace M Padilla, Jeffrey Caterino, Ping Zhang, et al. 2024. Rethinking human-AI collaboration in complex medical decision making: a case study in sepsis diagnosis. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–18.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110* (2021).

# A Related works

## A.1 Theorem-provers and autoformalization

In recent years, an increasing number of academic publications have begun to include mechanized proofs which are written with theorem provers, such as Lean (De Moura et al., 2015), Rocq (Huet et al., 1997), and Isabelle (Paulson, 1994). Writing in these formal languages is labor-intensive — even for experienced mathematicians. Through observations of how people construct formal proofs using theorem provers, Shi (Shi et al., 2025) found that users frequently struggle with the granularity required and face challenges in navigating large proof libraries to locate relevant lemmas and definitions. With the advancement of LLMs, there has been growing interest in using them to automate this tedious and technically demanding process.

Deep learning methods have been applied to the task of autoformalization with some success (Wu et al., 2022; Patel et al., 2025; Wang et al., 2025; Li et al., 2024). There are two key challenges. First, theorem proving languages are "low-resource" (Magueresse et al., 2020), making the translation task uniquely difficult. As a result, parallel datasets of informal-formal math statements are often synthetically generated through back-translation of formal statements (Gao et al., 2024; Ying et al., 2024; Wang et al., 2024). The resulting models trained off such data are not robust to the diversity of human input and fail to produce error-free formalizations. Second, even when evaluation accuracy appears high, models struggle to generalize to real-world use. Many benchmark datasets contain overly polished, textbook-style language (e.g., ProofNet (Azerbayev et al., 2023), while others (e.g., MiniF2F (Zheng et al., 2021), Putnam-AXIOM (Gulati et al., 2024), Lean-Workbook (Ying et al., 2024) ) provide only a single natural language version of each problem. These lack the stylistic and structural variety of student-authored or researcher-authored proofs. Additionally, many benchmarks focus on narrow mathematical domains, like Olympiad problems (Zheng et al., 2021; Gulati et al., 2024) that differ significantly from everyday academic or instructional proof writing.

## A.2 Domain Specific Language for mathematical proof construction

Prior research (Ionescu and Jansson, 2016) has explored the use of domain-specific languages to reduce ambiguity in mathematical proofs to support math education. Ionescu and Jansson (2016) observed that students often struggle with classical mathematics due to its context-dependent and ambiguous notation, while performing better in computer science courses that emphasize explicit syntax, thus they introduced the functional programming language Haskell to enforce unambiguous typing and function specification. Their observation on students highlights a similar gap between natural and disambiguated expressions identified in our ambiguity analysis. Furthermore, educational Interactive Theorem Provers (ITPs) such as Verbose Lean (Massot, 2024) and Waterproof (Wemmenhove et al., 2022) employ controlled natural language—a subset of natural language that can be accurately and efficiently processed by a computer—which maintains linguistic readability while preserving rigor of formal representation (Kuhn, 2014). However, since controlled natural languages limit syntax and vocabulary, and require extensive granularity, tools like Verbose Lean are not intended to simplify proof writing, but rather to facilitate students' understanding and transfer of reasoning to paper (Massot, 2024).

## A.3 Challenges in aligning natural language input in human-AI collaborative programming tasks

The misalignment between human-friendly natural language and the kind that LLMs can interpret reliably extends beyond mathematical autoformalization. In AI-assisted programming, multiple studies have highlighted the challenges novice programmers face in learning how to prompt effectively to communicate their intent (Liu et al., 2023; Jiang et al., 2022; Li et al., 2020; Nguyen et al., 2024; Kazemitabaar et al., 2024). Prior work found that only a small portion of naturalistic utterances are effective in guiding code generation, and users often feel they must learn the model's "syntax" (Liu et al., 2023; Jiang et al., 2022; Li et al., 2020). User's input are always underspecified thus LLM need to make assumptions in order to execute user's logic (Kazemitabaar et al., 2024). Moreover, novice programmers must learn to articulate program behavior at an abstraction level that aligns with what the model can interpret Liu et al. (2023); Nguyen et al. (2024).

To address these issues, Liu and Sarkar Liu et al. (2023) proposed grounded abstraction matching, where the user's input is first mapped to a formal representation and then translated back into natural language for user confirmation — mitigating abstraction mismatches. Jiang Jiang et al. (2022) developed an inline AI-assisted programming tool and proposed several design strategies to address ambiguity in user input. Other studies (Xu et al., 2022; Jiang et al., 2022) have also found that users were open to more constrained syntax or design affordances that confirm word recognition, viewing these as more helpful for generating reliable outputs than allowing unrestricted natural language input.

While prior work has largely focused on addressing the abstraction gap between natural language and formal representations as a unified challenge, our approach demonstrates that, within a narrower context, ambiguity can be decomposed into distinct types — and systematically resolved through targeted reformulations. Another key difference lies in the tolerance for ambiguity: in general AI-assisted programming, it is often acceptable to generate multiple interpretations and let users select the correct one. In contrast, certain contexts — including the educational setting of our research — require disambiguation to the extent that the model and the user are fully aligned, with only one plausible interpretation of the input remaining.

### A.4 Broader challenges of natural language interpretation in human–AI collaboration

The challenges in interpreting natural language input also extend to broader domains of human–AI collaboration, for example decision-making in medical contexts (Zhang et al., 2024), collaborative gameplay (Liang et al., 2019), and risk-sensitive tasks (Zhang et al., 2024), as well as task-oriented dialog systems (Li et al., 2020) and robotic interaction (Dragan et al., 2013).

Li and Chen Li et al. (2020) investigated users' challenges in identifying and repairing input ambiguity when interacting with audio-based agents, particularly when such ambiguity led to task failure. Their findings show that AI agents often rely on rigid communication patterns, forcing users to adapt their language rather than the system adapting to the user. In studying repair strategies for conversational breakdowns — defined as failures in the system's understanding of user intent — Li and Chen found that users often struggle to diagnose the cause of these breakdowns, and even when they do, their natural language repairs are frequently ineffective. To address this, Li and Chen designed a system for spoken task-oriented interactions on mobile devices that surfaces the system's internal state using GUI screenshots, helping users recognize misunderstandings and trace their causes.

Liang's work Liang et al. (2019) highlights that human communication often relies on implicit meaning — where listeners are expected to infer intent from context beyond the literal wording. Building on this, Liang developed an AI agent for collaborative video game decision-making that uses pragmatic reasoning to interpret and act on users' implicit communication.

Our research builds on these findings. Like Liang's work Liang et al. (2019), our preprocessing system leverages contextual information to infer the intended meaning behind ambiguous terms. At the same time, it addresses the challenge raised by Li and Chen Li et al. (2020) — helping users identify the specific ambiguities in their input that lead to system failure, so they don't have to guess the cause of breakdowns themselves.

# B Characterizing ambiguities in natural language proofs

## B.1 Exam questions used in the study

**Question A**
Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$. The eigenvalues and eigenvectors of $\mathbf{A}$ are given by $(\alpha_1, \vec{v}_1), (\alpha_2, \vec{v}_2), \ldots, (\alpha_n, \vec{v}_n)$, where all the $\alpha_i$, $1 \leq i \leq n$, are distinct. Similarly, the eigenvalues and eigenvectors of $\mathbf{B}$ are given by $(\beta_1, \vec{v}_1), (\beta_2, \vec{v}_2), \ldots, (\beta_n, \vec{v}_n)$, where all the $\beta_i$, $1 \leq i \leq n$, are distinct.

Prove that:

$$\mathbf{AB}\vec{x} = \mathbf{BA}\vec{x}$$

for any vector $\vec{x} \in \mathbb{R}^n$.

**Question B**
Prove that the set $\mathbf{S}$ is **not** a subspace of $\mathbb{R}^3$, given an invertible matrix $\mathbf{D} \in \mathbb{R}^{3\times3}$.

$$\mathbf{S} = \left\{\vec{c} \in \mathbb{R}^3 \mid \mathbf{D}\vec{x} = \vec{c},\ \vec{x} = x_1x_2x_3,\ x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0\right\}$$

**Question C**
Suppose $x$, $y$ are integers. Prove that if $xy$ and $x + y$ are both even, then both $x$ and $y$ must be even.

Figure 4: Three Proof questions used in this study.

## B.2 Sample student proofs

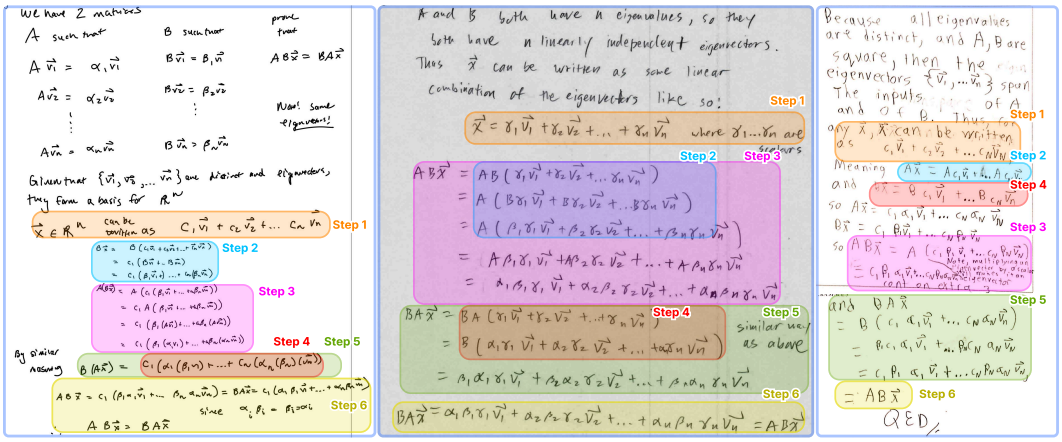


Figure 5: This figure includes 3 student written proofs for 1 exam question. All of them are using the same proof strategy, however, they have diverse ways of expressing the logic. The color highlight similar steps in their proofs.

## B.3 Baseline autoformalizer design

To avoid biasing our ambiguity categories toward any specific language model or prompt format, we tested multiple prompting strategies and model configurations. Following prior work in prompt engineering for autoformalization, we adopted the structure proposed by Murphy and Yang (Murphy et al., 2024) and Patel (Patel et al., 2025) as our primary reference. For **LLM model selection**, we compared Claude Sonnet 4 (Anthropic, 2025) and GPT-4 (Achiam et al., 2023), each on 6 student

proofs (2 per question). GPT-4 showed lower performance, frequently mixing Lean 3 and 4 syntax (23/65 steps), likely due to training bias. Claude Sonnet 4, while not error-free, performed more consistently. To identify the most effective **prompt structure**, we ran five experiments (E1–E5) on 6 proofs (Table: 1), varying two factors: inclusion of a standard staff solution (i.e., correct proof for the question at hand) paired with its Lean formalization, and whether student input was shown all at once or step-by-step.

In E1 and E2, we included both the staff solution and its formalization. This often led to overfitting: the model generated formalizations that mirrored the staff solution rather than reflecting the student's intent. On the other hand, E4 — using a step-by-step format with only the student natural language proof but without the former formalized Lean code — performed poorly, as the model lacked access to earlier variable or function definitions. E3 (whole proof with no staff solution) and E5 (step by step, no staff solution and with correct Lean code for the prior steps) showed similar accuracy, but the step-by-step strategy introduced a confound: accuracy on later steps was inflated when earlier steps in the proof were similar, since the model could learn from previously formalized examples. For instance, if Step 3 (proving $A = B$) fails due to ambiguity, but Step 4 (proving $C = D$) is correctly formalized using similar logic narration with similar ambiguity, it is unclear whether the model genuinely understood Step 4 or simply replicated the earlier pattern.

Based on these findings, we selected the E3 prompt structure — full proof input without staff solutions — as our baseline for final evaluations. The baseline prompt includes: the problem statement (in LaTeX), theorem declaration (in Lean), the complete student proof (in LaTeX), brief autoformalization instructions, and two unrelated example pairs (natural language step and corresponding formalization) to illustrate the output format.

Table 1: Prompt structure and accuracy across five experiments

| **Experiment** | **Q in NL** | **Lean decl.** | **Std. ans** | **Whole proof** | **Prev steps** | **Acc QA** | **Acc QB** | **Acc QC** |
|---|---|---|---|---|---|---|---|---|
| E1: Whole proof with standard answer | Y | Y | Y | Whole proof | — | 2/21 | 2/18 | 9/26 |
| E2: Step by step with standard answer | Y | Y | Y | Current step | Prev. steps | 3/21 | 3/18 | 9/26 |
| E3: Whole proof only | Y | N | N | Whole proof | — | 6/22 | 5/18 | 12/26 |
| E4: Step by step only | Y | N | N | Current step | Prev. steps | 1/21 | 1/18 | 9/26 |
| E5: Step by step (w/ Lean code for prev. steps) | Y | N | N | Current step | Prev. steps +Lean code | 7/22 | 5/18 | 18/26 |

### B.4 Methodology of characterizing ambiguity types

Each student proof was digitized using OCR (GPT-4o (Hurst et al., 2024)) with manual corrections for OCR errors. Proofs were segmented into steps based on visual layout and semantic cues from the handwritten structure, approximating the student's reasoning flow. The segmented proof steps were then processed by the baseline autoformalization model, which attempted to translate each step into Lean.

For all 24 student-written proofs (312 steps in total), the first author manually formalized faithful Lean code that closely reflected the students' intended reasoning for all steps. Then, the first author manually compared the autoformalized version with the correct version and looked for two failure types: (1) code that failed to compile (lack of correctness), and (2) code that compiled but misrepresented the student's intent (lack of semantic fidelity). Both failures stemmed from misalignment between natural language and model interpretation.

[4]This is a reasonable declaration for autoformalization in an educational setting, since the proofs under consideration have known solutions. As we see, this extra information does not actually improve autoformalization performance, since the student solution likely follows a different path.

The dataset is small in scale, as each proof was manually formalized, and faithful autoformalization accuracy was manually evaluated. Formalizing each standard solution took approximately four hours per question, and each student proof required an additional hour to formalize and compare against the autoformalized Lean code to identify autoformalization errors.

Then, we conducted a Reflexive Thematic Analysis (Braun et al., 2023) of the natural language steps which caused autoformalization error in our student proof samples. In accordance with this method, the first author labeled the ambiguities on a subset of these natural language steps, converged on broader ambiguity categories with clear definitions, and then completed the remaining labeling. Through this analysis, we were able to identify common ambiguity patterns that occurred in autoformalization, which we detail in the following sections. The full analysis pipeline is described in Figure 6.

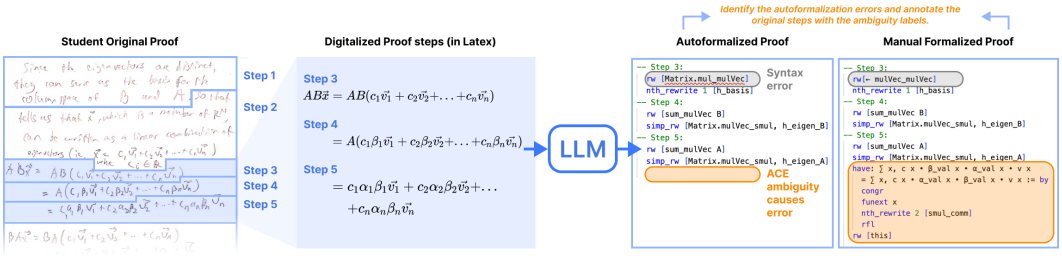


Figure 6: This figure shows the process for characterizing ambiguity types. The handwritten student proof is transformed to LaTeX using OCR (and is manually checked for accuracy). This LaTeX input is then provided to an LLM-based autoformalizer that generates Lean-4 code. We manually compared this code with an accurate human-written formal version of the proof to identify autoformalization errors, which leads to the misalignment categories we identified.

## C Ambiguity distribution across the three questions

As shown in Figure 7, autoformalization errors are attributed to four primary ambiguity categories **Operation Ambiguity** (24 cases), **Antecedent/Consequent/Evidence Ambiguity** (33), **Reference Ambiguity** (11), and **Granularity Ambiguity** (50). An additional **Syntactic or Miscellaneous** category captured 60 cases not attributable to ambiguity. In less than ten cases (excluding overlapping with syntax errors), a single proof step contained two distinct ambiguities, both of which needed to be resolved for successful disambiguation. These cases were double-counted and included in both relevant categories. The distribution of error types was influenced by the nature of each question: QA — primarily used a direct proof strategy and focused on equation manipulation — had more Antecedent/Consequent/Evidence Ambiguity and Granularity Ambiguity, likely due to long, underspecified steps. In contrast, QB and QC — both of which involved contradiction-based reasoning and allowed for multiple proof strategies — showed more failures stemming from Operation Ambiguity.

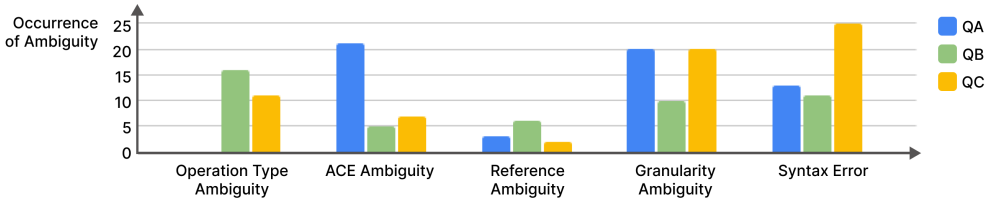


Figure 7: This figure shows the distribution of different ambiguity types on the 3 questions (QA, QB and QC).

# D Examples of each ambiguity types

## D.1 Detailed explanation and an example of Reference Ambiguity

**Reference Ambiguity capture cases where a proof step uses pronouns** (i.e., it, they, them) **or referential phrases** (i.e., in a similar way, by the reasons mentioned before). When students rely on these expressions without clearly restating their referents, the autoformalizer sometimes fails to recover the original meaning, leading to formalization errors.

**Example: Reference Ambiguity leads to incorrect reference prediction.** One student is asked to prove $\mathbf{AB}\vec{x} = \mathbf{BA}\vec{x}$. The proof strategy involves unfolding $\mathbf{AB}\vec{x}$ to $c_1(\beta_1\alpha_1\vec{v_1} + \ldots + \beta_n\alpha_n\vec{v_n})$ and unfolding $\mathbf{BA}\vec{x}$ to $c_1(\alpha_1\beta_1\vec{v_1} + \ldots + \alpha_n\beta_n\vec{v_n})$, showing that both sides are equal. The two unfoldings use a similar sequence of hypotheses. This student firstly wrote in detail the steps to unfold $\mathbf{AB}\vec{x}$, then wrote "*by similar reasoning*" and omitted the intermediate steps to unfold $\mathbf{BA}\vec{x}$ . While clear to humans, the autoformalizer cannot resolve "similar reasoning" and fails to formalize the step. For more details on this example, see Figure 8.

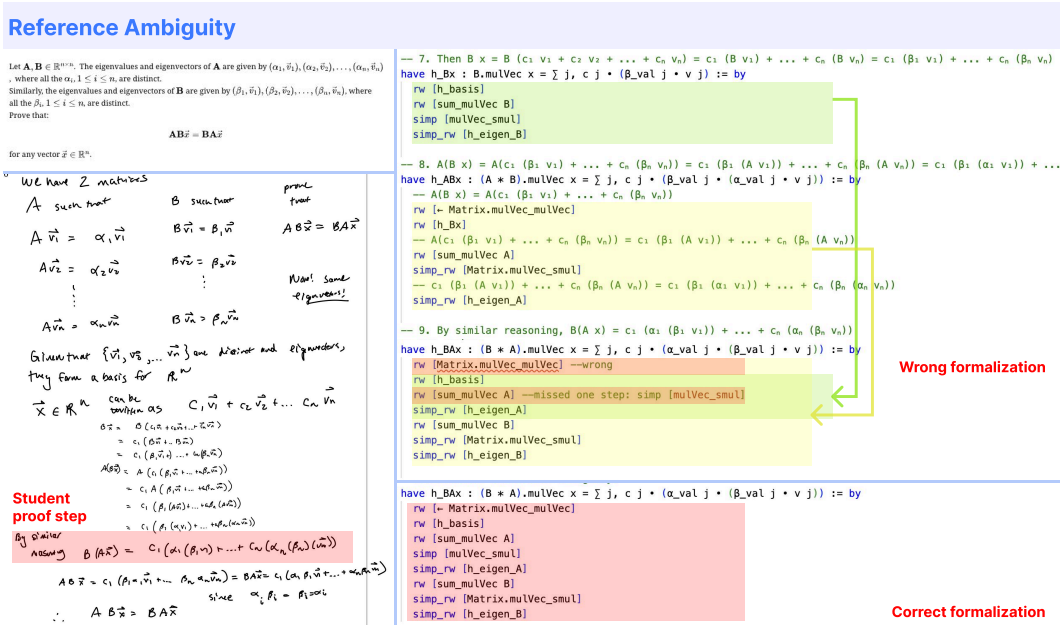


Figure 8: Reference Ambiguity example

## D.2 Detailed explanation and examples of Operation Ambiguity

**Operation Ambiguity captures cases where a proof step lacks explicit keywords indicating whether the step is an assertion or an declaration**, and if declaration, what type of declaration. Mathematical reasoning tasks usually consist of several known facts and one or more unsolved goals. A proof step either makes a declaration – declaring a new fact to be true in our context, or an assertion – a transformation that brings the current state closer to the goal. These transformations may move forward from known facts (forward reasoning) or work backward from the goal (backward reasoning). While a human reader could infer this information from contextual knowledge, the autoformalizer infers a wrong operation type.

From analyzing the written proofs, we categorized 4 different types of declarations. These include: (1) introducing a new variable (e.g., *Let* $n \in \mathbb{R}$), (2) adding a new condition to an existing variable (e.g., *Let* $n > 0$), (3) switching to a new sub-goal (e.g., shifting from proving $a$ is even to proving $b$ is even), or (4) declaring a proof strategy (e.g., indicating the use of contradiction).

**Example 1: Operation Ambiguity leads to declaration misread as assertion.** For QB (Figure 4), in the first step, one student writes "$\mathbf{D}\vec{x} = \vec{c}$," (Figure: 9) which appears to be an assertion, but is in

fact an declaration — introducing new variables $\vec{x}$ and $\vec{c} \in S$. The model, lacking context, treats the original proof step as restating the assumption in the question and omits formalization. In contrast, a correct formalization would explicitly introduce $\vec{c}$ and $\vec{x}$ as declarations. If rewritten in a more autoformalization-friendly form, the step would say *"Assume a new vector $\vec{x}$ and a vector $\vec{c} \in S$ such that $\mathbf{D}\vec{x} = \vec{c}$"*. Without this clarification, the model interprets the original statement as an assertion and outputs: *"This is part of the definition of S, no specific Lean code needed."*

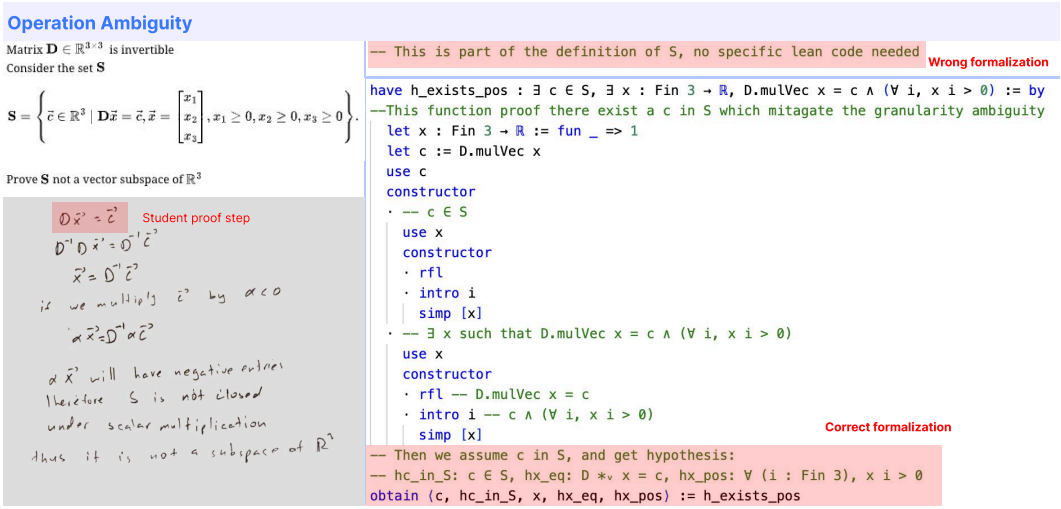


Figure 9: Operation Ambiguity example 1

**Example 2: Operation Ambiguity leads to assertion misread as declaration.** The opposite misclassification — incorrectly interpreting an declaration as an assertion — also occurs. For example, in QC (Figure 4), one student writes: *"To prove: If $x^2$ is even, then $x$ is even. implies if $x$ is odd, then $x^2$ is odd."* (Figure 10) This step can be interpreted in two ways. In one reading, the student is applying contraposition to prove the original statement, in which case the following steps should build arguments for "if $x$ is odd, then $x^2$ is odd." Alternatively, the student may be making a direct assertion. If no further justification is provided, the student likely considers the statement self-evident. In this example, in the following steps, the student continued to prove the contrapositive, suggesting it was a declaration. However, the LLM misinterprets it as an assertion and prematurely complete the proof — before the student's reasoning even begins. In contrast, human graders correctly interpret such ambiguity by drawing on context from the entire proof — something the baseline LLM fails to do.

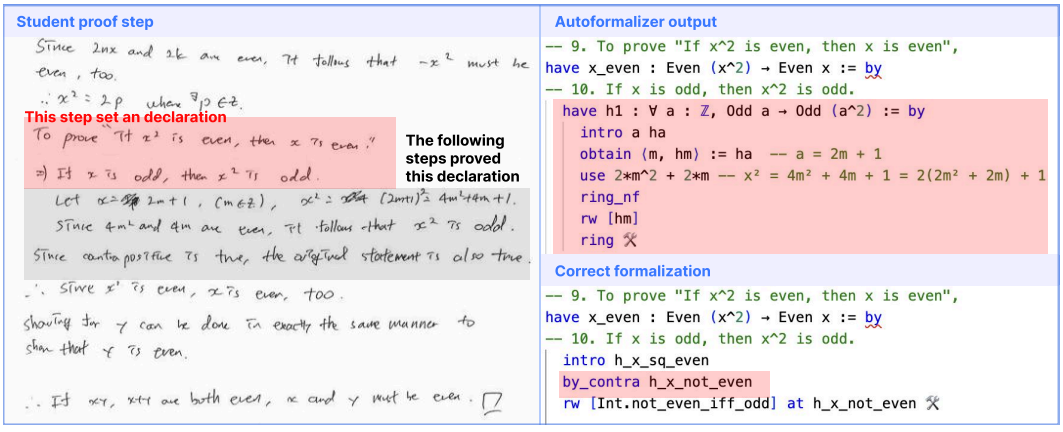


Figure 10: Operation Ambiguity example 2

### D.3 Detailed explanation examples of Antecedent/Consequent/Evidence Ambiguity

In the previous section, we defined an assertion as a transformation that brings the current state closer to the goal. Each assertion involves three components: an antecedent (the state being transformed), a consequent (resulting state), and evidence. **Antecedent/Consequent/Evidence Ambiguity refers to cases where a proof step lacks explicit markers for one or more of these components**, leading to mismatches between the model's formalization and the student's intended meaning. In our dataset students are not expected to identify the antecedent/consequent, and the dataset does not have such labels. Certain cue-words, such as "since" or "because", can indicate evidence, but these words are not consistently used, and using them as indications for evidence is not always reliable. Human readers rely on convention formats (e.g., antecedent on the left, consequent on the right, joined by an equals sign) and contextual knowledge. However, LLMs must infer all components based solely the proof text as none of them are explicitly marked.

**Example 1: ACE Ambiguity leads to misidentification of an assertion's structure.** In the following example (Figure 11), for QA (Figure: 4), what the student must show is $\mathbf{AB}\vec{x} = \mathbf{BA}\vec{x}$. In the first step, the student wrote: "$\mathbf{B}\vec{x} = \mathbf{B}(c_1\vec{v}_1 + c_2\vec{v}_2 + \ldots + c_n\vec{v}_n)$". Instead of choosing either left-hand side $\mathbf{AB}\vec{x}$, or right-hand side $\mathbf{BA}\vec{x}$ as an antecedent, the student begins with a sub-expression $\mathbf{B}\vec{x}$. In Lean, this shift requires a `have` statement to update the antecedent, but the model omits this. The autoformalized code applies the hypothesis `h_basis` to the wrong target $\mathbf{AB}\vec{x}$ instead of $\mathbf{B}\vec{x}$, producing syntactically valid but semantically incorrect code due to misalignment in the reasoning trajectory.

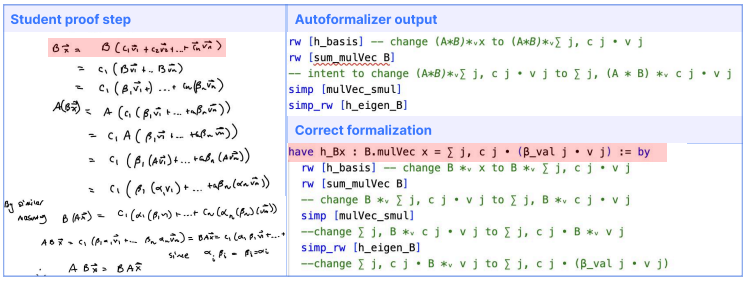


Figure 11: ACE ambiguity example 1

**Example 2: ACE Ambiguity leads to misidentification of the evidence.** This example (Figure 12) illustrates evidence ambiguity: a student begins a step with "*So, ...*", intending to build on the previous result. However, since the human input does not explicitly reference the prior step, the model fails to recognize that it should use the earlier step as evidence.

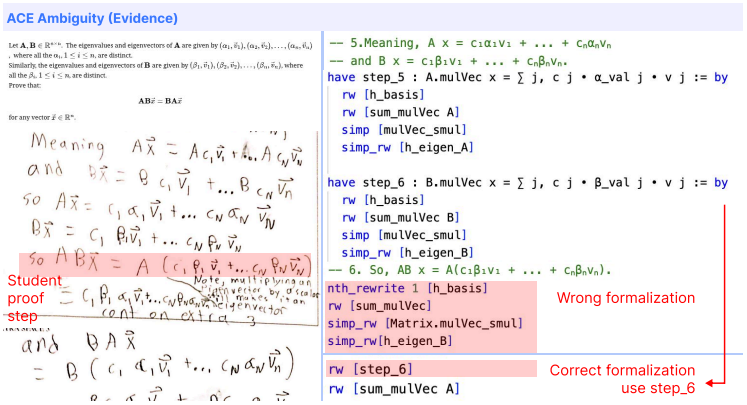


Figure 12: ACE ambiguity example 2

### D.4 Detailed explanation and examples of Granularity Ambiguity

**Example 1: Granularity Ambiguity which caused by skipping of lower level knowledge.** One student writes, "*if $xy$ is even, then at least one of $x$ or $y$ must be even,*" (Figure: 13) without further elaboration. The step is awarded full credit, suggesting that the omission is acceptable to human reader. However, its Lean formalization requires around ten additional sub-steps to justify the claim. In autoformalization-friendly language, such steps must be explicitly detailed, including not just the conclusion but the full logical pathway — similar to the level of precision expected in code.

**Example 2: Granularity Ambiguity which caused by omitting presuppositions.** In QB mentioned before (Figure: 4), one student introduces a new variable $\vec{v_1} \in S$ and states $\vec{v_1} = D\vec{x}$ and $\vec{x} > 0$, without first establishing that such a vector exists in the set $S$. The missing presupposition is: "There exists some vector in $S$ such that $\vec{x} > 0$." While acceptable in human grading, such omissions cause autoformalization to fail.

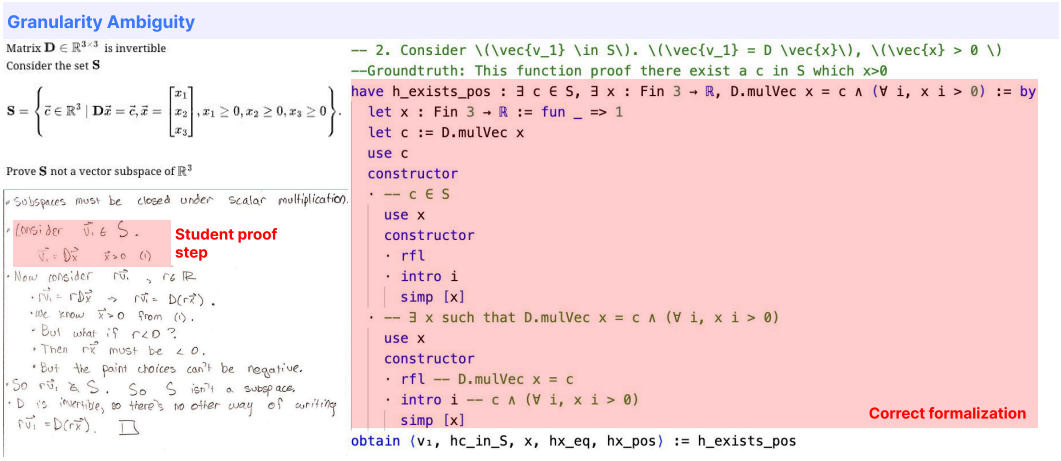


Figure 13: Granularity Ambiguity example

# E Development of the pre-processing system

## E.1 Preliminary Evaluation for constructing each pre-processing layer

To select suitable LLMs for Layers 1–3, we conducted a preliminary evaluation using several widely adopted LLM models on a subset of student proofs (20 proof steps). While we acknowledge that performance could be improved through better prompting, fine-tuning, or developing a task-specific model, our goal was to achieve reasonably strong performance at each layer to enable analysis of common LLM error types within the pipeline, and evaluate whether disambiguation could improve autoformalization accuracy.

**Layer 1.** In a preliminary trial on 20 proof steps containing four explicit references, among the models with best performance, Claude Sonnet 4 correctly resolved all four with no false positives. OpenAI GPT-4 (Achiam et al., 2023) also identified all four but incorrectly labeled five non-referential terms. OpenAI GPT-4o resolved three of four references. Based on these results, Claude Sonnet 4 was selected for the Reference Finder module.

**Layer 2.** Claude Sonnet 4, OpenAI GPT-4, and GPT-4o all performed comparably with the designed prompt, achieving accuracies of 19/20, 20/20, and 19/20, respectively. GPT-4 was selected for full evaluation.

**Layer 3.** In the trial run, LLMs showed limited ability to distinguish antecedents, consequents, and evidences within assertion steps. Among the models with best performance, Claude Sonnet 4 achieved the highest accuracy (63.8%), followed by GPT-o3 (OpenAI, 2025) at 56%, with other models performing worse. Claude Sonnet 4 was selected for full evaluation.

## E.2 Detailed System architecture

**Layer 1: Reference Resolution** This layer takes as input the natural language proof steps that have been digitized, manually corrected, and segmented using the same procedure described in the previous study. This layer consist of two modules, the Reference Finder and Reference Matcher. The Reference Finder is an LLM-based module which identifies pronouns and referential phrases, predicting their corresponding referents from the current or surrounding steps. This layer detects third-person pronouns such as *it, they, them, their* which refer to entities other than the writer ()OED2023, but excludes self-referential pronouns like *"we"* (e.g., "so that we can prove..."), following the distinction outlined by Ganesalingam in The Language of Mathematics (Ganesalingam, 2013). Referential phrases are defined as expressions that refer to prior content without restating it, such as "by a similar reason," "above," or "in the previous step."

The Reference Matcher then mechanically scans the original sentence for predicted pronoun or referential phrase, and inserts their predicted referent in angle brackets ("<>") immediately after the expression. For example, the sentence: "*Similarly, $a$ can be written as $x + y$.*" is transformed into: "*Similarly <Lemma A>, $a$ can be written as $x+y$.*"

**Layer 2: Operation Classification** This layer takes as input the natural language proof steps in which all pronouns and referential phrases have been replaced with their resolved referents. An LLM-based Operation Classifier determines whether each step is an declaration or an assertion. If both operations are present, the step is split into two sub-steps accordingly.

We define two main operation types as follows:

- **Assertion:** A step that progresses the proof by deriving new information from known results, hypotheses, or unsolved goals.
- **Declaration:** A step that sets up necessary context without directly progressing the proof. Declarations include five sub-types:
  - **Introduction of a hypothesis:** Introducing a new condition or case (e.g., "Assume $a + b = c$"; "Case 1: $n < 0$").
  - **Introduction of a new variable:** Declaring a new variable (e.g., "Let $x = a + b$"). Restating existing variables is excluded.

- **Stating the proof goal:** Declaring what is to be proven (e.g., "To prove that. . . ").
- **Declaring the proof strategy:** Indicating the method used (e.g., induction, contradiction, contraposition).
- **Repeating known facts:** Restating known declarations from the problem or prior steps.

The proofs are stored in JSON format, with each proof step containing an operation type as an attribute.

**Layer 3: Antecedent/ Consequent/ Evidence Identification** This layer takes as input the proof steps that have been classified as assertions. It includes two sub modules: an Antecedent/Consequent/Evidence Identifier and a State Matcher. The LLM-based identifier performs two tasks: (1) segments each assertion into smallest progression units, and (2) identifying the antecedent, consequent, and supporting evidence within each unit. And the State Matcher then mechanically verifies that each predicted phrase actually exists in the original proof step and is not hallucinated by the model.

A smallest progression unit is a minimal, self-contained statement that includes one antecedent, one consequent, and one piece of evidence, though the antecedent or evidence may be omitted in actual student writing. The antecedent is defined as the expression from which reasoning begins; it may be the unsolved goal, a part of it, or a known fact such as a hypothesis or lemma, and can appear in the current or previous step. If not present, "NA" is returned. The consequent is the derived conclusion and must appear in the current unit. The evidence is the justification linking antecedent and consequent; it may also come from the current or previous steps, and is labeled "NA" if missing.

The identified antecedent, consequent and evidence(s) are attributes for each steps in JSON.

**Layer 4: Granularity Supplement** The Granularity Supplement is a curated set of Lean code blocks designed to be included in the autoformalization prompt, to fill in reasoning steps often omitted by students but require non-trivial justification in Lean. Each code block is accompanied by documentation on its intended use, supported natural language variations, and guidelines for adapting it to specific proof contexts while maintaining correctness.

**The Autoformalization Layer** The autoformalization prompt is largely identical to the baseline prompt to ensure fair comparison. The only modifications are: (1) replacing the original free-form proof with the Layer 3 pre-processed version, and (2) appending the Granularity Supplement to support commonly omitted but non-trivial reasoning steps.

**Layer 5: Syntax Checker** The generated Lean code is run through the Lean 4 compiler. If a step fails to compile, it is not immediately marked as a failure. Instead, an LLM-based correction module attempts a syntax fix using the proof question, preceding Lean code, the error-prone step, the current unsolved goal, and the compiler error message to generate a corrected version. Notably, the current unsolved goal state is not part of the autoformalization prompt, but is dynamically available in the Lean environment after being compiled through the Lean compiler. It reflects the remaining proof obligation at each step — for example, if the goal is $A = B$ and the previous step proves $A = C$, the dynamic goal state would display $B = C$.

All modifications made by the correction module are recorded and later reviewed to ensure only syntactic errors were addressed. If the step remains uncompiled after one correction attempt, it is marked as a formalization failure.

# F Single layer evaluation: examples of failed predictions across layers

## F.1 Failed predictions in Layer 1 Reference Resolution

In the full evaluation across 24 student proofs (312 total steps), Claude Sonnet 4 achieved a reference resolution accuracy of **97%**, correctly resolving 26 out of 31 references and producing 3 false positives by mislabeling non-referential terms. The following paragraphs provide a detailed analysis of these remaining errors. This gave the following error types:

**References that are successfully detected but incorrectly resolved to wrong referents** . For example, in the sentence: "*Since $D$ is invertible, we know that $\alpha\vec{x}$ is the unique solution for $Dx = c$, then it's not satisfied that $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$,*" the Reference Finder correctly detects "it" but incorrectly resolves it to "*the condition that $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$*" instead of the intended referent, "$\alpha\vec{x}$."

**Undetected references** . The prompt primarily focused on identifying referential terms with positional cues (e.g., "the above step"), but references often appear in more varied forms. For example, one student wrote "$xy = 2k \ldots$ (1)" and later used "*substitute into 1*" to refer back to it — this reference was not detected.

**False positive reference detection** . In 3 cases, the model incorrectly labeled non-referential terms as references. For example, in the sentence "*Contrapositive is true, so the original statement is true,*" the model assigned the referent "*the contrapositive statement proven in the previous steps*" to "Contrapositive." While "contrapositive does not meet the strict criteria for a reference, the inferred referent is semantically equivalent, making the error relatively benign.

### F.2 Failed predictions in Layer 2 Operation Classification

On the complete set of 24 student proofs (312 steps), it achieved an overall accuracy of **94.4%** (295 correctly classified steps). The following paragraphs categorize and analyze the remaining errors.

**Assertion–Declaration Misclassification** : This ambiguity type identified in the qualitative study, cannot be 100% resolved by prediction by the Operation Classifier. The model made five errors misclassifying assertions as declarations, and seven errors in the opposite direction. Of these 12 cases, six led to autoformalization failures in the baseline, while six did not. An example of a **newly introduced error** is: "*Showing for $y$ is even can be done in exactly the same manner as $x$*," which could signal either a strategic declaration or a justified assertion, depending on context. In this case, the sentence constitutes the entire proof, with no further steps, so it should be classified as an assertion. However, the model incorrectly labeled it as an declaration. One example of a **remaining unsolved case** is found in QB (Figure 4), we mentioned before in the Operation Ambiguity example (Figure: 9)that the step $\mathbf{D}\vec{x} = \vec{c}$ contains an ambiguity in distinguishing assertion/ declaration. This sentence is misclassified as an assertion in the pre-processing system.

**Misclassification among declaration subtypes** (5 cases). For instance, for the question "*Suppose $x$ and $y$ are integer. Prove that if $xy$ and $x + y$ are both even, then both $x$ and $y$ must be even*", a student wrote: "*(1) $xy = 2k_1$ (2) $x + y = 2k_2$*" which introduces new variables $k_1$, $k_2$ and simultaneously states new hypotheses. However, the classifier only tagged it as an declaration introduction, overlooking the new variable introduction subtype.

### F.3 Failed predictions in Layer 3 ACE Prediction

**Misidentification of antecedent and consequent in continuous equations:** Errors emerge when students deviate from a clear left-to-right or top-to-bottom structure. Many responses lack linear formatting, making it difficult for the autoformalizer to infer transitions from language or formatting cues alone. For instance, Case 1 in Figure 14, the student wrote in a format of $A = B = C = D$, where each symbol represents a distinct term. The correct interpretation is: "*Step 1: $B = D$; Step 2: using the known equalities $A = B$ and $C = D$, infer $A = C$ from $B = D$.*" Accurate parsing of this reasoning requires awareness of the student's proof goal and the surrounding context — information not available from the sentence structure alone. In Case 2, a student again wrote $A = B = C = D$, but the intended reasoning was: "*Step 1: $A = B$; Step 2: $B = C$; Step 3: conclude $C = D$, given $A = D$*". Despite identical surface structure, these two steps reflect completely different logic flows. However, both steps were predicted as "*Step 1: $A = B$; Step 2: $B = C$; Step 3: $C = D$*" which is wrong in either case. This type of misidentification accounts for 67 of the 88 total errors observed in this layer.

**Misidentification of smallest progression unit:** In 17 cases, the model failed to segment a sentence into minimal reasoning steps. For example, $A = B = C = D$ was incorrectly parsed as *Step 1:* $A = B = C$, combining multiple logical progressions into a single unit.

**Misidentification of evidence:** The LLM generally predicted evidence correctly when guided by cues like "so" or "since," but hallucinated evidence not present in the original sentence. These were filtered out by the State Matcher. In 2 cases, valid evidence was incorrectly discarded because it was semantically correct but not an exact match to the original text.

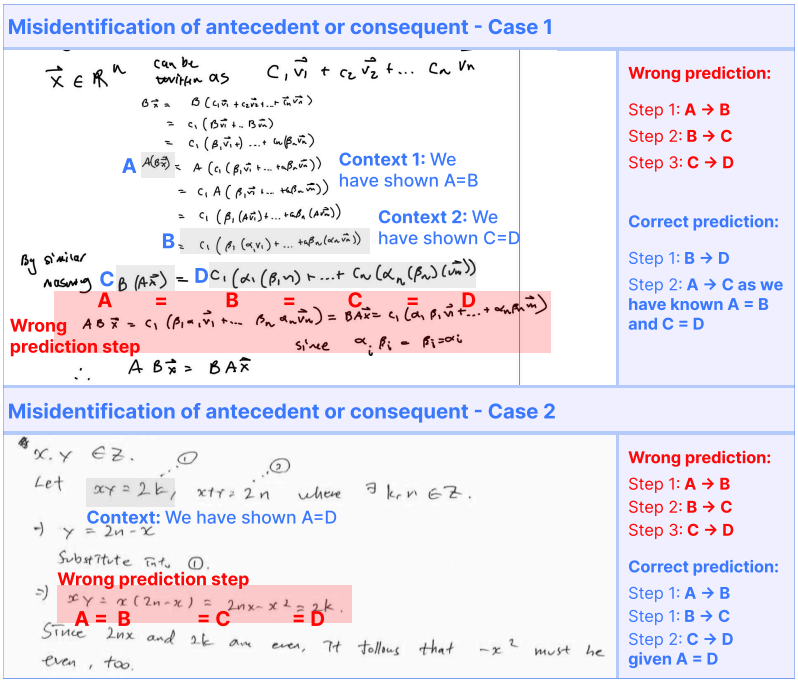


Figure 14: Examples of failed antecedent/consequent/evidence identification: In both cases, the proof step follows the format $A = B = C = D$, but the intended reasoning differs, with different antecedent/consequentstructures. The LLM failed to capture these differences, applying the same interpretation to both cases.

## G Autoformalization performance evaluation

We ran two experiments to evaluate whether disambiguated NL improves autoformalization accuracy. (1) We input the 24 proofs into the pre-processing system (Layers 1–4) and used its raw output (the disambiguated NL with prediction errors) for formalization. We then measured both the number of compiled lines and step accuracy. The resulting code was further processed by the post-processing Syntax Checker, after which we re-evaluated step accuracy. (2) We manually reviewed all outputs from the pre-processing system to produce a fully disambiguated NL without prediction errors. This version was then formalized, and we again measured compiled lines and step accuracy. Finally, we passed this resulting code through the Syntax Checker to obtain modified code and re-assessed step accuracy.

The number of compiled lines is defined as those that compile without errors **and** remain consistent with the student's original intent (verified manually).

The step error rate is calculated by marking any step that fails to compile or diverges from the student's intent as erroneous. Each such step (which may span several lines of code) is manually corrected, and evaluation continues on subsequent steps.

## H Syntax checker

The functionality of this layer has also been implemented in other autoformalization studies Azerbayev et al. (2023); Patel et al. (2025). In our system, since this is not the primary focus of the project, we adopt a relatively constraint approach. An LLM-based model takes as input: (1) the preceding Lean

code, (2) the current Lean code with errors, (3) the error message, (4) the unsolved goal, and (5) a syntax supplement file containing common syntax patterns. The model then generates three candidate fixes in a single call. If one of these fixes resolves the error and the issue falls under the category of syntax errors, the step is marked as successfully corrected by the syntax checker; otherwise, the step is marked as incorrect.

All 53 cases successfully corrected by the syntax checker involved steps where the natural language did not fall into any of the ambiguity types identified earlier. Many of these syntax issues were recurring patterns. Among the 53 corrected cases: a large portion of them were due to either redundant or missing use of tactics (i.e. specific Lean functions) like `simp` or `ring`. Five resulted from incorrect combinations or substitutions of tactics such as `apply`, `simp`, `rw`, `nth_rewrite`, and `rw[← ]`. Some other cases involved confusion between structurally similar expressions, such as `exact hx 0` versus `exact neg_pos.mpr (hx 0)`. The last few emerged from minor missing steps that did not substantially affect the meaning of the proof step.

# I Remaining failed formalizations after the post processing (syntax checker)

Among the 24 proof steps that remained failed to be formalized even after disambiguation, 20 were marked with *granularity ambiguity* in the ambiguity type analysis before. 8 of them involved the **omission of intermediate reasoning steps under the justification of structural similarity** — typically signaled by terms such as "similarly" or "by the same argument." The rest involved other granularity issues. Although our Reference Finder module was able to partially resolve these by expanding the reference (e.g., "similarly" $\rightarrow$ "similarly, referring to Step X to Step Y"), the elided content often does not follow the exact same reasoning process as the referenced steps, they are usually, either using similar lemmas on different variables or proving an analogous goal under different conditions. For instance, one student intends to prove "*If $x$ or $y$ is odd, then $xy$ or $x + y$ is odd*." The student spilt this goal into 3 sub-goals: "*Case 1: assume $x$ is odd, $y$ is even. Case 2: assume $x$ is even, $y$ is odd. Case 3: assume both $x$, $y$ are odd*." For case 1, the student explains in details how to progress from "x is odd and y is even" to the goal "xy or x+y is odd," however, for case 2 and 3 he wrote "By the same argument as above" and wrote a very concise proof. However, as in different cases, the condition for x and y are different, the proof process is not identical in Lean (i.e requires introduction of different variables in the intermediate steps), thus could not copy down the exact same strategy used in the first case to the subsequence cases. This subtle divergence leads to autoformalization failure.

# J Prompt for the LLM based layers

## J.1 Baseline autoformalization prompt (E3 prompt)

```
Here is the proof question you need to formalize:
{proof_question}

Here is Lean Theorem Declaration and definition you can use:
{lean_code_overhead}

Here is the proof written by a student in LaTeX:
{student_latex_proof}

Your task is to formalize this student's proof into Lean code
   ** step by step **. You must reuse the exact same theorem
   declaration, variable names, and assumptions provided in
   the overhead, so that your formalization can compile
   directly with it.

Go through the proof step by step according to the step number,
    for each step
** Each step need to correspond to some formalization code.
   Follow the student steps, the former step
need to be formalized before the later step could start. **

If this step can be formalize into one or more lines of lean
   code, ** write the student's original
natural language in comment first, and then write the lean code
    you formalized **

If this step cannot be formalize into any lean code, write down
    the original sentence in comment and
write in comment why it cannot be formalized

**DO NOT use have statment if the current step is progress on
   the unsolved goal or hypothesis, only
use have when the current step does not progress on the
   unsolved goal or hypothesis, or progress on
part of the unsolved goal**

Here are some examples. NOTE: These are just examples. The
   correct Lean4 code may not necessarily
use the propositions shown in these proofs.

Input:
1. so, a + b + c = a + (b + c) = a + (c + b)
2. Base case: \( n=0 \)  \( 2^(3 \cdot 0 + 1) + 5 = 7 \) is
   divisible by 7, proved.
3. Given that (v1, v2, ..., vn) are distinct and eigenvectors.
Output:
-- 1. so, a + b + c = a + (b + c) = a + (c + b)
rw [add_assoc]
rw [add_comm]

-- 2. Base case: \( n=0 \)  \( 2^(3 \cdot 0 + 1) + 5 = 7 \) is
   divisible by 7, proved.
 zero =>
    norm_num
```

```
-- 3. Given that (v1, v2, ..., vn) are distinct and
   eigenvectors.
-- No corresponding lean.
```

### J.2 Prompt for Layer 1

```
Proof question:
{proof_question}

The natural language proof
{student_proof}

Instructions:
- For each proof step, list all pronouns or references and
   specify what they refer to.
Example: In   Since   the eigenvectors are distinct, they can
   serve as the basis for the column space of B. So that tells
    us x can be expressed as a linear combination of
   eigenvectors,
'they'     'the eigenvectors'
'that'     'Since the eigenvectors are distinct, they can serve
    as the basis for the column space of B'.

- Example pronoun: it, they, them, their, which, that ...
- Example references: by a similar reason, above, former ...
- Do NOT mark pronouns referring to the author (e.g., 'we').
Example:\n
{pronoun_example}
```

### J.3 Prompt for Layer 2

```
You are a mathematics proof expert. Your task is to classify
   the action type for each proof step.
Proof question:
{proof_question}
**Proof steps to analyze**:
{student_proof}
Action type definition: Each proof step can be categorized as
   either an Assumption or an Assertion:
1. **Assumption**: The current step is not making progress in
   the proof, there are 5 sub-types:
**Assumption introduction**: introduce a hypothesis, like:
   assume a+b = c, or set a case for an existing variable,
   like: case 1: n<0
**New variable introduction**: introduce a new variable: let x
   = a+b (Restating variables already in the problem is not
   new introduction).
**State proof goal**: states what needs to be proven, often
   starts with: To prove
**State proof strategy**: states the overarching method (e.g.,
   induction, contraposition, contradiction).
**Repeat known**: repeat the facts that have already been known
    in the question
2. **Assertion**     Progresses the proof by deriving new
   information from the current state/ a subgoal / a
   hypothesis / a known result, sometimes provided with
   evidence of this progression.
**Your task**:
```

```
You are given a list of proof steps in json, analyse the proof
   **step by step** in order.
Each step could contain **multiple action type(s)**. If so,
   split it into sub-steps, each sub-step must have **exactly
   one action type**
Copy the text of the proof exactly. ** Do not modify or delete
   content.**
Output must be a valid JSON list of dictionaries in a format in
    the example below.
{action_type_example}
```

### J.4 Prompt for Layer 3

```
You are a mathematics proof expert. Your task is to identify
   the **start point**, **end point**, and **evidence** for
   each progression in a proof step.
Proof question:
{proof_question}
Proof steps to analyze:
{student_proof}
Definitions:
1. **Start point**: the expression from which the progression
   begins. This may be the current unsolved goal, a portion of
    the goal, or a known statement (e.g., hypothesis, lemma,
   or definition).
2. **End point**: the new expression or result obtained after
   applying the progression.
3. **Evidence**: the justification or reasoning used to connect
    the start to the end.
4. **Smallest progression unit**: a minimal self-contained
   progression consisting of one start, one end, and evidence.
    However, in the given proof steps, units may omit the
   start or evidence.
**Your task:**
1. You are given a list of proof steps in json, analyse the
   proof **step by step** in order. Each proof step is marked
   with an action type: assumption or assertion. Analyze the
   start, end and evidence for the **assertion steps**. Leave
   the assumption steps unchanged.
2. For **assertion** steps, extract one or more smallest
   progression units. Each unit should include one start (may
   omitted), one end, and evidence (may omitted). A few steps
   may contain no complete unit (e.g., a step state:   Since
   we have a =  c   may only serve as evidence). In such cases
   , group it with adjacent steps.
3. In each unit, analyze the start point, end point and
   evidence. **The end point must always be present in the
   current step.** The start point or evidence come from
   earlier steps or may be omitted. DO NOT FABRICATE OR INFER
   NEW CONTENT BEYOND THE PROVIDED PROOF STEPS.
4. For chained equalities (may or may not be grouped in one
   step), the start point is always the immediate left side of
    the equation, and end point is always the right side, like
    in a = b = c, there are two units, unit 1: a = b, start: a
   , end: b; unit 2: b = c, start: b, end: c.
5. Output must be a valid JSON list of dictionaries following
   the format in the examples below.
```

```
6. The content in    <>    explains ambiguous terms like
   pronouns or references before <>. These phrases may also
   serve as start, end, or evidence.
**Examples**
Example 1:
Input:
{\"step\": \"1\",
\"proof\": \"\\( A \\) and \\( B \\) both have \\( n \\)
   eigenvalues, so they both have \\( n \\) linearly
   independent eigenvectors.\",
\"action_type\": \"assertion\"}
Output:
{\"step\": \"1\",
\"proof\": \"\\( A \\) and \\( B \\) both have \\( n \\)
   eigenvalues, so they both have \\( n \\) linearly
   independent eigenvectors.\",
\"start\": \"\\( A \\) and \\( B \\) both have \\( n \\)
   eigenvalues\",
\"end\": \"they both have \\( n \\) linearly independent
   eigenvectors.\",
\"evidence\": \"NA\"}
Example 2:
Input:
{\"step\": \"1\",
\"proof\": \"(a+b)^2 = (a+b)(a+b) = a^2+ab+ab+b^2\",
\"action_type\": \"assertion\"}
Output:
{\"step\": \"1\",
\"proof\": \"(a+b)^2 = (a+b)(a+b)\",
\"start\": \"(a+b)^2\",
\"end\": \"(a+b)(a+b)\",
\"evidence\": \"NA\"},
{\"step\": \"1\",
\"proof\": \"(a+b)^2 = (a+b)(a+b)\",
\"start\": \"(a+b)(a+b)\",
\"end\": \"a^2+ab+ab+b^2\",
\"evidence\": \"NA\"}
Example 3:
Input:
{\"step\": \"3\",
\"proof\": \"=(a+b)(a+b) =a^2+ab+ab+b^2\",
\"action_type\": \"assertion\"}
Output:
{\"step\": \"3\",
\"proof\": \"(a+b)^2 = (a+b)(a+b)\",
\"start\": \"(a+b)^2\",
\"end\": \"(a+b)(a+b)\",
\"evidence\": \"NA\"},
{\"step\": \"1\",
\"proof\": \"(a+b)^2 = (a+b)(a+b)\",
\"start\": \"(a+b)(a+b)\",
\"end\": \"a^2+ab+ab+b^2\",
\"evidence\": \"NA\"}
Explain: the start point of the first progression unit ((a+b)
   ^2) is in the previous step (not fabricated)
```

### J.5 Granularity supplement file

Below is granularity supplement file for QA.

```
-- Codeblock 1:
-- This lemma solves: M.mulVec (    j, f j) =     j, M.mulVec (
   f j) (when M is matrix and f is vector)
-- This lemma also works for: M.mulVec (    j, c j     f j) =
       j, c j     M.mulVec (f j) (when cj are scalars)
lemma sum_mulVec {R : Type*} [CommSemiring R] {n : Type*} [
   Fintype n] -- M * (    j, f j) =     j, M * (f j)
  (M : Matrix n n R) (f : n     n     R) :
  M.mulVec (    j, f j) =     j, M.mulVec (f j) := by
  ext i
  simp [Matrix.mulVec, dotProduct, Finset.sum_apply]
  simp [Finset.mul_sum]
  rw [Finset.sum_comm]


-- Codeblock 2:
-- In hypothesis, we have (h_eigen_A :     j, A.mulVec (v j) =
     _val  j     v j), however, sometimes there is a scalar
    between B and v, like:     j, B *    c j     v j =     j, c
     j      _val  j     v j
-- We use simp [mulVec_smul]
-- Example application:
have:     j, B.mulVec (c j     v j) =     j, c j     B.mulVec (
   v j) := by
  simp [mulVec_smul]
-- This also works when there are multiple scalars:     j, A.
   mulVec (   j     c j     v j)
-- Alternative: congr 1; ext j; rw [mulVec_smul]


--Codeblock 3: user sometime ignore the step (AB)x = ABx,
-- We use rw [    Matrix.mulVec_mulVec]
-- Example application:
have: (A * B).mulVec x = A.mulVec (B.mulVec x) := by
  rw [    Matrix.mulVec_mulVec]
```

### J.6 Syntax checker prompt

```
You are a Lean 4 expert. The current Lean step has an error.
Your task is to provide **three alternative Lean code
   corrections**
that could fix the error, based on the given context.
Former Lean code (context):
{former_code}
The current Lean code with error:
{current_code}
Hypotheses and unsolved goal:
{hypo_goal}
Error message:
{error_msg}
{example}
Instructions:
- Identify why the current Lean step fails.
- Suggest 3 corrected Lean steps that may fix the issue.
- Do not modify the context unless necessary.
- Output in plain text, labeled clearly as Option 1, Option 2,
   Option 3.
Common error:
```

```
Mix of simp, rw, nth_rewrite, and rw[    ], if one of the
   tactic is error, try to replace it with one of these
```

### J.7 Syntax supplement file

Below is syntax supplement file for QA. QB and QC has no stntax supplement file.

```
1. Be careful when using mulVec_smul, when applying mulVec_smul
    in summation
e,g:     (    j, B *    c j     v j) j = (    j, c j      _val
    j     v j) j rw[mulVec_smul] is wrong, you need to use
   simp_rw[mulVec_smul]
e.g2:     (    j, c j     B *    v j) j = (    j, c j
    _val  j     v j) j rw[rw [h_eigen_B]] is wrong, you need
   to use simp_rw[h_eigen_B]
and also be careful for the scalar to apply mulVec_smul
e.g:     (A *    c j      _val  j     v j)  x   = (c j
    _val  j      _val  j     v j)  x   rw[mulVec_smul] is
   wrong, as there are two scalars in between, you need to use
    it twice rw[mulVec_smul] rw[mulVec_smul]

2. Be careful in distinguish mul_comm and smul_comm
e.g (c j      _val  j      _val  j     v j)  x   = ( _val  j
        _val  j     c j     v j)  x   should not use rw [
   mul_comm ( _val  j) ( _val  j)], should use rw [smul_comm (
    _val  j) ( _val  j)]

3. if you see goal like this     ( _val  j     c j     v j)
    x   = (c j      _val  j     v j)  x   or this     (    x,
   (c x *  _val  x)     v x) j = (    x, ( _val  x * c x)
   v x) j
where the only unsolved goal is the order of multiplication,
   try to use
simp
ring
to solve this
```

### J.8 The autoformalization prompt for the disambiguated NL

```
"You are a Lean 4 expert. Your task is to formalize a
   students proof into Lean 4 code. \n"
"Problem statement:\n"
f"{proof_question}\n\n"
"Theorem declaration and assumptions:\n"
f"{lean_code_overhead}\n\n"
"Students proof to formalize:\n"
f"{json_proof_steps_str}\n\n"
"Common micro-steps you may reference when filling gaps:\n"
f"{granularity_code}\n"
"Instructions:\n"
"1. The students proof is a JSON list of steps. Each step
   may include an action type, a start point, an end point,
   and possibly evidence. Formalize in order, **step by step
   .** totally follow students intent\n"
"2. If the action type is an assumption, this step introduces a
    variable or assumption to the proof. Write Lean to
   introduce it or adjust the goal (e.g., intro, cases/rcases,
    by_cases, by_contra, set, have, let). If the assumption
```

```
   repeats known information, produce no code for that step.\n
   "
"3. If the current entry does not have an   action   type
   column, this step is an assertion step, which makes
   progress for the current proof. For every assertion, write
   Lean code to progress the proof **from the start point to
   the end point**. If the student provides evidence, use the
   method in their evidence if possible.\n"
"4. If the end point is already present in the Lean environment
    or local context, output: --no formalization.\n"
"5. If the start point matches the current unsolved goal,
   directly formalize the step to progress toward solving it.
   (See example 3)\n"
"6. If the step does not directly operate on the unsolved goal,
    but proves a locally true fact useful later (part of the
   LHS/RHS of the unsolved goal or a fact from hypo or other
   theorem), **formalize it with    have    and prove it from
   the start to the end point**. (See example 1,2)\n\n"
"Example 1: \n"
"Current unsolved_goal:     7     14 * a + 21 * b\n"
"Input: \n"
"{\\\"proof\\\": \\\"14*a is dividable by 7\\\",\\n"
"\\\"start\\\": \\\"14*a\\\",\\n"
"\\\"end\\\": \\\"14*a is dividable by 7\\\"}\n"
"Output:\n"
"have h1 : 7     14 * a := by\n"
"   use 2 * a\n"
"   Ring\n\n"
"Example 2:\n"
"Current unsolved goal:     let S := {c |     x, D *    x = c
          (i : Fin 3), x i     0};   IsSubspace   (Fin 3
      ) S\n"
"Input:\n"
"{\\\"proof\\\": \\\"Assume \\\\(S\\\\) a subspace of \\\\(R^3
   \\\\)\\\",\\n"
"\\\"action_type\\\": \\\"assumption- assumption introduction
   \\\"}\n"
"Output:\n"
"intro S\n"
"by_contra h_subspace"
```