COOL: EFFICIENT AND RELIABLE CHAIN-ORIENTED OBJECTIVE LOGIC WITH NEURAL NETWORKS FEEDBACK CONTROL FOR PROGRAM SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Program synthesis methods, whether formal or neural-based, lack fine-grained control and flexible modularity, which limits their adaptation to complex software development. These limitations stem from rigid Domain-Specific Language (DSL) frameworks and neural network incorrect predictions. To this end, we propose the **Chain of Logic (CoL)**, which organizes synthesis stages into a chain and provides precise heuristic control to guide the synthesis process. Furthermore, by integrating neural networks with libraries and introducing a **Neural Network Feedback Control (NNFC)** mechanism, our approach modularizes synthesis and mitigates the impact of neural network mispredictions. Experiments on relational and symbolic synthesis tasks show that CoL significantly enhances the efficiency and reliability of DSL program synthesis across multiple metrics. Specifically, CoL improves accuracy by 70% while reducing tree operations by 91% and time by 95%. Additionally, NNFC further boosts accuracy by 6%, with a 64% reduction in tree operations under challenging conditions such as insufficient training data, increased difficulty, and multidomain synthesis. These improvements confirm COOL as a highly efficient and reliable program synthesis framework.

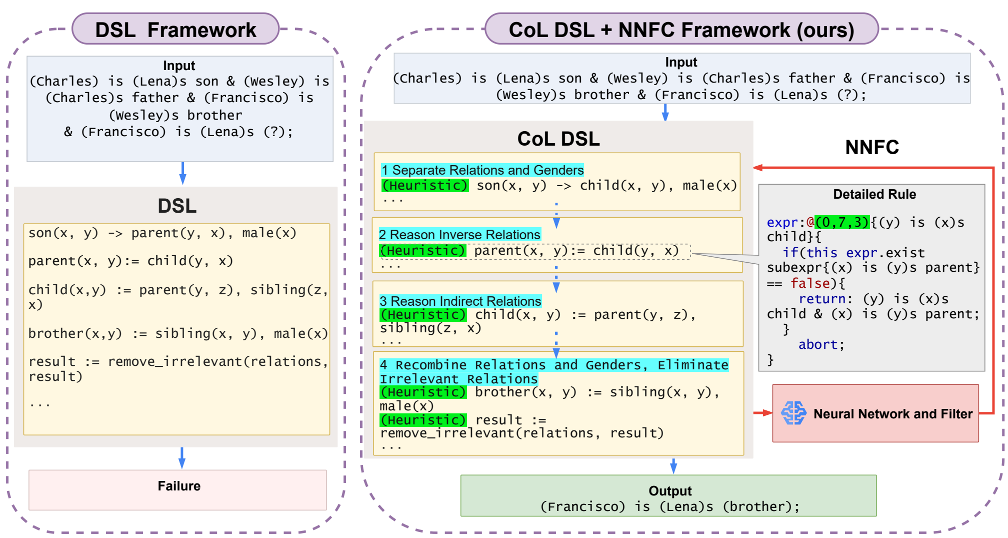

Figure 1: Chain-of-Logic (highlighted part) organizes the rule application into a structured sequence, enhancing the Domain-Specific Language (DSL) framework's ability to handle complex tasks. The Neural Network Feedback Control mechanism (red path) utilizes data during synthesis to improve the performance of the synthesis process dynamically.

1 Introduction

Program synthesis is becoming increasingly important in computer science for enhancing development efficiency Gulwani et al. (2017); Jin et al. (2024). Despite the effectiveness of current state-of-the-art methods in dealing with simple tasks, the complexity of modern software demands more advanced and sophisticated approaches Sobania et al. (2022).

To address these challenges, an effective solution must offer programmers fine-grained control and flexible modularity in the synthesis process Groner et al. (2014); Sullivan et al. (2001). First, fine-grained control tailors the synthesis path to specific tasks, ensuring the interpretability of the synthesis process. Secondly, flexible modularity enhances reusability and guarantees the quality of the entire program by ensuring the correctness of the modules Le et al. (2023).

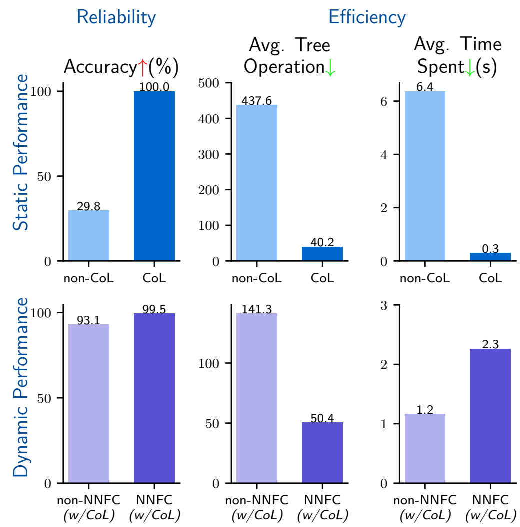

Figure 2: Performance Enhancements with CoL and NNFC. The CoL DSL surpasses non-CoL DSL in all metrics. While NNFC increases computation time due to neural network calls, it significantly boosts accuracy in dynamic experiments, enhancing reliability.

However, these principles are often overlooked in current state-of-the-art program synthesis methods. For example, symbolic approaches such as SyGus Alur et al. (2013), Escher Albarghouthi et al. (2013), and FlashFill++ Cambronero et al. (2023) struggle to scale to complex tasks because their traversal-based Domain-Specific Language (DSL) framework lacks fine-grained control. A compensatory strategy involves using neural networks for guidance or search space pruning, as seen in projects such as Neo Feng et al. (2018), LambdaBeam Shi et al. (2023a), Bustle Odena et al. (2020), DreamCoder Ellis et al. (2023), and Algo Zhang et al. (2023), but the control logic remains disconnected from the programmer. On the other hand, LLM-based projects like CodeGen Nijkamp et al. (2022), CodeX Finnie-Ansley et al. (2022), and Code Llama Roziere et al. (2023) allow programmers to control synthesis through prompt interactions. However, they lack modularity, as all tasks rely on the same LLM, making the logic vulnerable to biases in training data and leading to subtle errors that require manual verification. In summary, there is an urgent need for fine-grained control and flexible modularity to ensure the efficiency and reliability of these methods when tackling complex synthesis tasks.

In this paper, following the principles of fine-grained control and flexible modularity, we present **COOL (Chain-Oriented Objective Logic)**, a neural-symbolic framework for complex program synthesis. At the core of our approach, we introduce the **Chain-of-Logic (CoL)**, which integrates the functions of the activity diagram to enable fine-grained control Gomaa (2011). As illustrated in Figure 1, programmers can precisely organize rules into multiple stages and manage control flow using heuristics and keywords. Additionally, we leverage neural networks on top of CoL to dynamically fine-tune the synthesis process. For this purpose, we introduce **Neural Network Feedback Control (NNFC)** Turan & Jäschke (2024), which enhances future synthesis by learning from data generated during synthesis and suppresses neural network incorrect predictions through filtering. To ensure modularity, each neural network is bound with a specific CoL DSL, stored in separate library files for clear isolation and easy reuse. Thus, through the combination of CoL and NNFC, COOL achieves high efficiency and reliability when tackling complex synthesis tasks.

We conduct static experiments (constant domain and difficulty tasks, using pre-trained neural networks without further training) and dynamic experiments (mutative domain and difficulty tasks, where neural networks are created and continuously trained during the experiment) to evaluate the impact of CoL and NNFC on program synthesis. Figure 2 illustrates the significant improvements achieved by CoL and NNFC: In static experiments, CoL improves accuracy by 70%, while reducing

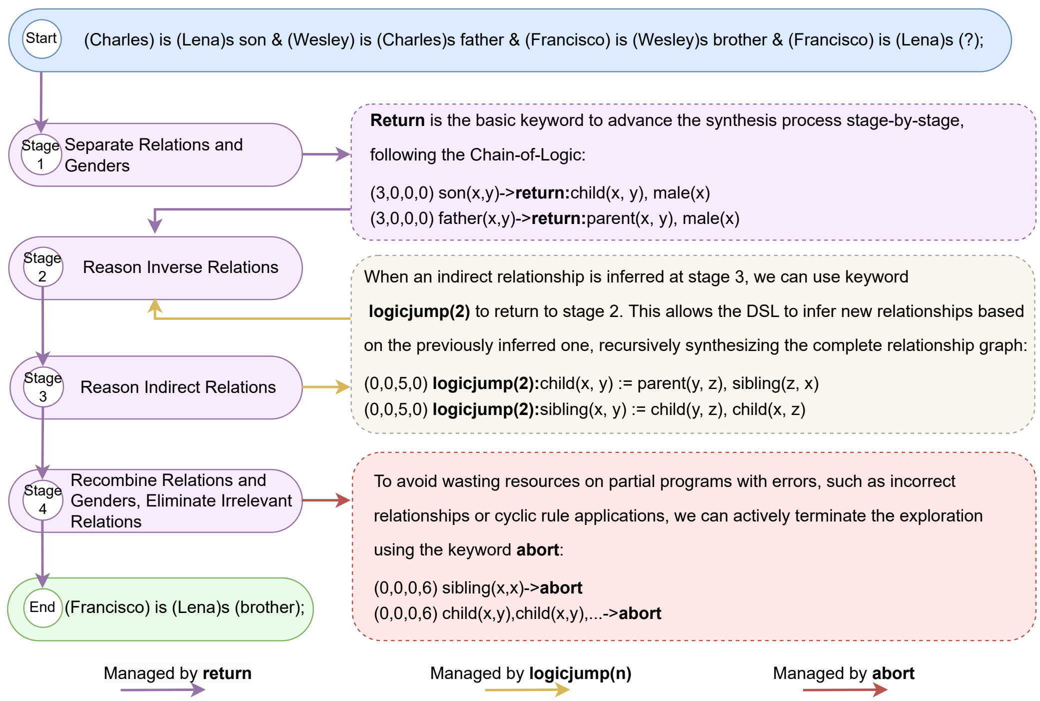

Figure 3: Chain-of-Logic. In this illustrative CoL DSL, each node represents a stage or activity where a set of rules can be applied to generate partial programs. The flow between stages is managed by keywords **return**, **logicjump(n)**, and **abort**, allowing for the implementation of complex control flow in program synthesis.

tree operations by 91% and time by 95%. In dynamic experiments, NNFC further increases the accuracy by 6%, with a 64% reduction in tree operations. The results underscore that achieving fine-grained control and flexible modularity can greatly improve efficiency and reliability in DSL program synthesis.

The contributions of our work are as follows:

1. We propose the **Chain-of-Logic (CoL)**, which enables fine-grained control in complex program synthesis by structuring rule applications into distinct and manageable stages.
2. We further introduce **Neural Network Feedback Control (NNFC)**, a dynamic correction mechanism for CoL that continuously learns from the synthesis process, ensuring modularity by pairing neural networks with specific CoL DSLs.
3. We present **COOL**, an efficient and reliable neural-symbolic framework for complex program synthesis, combining the strengths of CoL and NNFC to achieve fine-grained control and flexible modularity in DSL-based synthesis.

2 Method

In this section, we detail the implementation of CoL and NNFC, outlining the principles that ensure high efficiency and reliability for complex program synthesis tasks.

2.1 Chain-of-Logic (CoL)

Activity diagrams, widely used in software engineering, effectively describe how an initial state transitions to a final state through multiple stages. This feature aligns with the DSL-based program

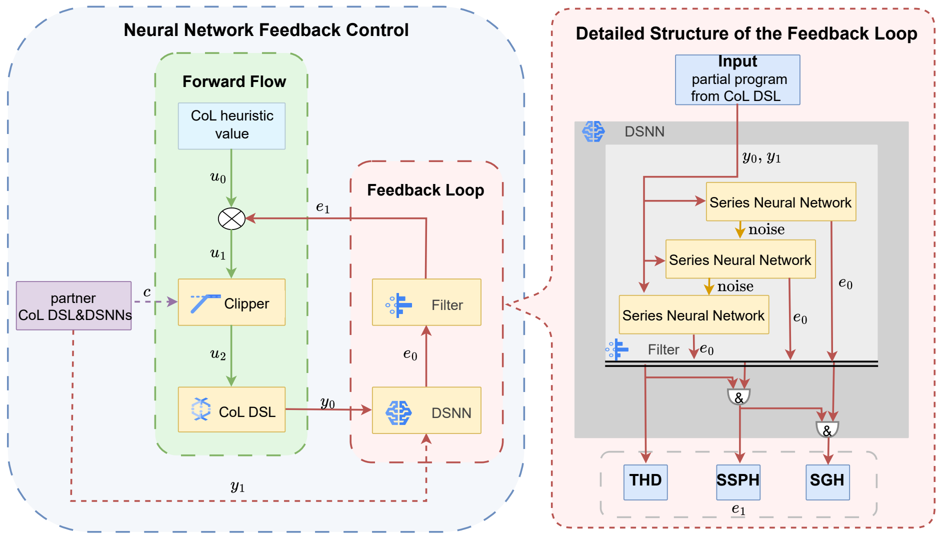

Figure 4: Neural Network Feedback Control. The left side illustrates the complete control loop of NNFC. In the forward flow (green path), heuristic values u guide the synthesis process as control signals. In the feedback loop (red path), the **DSNN (Domain-Specific Neural Network, the neural network paired with a DSL)** generates initial error signals e_0 from partial programs y. These singals are then filtered to produce high-quality error signals e_1, which adjust the initial heuristic values u_0. In multidomain synthesis, the CoL DSL and DSNN from the self-domain use partner domain information (dashed path) to clarify tasks and avoid competition, ensuring modularity. The right side details the feedback loop: The DSNN comprises multiple neural networks coupled in series via noise signals, with each network generating its own error signal e_0, then these signals with large discrepancies are filtered, retaining the final high-quality error signals e_1.

synthesis process. A DSL, defined as a context-free grammar, converts partial programs with nonterminal symbols into complete programs by applying given rules. However, as the rule set grows, DSL becomes inefficient in exploring partial programs. To enhance the efficiency of DSL, the Chain-of-Logic, drawing inspiration from activity diagrams, organizes rule applications during synthesis into a sequence of manageable stages, as illustrated in 3.

CoL improves the control flow of the DSL with two key features: *heuristic vectors* and *keywords*. Heuristic vectors specify the stages where rules apply and their corresponding values. For example, in Figure 1, a rule with the heuristic vector `(0,7,3)` is applicable in stages 2 and 3 with heuristic values of 7 and 3, respectively. These vectors form the core of CoL's control flow.

Second, CoL introduces three keywords—`return`, `logicjump(n)`, and `abort`—to dynamically choose the next stage during synthesis:

1. **return**: Ends the current rule, staying within current stage or advancing to following stages.
2. **logicjump(n)**: Jumps directly to the stage n, enabling branching and loops within CoL.
3. **abort**: Terminates the current synthesis branch, pruning the search space.

In summary, CoL provides fine-grained control through heuristic vectors and keywords. This structured and detailed approach enhances the efficiency of DSL synthesis.

2.2 Neural Network Feedback Control (NNFC)

While CoL enables programmers to fine-tune the synthesis process, the control flow may lack detail or vary by task. To this end, **Neural Network Feedback Control (NNFC)** dynamically refines control flow through feedback from neural networks, improving precision and adaptability. However, neural networks present the risk of generating incorrect predictions, threatening reliability.

Therefore, a robust control flow in NNFC is crucial to ensuring overall performance. As illustrated in Figure 4, NNFC enhances the CoL DSL in the following ways: In the forward flow, the Clipper prioritizes control signals aligned with DSNN guidance by capping any inconsistent signals, while the CoL DSL applies rules based on the adjusted heuristic values. Meanwhile, in the feedback loop, the DSNN generates error signals from partial programs across domains. To suppress the impact of mispredictions, the Filter refines these signals before they influence the forward flow.

The quality of the signals generated in the feedback loop directly determines the effectiveness of NNFC. If the error signals are of poor quality, NNFC may not only fail to provide additional improvements but also degrade CoL DSL performance. We ensure the error signal quality through an inner coupling structure within DSNN. As shown in Figure 4 (right), during synthesis tasks, DSNN processes partial programs using a series of sequentially connected neural networks. Each neural network takes both the partial programs and intermediate results from the preceding neural network as input, generating its own predictions. When errors occur in earlier networks, they propagate downstream as noise signals, amplifying at each stage. The difference in the outputs between these neural networks is positively correlated with the accumulated error. To mitigate this, we set a threshold to filter out signals with a significant difference in outputs. Finally, DSNN uses passed signals to generate multi-head outputs to fine-tune the forward flow:

1. **Task Detection Head (TDH)**: Improves modularity by determining whether the partial program contains components that the CoL DSL can process.
2. **Search Space Prune Head (SSPH)**: (Active when TDH is true) Evaluate the feasibility of synthesizing the final complete program from the current partial program, and CoL DSL will avoid exploring infeasible spaces.
3. **Search Guidance Head (SGH)**: (Active when both TDH and SSPH are true) Guides the CoL DSL in applying the most promising rules to the partial program.

By adopting filtering and multi-head outputs, the feedback loop delivers high-quality error signals to the forward path, ensuring that NNFC enhances the synthesis process on top of CoL.

3 Experiments

We conduct the experiments in two stages to evaluate the improvements introduced by CoL to DSL and to assess how NNFC further enhances performance. First, we carry out static experiments under fixed conditions, including task domain, difficulty level, and neural network. These controlled conditions allow us to accurately measure CoL's impact on performance. Next, we proceed with dynamic experiments, where conditions vary throughout. This dynamic setup evaluates NNFC's ability to improve reliability under changing situations.

3.1 Experimental Setup

Improvements of DSL by CoL and NNFC is evaluated across benchmarks using various metrics.

Benchmarks. We evaluate CoL and NNFC using relational and symbolic tasks with varying difficulty levels, as detailed in Table 1. Specifically, the relational tasks are drawn from the CLUTRR Sinha et al. (2019) dataset, where the goal is to synthesize programs that capture specific target relationships based on human common-sense reasoning. In contrast, the symbolic tasks are generated by GPT Achiam et al. (2023). They involve synthesizing standard quadratic equation programs from non-standard quadratic forms by performing manual calculation steps. Although

Table 1: Benchmark configurations. Relational benchmarks are divided into easy and difficult groups based on the number of relationship edges, while symbolic benchmarks are based on the number of nodes in the tree.

Benchmark Type	Difficulty Level A	Difficulty Level B
relational	300 tasks with 3 edges	200 tasks with 4 edges
symbolic	300 tasks with around 5 nodes	200 tasks with around 9 nodes

these tasks are simple for humans, they serve as a straightforward demonstration of how fine-grained control, derived from programmer expertise, can significantly improve program synthesis efficiency.

Metrics. Besides accuracy, we also focus on the following points: **(1) CPU Overhead** is assessed by the number of tree operations required for synthesis. **(2) Memory overhead** is assessed by the number of transformation pairs (a partial program paired with the rule to be applied)[1]. **(3) GPU Overhead** is measured by the number of neural network invocations. **(4) Time overhead** is referenced by the actual time spent on program synthesis tasks. **(5) Filtering Performance** is evaluated by the attenuation ratio of invalid to passed neural network predictions.

Chain-of-Logic. We utilize the CoL approach to enhance DSL by making the synthesis process more in line with human problem-solving strategies. For relational tasks, by mirroring the way humans typically reason about family relationships, CoL organizes the synthesis process into stages illustrated in Figure 3. For symbolic tasks, CoL structures the DSL to follow the manual quadratic equation simplification strategy, with stages such as expanding terms, extracting coefficients, permuting terms, and converting equations to standard form. The specific CoL DSL configurations are shown in Table 2, where the significant differences in DSLs highlights the generality of CoL.

Table 2: CoL DSL configurations. The DSL for relational benchmarks has a limited search space and shorter CoL, facing challenges from numerous production rules leading to larger trees. Conversely, the DSL for symbolic benchmarks offers an unlimited search space with a longer CoL, but the many permutation rules increase the risk of cyclic rule applications.

Benchmark	**Rules**					**Length of CoL**
	Total	**Production Rules**	**Reduction Rules**	**Recursive Rules**	**Permutation Rules**	
relational	40	36	2	16	0	4
symbolic	55	17	26	3	11	7

Groups. We use multiple groups to comprehensively evaluate CoL and NNFC (as shown in Table 3). First, in static experiments, we evaluate CoL by comparing DSL groups with and without CoL enhancements. Second, to isolate the impact of heuristic vectors—both as guides and as structuring tools for rule application—we create groups enhanced only by heuristic values. Third, we introduce groups enhanced by neural networks to assess whether combining CoL with neural networks yields better results and to explore the filtering effect of the inner coupling structure. In dynamic experi-

Table 3: Group configurations. Groups marked with ★ are the main experiments, those with ☆ are for ablation and extended experiments, and the unmarked group is the baseline.

Group	**Experiment**	**Pretrained DSNN**	**NNFC**	**Inner Coupling Structure**
DSL	static			
☆DSL (Heuristic)	static			
★CoL DSL	static, dynamic			
☆DSL+NN	static	✓		
☆DSL (Heuristic)+NN	static	✓		
☆ CoL DSL+NN	static	✓		
☆CoL DSL+NNFC	dynamic		✓	
☆DSL+NN (Cp)	static	✓		✓
☆DSL(Heuristic)+NN (Cp)	static	✓		✓
☆CoL DSL+NN (Cp)	static	✓		✓
☆CoL DSL+NN (Cp)	static	✓		✓
★CoL DSL+NNFC (Cp)	dynamic		✓	✓

[1]Each partial program must be completed with at most 1000 transformation pairs, though this may exceed 1000 if additional tasks are generated during synthesis.

ments, we design control groups with and without NNFC to evaluate its impact. Additionally, we include a group without the inner coupling structure to confirm its necessity.

Environment. Experiments are carried out on a computer equipped with an Intel i7-14700 processor, a GTX 4070 GPU, and 48GB RAM.

3.2 Static Experiments

We start with static experiments. With the task domain, difficulty level, and neural network conditions unchanged in each group, a series of controlled experiments confirm that CoL has remarkably boosted DSL program synthesis in all metrics.

The results in Table 4 clearly demonstrate that **CoL significantly improves accuracy while minimizing overhead**. Most notably, CoL improves the accuracy of the DSL from less than 50% to 100% across both relational and symbolic benchmarks. Additionally, CoL achieves remarkable reductions in relational tasks, cutting tree operations by 90%, transformation pairs by 88%, and time by 95%. Similarly, in symbolic tasks, CoL reduces tree operations by 92%, transformation pairs by 96%, and time by 97%. These findings showcase CoL's substantial impact on improving performance across all key metrics.

Table 4: Static performance of DSL and CoL DSL for relational and symbolic tasks. CoL DSL significantly outperforms DSL in all metrics.

Benchmark	Group	Accuracy↑ (%)	Avg. Tree Operation↓	Avg. Transformation Pair↓	Avg. Time Spent↓(s)
relational	DSL	11.3	463.9	1432.2	9.43
	CoL DSL	**100.0**	**46.6**	**177.8**	**0.48**
symbolic	DSL	48.3	411.2	2285.3	3.31
	CoL DSL	**100.0**	**33.8**	**92.7**	**0.11**

Further ablation and extension experiments clarify the sources of CoL's enhancement, confirm CoL's effective integration with neural networks, and explore when filtering via inner coupling structures is most beneficial. Our findings are as follows:

First, **CoL's enhancement stems from both heuristics and structured rule application stages**. As illustrated in Figure 5, the DSL (Heuristic) group outperforms the DSL group in most metrics, and the CoL DSL group significantly surpasses DSL (Heuristic) in all metrics. Such results indicate that CoL positively impacts synthesis by guiding and structuring rule application. Moreover, on top of guidance, the structured rule application stages achieve greater improvement.

Second, **integrating CoL with neural networks further improves the search efficiency.** As shown in Figure 5, despite additional GPU and time overhead, the top-performing CoL DSL + NN group reduces tree operations by 43% and transformation pairs by 19% in relational tasks compared to the CoL DSL group. In symbolic tasks, the CoL DSL + NN (Cp) group reduces tree operations by 64% and transformation pairs by 46%. The results showcase that neural networks can further narrow the search space for program synthesis beyond CoL. Importantly, the group with the inner coupling structure outperforms non-neural groups in both tasks. In contrast, the group without it presents an accuracy decline in symbolic tasks, validating the structure's role in improving reliability.

Third, **the inner coupling structure is more effective when error tolerance is low**. As indicated in Figure 5, for symbolic tasks, CoL DSL-based groups with the inner coupling structure significantly outperform those without it. However, for relational tasks and DSL-based groups (without CoL or heuristic), those without such structure perform better. This difference indicates that the filtering effect of the inner coupling structure comes at a cost: it filters out both incorrect and correct predictions. So, its effectiveness depends on the positive impact of eliminating incorrect predictions outweighing the loss of correct ones. Therefore, for relational tasks with a limited search space and DSL-based groups with higher error tolerance, the cost of filtering outweighs the benefit. However, in symbolic tasks, where avoiding errors is more critical, CoL DSL-based groups benefit significantly from the inner coupling structure.

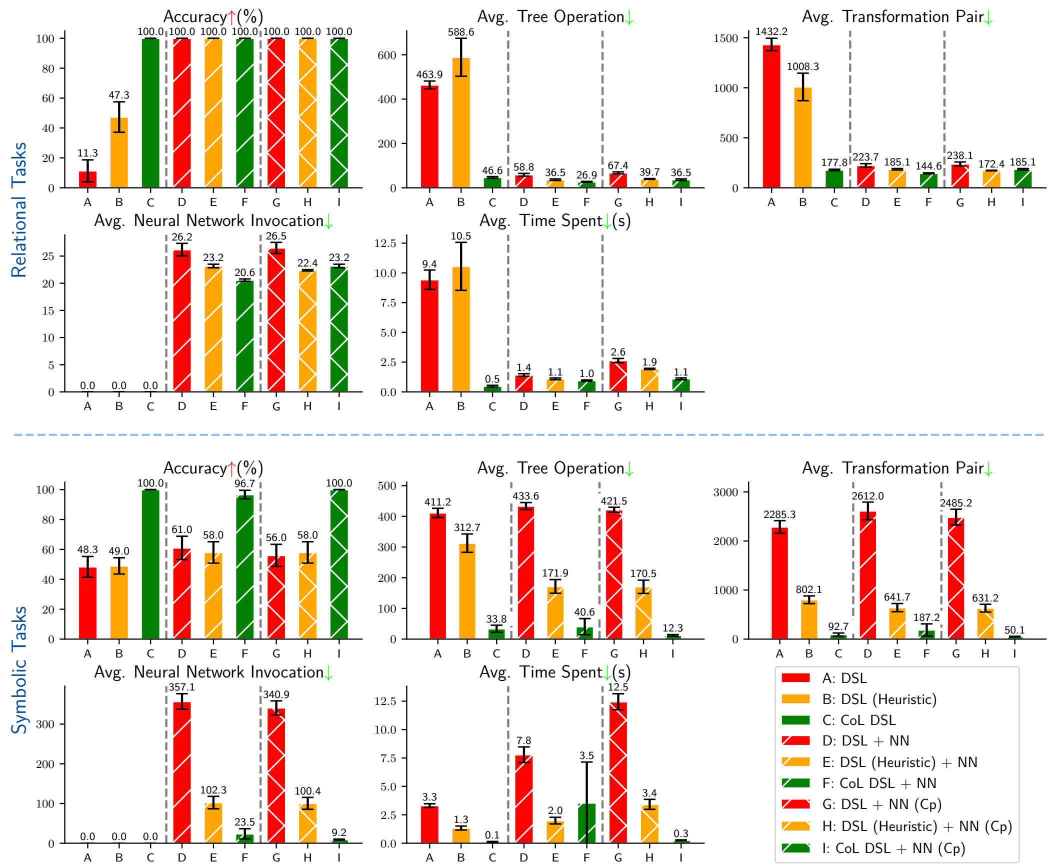

Figure 5: Static performance on relational and symbolic tasks at difficulty level A. CoL DSL-based groups outperform DSL (Heuristic) and DSL groups. Performance varies for DSNN-enhanced groups with the inner coupling structure. Error bars show 95% confidence intervals across 6 batches.

3.3 Dynamic Experiments

Static experiments confirm CoL's improvements on DSL and its enhancement with neural networks. However, real-world program synthesis involves varying task domains and difficulty, facing the risk of neural network mispredictions due to underperformance. Therefore, we introduce these factors in dynamic experiments to evaluate how NNFC further improves the performance of CoL DSL.

Table 5: Dynamic performance of CoL DSL and CoL DSL+NNFC(Cp). NNFC significantly improves the dynamic performance of CoL DSL in accuracy, tree operations, and transformation pairs.

Benchmark	**Group**	**Accuracy↑ (%)**	**Avg. Tree Operation↓**	**Avg. Transformation Pair↓**	**Avg. Neural Network Invocation↓**	**Avg. Time Spent↓(s)**
relational	CoL DSL	**100.0**	70.0	259.8	**0**	**1.05**
	CoL DSL+NNFC (Cp)	**100.0**	**54.6**	**224.5**	21.7	2.08
symbolic	CoL DSL	82.6	233.5	977.1	**0**	1.42
	CoL DSL+NNFC (Cp)	**99.4**	**50.3**	**222.2**	21.6	**1.12**
multi-domain	CoL DSL	97.5	115.2	367.6	**0**	**0.99**
	CoL DSL+NNFC (Cp)	**99.0**	**45.6**	**250.5**	72.84	3.91

The results in Table 5 confirm that **NNFC significantly enhances the reliability of CoL DSL in challenging conditions.** As task difficulty increases and multidomain scenarios emerge, the accuracy of the CoL DSL group declines compared to its performance in static experiments. However, the NNFC-enhanced group maintains an accuracy of at least 99%, demonstrating its strong reliability in challenging situations. Additionally, compared with the original CoL DSL group, it reduces tree operations by 22% and transformation pairs by 14%. For symbolic tasks, despite the added time for neural network invocations, the NNFC-enhanced group still shortens the time spent by 21%.

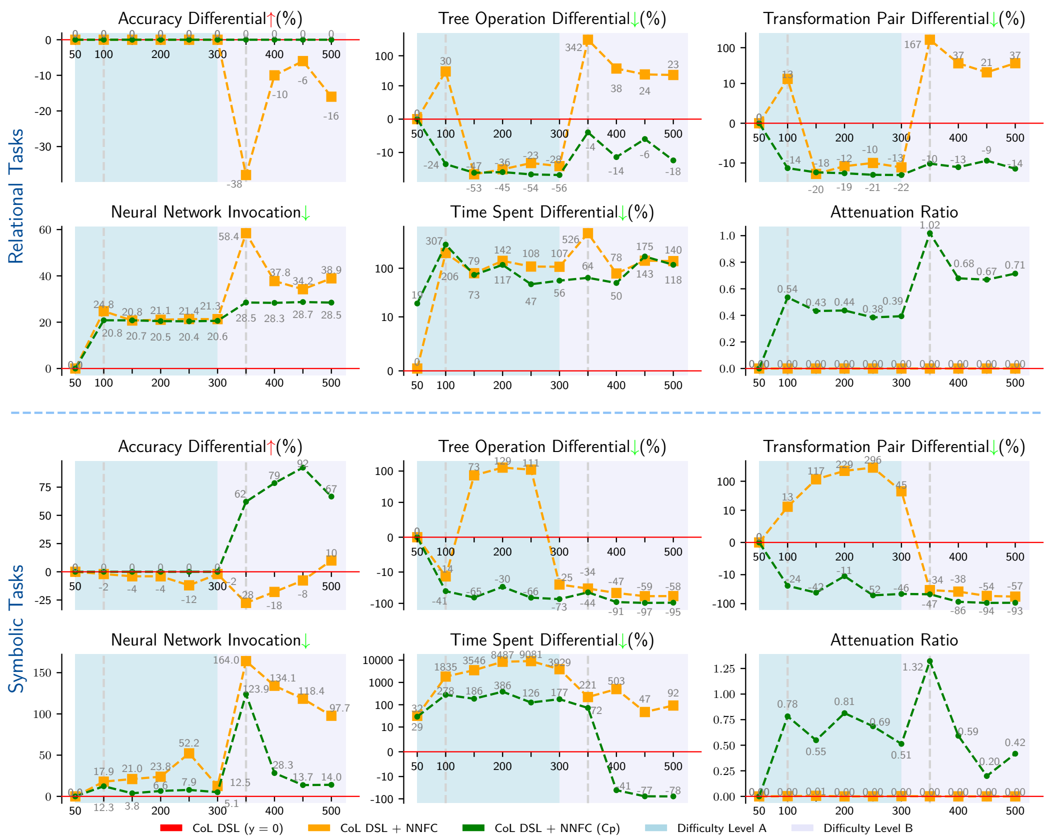

Figure 6: Dynamic performance differential to CoL DSL in singledomain tasks. The NNFC group without the inner coupling structure shows 12 accuracy declines across 20 batches, while the group with the structure shows none. Each batch consists of 50 tasks, and NNFC continuously trains DSNNs using generated data after each batch, starting from scratch.

Further ablation experiments confirm that **reliability provided by NNFC primarily stems from the filtering effect of the inner coupling structure.** As shown in Figures 6 and 7, the inner coupling structure reduces the occurrence of accuracy declines due to DSNN mispredictions by 94%. Additionally, the dynamic performance reveals how the inner coupling structure enhances NNFC:

In the scenarios where a DSNN underperforms due to issues such as insufficient training data Mikołajczyk & Grochowski (2018) (as seen in Figure 6, tasks 51-100), inadequate generalization to more challenging tasks Yosinski et al. (2014); Wei et al. (2019) (Figure 6, tasks 301-350), and catastrophic forgetting when tasks from a new domain are learned Kirkpatrick et al. (2017); Van de Ven & Tolias (2019) (Figure 7, tasks 1-100), incorrect predictions lead the actual synthesis path to deviate from the CoL, which in turn causes inefficiency and reduced accuracy. During these phases, for NNFC with the inner coupling structure, the attenuation ratio spikes, indicating that a large percentage of neural network predictions are filtered out. Consequently, the inner coupling structure ensures that the synthesis process adheres to the CoL, effectively mitigating the negative impact of DSNN mispredictions and enhancing reliability.

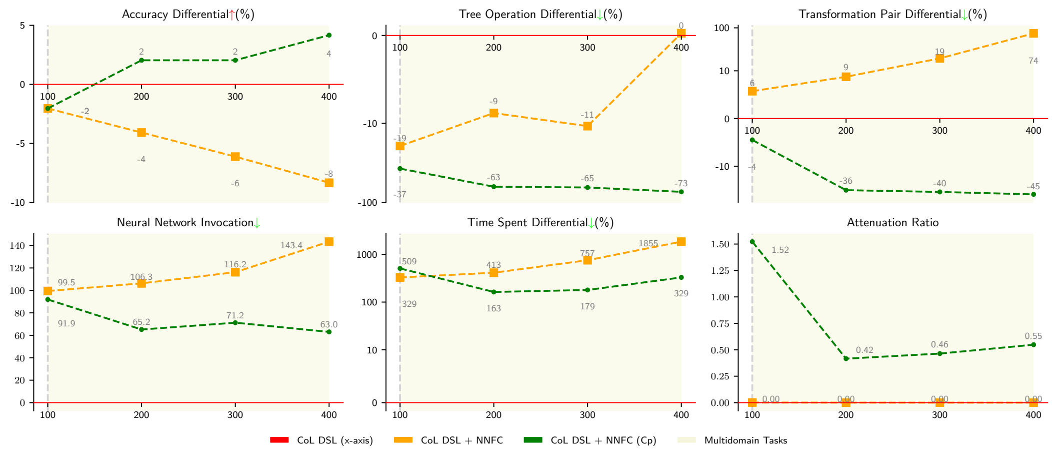

Figure 7: Dynamic performance differential to CoL DSL in multidomain tasks. The NNFC group without an inner coupling structure degrades across all 4 batches, while the group with the structure experiences degradation only in the first batch. Each batch includes 50 relational and 50 symbolic tasks, and DSNNs are continuously trained from those for tasks at difficulty level A in Figure 6.

As the DSNN improves and reaches a relatively stable state (as seen in Figure 6, tasks 101-300, 351-500, and Figure 7, tasks 101-400), the attenuation ratio shows a decreasing trend accordingly. This adaptive adjustment demonstrates how the inner coupling structure dynamically regulates the DSNN's impact, leveraging neural network contributions while mitigating risks to ensure both efficiency and reliability in program synthesis.

4 Related Work

Neural Search Optimization: Neural networks are key for optimizing search in program synthesis. Projects like Kalyan et al. (2018); Zhang et al. (2023) and Li et al. (2024) use neural networks to provide oracle-like guidance, while Neo Feng et al. (2018), Flashmeta Polozov & Gulwani (2015), and Concord Chen et al. (2020) prune search spaces with infeasible partial programs. COOL employs both strategies to enhance efficiency.

Multi-step Program Synthesis: Chain-of-Thought (CoT) Wei et al. (2022) enhances LLMs by breaking tasks into subtasks. Projects like Zhou et al. (2022); Shi et al. (2023b) and Zheng et al. (2023) use this in program synthesis. Compared to CoT, which directly decomposes tasks, CoL does so indirectly by constraining rule applications.

Reinforcement Learning: Reinforcement learning improves neural agents in program synthesis through feedback, as seen in Eberhardinger et al. (2023); Liu et al. (2024); Bunel et al. (2018), Concord Chen et al. (2020), and Quiet-STaR Zelikman et al. (2024). NNFC similarly refines control flow but serves an auxiliary role for programmer strategies in synthesis rather than dominating it.

5 Conclusion

We explored fine-grained control and flexible modularity for complex program synthesis through the Chain-Oriented Objective Logic (COOL) framework. Inspired by activity charts and control theory, we developed Chain-of-Logic (CoL) and Neural Network Feedback Control (NNFC) to achieve these goals. Static and dynamic experiments across relational, symbolic, and multidomain tasks demonstrated that COOL offers strong efficiency and reliability. We believe that continued research and refinement of CoL and NNFC will inspire advancements not only in program synthesis but also in broader areas of neural network reasoning.

References

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

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pp. 934–950. Springer, 2013.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. Flashfill++: Scaling programming by example by cutting to the chase. *Proceedings of the ACM on Programming Languages*, 7(POPL):952–981, 2023.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34: 22196–22208, 2021.

Xiuying Chen, Mingzhe Li, Xin Gao, and Xiangliang Zhang. Towards improving faithfulness in abstractive summarization. *Advances in Neural Information Processing Systems*, 35:24516–24528, 2022.

Y Chen, C Wang, O Bastani, I Dillig, and Y Feng. Program synthesis using deduction-guided reinforcement learning. In *Computer Aided Verification32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, volume 12225, pp. 587–610, 2020.

Guofeng Cui and He Zhu. Differentiable synthesis of program architectures. *Advances in Neural Information Processing Systems*, 34:11123–11135, 2021.

Iddo Drori, Sarah Zhang, Reece Shuttleworth, Leonard Tang, Albert Lu, Elizabeth Ke, Kevin Liu, Linda Chen, Sunny Tran, Newman Cheng, et al. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.

Manuel Eberhardinger, Johannes Maucher, and Setareh Maghsudi. Towards explainable decision making with neural program synthesis and library learning. In *NeSy*, pp. 348–368, 2023.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, pp. 10–19, 2022.

Hassan Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

Rudolf Groner, Marina Groner, and Walter F Bischof. *Methods of heuristics*. Routledge, 2014.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*, 2019.

Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. *Programming with TensorFlow: solution for edge computing applications*, pp. 87–104, 2021.

Alon Jacovi and Yoav Goldberg. Towards faithfully interpretable nlp systems: How should we define and evaluate faithfulness? *arXiv preprint arXiv:2004.03685*, 2020.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

Paul King. A history of the groovy programming language. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–53, 2020.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*, 2023.

Michael E Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*, volume 39. Bell Laboratories Murray Hill, NJ, 1975.

Wei Li, Wenhao Wu, Moye Chen, Jiachen Liu, Xinyan Xiao, and Hua Wu. Faithfulness in natural language generation: A systematic survey of analysis, evaluation and optimization methods. *arXiv preprint arXiv:2203.05227*, 2022.

Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*, pp. 280–301. Springer, 2024.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems*, 31, 2018.

Max Liu, Chan-Hung Yu, Wei-Hsu Lee, Cheng-Wei Hung, Yen-Chun Chen, and Shao-Hua Sun. Synthesizing programmatic reinforcement learning policies with large language model guided search. *arXiv preprint arXiv:2405.16450*, 2024.

Agnieszka Mikołajczyk and Michał Grochowski. Data augmentation for improving deep learning in image classification problem. In *2018 international interdisciplinary PhD workshop (IIPhDW)*, pp. 117–122. IEEE, 2018.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Maxwell Nye, Armando Solar-Lezama, Josh Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *Advances in Neural Information Processing Systems*, 33:10832–10842, 2020.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 107–126, 2015.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. Lambdabeam: Neural program search with higher-order functions and lambdas. *Advances in Neural Information Processing Systems*, 36:51327–51346, 2023a.

Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*, 2023b.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L Hamilton. Clutrr: A diagnostic benchmark for inductive reasoning from text. *arXiv preprint arXiv:1908.06177*, 2019.

Dominik Sobania, Martin Briesch, and Franz Rothlauf. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pp. 1019–1027, 2022.

Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s): 1–25, 2014.

Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108, 2001.

Evren Mert Turan and Johannes Jäschke. Closed-loop optimisation of neural networks for the design of feedback policies under uncertainty. *Journal of Process Control*, 133:103144, 2024.

Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

Colin Wei, Jason D Lee, Qiang Liu, and Tengyu Ma. Regularization matters: Generalization and optimization of neural nets vs their induced kernel. *Advances in Neural Information Processing Systems*, 32, 2019.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, and Xiaojie Guo. Graph neural networks: foundation, frontiers and applications. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4840–4841, 2022.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36:54769–54784, 2023.

Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. Outline, then details: Syntactically guided coarse-to-fine code generation. In *International Conference on Machine Learning*, pp. 42403–42419. PMLR, 2023.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

A Rule in CoL DSL

In addition to the heuristic vector and keywords, COOL extends the flexibility of the synthesis process by enhancing DSL rules. These enhancements are exemplified in Figure 8, which clarifies the rule introduced in Figure 1.

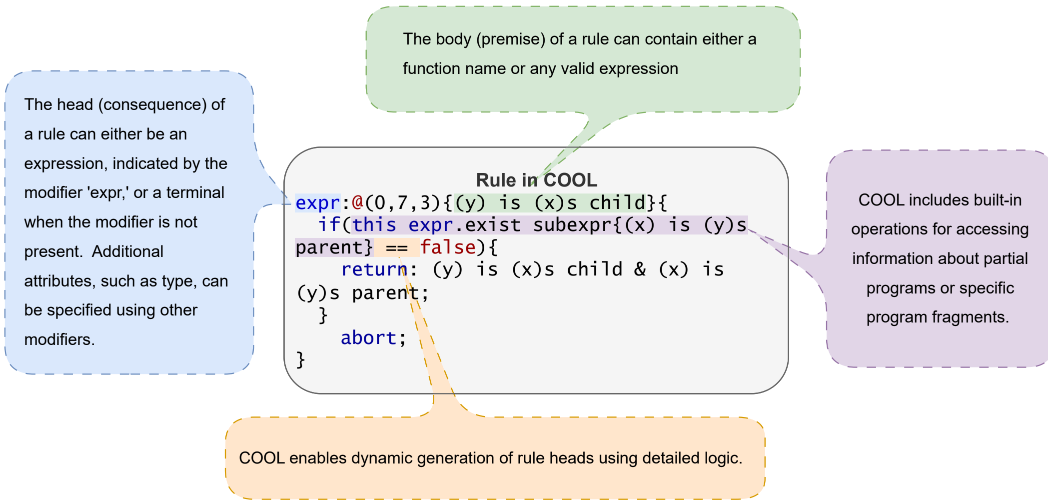

Figure 8: DSL rules in COOL. The framework allows for defining rule heads using expressions or terminals, which are enhanced with modifiers for additional attributes. Rule bodies can incorporate any valid expression or function name. Besides, COOL provides built-in operations for accessing program fragment information and facilitates dynamic rule head generation.

B Stage Progression Driven by Heuristic Vectors

Let s denote the CoL stage, h donate the heuristic value, and n donate the length of CoL. A rule's heuristic vector can be mathematically represented as:

$$\mathbf{H} = \{(s_0, h_0), (s_1, h_1), \ldots, (s_n, h_n)\}, \quad n \in \mathbb{N}^+ \tag{1}$$

Upon applying a rule with heuristic vector $\mathbf{H}$, the subsequent stage, s_{next}, can only advance or remain the same, and the next stage should be as close to the current stage as possible:

$$\min s_{\text{next}} \quad \text{such that} \quad \exists(s_{\text{next}}, h_{\text{next}}) \in \mathbf{H} \quad \text{and} \quad s_{\text{next}} \geq s_{\text{current}} \tag{2}$$

C Neural Networks in DSNN

COOL performs synthesis tasks using Three-Address Code (TAC), also utilized as input by DSNN. TAC serves as an intermediate representation (IR), allowing program synthesis to be conducted without the constraints of specific DSL syntax or the machine code format of the execution platform Sujeeth et al. (2014). As TAC embodies both the graphical properties of a syntax tree and the sequential properties of execution, the design of the neural network must be capable of capturing these dual characteristics.

The detailed layer architecture of neural networks in DSNN is illustrated in Figure 9. The processing flow consists of the following steps:

1. **Embedding Node Features:** We start by employing embedding layers with learning capabilities. These layers convert categorical inputs into dense, continuous vectors, which enhances the stability and efficiency of subsequent processing layers Hrinchuk et al. (2019).

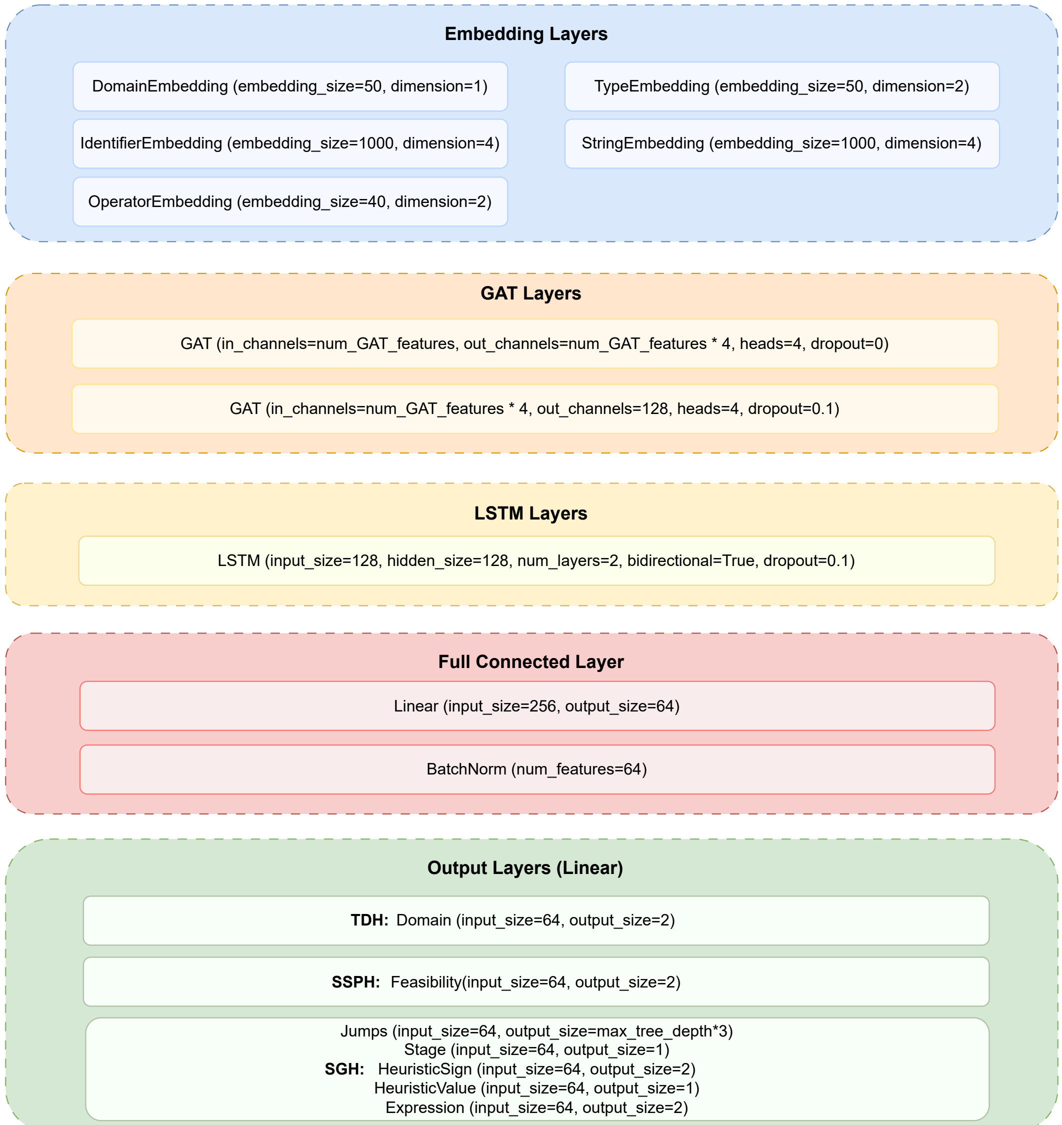

Figure 9: Layer architecture of neural networks in DSNN. Each neural network consists of embedding layers for domains, types, identifiers, strings, and operators, followed by GAT layers for tree feature extraction. LSTM layers provide sequential modeling for programs, with fully connected layers combining the outputs. Various output layers handle domain identification for task detection, feasibility judgment for search space pruning, tree jumps, stage prediction, heuristic constraint (sign and value), and constraint on the type of rule's head (expression or terminal) for search guidance.

2. **Graph Feature Extraction:** Next, we use a Graph Neural Network (GNN) to extract graph features from each line of TAC code Drori et al. (2022); Wu et al. (2022). To adaptively extract intricate details such as node types, graph attention (GAT) layers are applied after the embedding layers Velickovic et al. (2017).

3. **Sequential Feature Processing:** We adopt Long Short-Term Memory (LSTM) networks to capture the sequential features inherent in TAC Chen et al. (2021); Nye et al. (2020). Recognizing the equal importance of each TAC line, bidirectional LSTM layers are employed following the GAT layers to enrich the contextual understanding Huang et al. (2015).

4. **Multi-Head Output:** Finally, the processed data is channeled through multiple output layers to prevent task interference and ensure clarity in results.

Figure 4 (right) illustrates using three neural network units arranged in series to construct the internal coupling structure of DSNN. Labeling these neural networks with A, B, and C in order of their sequence, Table 6 details the specific input features for each network: Neural network B receives its input feature ”applied” from network A’s output feature ”jumps,” while network C’s input features ”applied” and ”next stage” are derived from the output features ”jumps” and ”next stage” of network B. The output features of three neural network units are consistent and comparable, Table 7 presents the output features of the neural networks.

It is necessary to note that the DSNNs without internal coupling structures in Table 3 contain only neural network A.

Table 6: Input features of neural networks in DSNN. Each entry specifies the feature, its size, and the neural networks it pertains to, along with a description of its role. These features contribute to the neural network’s understanding of the syntax tree’s structure and semantics, aiding in the accurate synthesis of programs.

Feature	**Feature Size**	**Neural Network**	**Signification**
grounded	2	A, B, C	The node is in a fully specified expression.
domain	1	A, B, C	Domain of the subtask represented by the subtree where the node is located.
root	2	A, B, C	The tree representing the subtask is rooted at this node.
non-terminal	2	A, B, C	The node is a non-terminal.
type	1	A, B, C	Type of the node.
identifier	1	A, B, C	Identifier of the node.
string	1	A, B, C	The node contains a string as the immediate value.
number	1	A, B, C	The node contains a number as the immediate value.
operator	1	A, B, C	The node is an operator.
current stage	1	A, B, C	Current CoL stage (valid when this node is grounded).
operand position	3	A, B, C	Placement of nodes in a binary operation tree (left operand node, right operand node, operation node).
applied	1	B, C	A rule is applied to the subtree rooted at this node (derived from the output feature “**jumps**” of the previous neural network).
next stage	1	C	The CoL stage to advance to after applying the rule (derived from the output feature “**next stage**” of the previous neural network).

D SIGNAL CLIPPER

The Clipper, as illustrated in Figure 4 (left), caps signals that do not align with the DSNN guidance to zero:

$$u_2 = \begin{cases} 0 & \text{if } u_1 > 0 \text{ and current rule doesn't align with} \\ & \text{the guidance and there exists another rule in} \\ & \text{the search space that aligns with the guidance} \\ u_1 & \text{otherwise} \end{cases} \tag{3}$$

Table 7: Output features of neural networks in DSNN. These features provide comprehensive optimizations for CoL DSL during program synthesis, including task detection, search space pruning, and search guidance.

Feature	Feature Size	Neural Network	Signification
domain	2	A, B, C	Relevance of task domains to DSNN.
feasibility	2	A, B, C	Feasibility of synthesizing the complete program.
jumps	max_tree_depth*3	A, B, C	The path from the tree's root to the subtree's root where the rule is applied (jump left, right, or stop in each step).
next stage	1	A, B, C	The CoL stage to advance to after applying the rule.
heuristic sign	2	A, B, C	Sign of the rule's heuristic value.
heuristic value	1	A, B, C	Rule's heuristic value.
expression	2	A, B, C	Type of rule's head (expression or terminal).

D.1 A* Search in Program Synthesis

During the exploration phase of program synthesis, we leverage the A* algorithm to perform the heuristic search. This algorithm is renowned for its efficacy in discrete optimization tasks, utilizing heuristic guidance to navigate the search space effectively Hart et al. (1968). Each action or decision is associated with a specific cost in this context. By evaluating the cumulative cost of actions taken so far and the estimated costs of future actions, A* seeks to determine the path with the least overall cost. In our approach, heuristic values promoting forward progression are considered rewards. Therefore, we treat them as negative costs in calculations. Algorithm 1 illustrates the implementation details.

Algorithm 1 Search Algorithm for DSL Program Synthesis

procedure A* Search($initialPartialProgram, u_2$)
  $openSet \leftarrow$ priority queue containing only the initial partial program
  $gScore[startPartialProgram] \leftarrow 0$ ▷ cost from start
  $fScore[startPartialProgram] \leftarrow 0$
  while $openSet \neq \emptyset$ **do**
    $currentProgram \leftarrow openSet$.pop() ▷ The partial program in openSet with lowest fScore value
    if $currentProgram$ is complete program **then**
      return Success
    end if
    for each $neighbor$ of $currentProgram$ **do** ▷ Neighbor is a program directly obtained by applying a rule to the current program
      $tentative_g Score \leftarrow gScore[current] - u_2[neighbor]$
      if $tentative_g Score < gScore[neighbor]$ **then**
        $cameFrom[neighbor] \leftarrow current$
        $gScore[neighbor] \leftarrow tentative_g Score$
        $fScore[neighbor] \leftarrow gScore[neighbor] - u_2[neighbor]$
        if $neighbor \notin openSet$ **then**
          $openSet$.add($neighbor$)
        end if
      end if
    end for
  end while
  return Failure
end procedure

E IMPLEMENTATION TOOLCHAIN

To fully implement the CoL DSL and adapt it to NNFC, we choose to build COOL from the ground up rather than extending existing DSL frameworks such as Xtext Bettini (2016) or Groovy King (2020). We use C++ as the primary language to meet the execution efficiency requirements for the numerous tree operations inherent in the DSL program synthesis process. For development efficiency, we utilize Lex Lesk & Schmidt (1975) and YACC Johnson et al. (1975) for syntax and semantic parsing, respectively. The neural network components are implemented in Python, leveraging the PyTorch library Imambi et al. (2021) to support machine learning tasks effectively. Table 8 shows the detailed code effort involved in developing the different components of COOL across various programming languages.

Table 8: Code Effort in COOL. Components of COOL are developed across different programming languages.

Language	Lines	Components
C++	60k	framework and CoL DSL solver
Python	3k	DSNN
Lex	1k	syntax parser
YACC	2k	semantic parsers

F OPTIMIZATION STRATEGY

In practice, we observe that as the CoL length increases, the frequency of skipping stages rises. While skipping can lead to shorter synthesis paths and improved efficiency, it may cause task failures by omitting necessary stages. To manage this, we propose two strategies:

1. Gradient-Based Regulation: We employ gradient-based regulation, a widely used strategy in program synthesis Cui & Zhu (2021); Liang et al. (2018); Chaudhuri et al. (2021). By evaluating the slope or rate of change between consecutive stages, gradients help us make dynamic adjustments to synthesis paths. In our approach, we regulate skipping by applying a gradient to the heuristic values at each stage in the CoL. We encourage skipping when the heuristic gradient from one stage to the next is positive. Conversely, if the gradient is negative, we suppress skipping.
2. NNFC Regulation: Once we establish a feasible synthesis path, we can treat partial programs derived through skipping as infeasible. Then, we will utilize the feedback loop to suppress unwarranted skipping actions. However, since these partial programs might still contain feasible solutions, we need further investigation to understand and fully leverage the potential impact of this data.

In our experiments, we prioritize accuracy by suppressing skipping behavior, ensuring essential stages are included in synthesis paths.

G FUTURE WORK

In future work, we aim to enhance the capability of the COOL framework by exploring the implementation of CoL and NNFC in more complex scenarios, such as managing dependencies among DSL libraries and object-oriented development. We plan to facilitate community collaboration by developing more DSL libraries to expand COOL's applications. Additionally, we are interested in integrating COOL with language models. As these models evolve, ensuring ethical and accurate reasoning becomes increasingly crucial Jacovi & Goldberg (2020); Chen et al. (2022); Li et al. (2022). The COOL framework, including CoL's constraints on rule application and NNFC's structured agent interactions, helps to enhance reasoning faithfulness, preventing harmful reasoning logic. We hope our work will serve as a reliable bridge for interaction and understanding between human cognitive processes and language model reasoning.

H CoL DSL for Relational Tasks

We present only the specific code for the CoL DSL group, while the code for the DSL and DSL (Heuristic) groups, referenced in Table 3, is not displayed. This omission is because their differences from the CoL DSL group are confined to their heuristic vectors. In both the DSL and DSL (Heuristic) groups, the heuristic vectors have a dimension of 1. However, the DSL group employs a fixed heuristic value of -1, whereas the DSL (Heuristic) group utilizes variable values. The experimental codes are presented concisely, showcasing only the framework. Please refer to the attached supplementary materials for the complete content.

```
//1 Separate Relations and Genders
expr:@(9){(a) is (b)s grandson}{
    return:(a) is male &  (a) is (b)s grandchild & (b) is (a)s
     ↪  grandparent;
}
...

//2 Reason Inverse Relations
expr:@(0,7,3){(a) is (b)s grandchild}{
    if(this expr.exist subexpr{(b) is (a)s grandparent} == false){
        return: (a) is (b)s grandchild & (b) is (a)s grandparent;
    }
    abort;
}
...

//3 Reason Indirect Relations
expr:@(0,0,5){(a) is (b)s sibling}{
    placeholder:p1;
    while(this expr.find subexpr{(p1) is (a)s sibling}){
        if(this expr.exist subexpr{(p1) is (b)s sibling} == false
         ↪  && p1 != b){
return: (a) is (b)s sibling & (p1) is (b)s sibling;
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(p1) is (a)s parent}){
        if(this expr.exist subexpr{(p1) is (b)s parent} == false){
return: (a) is (b)s sibling & (p1) is (b)s parent;
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(p1) is (a)s pibling}){
        if(this expr.exist subexpr{(p1) is (b)s pibling} ==
         ↪  false){
return: (a) is (b)s sibling & (p1) is (b)s pibling;
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(p1) is (a)s grandparent}){
        if(this expr.exist subexpr{(p1) is (b)s grandparent} ==
         ↪  false){
return: (a) is (b)s sibling & (p1) is (b)s grandparent;
        }
        p1.reset();
    }
```

```
    p1.reset();
    abort;
}
...

//4 Recombine Relations and Genders, Eliminate Irrelevant
↪  Relations
expr:@(0,0,0,8){(a) is (b)s ($relation)}{
    //immediate family
    placeholder:p1;
    while(this expr.find subexpr{(a) is (b)s grandchild}){
        if(this expr. exist subexpr{(a) is male}){
return: $relation == "grandson";
        }
        if(this expr.exist subexpr{(a) is female}){
return:$relation == "granddaughter";
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(a) is (b)s child}){
        if(this expr. exist subexpr{(a) is male}){
return: $relation == "son";
        }
        if(this expr.exist subexpr{(a) is female}){
return:$relation == "daughter";
        }
        p1.reset();
    }
    ...
    abort;
}
...
expr:@(0,0,0,10){a & ($b == c)}{
    return:b == c;
}
...
```

I CoL DSL for Symbolic Tasks

```
// Common Transformations
expr:@(2,2,2,2,2){0+#a}{
    return:a;
}
expr:@(2,2,2,2,2){#a+0}{
    return:a;
}
...

// 1 Expand Square Terms
expr:@(5,0,0,0){(#?a + #?b)^2}{
    return:a^2+2*a*b+b^2;
}
expr:@(5,0,0,0){(#?a - #?b)^2}{
    return:a^2+(-2)*a*b+b^2;
}
expr:@(6,0,0,0){(#a*#b)^2}{
    return:a^2*b^2;
```

```
}
...

// 2 Expand Bracketed Terms
expr:@(0,4,0,0,0){#?a-(#?b+#?c)}{
    return:a-b-c;
}
expr:@(0,3.8,0,0,0){(#?b+#?c)*#?a}{
    return:b*a+c*a;
}
...

// 3 Extract Coefficients
expr:@(0,0,5,0){$x*a}{
    return:a*x;
}
expr:@(0,0,4.8,0){(immediate:a*$x)*(immediate:b*$x)}{
    new:tmp = a*b;
    return:tmp*x^2;
}
expr:@(0,0,4.6,0){$x*(a*$x)}{
    return:a*x^2;
}
...

// 4 Re-Express Negative Coefficients
expr:@(0,0,0,3.5,0){#a-$x}{
    placeholder:p1;
    placeholder:p2;
    if(x.exist subexpr{p1*p2}){
        abort;
    }
    return:a+(-1)*x;
}
expr:@(0,0,0,3.7,0){#a-immediate:b*$x}{
    new:tmp = 0 - b;
    return:a+tmp*x;
}
...

//5 Arrange Terms in Descending Order, Combine Like Terms
expr:@(0,0,0,0,3){immediate:a*$x+immediate:b*$x}{
    new:tmp = a+b;
    return:tmp*x;
}
expr:@(0,0,0,0,2.8){a1*$x+a2*$x^2}{
    return:a2*x^2+a1*x;
}
...

//6 Convert to Standard Form
expr:@(0,0,0,0,0,2.5){a*$x^2+b*x == #d}{
    return: a*$x^2+b*x + 0 == d;

}
expr:@(0,0,0,0,0,2.5){b*$x == $d}{

    if(d.exist subexpr{x^2}){
        return: 0*x^2 + b*x + 0 == d;
```

```
    }else {
        abort;
    }
}
expr:@(0,0,0,0,0,-4){$a==$b}{
    return:b==a;
}
...

//7 Apply Solution Formula
@(0,0,0,0,0,0,0,10){a*$x^2+b*x+c==0}{
    if(b^2-4*a*c<0){
        x="null";
    }
    else {
        new:x1=(-b+(b^2-4*a*c)^0.5)/(2*a);
        new:x2=(-b-(b^2-4*a*c)^0.5)/(2*a);
        x={x1,x2};
    }
};
```

J Relational Tasks at Difficulty Level A

```
#load(family) // Load the CoL DSL library for Relational Tasks
new:relation = "";
// [Francisco]'s brother, [Wesley], recently got elected as a
 ↪  senator. [Lena] was unhappy with her son, [Charles], and his
 ↪  grades. She enlisted a tutor to help him. [Wesley] decided to
 ↪  give his son [Charles], for his birthday, the latest version
 ↪  of Apple watch.
// Ans: (Francisco) is (Lena)s brother
new:Lena = "Lena";
new:Charles = "Charles";
new:Wesley = "Wesley";
new:Francisco = "Francisco";
(Charles) is (Lena)s son & (Wesley) is (Charles)s father &
 ↪  (Francisco) is (Wesley)s brother & (Francisco) is (Lena)s
 ↪  ($relation);
relation-->"#FILE(SCREEN)";

// [Clarence] woke up and said hello to his wife, [Juanita].
 ↪  [Lynn] went shopping with her daughter [Felicia]. [Felicia]'s
 ↪  sister [Juanita] was too busy to join them.
// Ans: (Lynn) is (Clarence)s mother-in-law
new:Clarence = "Clarence";
new:Juanita = "Juanita";
new:Felicia = "Felicia";
new:Lynn = "Lynn";
(Juanita) is (Clarence)s wife & (Felicia) is (Juanita)s sister &
 ↪  (Lynn) is (Felicia)s mother & (Lynn) is (Clarence)s
 ↪  ($relation);
relation-->"#FILE(SCREEN)";
...
```

K RELATIONAL TASKS AT DIFFICULTY LEVEL B

```
#load(family) // Load the CoL DSL library for Relational Tasks
new:relation = "";
// [Antonio] was happy that his son [Bernardo] was doing well in
 ↪  college. [Dorothy] is a woman with a sister named [Tracy].
 ↪  [Dorothy] and her son [Roberto] went to the zoo and then out
 ↪  to dinner yesterday. [Tracy] and her son [Bernardo] had lunch
 ↪  together at a local Chinese restaurant.
// Ans: (Roberto) is (Antonio)s nephew
new:Antonio = "Antonio";
new:Bernardo = "Bernardo";
new:Tracy = "Tracy";
new:Dorothy = "Dorothy";
new:Roberto = "Roberto";
(Bernardo) is (Antonio)s son & (Tracy) is (Bernardo)s mother &
 ↪  (Dorothy) is (Tracy)s sister & (Roberto) is (Dorothy)s son &
 ↪  (Roberto) is (Antonio)s ($relation);
relation-->"#FILE(SCREEN)";

// [Bernardo] and his brother [Bobby] were rough-housing. [Tracy],
 ↪  [Bobby]'s mother, called from the other room and told them to
 ↪  play nice. [Aaron] took his brother [Bernardo] out to get
 ↪  drinks after a long work week. [Tracy] has a son called
 ↪  [Bobby]. Each day they go to the park after school. ans:
 ↪  (Bobby) is (Aaron)s brother
new:Aaron = "Aaron";
new:Bernardo = "Bernardo";
new:Bobby = "Bobby";
new:Tracy = "Tracy";
(Bernardo) is (Aaron)s brother & (Bobby) is (Bernardo)s brother &
 ↪  (Tracy) is (Bobby)s mother & (Bobby) is (Tracy)s son & (Bobby)
 ↪  is (Aaron)s ($relation);
relation-->"#FILE(SCREEN)";
...
```

L SYMBOLIC TASKS AT DIFFICULTY LEVEL A

```
#load(quadratic) // Load the CoL DSL library for Symbolic Tasks
new:x = 1;
6*$x^2 == 3*x - 7;
x-->"#FILE(SCREEN)";
($x - 6)*(x + 3) == x;
x-->"#FILE(SCREEN)";
...
```

M SYMBOLIC TASKS AT DIFFICULTY LEVEL B

```
#load(quadratic) // Load the CoL DSL library for Symbolic Tasks
new:x = 1;
$x*($x + 11) == 16*($x + 22);
x-->"#FILE(SCREEN)";
$x*(36*$x + 50) - 11*(19 - 30*$x) == $x^2;
x-->"#FILE(SCREEN)";
...
```

N MULTIDOMAIN TASKS

```
#load(quadratic) // Load the CoL DSL library for Symbolic Tasks
#load(family) // Load the CoL DSL library for Relational Tasks
new:x = 1;
$x^2 - 4*$x == 6;
x --> "#FILE(SCREEN)";
...
new:relation = "";
// [Dolores] and her husband [Don] went on a trip to the
 ↪  Netherlands last year. [Joshua] has been a lovely father of
 ↪  [Don] and has a wife named [Lynn] who is always there for him.
// Ans: (Dolores) is (Lynn)s daughter-in-law
new:Lynn = "Lynn";
new:Joshua = "Joshua";
new:Don = "Don";
new:Dolores = "Dolores";
(Joshua) is (Lynn)s husband & (Don) is (Joshua)s son & (Dolores)
 ↪  is (Don)s wife & (Dolores) is (Lynn)s ($relation);
relation-->"#FILE(SCREEN)";
...
```

O PARTIAL PROGRAM AS NEURAL NETWORK INPUT


```
"codeTable": [
    {
        "boundtfdomain": "",
        "grounded": false,
        "operand1": {
            "argName": "x",
            "argType": "identifier",
            "changeable": 1,
            "className": "",
            "isClass": 0
        },
        "operand2": {
            "argName": "2",
            "argType": "number",
            "changeable": 0,
            "className": "",
            "isClass": 0
        },
        "operator": {
            "argName": "^",
            "argType": "other"
        },
        "result": {
            "argName": "1418.4",
            "argType": "identifier",
            "changeable": 1,
            "className": "",
            "isClass": 0
        },
        "root": false
    },
    ...
]
```