LILO: Learning Interpretable Libraries by Compressing and Documenting Code

Gabriel Grand[1,2*] **Lionel Wong**[2] **Matthew Bowers**[1] **Theo X. Olausson**[1] **Muxin Liu**[3]
Joshua B. Tenenbaum[1,2] **Jacob Andreas**[1]
[1]MIT CSAIL [2]MIT Brain and Cognitive Sciences [3]Harvey Mudd College

Abstract

While large language models (LLMs) now excel at code *generation*, a key aspect of software development is the art of *refactoring*: consolidating code into libraries of reusable and readable programs. In this paper, we introduce LILO, a neurosymbolic framework that iteratively synthesizes, compresses, and documents code to build libraries tailored to particular problem domains. LILO combines LLM-guided program synthesis with recent algorithmic advances in automated refactoring from STITCH: a symbolic compression system that efficiently identifies optimal λ-abstractions across large code corpora. To make these abstractions *interpretable*, we introduce an auto-documentation (AutoDoc) procedure that infers natural language names and docstrings based on contextual examples of usage. In addition to improving human readability, we find that AutoDoc boosts performance by helping LILO's synthesizer to interpret and deploy learned abstractions. We evaluate LILO on three inductive program synthesis benchmarks for string editing, scene reasoning, and graphics composition. Compared to existing methods—including the state-of-the-art library learning algorithm DreamCoder—LILO solves more complex tasks and learns richer libraries that are grounded in linguistic knowledge.

1 Introduction

Large language models (LLMs) are growing highly adept at programming in many settings: completing partially-written code [1–3], conversing with programmers [4, 5], and even solving competition-level programming puzzles [6–9]. However, human software engineers are principally concerned with building *libraries* that can be applied to entire problem domains. To this end, a key aspect of software development is the art of *refactoring* [10, 11]: identifying abstractions that make the codebase more concise, reusable, and readable. Solving this multi-objective optimization will require broadening the scope of existing code completion tools to the longer-horizon setting of *library learning*.

In this paper, we combine LLMs with recent algorithmic advances in automated refactoring from the programming languages (PL) literature to learn libraries of reusable function abstractions. Our approach draws inspiration from DREAMCODER [12], an iterative Wake-Sleep algorithm that alternates between searching for solutions to programming tasks (*Wake*) and refactoring shared abstractions into a library (*Sleep*) that in turn helps to guide search. Unlike standard deep learning approaches, DreamCoder can make strong generalizations from just a handful of examples, and the model's conceptual knowledge is represented symbolically. However, DreamCoder is extremely computationally intensive, requiring more than two CPU-*months* to learn a single domain (see Ellis et al., Apx. J). Much of this search time is spent discovering a basic set of abstractions that human programmers typically already know, or might be able to grok quickly based on having solved problems in other

*Correspondence to gg@mit.edu. *The full version of this paper is available at* arxiv.org/abs/2310.19791. *This is an abbreviated version for the Intrinsically-Motivated and Open-Ended Learning Workshop at NeurIPS 2023. Code and data available at* github.com/gabegrand/lilo.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

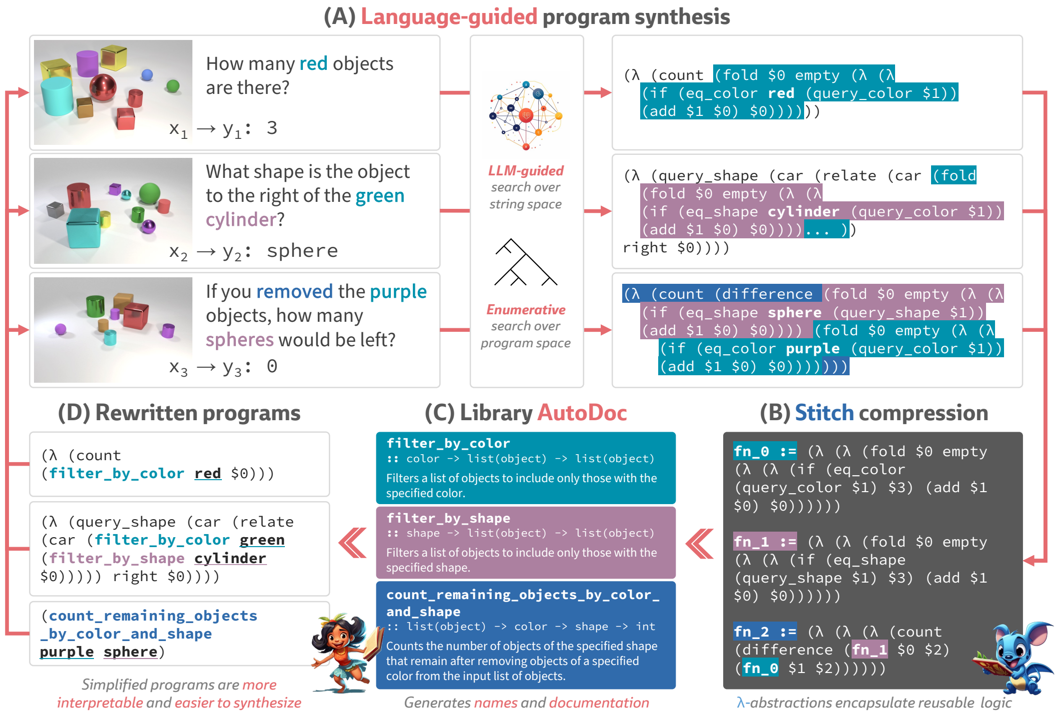

Figure 1: **Overview of the LILO learning loop.** (A) LILO synthesizes programs based on natural language task descriptions using a dual-system search model. To refactor a set of program solutions, LILO integrates a compression algorithm called STITCH (B) with LLM-generated auto-documentation (C) to produce an interpretable library of λ-abstractions. This search-compress-document loop simplifies the structure of program solutions (A vs. D), making it easier to solve more complex tasks on future iterations.

domains. Moreover, DreamCoder libraries are not necessarily interpretable, requiring both domain expertise and knowledge of lambda calculus to decipher.

To address these issues, we introduce **LILO**, a neurosymbolic program synthesis framework that leverages LLMs in two novel ways: (1) to expedite the discovery of program solutions during search, and (2) to improve the interpretability of learned libraries through auto-documentation. We evaluate LILO against a language-guided DreamCoder variant and an LLM baseline on three challenging program synthesis domains: **string editing** (REGEX) [13], **scene reasoning** (CLEVR) [14], and **graphics composition** (LOGO) [15]. On all three domains, LILO solves more tasks than both models and learns abstractions that are intractable to discover with existing methods.

2 LILO: Library Induction with Language Observations

LILO builds on a long line of work in inductive program synthesis, which we review in Apx. A. Algorithmically, LILO (Alg. 1) has a similar structure to existing approaches [12, 16–18] that alternate between search and refactoring. To complete the loop, we introduce an auto-documentation procedure (AutoDoc) that infers names for these abstractions, rendering them legible to LLM-guided synthesis.

Dual-system program search (Fig. 1A). Inspired by dual process theories of cognition [19–21], LILO is equipped with two kinds of search procedures. As in DreamCoder and LAPS [22], we use a task-conditioned PCFG to perform "slow" enumerative search in program space. Additionally, we introduce a "fast" approximate search model in string space that leverages LLMs. We procedurally construct few-shot prompts (Apx. B.2) consisting of three parts: (1) A library specification, (2) a set of task solutions, and (3) a linguistic description of the target task. For each completion, we run parsing, type inference, and execution checks to identify valid programs that solve the target task.

Refactoring via Stitch compression (Fig. 1B). As the learner solves more tasks, the solution set will grow to contain many recurring program fragments that we wish to refactor. In library learning systems that rely on enumeration, refactoring improves search efficiency by avoiding the need to rediscover key building blocks for each new task. Analogously, in LILO, refactoring makes the generation task easier: a LLM equipped with a library of abstractions can deploy entire blocks of code with just a few tokens. We leverage recent algorithmic advances from STITCH [23]: a symbolic compression system that identifes reusable abstractions in large datasets of lambda calculus programs and achieves 100–10000x efficiency improvements over existing methods.

Library auto-documentation (Fig. 1C). Unlike traditional program synthesis methods, LLMs (like human programmers) are sensitive to function names [24–26]. However, PL tools are typically not equipped to write human-readable function names, instead outputting anonymous lambda abstractions (e.g., `fn_0`, Fig. 1B). In early experiments, we observed that naively providing a LLM with Stitch abstractions measurably degraded its ability to solve tasks (§3). Motivated by these findings, as part of LILO, we introduce a *library auto-documentation* (AutoDoc) procedure inspired by ideas from code deobfuscation [27–29]. During AutoDoc, we sequentially prompt a LLM to produce a name and docstring for each abstraction in the library (Fig. 6). In §3, we explore how AutoDoc benefits downstream synthesis performance, yielding both richer and more interpretable libraries.

3 Experiments and Results

Experiment setup. Our experiments are designed to simulate a "lifelong learning" setting where the learner must generalize a small set of seed examples to a broader space of tasks that range in complexity. We sequentially perform two experiments that test different aspects of models' learning. First, in **online synthesis**, each model runs for a fixed number of iterations, continually updating its library (if applicable) and attempting to solve test tasks. Next, in **offline synthesis**, we freeze the final library $\mathcal{L}_f$ from each online synthesis run and perform enumerative search with no language guidance for a fixed time budget. We hold the hyperparameters of the search fixed so that performance depends entirely on $\mathcal{L}_f$ and not on the original model. Thus, the offline synthesis evaluations provide a controlled comparison of the off-the-shelf utility of different learned libraries.

Models and metrics. We compare LILO against two baselines: a language-guided DreamCoder variant [22] and a non-library learning baseline (LLM Solver). For LLM-guided search, we queried OpenAI's Codex model (`code-davinci-002`) with up to 4 prompts per task, sampling 4 completions per prompt. For AutoDoc, we found that OpenAI's newer instruction-tuned models (`gpt-3.5-turbo` and `gpt-4`) better adhered to the AutoDoc task and schema. Further implementation details can be found in Apxs. B.4–B.5. To study the effects of the different LILO components, we introduce ablated variants that remove the enumerative search and/or AutoDoc steps. Tab. 1 gives the full breakdown of our experimental results. Throughout, comparisons between models are expressed in terms of absolute percentage point changes in mean solve rates on an i.i.d. test set.

	REGEX			CLEVR			LOGO		
MODEL	*max*	*mean*	*std*	*max*	*mean*	*std*	*max*	*mean*	*std*
DreamCoder	45.60	43.93	1.53	97.09	94.50	2.44	36.94	28.53	13.79
LLM Solver	90.00	76.13	12.04	90.29	88.67	1.48	41.44	32.13	8.07
LLM Solver (+ Search)	91.20	76.60	13.02	97.09	96.44	0.56	45.05	37.84	6.80
LILO (✂ Search / AutoDoc)	59.40	53.20	5.38	93.20	85.76	9.72	45.05	21.02	20.88
LILO (✂ Search)	63.80	62.93	1.50	94.17	88.03	8.26	30.63	21.02	9.46
LILO	**93.20**	**77.07**	14.14	**99.03**	**96.76**	3.12	**73.87**	**48.95**	22.15
Base DSL	22.00	22.00	0.00	29.13	29.13	0.00	0.90	0.90	0.00
DreamCoder	42.00	41.60	0.40	94.17	91.59	2.97	36.04	30.63	7.85
LLM Solver*	48.60	43.00	5.17	91.26	89.64	2.02	36.04	27.33	7.56
LLM Solver (+ Search)*	63.40	55.67	7.51	91.26	89.00	3.92	28.83	27.63	1.04
LILO (✂ Search / AutoDoc)	60.80	50.73	8.85	95.15	93.85	2.24	**51.35**	30.63	18.22
LILO (✂ Search)	57.60	56.20	2.25	**96.12**	**95.79**	0.56	28.83	26.13	3.25
LILO	**71.40**	**64.27**	6.31	**96.12**	92.56	6.17	50.45	**41.14**	8.66

Table 1: **Task solution rates for online (upper) and offline (lower) synthesis experiments.** We report the best (*max*), average (*mean*), and standard deviation (*std*) test solve rates across model runs. In each mean column, results within 1σ of the best (**bold**) result are underlined. *Asterisk indicates $\mathcal{L}_f$ computed *post-hoc*.

LILO achieves the strongest overall performance in online synthesis. As observed in Fig. 2, LILO significantly outperforms DreamCoder on REGEX (+33.14) and LOGO (+20.42). It also achieves small improvements on CLEVR (+2.26), though DreamCoder is already quite strong on this domain. LILO also improves on the LLM Solver baseline by +0.94–16.82, thanks in part to its ability to

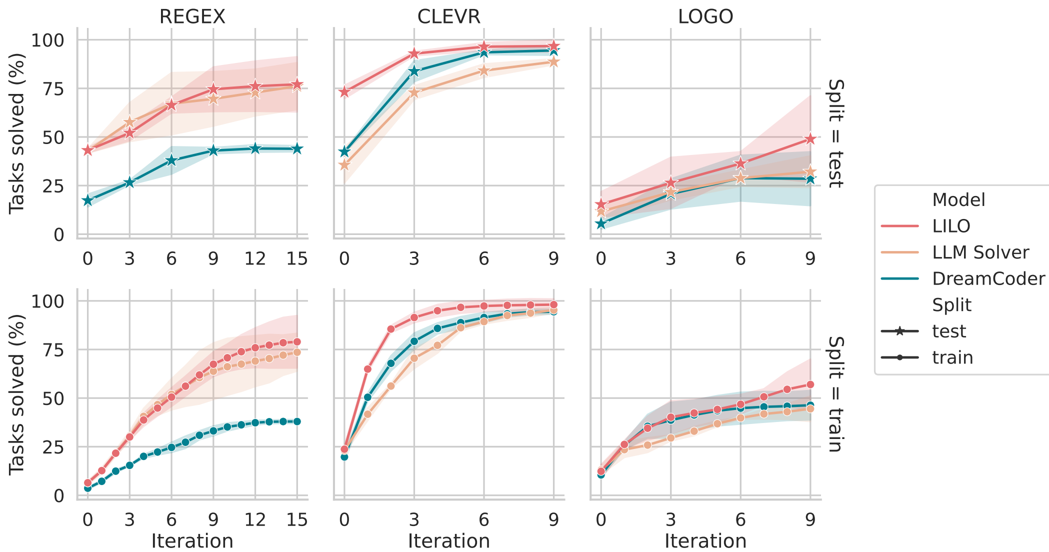

Figure 2: **Learning curves during online synthesis.** Within each plot, the x-axis tracks the experiment iteration and the y-axis shows the percent of tasks solved (top = test, bottom = train). Error bars show standard deviation across 3 randomly-seeded runs.

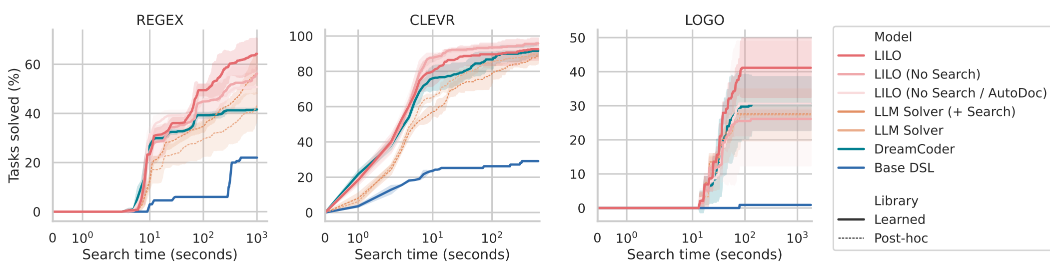

Figure 3: **Evaluating library quality via offline synthesis.** We run a timed enumerative search (x-axis; note the log-scale) with the final library $\mathcal{L}_f$ learned by each model in online synthesis or inferred *post-hoc*. In this setting, LILO's $\mathcal{L}_f$ expedites discovery of test task solutions (y-axis) even without language guidance.

discover novel program structures via enumerative search. To isolate the effects of search, we ran an ablation [LILO (✂ Search)] as well as an augmented baseline [LLM Solver (+ Search)]. We find that search is most helpful on LOGO, which requires certain domain-specific program structures (e.g., how to draw a "snowflake" or "staircase"; see Fig. 4) that are difficult to infer from language alone.

Auto-documentation unlocks effective contextual usage of abstractions. Early experiments revealed a puzzling finding: providing the LLM with abstractions did not help—and in some cases, hurt—online synthesis performance [Tab. 1, LILO (✂ Search / AutoDoc)]. Relative to the LLM Solver baseline, we observed solution rate changes of -30.60 (REGEX), -2.91 (CLEVR), and -11.11 (LOGO) after introducing Stitch compression [Tab. 1, LILO (✂ Search / AutoDoc)]. Qualitative inspection found that Codex struggled to deploy anonymous abstractions in context. After introducing AutoDoc, we saw mean improvements of $+9.73$ (REGEX) and $+2.27$ (CLEVR) over the naive condition.

LILO libraries generalize well even in the absence of language. In our offline synthesis experiments, we tested each model's final library $\mathcal{L}_f$ in an off-the-shelf enumerative search with no language guidance (Fig. 3). As the baseline for each domain, we measure synthesis performance in $\mathcal{L}_0$ (Base DSL). As expected, we can significantly outperform $\mathcal{L}_0$ using library learning: DreamCoder's $\mathcal{L}_f$ improves on $\mathcal{L}_0$ by $+19.6$–62.5 and LILO's $\mathcal{L}_f$ adds $+1.0$–22.7 over DreamCoder. LILO's $\mathcal{L}_f$ also outperforms libraries derived *post-hoc* from the two LLM Solver baselines, highlighting the benefits of performing compression and documentation in-the-loop. As these results demonstrate, LILO learns high-quality libraries that generalize well to downstream synthesis tasks even when no language annotations are available at test time.

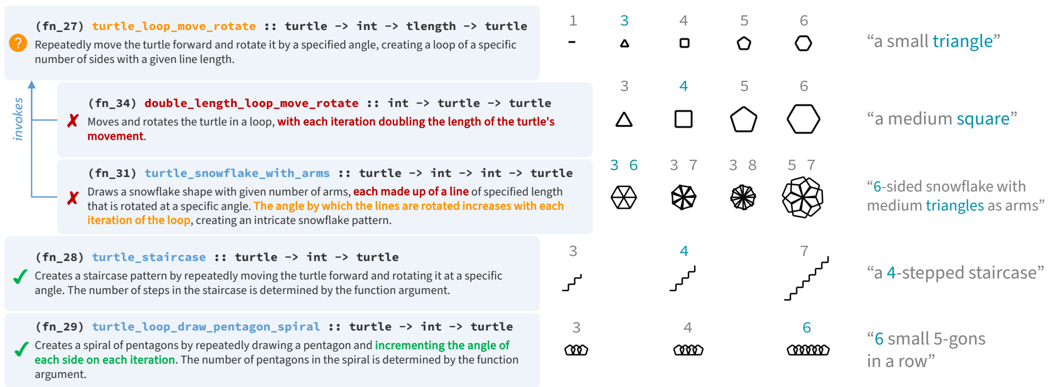

Figure 4: **Qualitative inspection of learned LOGO library.** Highlights indicate ambiguities (orange) and errors (red) in naming and documentation that may affect code comprehension, which we discuss below.

Libraries learned by LILO exhibit examples of hierarchical reuse. For instance, in the LOGO library (Fig. 4 and Apx. C.2.3), the top abstraction is a general method for drawing polygons that is invoked by several higher-level abstractions. Similarly, in the CLEVR library (Fig. 1 and Apx. C.2.2), a set of learned `filter` operations over color, shape, material, etc. supports a higher layer of more specialized abstractions. These examples showcase how LILO builds on one of the main strengths of DreamCoder—the ability to bootstrap hierarchies of learned concepts—while improving the richness and interpretability of libraries through documentation.

AutoDoc occasionally struggles to infer semantics. For instance, in LOGO (Fig. 4), `fn_27` and `fn_34` are assigned relatively uninformative names that emphasize their implementation (looping `move` and `rotate`) but not their behavior (drawing polygons). Moreover, AutoDoc occasionally "doubles down" on particular statements that may be correct in one context but not another. For example, it correctly notes that `fn_27` works by "incrementing the angle of each side on each iteration," but this idea is ambiguous in `fn_31` (*which angle?*) and incorrect in `fn_34` (*the length is constant, not doubling*). In addition to affecting interpretability, these semantic errors may also impact downstream synthesis performance in LLM-guided search. Future work could adopt self-consistency and verification techniques [30, 31] to improve the quality of AutoDoc generations.

4 Discussion and Conclusion

While LILO improves on prior library learning approaches, notably, the LLM-only baseline also demonstrates the ability to bootstrap its performance over time. This result aligns with recent successes in automated prompting [32, 33], suggesting that transformer attention can be viewed as implementing a form of *non-compressive* library learning where information is accumulated in the prompt. However, it is unclear whether this approach will scale to large software libraries: as context length grows, key information may be ignored due to ordering effects [34–36]. Accordingly, an important line of research looks to equip LLMs with long-term memory through retrieval [37, 38], self-reflection [39], or combinations of both that enable learning libraries of programmatic skills in embodied environments [40]. Currently, these approaches face the common challenge of determining what information to preserve, leading to a large space of *ad hoc* heuristics.

LILO offers a principled approach to the consolidation of knowledge in a lifelong learning setting, adding compression to a growing toolkit of LLM integrations with symbolic computation [41, 42]. Moreover, given Stitch's algorithmic generality, extending LILO to imperative languages (e.g., Python) reduces to a tractable and compelling PL research problem. Thus, LILO offers a blueprint for collaboration between the ML and PL communities towards the longstanding goal of learning interpretable software libraries that enable solutions to novel problem domains.

Acknowledgments

This work benefited greatly from discussions with colleagues in academia and industry, including Sam Acquaviva, Ekin Akyürek, Jacob Austin, Armando Solar-Lezama, and Belinda Li. We are thankful to Jack Feser for his OCaml wizardry. Finally, LILO owes a significant debt to ideas and infrastructure pioneered by Kevin Ellis, and we are deeply grateful for his feedback and guidance.

The authors gratefully acknowledge support from the MIT Quest for Intelligence, the MIT-IBM Watson AI Lab, the Intel Corporation, AFOSR, DARPA, and ONR. GG and MB are supported by the National Science Foundation (NSF) under Grant No. 2141064. GG was additionally supported by the MIT Presidential Fellowship. TXO is supported by the Herbert E. Grier (1933) and Dorothy J. Grier Fund Fellowship and the DARPA ASKEM program (award #HR00112220042). GG, MB, TXO, and JA are supported by the National Science Foundation (NSF) and Intel through NSF Grant CCF:2217064. JA is supported by NSF Grant IIS-2144855. LW and JBT received support from AFOSR Grant #FA9550-19-1-0269, the MIT-IBM Watson AI Lab, ONR Science of AI and the DARPA Machine Common Sense program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of sponsors.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[2] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *ArXiv preprint*, abs/2204.05999, 2022. URL https://arxiv.org/abs/2204.05999.

[3] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: May the source be with you!, 2023. URL https://arxiv.org/abs/2305.06161.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[5] 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. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iaYcJKpY2B_.

[6] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *ArXiv preprint*, abs/2105.09938, 2021. URL https://arxiv.org/abs/2105.09938.

[7] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[8] Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *ArXiv preprint*, abs/2207.14502, 2022. URL https://arxiv.org/abs/2207.14502.

[9] OpenAI. GPT-4 Technical Report, 2023. URL https://arxiv.org/abs/2303.08774.

[10] William H Brown, Raphael C Malveau, Hays W "Skip" McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[11] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *ArXiv preprint*, abs/1709.08439, 2017. URL https://arxiv.org/abs/1709.08439.

[12] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 835–850, 2021.

[13] Jacob Andreas, Dan Klein, and Sergey Levine. Learning with latent language. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2166–2179, New Orleans, Louisiana, 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1197. URL https://aclanthology.org/N18-1197.

[14] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 1988–1997. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.215. URL https://doi.org/10.1109/CVPR.2017.215.

[15] Harold Abelson and Andrea diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. The MIT Press, 1986. ISBN 9780262362740. doi: 10.7551/mitpress/6933.001.0001. URL https://doi.org/10.7551/mitpress/6933.001.0001.

[16] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

[17] Eyal Dechter, Jonathan Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1302–1309. IJCAI/AAAI, 2013. URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6890.

[18] Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 4(26):eaav3150, 2019.

[19] Steven A Sloman. The empirical case for two systems of reasoning. *Psychological bulletin*, 119(1):3, 1996.

[20] Jonathan St.B.T. Evans. In two minds: dual-process accounts of reasoning. *Trends in Cognitive Sciences*, 7(10):454–459, 2003. ISSN 1364-6613. doi: https://doi.org/10.1016/j.tics.2003.08.012. URL https://www.sciencedirect.com/science/article/pii/S1364661303002250.

[21] Daniel Kahneman. *Thinking, fast and slow*. Farrar, Straus and Giroux, New York, 2011. ISBN 9780374275631 0374275637.

[22] Catherine Wong, Kevin Ellis, Joshua B. Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 11193–11204. PMLR, 2021. URL http://proceedings.mlr.press/v139/wong21a.html.

[23] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), 2023. doi: 10.1145/3571234. URL https://doi.org/10.1145/3571234.

[24] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=PH5PH9ZO_4.

[25] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 39–51, Virtual South Korea, 2022. ACM. ISBN 978-1-4503-9379-9. doi: 10.1145/3533767.3534390.

[26] Antonio Valerio Miceli-Barone, Fazl Barez, Ioannis Konstas, and Shay B. Cohen. The Larger They Are, the Harder They Fail: Language Models do not Recognize Identifier Swaps in Python, 2023. URL https://arxiv.org/abs/2305.15507.

[27] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. DOBF: A de-obfuscation pre-training objective for programming languages. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 14967–14979, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/7d6548bdc0082aacc950ed35e91fcccb-Abstract.html.

[28] Pratyusha Sharma, Antonio Torralba, and Jacob Andreas. Skill induction and planning with latent language. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1713–1726, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.120. URL https://aclanthology.org/2022.acl-long.120.

[29] 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. ISSN 2475-1421. doi: 10.1145/3571226.

[30] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *ArXiv preprint*, abs/2203.11171, 2022. URL https://arxiv.org/abs/2203.11171.

[31] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*, 2023.

[32] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

[33] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models, 2022. URL https://arxiv.org/abs/2210.03629.

[34] Joe O'Connor and Jacob Andreas. What context features can transformer language models use? In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 851–864, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.70. URL https://aclanthology.org/2021.acl-long.70.

[35] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8086–8098, Dublin, Ireland, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.556. URL https://aclanthology.org/2022.acl-long.556.

[36] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts, 2023. URL https://arxiv.org/abs/2307.03172.

[37] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=TrjbxzRcnf-.

[38] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior, 2023. URL https://arxiv.org/abs/2304.03442.

[39] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, 2023. URL https://arxiv.org/abs/2303.11366.

[40] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, 2023. URL https://arxiv.org/abs/2305.16291.

[41] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *ArXiv preprint*, abs/2302.04761, 2023. URL https://arxiv.org/abs/2302.04761.

[42] Stephen Wolfram. ChatGPT gets its "Wolfram Superpowers", 2023. URL https://writings.stephenwolfram.com/2023/03/chatgpt-gets-its-wolfram-superpowers/.

[43] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

[44] Ray J Solomonoff. A formal theory of inductive inference. Part I. *Information and control*, 7 (1):1–22, 1964.

[45] Andrei Nikolaevich Kolmogorov. Three approaches for defining the concept of information quantity. *Problemy peredaci informacii*, 1:3–11, 1965.

[46] Gregory J Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM (JACM)*, 13(4):547–569, 1966.

[47] Mark Johnson. PCFG models of linguistic tree representations. *Computational Linguistics*, 24 (4):613–632, 1998. URL https://aclanthology.org/J98-4004.

[48] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[49] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 639–646. Omnipress, 2010. URL https://icml.cc/Conferences/2010/papers/568.pdf.

[50] Miltiadis Allamanis, Earl T. Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. Mining Semantic Loop Idioms. *IEEE Transactions on Software Engineering*, 44 (7):651–668, 2018. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2018.2832048.

[51] Eui Chul Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10824–10834, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/cff34ad343b069ea6920464ad17d4bcf-Abstract.html.

[52] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[53] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

[54] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=rJ0JwFcex.

[55] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 2017. URL http://proceedings.mlr.press/v70/devlin17a.html.

[56] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4861–4870. PMLR, 2019. URL http://proceedings.mlr.press/v97/nye19a.html.

[57] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. doi: 10.1126/science.aab3050. URL https://www.science.org/doi/abs/10.1126/science.aab3050.

[58] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6062–6071, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/6788076842014c83cedadbe6b0ba0314-Abstract.html.

[59] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 472–483, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635901. URL https://doi.org/10.1145/2635868.2635901.

[60] Tessa A. Lau and Daniel S. Weld. Programming by demonstration: An inductive learning formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, pages 145–152, Los Angeles California USA, 1998. ACM. ISBN 978-1-58113-098-0. doi: 10.1145/291080.291104.

[61] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning better abstractions with e-graphs and anti-unification. *Proc. ACM Program. Lang.*, 7(POPL):396–424, 2023. doi: 10.1145/3571207. URL https://doi.org/10.1145/3571207.

[62] Mehdi Manshadi, Daniel Gildea, and James Allen. Integrating programming by example and natural language programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, page 661–667. AAAI Press, 2013.

[63] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 792–800. AAAI Press, 2015. URL http://ijcai.org/Abstract/15/117.

[64] Percy Liang, Michael Jordan, and Dan Klein. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 590–599, Portland, Oregon, USA, 2011. Association for Computational Linguistics. URL https://aclanthology.org/P11-1060.

[65] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013. doi: 10.1162/tacl_a_00209. URL https://aclanthology.org/Q13-1005.

[66] David L. Chen and Raymond J. Mooney. Learning to interpret natural language navigation instructions from observations. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3701.

[67] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.

[68] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural Language to Code Generation in Interactive Data Science Notebooks, 2022. URL https://arxiv.org/abs/2212.09248.

[69] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. Parsel: A (De-)compositional Framework for Algorithmic Reasoning with Language Models, 2022. URL https://arxiv.org/abs/2212.10561.

[70] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993. URL https://aclanthology.org/J93-2003.

[71] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 39–48. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.12. URL https://doi.org/10.1109/CVPR.2016.12.

[72] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 804–813. IEEE Computer Society, 2017. doi: 10.1109/ICCV.2017.93. URL https://doi.org/10.1109/ICCV.2017.93.

[73] David C Knill and Whitman Richards. *Perception as Bayesian inference*. Cambridge University Press, 1996.

[74] Daniel Kersten, Pascal Mamassian, and Alan Yuille. Object perception as Bayesian inference. *Annu. Rev. Psychol.*, 55:271–304, 2004.

[75] Alan Yuille and Daniel Kersten. Vision as Bayesian inference: analysis by synthesis? *Trends in cognitive sciences*, 10(7):301–308, 2006.

[76] Tai Sing Lee and David Mumford. Hierarchical Bayesian inference in the visual cortex. *JOSA A*, 20(7):1434–1448, 2003.

[77] Jiajun Wu, Ilker Yildirim, Joseph J. Lim, Bill Freeman, and Joshua B. Tenenbaum. Galileo: Perceiving physical object properties by integrating a physics engine with deep learning. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 127–135, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/d09bf41544a3365a46c9077ebb5e35c3-Abstract.html.

[78] Ilker Yildirim, Mario Belledonne, Winrich Freiwald, and Josh Tenenbaum. Efficient inverse graphics in biological face processing. *Science Advances*, 6(10):eaax5979, 2020. doi: 10.1126/sciadv.aax5979. URL https://www.science.org/doi/abs/10.1126/sciadv.aax5979.

[79] Jiajun Wu, Joshua B. Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 7035–7043. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.744. URL https://doi.org/10.1109/CVPR.2017.744.

[80] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic VQA: disentangling reasoning from vision and language understanding. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 1039–1050, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/5e388103a391daabe3de1d76a6739ccd-Abstract.html.

[81] Nishad Gothoskar, Marco F. Cusumano-Towner, Ben Zinberg, Matin Ghavamizadeh, Falk Pollok, Austin Garrett, Josh Tenenbaum, Dan Gutfreund, and Vikash K. Mansinghka. 3DP3: 3D scene perception via probabilistic programming. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 9600–9612, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/4fc66104f8ada6257fa55f29a2a567c7-Abstract.html.

[82] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.deelio-1.10. URL https://aclanthology.org/2022.deelio-1.10.

[83] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

[84] Thomas N. Theis and H.-S. Philip Wong. The end of Moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017. doi: 10.1109/MCSE.2017.29.

Appendix

Table of Contents

A Background: Program Search and Library Learning

Program synthesis. In inductive program synthesis [43], we are given a library of primitives $\mathcal{L} = \{f_1, f_2, \ldots\}$ that forms a **domain-specific language (DSL)**. For a given programming task $t = \{(x_i, y_i)\}$ specified as a set of input-output pairs, the goal is to find a program $\pi : \forall_i \pi(x_i) = y_i$ that correctly maps all inputs to outputs, denoted $\pi \vdash t$. However, a typical task admits many such solutions that will not necessarily generalize (for instance, a simple lookup table). To address this inherent under-specification, concern is given to finding an *optimal* program $\hat{\pi} \vdash t$ with respect to descriptive complexity [44–46]. This optimization is naturally framed in terms of probabilistic inference:

$$\arg\max_{\pi} \log p(\pi \mid t, \mathcal{L}) = \arg\max_{\pi} \left[\log p(t \mid \pi) + \log p(\pi \mid \mathcal{L})\right] \tag{1}$$

In a typical setting, the likelihood $p(t \mid \pi) \triangleq \mathbb{1}_{\pi \vdash t}$ is computed via program execution, while the prior $p(\pi \mid \mathcal{L}) \triangleq \prod_{f \in \pi} p(f \mid \mathcal{L})$ is defined under a probabilistic context free grammar (PFCG; 47) that assigns a weight $0 \leq \theta_f \leq 1$ to each primitive $f \in \mathcal{L}$. This is equivalent to a weighted *description length* prior, where longer programs have lower probability.

This formulation highlights the central challenge of program synthesis: historically, approaches to Eq. 1 have inevitably involved enumerative search through a combinatoral space of programs. A range of techniques have been proposed to improve search tractability, including type-directed synthesis [48], Monte Carlo approximation [49–51], and neural network guidance [52–56, 12]. However, even with these methods, traditional program synthesis hinges critically on DSL design. Omission of key primitives can make complex tasks unsolvable, while inclusion of extraneous primitives can make search intractable. Consequently, DSL engineering is a painstaking process that requires significant expertise to anticipate common patterns across tasks in a domain.

Library learning. While classical approaches focus on synthesizing the best program for a task specification given a fixed DSL (as in Eq. 1), programmers in the wild are typically concerned with solving entire problem domains. Given the difficulty of manual DSL engineering, a natural evolution is to include $\mathcal{L}$ itself as part of the optimization problem. This is the main intuition behind *library learning* methods [49, 17, 57, 51, 18, 58, 12], which start with a collection of tasks $\mathcal{T} = \{t_1, t_2, \ldots\}$ and a base library $\mathcal{L}_0$, and jointly infer an expanded library $\mathcal{L} = \mathcal{L}_0 \cup \{f_1^*, \ldots, f_k^*\}$ that includes additional **abstractions** f^* built from $\mathcal{L}_0$ (Fig. 1B) and programs $\Pi = \{\pi_1, \pi_2, \ldots\}$ written in terms of $\mathcal{L}$:

$$\arg\max_{\Pi,\mathcal{L}} \log p(\Pi, \mathcal{L} \mid \mathcal{T}, \mathcal{L}_0) = \arg\max_{\Pi,\mathcal{L}} \left[\sum_{t \in \mathcal{T}} \log p(t \mid \pi_t) + \log p(\pi_t \mid \mathcal{L})\right] + \log p(\mathcal{L} \mid \mathcal{L}_0) \tag{2}$$

This objective carries over the program prior and likelihood from Eq. 1, but introduces a distribution over libraries $p(\mathcal{L} \mid \mathcal{L}_0)$, typically also defined in terms of description length. Intuitively, Eq. 2 is optimized by inventing abstractions that are both *reusable*, simplifying the solutions to multiple tasks in $\mathcal{T}$; and *concise*, ideally building on one another hierarchically so as to share logic. [12] approximate Eq. 2 via coordinate ascent, alternating between a **search step**, which holds the library fixed and searches for task solutions Π, and a **refactoring step**, which extracts common structure from the solution set to update $\mathcal{L}$. The tractability of this approach hinges critically on the ability to do efficient refactoring, which we discuss further in §2.

Refactoring and compression. Various algorithms for refactoring have been proposed using combinatory logic [49], tree substitution grammars [59, 50, 51], version spaces [60, 12], and e-graphs [61]. In LILO, we cast refactoring as a *compression problem* over a corpus of programs

$$f^* = \underset{\mathcal{L}}{\text{COMPRESS}}(\Pi) = \arg\min_{f} |f| + \sum_{\pi \in \Pi} |\underset{\mathcal{L} \cup \{f\}}{\text{REWRITE}}(\pi)| \tag{3}$$

where the goal is to identify abstractions with minimal description length $|f|$ that facilitate efficient rewriting of Π. However, performing even a single round of compression as in Eq. 3 necessitates an efficient search strategy. In LILO, we leverage recent algorithmic advances from STITCH [23]: a symbolic compression system that uses branch-and-bound search to identify reusable abstractions in large datasets of lambda calculus programs. As Bowers et al. demonstrate, Stitch is 1000–10000x faster and 100x more memory efficient than DreamCoder's compression algorithm. Nevertheless, prior analyses were limited to static program corpora; in LILO, we perform the first experiments using Stitch as part of a program synthesis loop. We find Stitch similarly performant on our domains, typically running in seconds on a single CPU. These efficiency improvements enable us to re-derive the entire library from $\mathcal{L}_0$ at every iteration (Alg. 1 line 9). While many abstractions remain stable across iterations, this "deep refactoring" allows LILO to discard suboptimal abstractions discovered early in learning.

Leveraging language guidance. Given the size of the search space, generic priors such as description length are not always sufficient to solve Eq. 1; for this reason, a line of work considers natural language task descriptions d_t as an additional source of learning signal [62, 63, 51]. Traditionally, making use of such descriptions has required learning a domain-specific semantic parsing model [64–66]. More recent work [67–69] uses LLMs, which excel when $\mathcal{L}$ resembles a common programming language that is well-represented in pretraining.

In library learning settings—where $\mathcal{L}$ is novel by construction—it is currently less clear how to leverage language. In LAPS (Language for Abstraction and Program Search), [22] generalize Eq. 2 to condition on d_t by fitting an inverted "program-to-language" translation model. However, learning this mapping from scratch necessitates the use of a small alignment model (*IBM Model 4*; 70) that makes strict token-to-token decomposition assumptions. In LILO, we take the opposite approach: we start with a large model that already has strong priors over the joint distribution of language and code; then, we adapt the *library* to resemble this distribution by building up contextual examples and documentation. In contrast to simply picking a more common $\mathcal{L}$ (e.g., Python) to work in, this procedure enables us to *learn* a new $\mathcal{L}$ on-the-fly that is both optimized for the domain and grounded in natural language.

B Methods

B.1 LILO Algorithm

Algorithm 1 Library learning loop with LILO

function LILOLEARNING($\mathcal{L}_0, \mathcal{T}$)
  $\mathcal{L} \leftarrow \mathcal{L}_0$ ▷ *Initialize library with base DSL*
  $\Pi \leftarrow \{t : \emptyset \mid t \in \mathcal{T}\}$ ▷ *Initialize task solution set*
  for $i = 1, \ldots, N$ **do**
    for $t \in \mathcal{T}$ **do** ▷ *Run LLM Solver*
      $\Pi_t \leftarrow \Pi_t \cup \mathrm{LLM}(\texttt{TaskPrompt}(\mathcal{L}, \Pi, d_t))$
    $\Pi \leftarrow \Pi \cup \mathrm{SEARCH}(\mathcal{L}_i, \mathcal{T})$ ▷ *Run enumerative search (skipped in ✂ Search)*
    $\{f_1^*, \ldots, f_k^*\} \leftarrow \mathrm{COMPRESS}(\mathcal{L}, \Pi, k)$ ▷ *Generate new abstractions*
    $\mathcal{L} \leftarrow \mathcal{L}_0 \cup \{f_1^*, \ldots, f_k^*\}$
    $\Pi \leftarrow \mathrm{REWRITE}(\mathcal{L}, \Pi)$
    for $\alpha \in \{f_1^*, \ldots, f_k^*\}$ **do** ▷ *Document abstractions (skipped in ✂ AutoDoc)*
      $\mathcal{D} \leftarrow \mathrm{LLM}(\texttt{AutoDocPrompt}(\mathcal{L}, \Pi, \alpha))$
      $\mathcal{L} \leftarrow \texttt{add\_docs}(\mathcal{L}, \alpha, \mathcal{D})$
  return $\mathcal{L}, \Pi$ ▷ *Return final library and task solutions*

B.2 LLM Solver Prompt

We introduce a "fast" approximate search model in string space that leverages the strong inductive biases learned by LLMs. Formally, we write $p_{\text{LLM}}(y \mid x)$ to denote the distribution over strings y produced by a language model prompted with string x. Then, for some target task $\hat{t}$, our goal is to approximate the conditional distribution over programs

$$p(\pi_{\hat{t}} \mid \mathcal{L}, \Pi, d_{\hat{t}}) \approx p_{\text{LLM}}(\langle \pi_{\hat{t}} \rangle \mid \underbrace{\langle f \mid f \in \mathcal{L} \rangle}_{\text{library functions}} \circ \underbrace{\langle (d_t, \pi_t) \mid \pi_t \sim \Pi \rangle}_{\text{program examples}} \circ \underbrace{\langle d_{\hat{t}} \rangle}_{\text{task desc.}}) \tag{4}$$

where $\langle \ldots \rangle$ and $\circ$ denote string serialization and concatenation, respectively. To sample from the distribution in Eq. 4, we procedurally construct few-shot prompts consisting of three parts: (1) A library description that enumerates the available primitives and any learned abstractions, (2) a set of exemplars consisting of description-solution pairs $(d_t, \pi_t) \sim \Pi$ sampled from the set of solved tasks, and (3) a description of the target task $d_{\hat{t}}$. For each completion, we run parsing, type inference, and execution checks to identify valid programs that solve the target task. Fig. 5 (below) illustrates the composition of a typical prompt; Apx. C.5 contains additional details on how examples are sampled.

Domain header & library functions

A

```
You are an expert programmer working in a language based on λ calculus. Your goal is to
write programs that accomplish the tasks specified by the user. This is a domain-specific
language for Logo turtle graphics. Write programs using the available functions:

logo_unit_angle :: tangle
logo_unit_line :: tlength
logo_zero_angle :: tangle
logo_zero_line :: tlength
logo_divide_angle :: tangle -> int -> tangle
logo_multiply_angle :: tangle -> int -> tangle
logo_divide_line :: tlength -> int -> tlength
logo_multiply_line :: tlength -> int -> tlength
logo_add_angles :: tangle -> tangle -> tangle
logo_subtract_angles :: tangle -> tangle -> tangle
logo_lift_pen :: (turtle -> turtle) -> turtle -> turtle
logo_move_pen_forward_rotate :: tlength -> tangle -> turtle -> turtle
logo_get_set_function_pen :: (turtle -> turtle) -> turtle -> turtle
logo_IFTY :: int
logo_epsilon_angle :: tangle
logo_epsilon_line :: tlength
logo_for_loop :: int -> (int -> turtle -> turtle) -> turtle -> turtle
```

Example task / program pairs *(sampled from solved tasks)*

B

```
Here are some example programs:

-- a small 8 gon
(λ (logo_for_loop 9 (λ (λ (logo_move_pen_forward_rotate logo_unit_line (logo_divide_angle
logo_unit_angle 8) $0))) $0))

-- a medium 6 gon (λ (logo_for_loop 6 (λ (λ (logo_move_pen_forward_rotate
(logo_multiply_line logo_unit_line 2) (logo_divide_angle logo_unit_angle 6) $0))) $0))

-- 5 small squares in a row
(λ (logo_for_loop 5 (λ (λ (logo_move_pen_forward_rotate logo_zero_line (logo_divide_angle
logo_unit_angle 4) (logo_for_loop 7 (λ (λ (logo_move_pen_forward_rotate logo_unit_line
(logo_divide_angle logo_unit_angle 4) $0))) $0)))) $0))

-- a vertical short line
(λ (logo_move_pen_forward_rotate logo_zero_line (logo_divide_angle logo_unit_angle 4)
(logo_move_pen_forward_rotate logo_unit_line logo_epsilon_angle $0)))

-- a small 9 gon
(λ (logo_for_loop logo_IFTY (λ (λ (logo_move_pen_forward_rotate logo_unit_line
(logo_divide_angle logo_unit_angle 9) $0))) $0))

<additional examples to fill context window>
```

Task description *(unsolved)*

C

```
-- 5 small triangles in a row
```

LLM completions

🏆 = solution
✅ = valid
❌ = invalid

D

```
✅ (λ (logo_for_loop 7 (λ (λ (logo_move_pen_forward_rotate logo_zero_line
(logo_divide_angle logo_unit_angle 3) (logo_for_loop 3 (λ (λ (logo_move_pen_forward_rotate
logo_unit_line (logo_divide_angle logo_unit_angle 3) $0))) $0)))) $0))

❌ (λ (logo_for_loop 5 (λ (λ (logo_for_loop 7 (λ (λ (logo_move_pen_forward_rotate
logo_unit_line (logo_divide_angle logo_unit_angle 3) $0))) $0)))) $0))

🏆 (λ (logo_for_loop 5 (λ (λ (logo_move_pen_forward_rotate logo_zero_line
(logo_divide_angle logo_unit_angle 3) (logo_for_loop 5 (λ (λ (logo_move_pen_forward_rotate
logo_unit_line (logo_divide_angle logo_unit_angle 3) $0))) $0)))) $0))

✅ (λ (logo_for_loop 5 (λ (λ (logo_move_pen_forward_rotate logo_zero_line
(logo_divide_angle logo_unit_angle 4) (logo_for_loop 7 (λ (λ (logo_move_pen_forward_rotate
logo_unit_line (logo_divide_angle logo_unit_angle 3) $0))) $0)))) $0))
```

Figure 5: **Anatomy of an LLM solver prompt.** (A) Each prompt begins with a short domain description followed by an autogenerated list of the DSL primitives and their type signatures. (B) We randomly sample task solutions and their language descriptions to construct the prompt body. (C) The final line of the prompt contains a target task description for an unsolved task. (D) We sample and parse $N = 4$ completions from the LLM, filter out invalid programs, and check for task solutions.

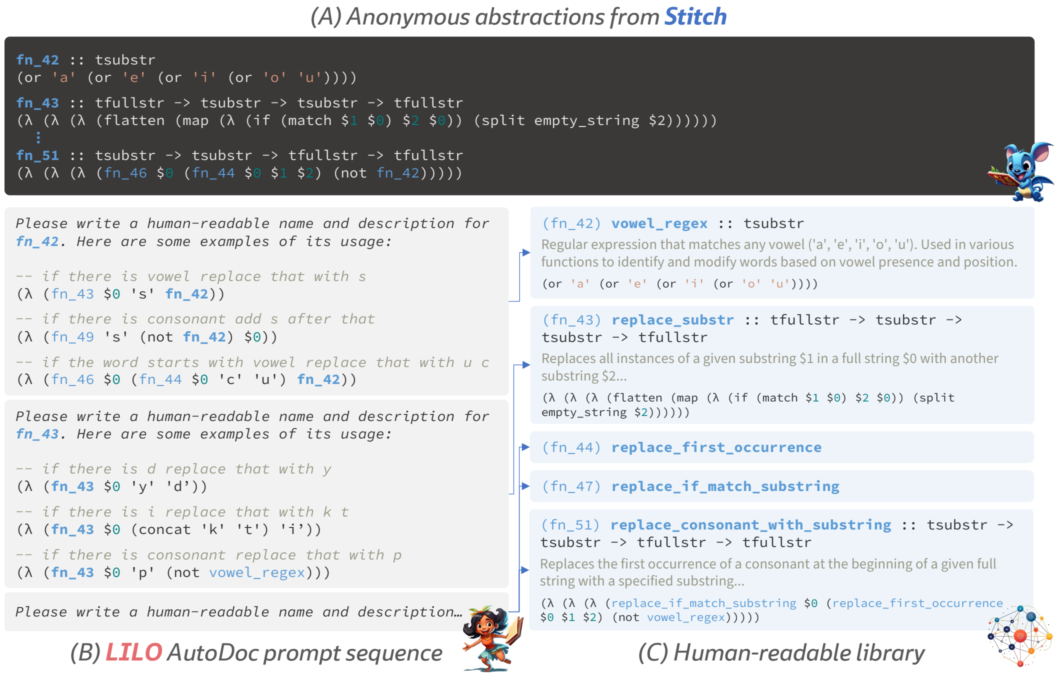

Figure 6: **LILO library auto-documentation (AutoDoc) workflow in the REGEX domain.** For each Stitch abstraction (A), we prompt an instruction-tuned LLM with usage examples from solved tasks (B) to generate a human-readable name and description (C). The chat-style structure of AutoDoc allows naming choices to cascade sequentially; e.g., `replace_consonant_with_substring` (`fn_51`) refers back to `vowel_regex` (`fn_42`) and other named abstractions in a consistent and interpretable manner.

B.3 Auto-Documentation Prompt

In this prototypical example in the REGEX domain, the LLM has solved some problems that require vowel substitutions. During compression, Stitch pulls out the expression `(or 'a' (or 'e' (or 'i' (or 'o' 'u'))))` for occurring commonly in the solution set and defines it as an anonymous arity-0 function (i.e., a constant). Subsequently, AutoDoc names this abstraction `vowel_regex`, which forms the basis for more complex expressions. For instance, *consonant* is expressed as `(not vowel_regex)`, which in turn is used to define an abstraction for consonant replacement. In §3, we explore how AutoDoc benefits downstream synthesis performance, yielding both richer and more interpretable libraries.

For reproducibility, we provide an example of the full text of an AutoDoc prompt sequence for the REGEX domain below. The prompt is composed of multiple pieces that are sent in serial as messages to the ChatGPT interface. The sequence begins with a header message describing the DSL. For pedagogical clarity, we consider the case where every abstraction except the final one have already assigned names. Thus, the header contains a mostly-documented library with the final `fn_51` remaining anonymous.

```
You are writing software documentation. Your goal is to write human-readable names for
the following library functions:

vowel_or :: tsubstr
(regex_or 'a' (regex_or 'e' (regex_or 'i' (regex_or 'o' 'u'))))
{- Matches any single vowel character ('a', 'e', 'i', 'o', 'u') using 'regex_or'
function. -}

replace_and_flatten :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(lambda (lambda (lambda (regex_flatten (regex_map (lambda (regex_if (regex_match $2 $0)
$1 $0)) (regex_split $1 $2))))))
{- Replaces all instances of a given substring with another substring, and returns the
resulting string flattened into one string. The first argument is the input string, the
second argument is the substring to be replaced, and the third argument is the
substring to use instead of the replaced substring. -}
```

```
                    ... <fn_44 - fn_50 omitted for concision> ...

fn_51 :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(lambda (lambda (lambda (regex_flatten (regex_cons $0 (regex_cons $1 (regex_cdr
(split_string_into_list $2))))))))
```

We then send a message prompting the LLM to document `fn_51`. At the end of the message, we request that the LLM encode the reply into a particular JSON format to facilitate downstream parsing.

```
Consider the following anonymous function:

fn_51 :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(lambda (lambda (lambda (regex_flatten (regex_cons $0 (regex_cons $1 (regex_cdr
(split_string_into_list $2))))))))

Here are some examples of its usage:

-- if the word starts with consonant any letter replace that with v d
(lambda (regex_if (regex_match (regex_not vowel_or) (regex_car (split_string_into_list
$0))) (fn_51 (regex_flatten (regex_cdr (split_string_into_list $0))) 'd' 'v') $0))

-- if the word starts with any letter vowel add q before that
(lambda (regex_if (regex_match vowel_or (regex_car (regex_cdr (split_string_into_list
$0)))) (fn_51 $0 (regex_car (split_string_into_list $0)) 'q') $0))

-- if the word starts with vowel replace that with u c
(lambda (regex_if (regex_match vowel_or (regex_car (split_string_into_list $0))) (fn_51
(regex_flatten (split_string_into_list $0)) 'c' 'u') $0))

            ... <additional usage examples omitted for concision> ...

Please write a human-readable name and description for `fn_51` in the JSON format shown
below.
Your `readable_name` should be underscore-separated and should not contain any spaces.
It should also be unique (not existing in the function library above).
If you cannot come up with a good name, please set `readable_name` to `null`.

{
    "anonymous_name": "fn_51",
    "readable_name": TODO,
    "description": TODO
}
```

We encountered difficulties in coaxing Codex to perform the AutoDoc task: the resulting function names were variable in quality, did not reliably capture the function semantics, and were embedded in generations that did not always adhere to the desired output specification. Instead, we take advantage of OpenAI's instruction-tuned `gpt-3.5-turbo` and `gpt-4` models, which we found adhered to the desired output JSON schema 100% of the time and never chose to return `null` for `readable_name`. We experimented with both `gpt-3.5-turbo` and `gpt-4` for AutoDoc and found both resulted in comparable synthesis performance on REGEX. However, GPT-4 was significantly slower: whereas `gpt-3.5-turbo` averaged 10-20 seconds for one iteration of AutoDoc, `gpt-4` averaged upwards of 2 minutes per iteration. We therefore chose to use `gpt-3.5-turbo` in the experiments reported in §3.

Unlike for the LLM Solver, we do not provide any few-shot examples of the desired transformations; all of this behavior is *zero-shot*, making AutoDoc an extremely domain-general technique that is easy to implement across a variety of settings.

B.4 Implementation Details

We provide a brief summary of key implementation details relevant to the experiments that are not reported in §3. We ran all experiments on AWS EC2 instances with machine specs tailored to suit the computational workload of each experiment.

Enumerative search. For experiments involving enumerative search, which is an embarrassingly parallel workload that scales linearly with the number of available CPUs, we ran on 96-CPU `c5.24xlarge` instances. These machines have the highest CPU count in the `c5` machine class. To take maximal advantage of the CPU parallelism, we set `batch_size=96` for these experiments (i.e., each iteration searches for solutions for a subset of 96 tasks). A convenient consequence of this implementation choice is that each task is allocated to a single, dedicated CPU, so the overall wall clock runtime of a single search iteration is equal to the per-task enumeration time budget. We set the enumeration budget on a per-domain basis using the timeouts from [22] (REGEX = 1000s, CLEVR = 600s, LOGO = 1800s). We ran DreamCoder until convergence on all domains. For CLEVR and LOGO, we performed 10 iterations of search, while for REGEX, we observed that the solve rate was still increasing at iteration 10, so we used a higher search budget of 16 iterations for this domain. Following [22] and based on a common practice in machine learning, we limited evaluation of the test set to every 3 iterations due to the computational cost of enumerative search.

GPT language models. For experiments in which GPT LLMs perform program search, the bulk of the computational workload is effectively offloaded to OpenAI's servers. Locally, the only requirements are that our machine is able to make API queries, process the results, and run compression. Accordingly, these experiments are run on `c5.2xlarge` machines with 8 CPUs each. (For experiments involving combinations of GPT queries and DreamCoder search, we use the larger `c5.24xlarge` machines.) To ensure comparability in solver performance between LLM-based and enumerative search-based experiments, we also run the LLM experiments with `batch_size=96` so that the learning timelines are aligned.

Our use of Codex for LLM-guided search was strongly motivated by resource considerations: we accessed Codex through OpenAI's free beta program for researchers, which saved thousands of USD over the project lifetime (see Apx. C.6 for a cost analysis) and afforded higher rate limits than paid GPT models. To preserve reproducibility, we make all Codex generations available at: `github.com/gabegrand/lilo`.

Stitch. For compression, we make use of the Stitch Python bindings, which interface with a fast backend written in Rust (`https://stitch-bindings.readthedocs.io/en/stable/`). Stitch exposes various hyperparameters, the most important of which are `iterations`, which governs the number of abstractions produced, and `max-arity`, which governs the maximum number of arguments that each abstraction can take. For all experiments, we set these to a constant `iterations=10` and `max-arity=3`. We note that Stitch will only produce an abstraction if it is *compressive*; i.e., it appears in multiple programs, and rewriting the corpus in terms of the abstraction reduces the overall description length. For this reason, in rare cases early on in learning, when only a handful of solved programs are available, the actual library size can be smaller than `iterations`. This behavior is beneficial in that it avoids introducing abstractions that have no utility and that might potentially negatively affect performance.

A summary of hyperparameters can be found in Apx. B.5. For further implementation details, we refer to our codebase: `github.com/gabegrand/lilo`.

B.5 Hyperparameters

We provide a summary of all key hyperparameters used in each component of LILO.

DreamCoder

Batch size:	96 tasks
Global iterations:	10 (CLEVR, LOGO), 16 (REGEX)
Search timeouts:	600s (CLEVR), 1000s (REGEX), 1800s (LOGO)
Neural recognition model:	10K training steps / iteration

Stitch

Max iterations:	10 (Controls max library size)
Max arity:	3 (Controls max arity of abstractions)

LILO: LLM Synthesizer

Prompts per task:	4
Samples per prompt:	4
GPT Model:	`code-davinci-002`
Temperature:	0.90
Max completion tokens β:	4.0x (Multiplier w/r/t the final prompt program.)

LILO: AutoDoc

Max usage examples: 10
GPT Model: `gpt-3.5-turbo-0301 / gpt-4-0314`
Top-P: 0.10
Max completion tokens: 256

C Experiments and Results

C.1 Domain Details

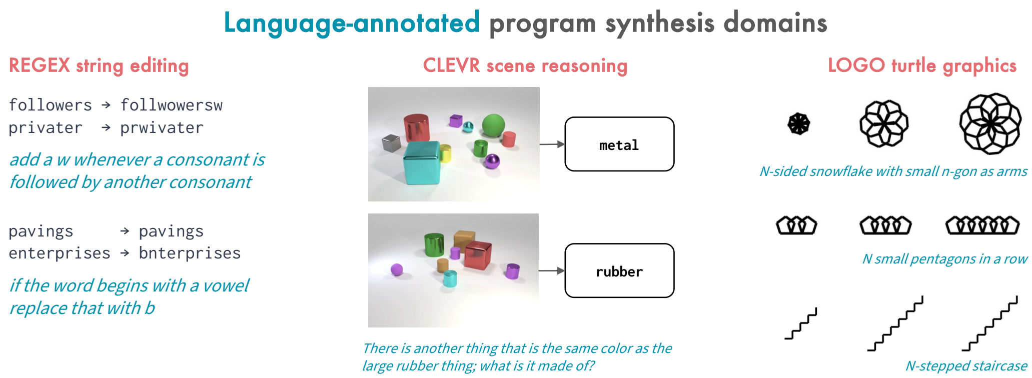

Figure 7: **Overview of domains.** We evaluate LILO on three *language-annotated* program synthesis domains: *string editing* with regular expressions, *scene reasoning* on the CLEVR dataset, and *graphics composition* in the 2D Logo turtle graphics language.

	#Tasks		Description length		String length	
Domain	Train	Test	Train	Test	Train	Test
REGEX	491	500	38.95 ± 26.11	41.03 ± 27.02	276.47 ± 179.92	262.74 ± 172.69
CLEVR	191	103	32.95 ± 15.78	30.82 ± 15.49	361.62 ± 182.06	387.44 ± 184.19
LOGO	200	111	24.65 ± 8.71	27.79 ± 8.19	250.98 ± 92.75	287.17 ± 89.65

Table 2: **Summary statistics for the domains used in this paper.** Description length is the number of terminals, lambda-abstractions and applications necessary to uniquely describe the ground truth program for each task; string length is the length of each program in terms of characters. Both are reported as the mean over the entire dataset plus/minus one standard deviation.

REGEX: String editing. We evaluate on a domain of *structured string transformation problems*–a classic task in inductive program synthesis [60]. The dataset, originally introduced in [13], contains procedurally-generated regular expressions that implement transformations on strings (e.g., *if the word ends with a consonant followed by "s", replace that with b*). Task examples consist of input/output pairs where the inputs are strings randomly sampled from an English dictionary and the outputs are the result of applying a particular string transformation. Following prior work [12, 22], the base DSL in this domain contains functional various programming primitives for string manipulation (`map`, `fold`, `cons`, `car`, `cdr`, `length`, `index`) and character constants. Each example comes with a synthetic language description of the task, which was generated by template based on human annotations [13].

CLEVR: Scene reasoning. We extend our approach to a *visual question answering* (VQA) task based on the CLEVR dataset [14]. Following successful efforts in modeling VQA as program synthesis [71, 72], each synthesis task is specified by a structured input scene and a natural language question. Outputs can be one of several types, including a number (*how many red rubber things are there?*), a boolean value (*are there more blue things than green?*), or another scene (*what if all of the red things turned blue?*). The dataset, designed by [22], uses a modified subset of the original CLEVR tasks and introduces new task types that require imagining or generating new scenes (e.g., *how many metal things would be left if all the blue cylinders were removed?*) that require learning new abstractions. The base DSL includes functional programming primitives similar to the regular expression domain, with domain-specific query functions and constants (e.g., `get_color(x)`; `get_shape(x)`; `blue`; `cube`). Input scenes are specified *symbolically* as scene graphs consisting of an array of structured objects defined as a dictionary of their attributes, and programs are designed to manipulate these structured arrays. Synthetic language annotations were generated based on the original high-level templates in [14] and human annotations were collected by [22].

LOGO: Turtle graphics. Following in a long tradition of modeling *vision as inverse graphics*, [73–81] we evaluate on a domain of *compositional drawing problems*. The dataset, originally introduced in [22] and based on a simpler dataset from [12], contains programs that generate shapes and designs in a vector graphics language. The DSL is based on Logo Turtle graphics [15], which originated from early symbolic AI research. Program expressions control the movement and direction of a pen (classically represented as a Turtle) on a canvas and can involve complex symmetries and recursions (e.g., *a seven sided snowflake with a short line and a small triangle as arms; a small triangle connected by a big space from a small circle*). The base DSL includes `for` loops, a stack for saving/restoring the pen state, and arithmetic on angles and distances [12]. Synthetic language annotations were generated with high-level templates over the objects and relations in each task; human annotations were collected by [22].

C.2 Learned Libraries and Graphical Maps

We generated graphical visualizations of the libraries learned by the best LILO model for each domain. Each graph includes the DSL primitives, the learned and named abstractions, and a random sample of 3 solved tasks that invoke each abstraction. Arrows indicate direction of reference; i.e., `fn_1 -> fn_2` indicates that `fn_1` invokes `fn_2`, and analogously for the tasks.

C.2.1 Library for REGEX

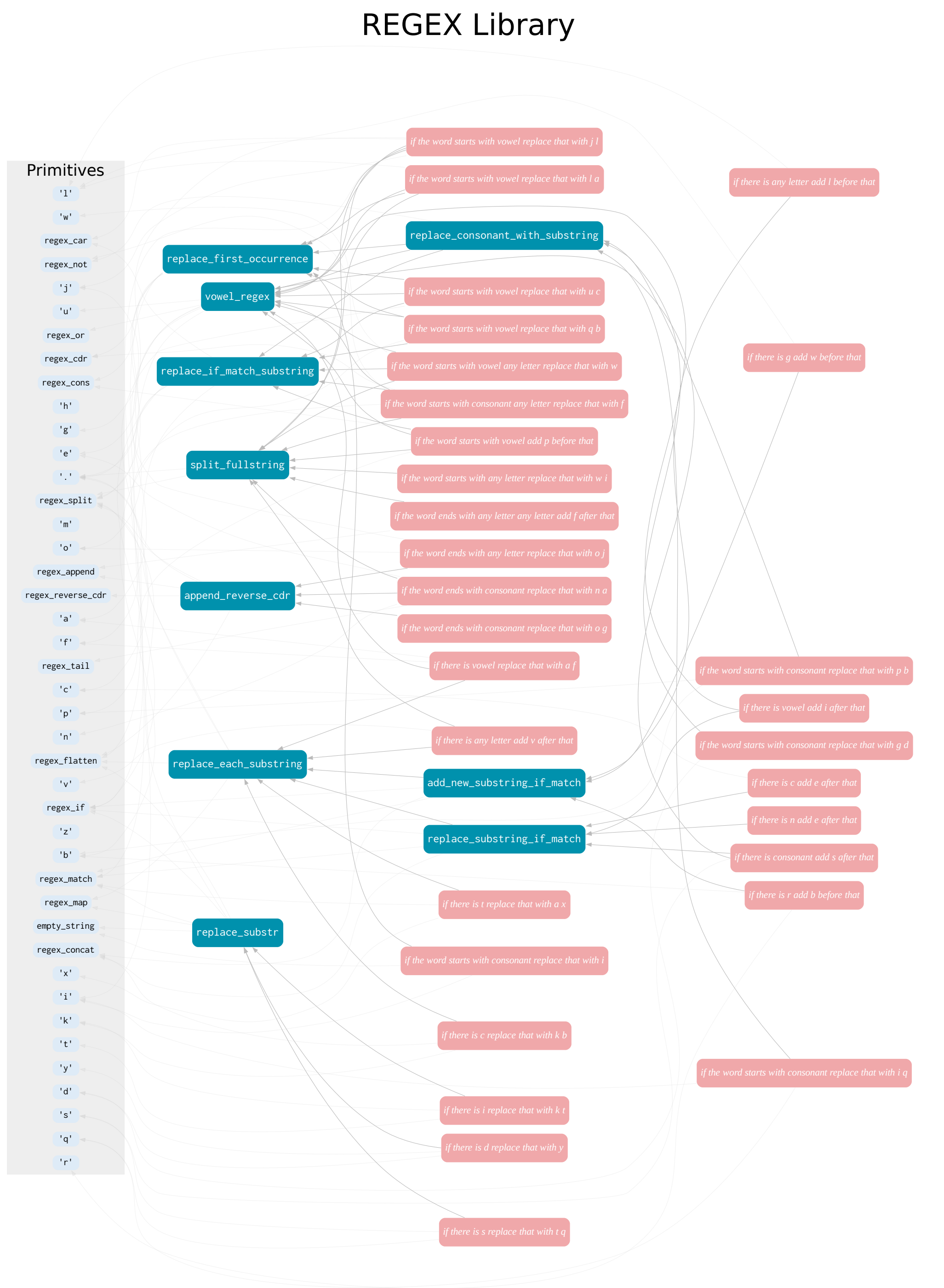

Figure 8: **Graphical map of REGEX library learned by LILO.** Named abstractions (turquoise) are hierarchically composed of other abstractions and ground out in the base DSL primitives (gray box).

```
(fn_42) vowel_regex :: tsubstr
(regex_or 'a' (regex_or 'e' (regex_or 'i' (regex_or 'o' 'u'))))
{- Regular expression that matches any vowel ('a', 'e', 'i', 'o', 'u'). Used in various
functions to identify and modify words based on vowel presence and position. -}

{- Example usages -}
--if there is consonant add s after that
(λ (replace_substring_if_match 's' (regex_not vowel_regex) $0))
--if the word starts with vowel replace that with j l
(λ (regex_if (regex_match (regex_not vowel_regex) (regex_car (split_fullstring $0))) $0
(replace_first_occurrence $0 'l' 'j')))
--if the word starts with vowel replace that with u c
(λ (replace_if_match_substring $0 (replace_first_occurrence $0 'c' 'u') vowel_regex))

(fn_43) replace_substr :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_flatten (regex_map (λ (regex_if (regex_match $1 $0) $2 $0))
(regex_split empty_string $2))))))
{- Replaces all instances of a given substring $1 in a full string $0 with another
substring $2. The substrings are separated by empty spaces. -}

{- Example usages -}
--if there is d replace that with y
(λ (replace_substr $0 'y' 'd'))
--if there is i replace that with k t
(λ (replace_substr $0 (regex_concat 'k' 't') 'i'))
--if there is s replace that with t q
(λ (replace_substr $0 (regex_concat 't' 'q') 's'))

(fn_44) replace_first_occurrence :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_flatten (regex_cons $0 (regex_cons $1 (regex_cdr (regex_split '.'
$2))))))))
{- Replaces the first occurrence of a substring $1 in a full string $0 with another
substring $2. The substrings are separated by periods. -}

{- Example usages -}
--if the word starts with vowel replace that with q b
(λ (replace_if_match_substring $0 (replace_first_occurrence $0 'b' 'q') vowel_regex))
--if the word starts with consonant replace that with i
(λ (replace_first_occurrence $0 empty_string 'i'))
--if the word starts with vowel replace that with l a
(λ (regex_if (regex_match (regex_not vowel_regex) (regex_car (split_fullstring $0))) $0
(replace_first_occurrence $0 'a' 'l')))

(fn_45) replace_each_substring :: tfullstr -> (tsubstr -> tsubstr) -> tfullstr
(λ (λ (regex_flatten (regex_map $0 (regex_split '.' $1)))))
{- Replaces each substring separated by periods in a given full string with a new
substring. The new substring can be manipulated with a λ function that takes each
substring as input. -}

{- Example usages -}
--if there is t replace that with a x
(λ (replace_each_substring $0 (λ (regex_if (regex_match 't' $0) (regex_concat 'a' 'x')
$0))))
--if there is vowel replace that with a f
(λ (replace_each_substring $0 (λ (regex_if (regex_match vowel_regex $0) (regex_concat
'a' 'f') $0))))
--if there is c replace that with k b
(λ (replace_each_substring $0 (λ (regex_if (regex_match 'c' $0) (regex_concat 'k' 'b')
$0))))

(fn_46) replace_if_match_substring :: tfullstr -> tfullstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_if (regex_match $0 (regex_car (regex_split '.' $2))) $1 $2))))
{- Replaces a given substring $2 in a full string $0 with another substring $1 if the
beginning of the string matches the target substring. All substrings are separated by
periods. -}
```

```
{- Example usages -}
--if the word starts with vowel add p before that
(λ (replace_if_match_substring $0 (regex_flatten (regex_cons 'p' (split_fullstring
$0))) vowel_regex))
--if the word starts with consonant any letter replace that with f
(λ (replace_if_match_substring $0 (regex_flatten (regex_cons 'f' (regex_cdr (regex_cdr
(split_fullstring $0))))) (regex_not vowel_regex)))
--if the word starts with vowel any letter replace that with w
(λ (replace_if_match_substring $0 (regex_flatten (regex_cons 'w' (regex_cdr (regex_cdr
(split_fullstring $0))))) vowel_regex))

(fn_47) add_new_substring_if_match :: tsubstr -> tsubstr -> tfullstr -> tfullstr
(λ (λ (λ (replace_each_substring $0 (λ (regex_if (regex_match $2 $0) (regex_concat $3
$0) $0))))))
{- Replaces each substring separated by periods in a given full string with a new
substring, if a specified substring is found. The new substring can be manipulated with
a λ function that takes each substring as input. -}

{- Example usages -}
--if there is g add w before that
(λ (add_new_substring_if_match 'w' 'g' $0))
--if there is any letter add l before that
(λ (add_new_substring_if_match 'l' '.' $0))
--if there is r add b before that
(λ (add_new_substring_if_match 'b' 'r' $0))

(fn_48) append_reverse_cdr :: tfullstr -> tsubstr -> tfullstr
(λ (λ (regex_flatten (regex_append $0 (regex_reverse_cdr (regex_split '.' $1))))))
{- Appends a new substring to the end of the given full string and reverses the order
of all substrings except for the last one (which is removed). -}

{- Example usages -}
--if the word ends with consonant replace that with o g
(λ (append_reverse_cdr $0 (regex_concat 'o' 'g')))
--if the word ends with consonant replace that with n a
(λ (regex_if (regex_match 'e' (regex_tail (split_fullstring $0))) $0
(append_reverse_cdr $0 (regex_concat 'n' 'a'))))
--if the word ends with any letter replace that with o j
(λ (append_reverse_cdr $0 (regex_concat 'o' 'j')))

(fn_49) replace_substring_if_match :: tsubstr -> tsubstr -> tfullstr -> tfullstr
(λ (λ (λ (replace_each_substring $0 (λ (regex_if (regex_match $2 $0) (regex_concat $0
$3) $0))))))
{- Replaces each substring separated by periods in a given full string with a new
substring, if a specified substring is found, using a λ function that takes the current
substring as input and replaces it with a new substring based on a condition. -}

{- Example usages -}
--if there is vowel add i after that
(λ (replace_substring_if_match 'i' vowel_regex $0))
--if there is c add e after that
(λ (replace_substring_if_match 'e' 'c' $0))
--if there is n add e after that
(λ (replace_substring_if_match 'e' 'n' $0))

(fn_50) split_fullstring :: tfullstr -> list(tsubstr)
(λ (regex_split '.' $0))
{- Splits a given full string into a list of substrings separated by periods. -}

{- Example usages -}
--if the word ends with any letter any letter add f after that
(λ (regex_flatten (regex_append (regex_concat 'f' empty_string) (split_fullstring
$0))))
```

```
--if the word starts with any letter replace that with w i
(λ (regex_flatten (regex_cons (regex_concat 'w' 'i') (regex_cdr (split_fullstring
$0)))))
--if there is any letter add v after that
(λ (replace_each_substring $0 (λ (regex_tail (regex_map (λ (regex_concat $1 'v'))
(split_fullstring $1))))))

(fn_51) replace_consonant_with_substring :: tsubstr -> tsubstr -> tfullstr -> tfullstr
(λ (λ (λ (replace_if_match_substring $0 (replace_first_occurrence $0 $1 $2) (regex_not
vowel_regex)))))
{- Replaces the first occurrence of a consonant at the beginning of a given full string
with a specified substring. The target substring can also be modified before
replacement using another specified substring. -}

{- Example usages -}
--if the word starts with consonant replace that with i q
(λ (replace_consonant_with_substring 'i' 'q' $0))
--if the word starts with consonant replace that with g d
(λ (replace_consonant_with_substring 'g' 'd' $0))
--if the word starts with consonant replace that with p b
(λ (replace_consonant_with_substring 'p' 'b' $0))
```

C.2.2 Library for CLEVR

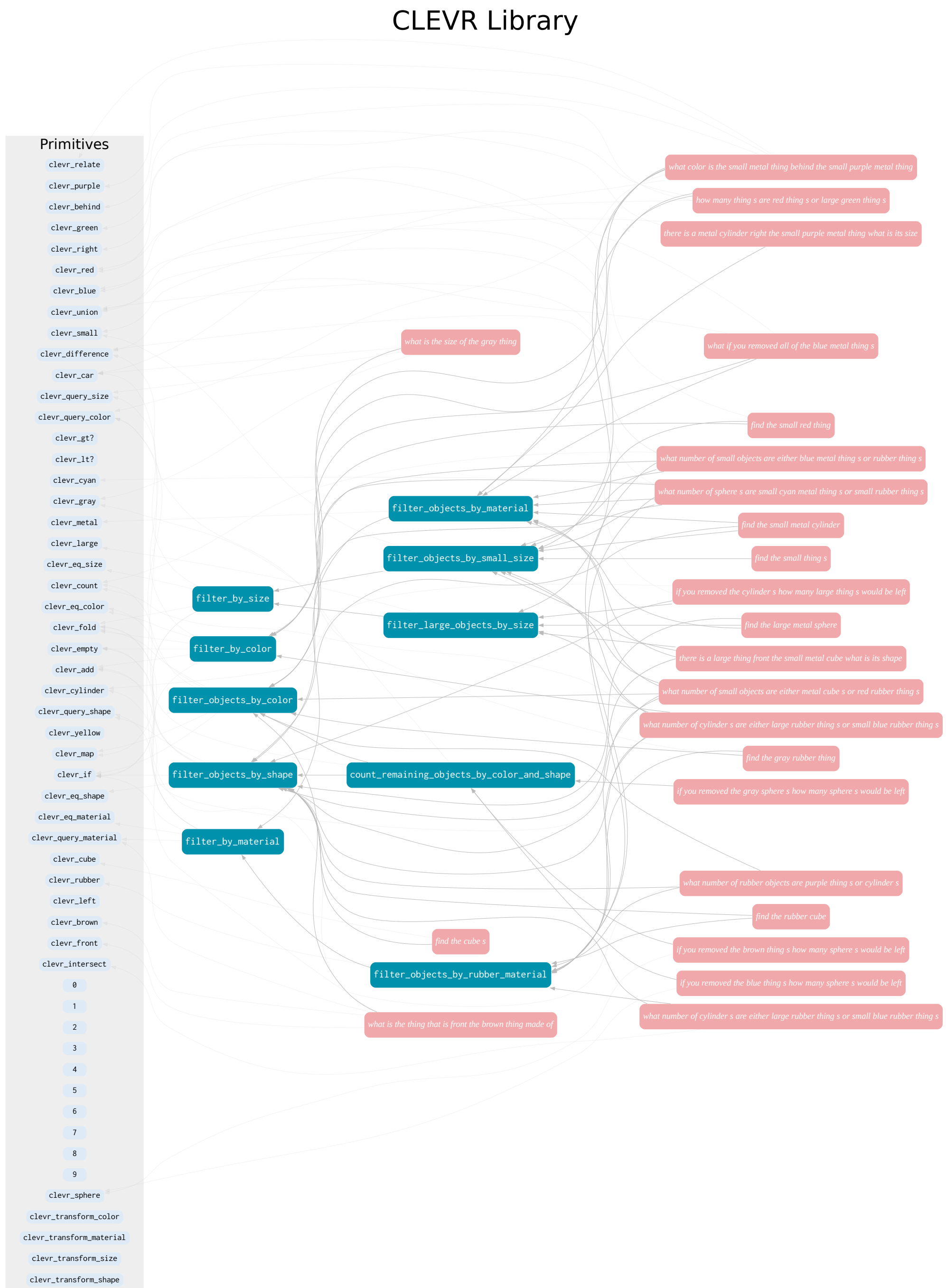

Figure 9: **Graphical map of CLEVR library learned by LILO.** Named abstractions (turquoise) are hierarchically composed of other abstractions and ground out in the base DSL primitives (gray box).

```
(fn_54) filter_by_size :: tclevrsize -> list(tclevrobject) -> list(tclevrobject)
(λ (λ (clevr_fold $0 $0 (λ (λ (clevr_map (λ (clevr_if (clevr_eq_size (clevr_query_size
$0) $4) $0 $2)) $0))))))
{- Returns a list of objects in the input list that have the specified size. -}

(fn_55) filter_by_color :: tclevrcolor -> list(tclevrobject) -> list(tclevrobject)
(λ (λ (clevr_fold $0 clevr_empty (λ (λ (clevr_if (clevr_eq_color (clevr_query_color
$1) $3) (clevr_add $1 $0) $0))))))
{- Returns a list of objects in the input list that have the specified color. -}

{- Example usages -}
--what color is the small metal thing behind the small purple metal thing
(λ (clevr_query_color (clevr_car (filter_objects_by_material
(filter_objects_by_small_size (clevr_relate (clevr_car (filter_by_color clevr_purple
(filter_objects_by_material (filter_objects_by_small_size $0)))) clevr_behind $0))))))
--what is the size of the gray thing
(λ (clevr_query_size (clevr_car (filter_by_color clevr_gray $0))))
--how many thing s are red thing s or large green thing s
(λ (clevr_count (clevr_union (filter_by_color clevr_red $0)
(filter_large_objects_by_size (filter_by_color clevr_green $0)))))

(fn_56) filter_by_material :: tclevrmaterial -> list(tclevrobject) ->
list(tclevrobject)
(λ (λ (clevr_fold $0 clevr_empty (λ (λ (clevr_if (clevr_eq_material
(clevr_query_material $1) $3) (clevr_add $1 $0) $0))))))
{- Returns a list of objects in the input list that have the specified material. -}

(fn_57) filter_objects_by_shape :: tclevrshape -> list(tclevrobject) ->
list(tclevrobject)
(λ (λ (clevr_fold $0 clevr_empty (λ (λ (clevr_if (clevr_eq_shape (clevr_query_shape
$1) $3) (clevr_add $1 $0) $0))))))
{- Filters a list of objects to include only those with the specified shape. -}

{- Example usages -}
--find the cube s
(λ (filter_objects_by_shape clevr_cube $0))
--find the rubber cube
(λ (filter_objects_by_rubber_material (filter_objects_by_shape clevr_cube $0)))
--if you removed the cylinder s how many large thing s would be left
(λ (clevr_count (clevr_difference (filter_large_objects_by_size $0)
(filter_objects_by_shape clevr_cylinder $0))))

(fn_58) filter_objects_by_color :: tclevrcolor -> list(tclevrobject) ->
list(tclevrobject)
(λ (λ (clevr_fold $0 $0 (λ (λ (clevr_map (λ (clevr_if (clevr_eq_color
(clevr_query_color $0) $4) $0 $2)) $0))))))
{- Returns a list of objects in the input list that have the specified color. -}

{- Example usages -}
--find the gray rubber thing
(λ (filter_objects_by_rubber_material (filter_objects_by_color clevr_gray $0)))
--what is the thing that is front the brown thing made of
(λ (clevr_query_material (clevr_car (clevr_relate (clevr_car (filter_objects_by_color
clevr_brown $0)) clevr_front $0))))
--what number of small objects are either metal cube s or red rubber thing s
(λ (clevr_count (filter_objects_by_small_size (clevr_union (filter_objects_by_material
(filter_objects_by_shape clevr_cube $0)) (filter_objects_by_rubber_material
(filter_objects_by_color clevr_red $0))))))

(fn_59) filter_objects_by_small_size :: list(tclevrobject) -> list(tclevrobject)
(λ (filter_by_size clevr_small $0))
{- Returns a list of objects in the input list that are small in size. -}

{- Example usages -}
```

```
--find the small red thing
(λ (filter_objects_by_small_size (filter_objects_by_color clevr_red $0)))
--find the small thing s
(λ (filter_objects_by_small_size $0))
--what number of small objects are either blue metal thing s or rubber thing s
(λ (clevr_count (filter_objects_by_small_size (clevr_union
(filter_objects_by_rubber_material $0) (filter_objects_by_material
(filter_objects_by_color clevr_blue $0))))))

(fn_60) filter_objects_by_material :: list(tclevrobject) -> list(tclevrobject)
(λ (filter_by_material clevr_metal $0))
{- Returns a list of objects in the input list that have the specified material. -}

{- Example usages -}
--there is a metal cylinder right the small purple metal thing what is its size
(λ (clevr_if (clevr_eq_shape clevr_cube (clevr_query_shape (clevr_car (clevr_relate
(clevr_car (clevr_union $0 (filter_objects_by_material $0))) clevr_right $0))))
clevr_small clevr_large))
--what if you removed all of the blue metal thing s
(λ (clevr_difference $0 (filter_objects_by_color clevr_blue (filter_objects_by_material
$0))))
--find the small metal cylinder
(λ (filter_objects_by_small_size (filter_objects_by_material (filter_objects_by_shape
clevr_cylinder $0))))

(fn_61) count_remaining_objects_by_color_and_shape :: list(tclevrobject) -> tclevrcolor
-> tclevrshape -> int
(λ (λ (λ (clevr_count (clevr_difference (filter_objects_by_shape $0 $2)
(filter_objects_by_color $1 $2))))))
{- Counts the number of objects that remain after removing objects of a specified color
and shape from the input list of objects. -}

{- Example usages -}
--if you removed the brown thing s how many sphere s would be left
(λ (count_remaining_objects_by_color_and_shape $0 clevr_brown clevr_sphere))
--if you removed the red cube s how many cube s would be left
(λ (count_remaining_objects_by_color_and_shape $0 clevr_red clevr_cube))
--if you removed the cyan cylinder s how many cylinder s would be left
(λ (count_remaining_objects_by_color_and_shape $0 clevr_cyan clevr_cylinder))

(fn_62) filter_objects_by_rubber_material :: list(tclevrobject) -> list(tclevrobject)
(λ (filter_by_material clevr_rubber $0))
{- Returns a list of objects in the input list that have rubber as their material. -}

{- Example usages -}
--what number of sphere s are small cyan metal thing s or small rubber thing s
(λ (clevr_count (clevr_union (filter_objects_by_material (filter_objects_by_small_size
(filter_by_color clevr_cyan (filter_objects_by_shape clevr_sphere $0))))
(filter_objects_by_rubber_material (filter_objects_by_small_size
(filter_objects_by_shape clevr_sphere $0))))))
--what number of rubber objects are purple thing s or cylinder s
(λ (clevr_count (filter_objects_by_rubber_material (clevr_union
(filter_objects_by_shape clevr_cylinder $0) (filter_objects_by_color clevr_purple
$0)))))
--what number of cylinder s are either large rubber thing s or small blue rubber thing s
(λ (clevr_count (clevr_intersect (filter_objects_by_rubber_material $0)
(filter_objects_by_shape clevr_cylinder $0))))

(fn_63) filter_large_objects_by_size :: list(tclevrobject) -> list(tclevrobject)
(λ (filter_by_size clevr_large $0))
{- Returns a list of objects in the input list that are large in size. -}

{- Example usages -}
--find the large metal sphere
```

```
(λ (filter_large_objects_by_size (filter_objects_by_material (filter_objects_by_shape
clevr_sphere $0))))
--there is a large thing front the small metal cube what is its shape
(λ (clevr_query_shape (clevr_car (filter_large_objects_by_size (clevr_relate (clevr_car
(filter_objects_by_small_size (filter_objects_by_material (filter_objects_by_shape
clevr_cube $0)))) clevr_front $0)))))
--what number of cylinder s are either large rubber thing s or small blue rubber thing s
(λ (clevr_count (filter_objects_by_shape clevr_cylinder (clevr_union
(filter_objects_by_rubber_material (filter_large_objects_by_size $0))
(filter_objects_by_small_size (filter_by_color clevr_blue
(filter_objects_by_rubber_material $0)))))))
```

C.2.3 Library for LOGO

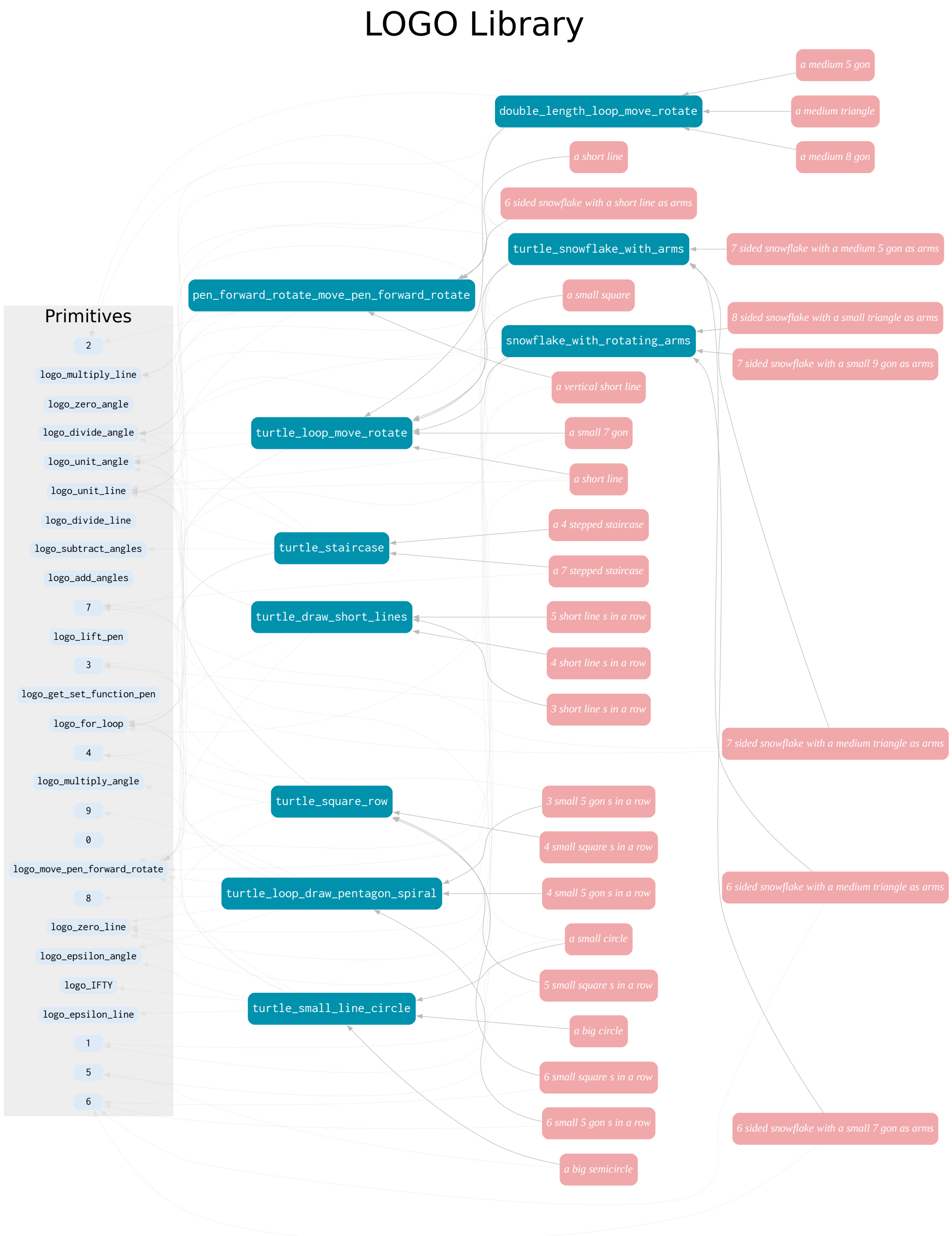

Figure 10: **Graphical map of LOGO library learned by LILO.** Named abstractions (turquoise) are hierarchically composed of other abstractions and ground out in the base DSL primitives (gray box).

```
(fn_27) turtle_loop_move_rotate :: turtle -> int -> tlength -> turtle
(λ (λ (λ (logo_for_loop $1 (λ (λ (logo_move_pen_forward_rotate $2 (logo_divide_angle
logo_unit_angle $3) $0))) $2))))
{- Repeatedly move the turtle forward and rotate it by a specified angle, creating a
loop of a specific number of sides with a given line length. -}

{- Example usages -}
--a small square
(λ (turtle_loop_move_rotate $0 4 logo_unit_line))
--a small 7 gon
(λ (turtle_loop_move_rotate $0 7 logo_unit_line))
--a short line
(λ (turtle_loop_move_rotate $0 1 logo_unit_line))

(fn_28) turtle_staircase :: turtle -> int -> turtle
(λ (λ (logo_for_loop $0 (λ (λ (logo_move_pen_forward_rotate logo_unit_line
(logo_divide_angle logo_unit_angle 4) (logo_move_pen_forward_rotate logo_unit_line
(logo_subtract_angles logo_unit_angle (logo_divide_angle logo_unit_angle 4)) $0))))
$1)))
{- Creates a staircase pattern by repeatedly moving the turtle forward and rotating it
at a specific angle. The number of steps in the staircase is determined by the function
argument. -}

{- Example usages -}
--a 4 stepped staircase
(λ (turtle_staircase $0 4))
--a 7 stepped staircase
(λ (turtle_staircase $0 7))
--a 4 stepped staircase
(λ (turtle_staircase $0 4))

(fn_29) turtle_loop_draw_pentagon_spiral :: turtle -> int -> turtle
(λ (λ (logo_for_loop $0 (λ (λ (logo_move_pen_forward_rotate logo_zero_line
(logo_multiply_angle logo_epsilon_angle 8) (logo_for_loop 9 (λ (λ
(logo_move_pen_forward_rotate logo_unit_line (logo_multiply_angle logo_epsilon_angle 8)
$0))) $0)))) $1)))
{- Creates a spiral of pentagons by repeatedly drawing a pentagon and incrementing the
angle of each side on each iteration. The number of pentagons in the spiral is
determined by the function argument. -}

{- Example usages -}
--4 small 5 gon s in a row
(λ (turtle_loop_draw_pentagon_spiral $0 4))
--3 small 5 gon s in a row
(λ (turtle_loop_draw_pentagon_spiral $0 3))
--6 small 5 gon s in a row
(λ (turtle_loop_draw_pentagon_spiral $0 6))

(fn_30) turtle_square_row :: turtle -> int -> turtle
(λ (λ (logo_for_loop $0 (λ (λ (logo_move_pen_forward_rotate logo_zero_line
(logo_divide_angle logo_unit_angle 4) (logo_for_loop 7 (λ (λ
(logo_move_pen_forward_rotate logo_unit_line (logo_divide_angle logo_unit_angle 4)
$0))) $0)))) $1)))
{- Draws a row of small squares using repeated forward motion and rotation. The number
of squares in the row is determined by the function argument. -}

{- Example usages -}
--4 small square s in a row
(λ (turtle_square_row $0 4))
--6 small square s in a row
(λ (turtle_square_row $0 6))
--5 small square s in a row
(λ (turtle_square_row $0 5))
```

```
(fn_31) turtle_snowflake_with_arms :: turtle -> int -> int -> turtle
(λ (λ (λ (logo_for_loop $0 (λ (λ (turtle_loop_move_rotate
(logo_move_pen_forward_rotate logo_zero_line (logo_divide_angle logo_unit_angle $2) $0)
$3 (logo_multiply_line logo_unit_line 2)))) $2))))
{- Draws a snowflake shape with given number of arms, each made up of a line of specified
length that is rotated at a specific angle. The angle by which the lines are rotated
increases with each iteration of the loop, creating an intricate snowflake pattern. -}

{- Example usages -}
--7 sided snowflake with a medium 5 gon as arms
(λ (turtle_snowflake_with_arms $0 5 7))
--6 sided snowflake with a medium triangle as arms
(λ (turtle_snowflake_with_arms $0 3 6))
--7 sided snowflake with a medium triangle as arms
(λ (turtle_snowflake_with_arms $0 3 7))

(fn_32) turtle_small_line_circle :: turtle -> int -> turtle
(λ (λ (logo_for_loop logo_IFTY (λ (λ (logo_move_pen_forward_rotate (logo_multiply_line
logo_epsilon_line $2) logo_epsilon_angle $0))) $1)))
{- Moves the turtle forward and rotates it repeatedly to draw a small circle with a
given line length. The number of iterations is determined by the function argument. -}

{- Example usages -}
--a small circle
(λ (logo_for_loop 7 (λ (λ (turtle_small_line_circle $0 1))) $0))
--a big semicircle
(λ (turtle_small_line_circle $0 5))
--a big circle
(λ (logo_for_loop 7 (λ (λ (turtle_small_line_circle $0 5))) $0))

(fn_33) snowflake_with_rotating_arms :: turtle -> int -> int -> turtle
(λ (λ (λ (logo_for_loop $0 (λ (λ (turtle_loop_move_rotate
(logo_move_pen_forward_rotate logo_zero_line (logo_divide_angle logo_unit_angle $2) $0)
$3 logo_unit_line))) $2))))
{- Draws a snowflake shape with given number of arms, each made up of a line of specified
length that is rotated at a specific angle. The angle by which the lines are rotated
increases with each iteration of the loop, creating an intricate snowflake pattern. -}

{- Example usages -}
--7 sided snowflake with a small 9 gon as arms
(λ (snowflake_with_rotating_arms $0 9 7))
--6 sided snowflake with a small 7 gon as arms
(λ (snowflake_with_rotating_arms $0 7 6))
--8 sided snowflake with a small triangle as arms
(λ (snowflake_with_rotating_arms $0 3 8))

(fn_34) double_length_loop_move_rotate :: int -> turtle -> turtle
(λ (λ (turtle_loop_move_rotate $0 $1 (logo_multiply_line logo_unit_line 2))))
{- Moves and rotates the turtle in a loop, with each iteration doubling the length of
the turtle's movement. -}

{- Example usages -}
--a medium 5 gon
(λ (double_length_loop_move_rotate 5 $0))
--a medium triangle
(λ (double_length_loop_move_rotate 3 $0))
--a medium 8 gon
(λ (double_length_loop_move_rotate 8 $0))

(fn_35) turtle_draw_short_lines :: turtle -> int -> turtle
(λ (λ (logo_for_loop $0 (λ (λ (logo_move_pen_forward_rotate logo_unit_line
logo_unit_angle $0))) $1)))
{- Draws a specified number of short lines in a row using repeated forward motion and
rotation. -}
```

```
{- Example usages -}
--5 short line s in a row
(λ (turtle_draw_short_lines $0 5))
--4 short line s in a row
(λ (turtle_draw_short_lines $0 4))
--3 short line s in a row
(λ (turtle_draw_short_lines $0 3))

(fn_36) pen_forward_rotate_move_pen_forward_rotate :: turtle -> int -> tlength -> turtle
(λ (λ (λ (logo_move_pen_forward_rotate $0 (logo_divide_angle logo_unit_angle $1)
(logo_move_pen_forward_rotate logo_unit_line (logo_divide_angle logo_unit_angle 2)
$2)))))
{- Moves the turtle forward and rotates it at a given angle. Then moves the turtle
forward again and rotates it at half the angle, creating a pivot point for the turtle
to change direction. The distance the turtle moves each time is determined by a given
length parameter. -}

{- Example usages -}
--a vertical short line
(λ (pen_forward_rotate_move_pen_forward_rotate $0 4 logo_zero_line))
--a short line
(λ (pen_forward_rotate_move_pen_forward_rotate $0 2 logo_unit_line))
--6 sided snowflake with a short line as arms
(λ (logo_for_loop 7 (λ (λ (pen_forward_rotate_move_pen_forward_rotate $0 3
logo_unit_line))) $0))
```

C.3 Benchmark Comparison to Prior Work

Language	Model	Strings (n_{test} = 500)	Graphics (n_{test} = 111)		Scenes (n_{test} = 115)	
		% Solved	% Solved (Best)	% Solved (Mean)	% Solved (Curric.)	% Solved (Mean.)
Synth train/test	DreamCoder (no language)	33.4	49.55	42. 64	67.80	73.9
Synth train/test	Multimodal (no generative translation model)	46.00	26.12	23.20	76.50	49.5
Synth train/test	LAPS in neural search	52.20	92.79	52.93	95.6	88.1
Synth train/test	LAPS + mutual exclusivity	**57.00**	86.49	80.18	**96.5**	82.3
Synth train/test	LAPS + ME + language-program compression	54.60	**98.19**	**81.98**	95.6	**95.9**
Synth train/human test	LAPS + ME + language-program compression	54.60	89.20	–	97.4	–
Human train/human test	LAPS + ME + language-program compression	48.60	58.55	–	95.6	–
No language at test						
No language on train/test	Original DSL; Enumerative	0.06	0.00	–	27.8	–
No language on train/test	DreamCoder (best library): Enumerative	27.2	41.44	–	53.6	–
No lang at test	LAPS (best library): Enumerative	33.2	62.16	–	93.04	–
No lang at test	LAPS (best library): example-only neural synthesis	**52.4**	**91.0**	–	95.6	–

Table 3: **Percent held-out test-tasks solved for LAPS.** *Best* reports the best model across replications; *Mean* averages across replications. (Reproduced from Wong et al. [22].)

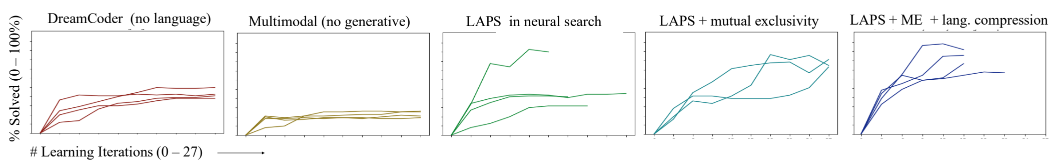

Figure 11: Learning curves comparing baselines and LAPS models in Table 3, showing % heldout tasks solved on the graphics domain over random training task orderings. (Reproduced from 22.)

Our results from §3 are directly comparable to those from Wong et al. [22]. The primary results from that work are reproduced in Tab. 3, where Strings corresponds to REGEX, Graphics corresponds to LOGO, and Scenes corresponds to CLEVR. The DreamCoder baseline from our work, which uses the language-conditioned recognition model from [22], is comparable to the "LAPS in neural search" condition in Tab. 3, with the key difference being that we do not use the IBM translation model component. (We also run on larger batch sizes to take full advantage of the available CPU parallelism on our cloud hardware.)

On REGEX (Strings), with the use of LLMs for search, our LLM Solver and LILO conditions perform significantly better (93.20 best vs. 57.00 best) than this prior work, even without explicitly computing language/program alignment via a translation model. On CLEVR (Scenes), our models perform comparably to LAPS: the DreamCoder baseline already solves almost all of the tasks in the test set (97.09 best), and LILO brings the best solve rate up to 99.03.

Finally, on LOGO (Graphics), our models generally underperform with respect to the results reported in LAPS (73.87 LILO best vs. 92.79 LAPS best). It is worth noting that the best run from LAPS on this domain appears to be an outlier (see Fig. 11, *LAPS in neural search*), so a comparison of average results (48.95 LILO mean vs. 52.93 LAPS mean) may be more appropriate. Moreover, even matching the $1800s$ search time, we were unable to obtain a DreamCoder run that matches their equivalent LAPS baseline on this domain (28.53 DreamCoder (ours) vs. 42.64 DreamCoder (LAPS)). This finding suggests that the LOGO domain is particularly well-suited to the token-to-token assumptions made by the IBM translation model from [22]. It is also worth noting that only the DreamCoder and LILO conditions, which train a CNN-guided neural recognition model as part of enumerative search, have the ability to condition on the LOGO drawings. In particular, the conditions that rely exclusively on LLM-guided search must infer what to draw solely based on the task descriptions; an exceedingly difficult generalization task.

C.4 Experiments with Human Language Descriptions

Each of our domains provides a default set of language task descriptions that were generated synchronously with the ground truth program(s) for each task. Following [22], we use these synthetic language annotations for our primary experiments, as these descriptions correspond closely and systematically to the target programs. To test generalization to real-world applications, we also evaluated our methods on human language annotations sourced from Mechanical Turk. These were collected by [22], with the exception of the REGEX domain, for which the annotations were sourced from the original [13].

We ran experiments with a key subset of model conditions to compare performance on human vs. synthetic language. Fig. 12 and Tab. 4 summarize the results from these experiments. In general, synthesis performance with human language is upper-bounded by performance with synthetic language. This is expected, as the human language contains a wide range of lexical and syntactic variations. For instance, for an individual LOGO task involving drawing a snowflake, human annotations range from "3 sided snowflake with arms that are lines with a semi circle at the end" to "3 candy cane shapes with spaces in them," with one annotator simply stating, "boomerang." Compared to the more templated synthetic language, the broad variation present in the human annotations makes it more difficult to infer a mapping between the language and target programs.

Our experiments reveal that both search and library learning appear to be important to achieving robust performance on human language. DreamCoder achieves remarkably consistent performance between the two language types. In contrast, the LLM Solver baseline degrades markedly on CLEVR and LOGO with human descriptions. We see that adding search [LLM Solver (+ Search)] helps to mitigate this gap. Introducing the full library learning pipeline [LILO] further improves robustness to human language, while achieving better overall performance than DreamCoder.

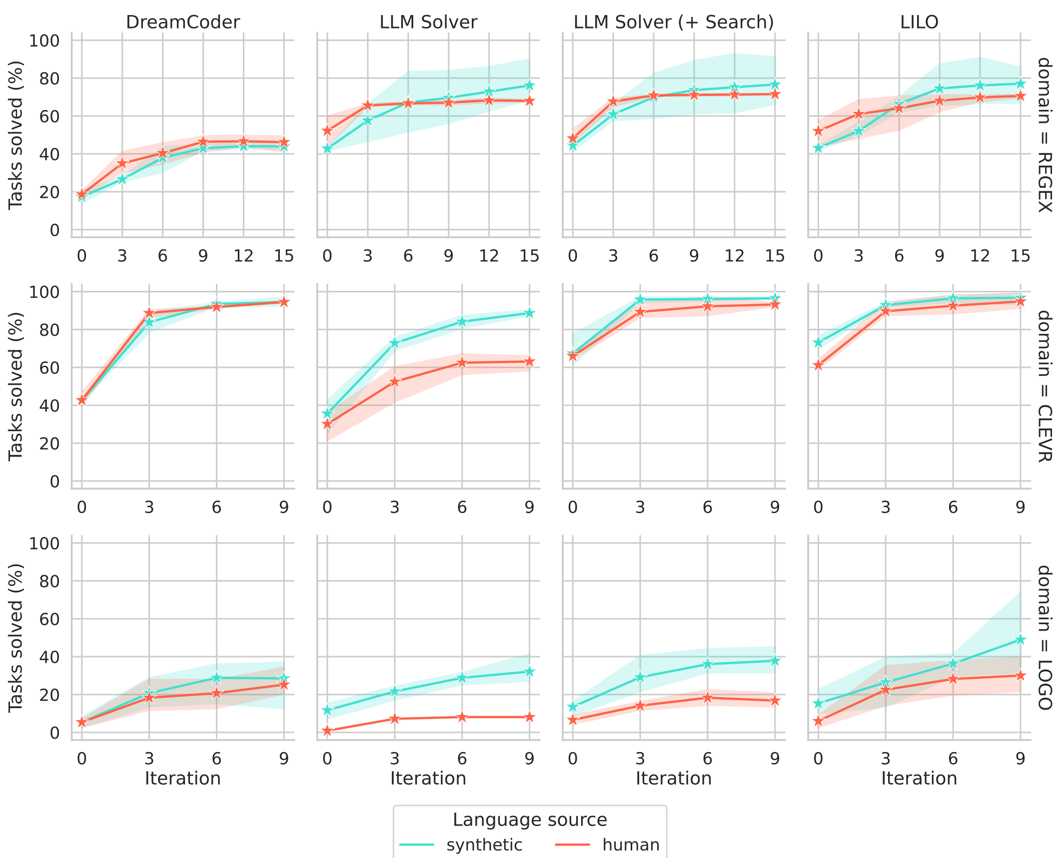

Figure 12: Learning curves illustrating performance of select models on human vs. synthetic language annotations.

	Synthetic language - Tasks solved (%)								
	REGEX			CLEVR			LOGO		
Model	max	$mean$	std	max	$mean$	std	max	$mean$	std
DreamCoder	45.60	43.93	1.53	97.09	94.50	2.44	36.94	28.53	13.79
LLM Solver	90.00	76.13	12.04	90.29	88.67	1.48	41.44	32.13	8.07
LLM Solver (+ Search)	91.20	76.60	13.02	97.09	96.44	0.56	45.05	37.84	6.80
LILO	93.20	77.07	14.14	99.03	96.76	3.12	73.87	48.95	22.15
	Human language - Tasks solved (%)								
DreamCoder	49.40	46.20	4.39	95.15	94.50	0.56	34.23	25.23	7.85
LLM Solver	68.60	68.00	0.60	66.02	63.11	4.23	8.11	8.11	–
LLM Solver (+ Search)	71.60	71.53	0.12	94.17	93.20	0.97	20.72	16.82	3.64
LILO	71.40	70.60	0.92	99.03	94.82	3.92	39.64	30.03	9.07

Table 4: Solution rates of select models on human vs. synthetic language annotations.

C.5 Task-Example Selection Methods

Our LLM-guided program synthesis method (Eq. 4) requires selecting a set of few-shot examples for prompting. As the set of solved tasks grows, the set of possible examples exceeds the size of the LLM context window. This issue particularly affects non-compressive methods, such as the LLM Solver baseline. However, even with program compression—which substantially reduces the length of the program examples—LILO still requires subsampling from the total set of possible examples.

We experimented with two different methods for task example selection: a naive random sampling method and a task-example selection method [82] based on cosine similarity between the task descriptions of the example $\vec{d_x}$ and the target $\vec{d_t}$:

$$\text{score}(\vec{d_x}, \vec{d_t}) = \frac{\vec{d_x} \cdot \vec{d_t}}{\|\vec{d_x}\| \|\vec{d_t}\|}$$

In our implementation, we used embeddings from `text-embedding-ada-002` via the OpenAI API to pre-compute pairwise similarities between all task descriptions in each domain. For both selection methods, we construct the prompt dynamically to fit as many examples as possible.

We ran a head-to-head comparison between the two sampling methods for our main LILO model. As Fig. 13 and Tab. 5 show, we did not observe a significant improvement from the cosine similarity example selection method, though introducing determinism did have the effect of reducing the variance across runs in the REGEX domain. In absence of evidence justifying additional methods complexity, we chose to use random sampling for the results reported in §3.

It is possible that the use of compression in LILO reduces the need for targeted example selection, since we are able to fit approx. 20-40 examples per prompt across all domains. We also noted a tendency for the cosine similarity sampling to be oversensitive to superficial lexical overlap in the task descriptions; e.g., two tasks might involve very different programs but both include the word "six" as an argument, resulting in high cosine similarity. Thus, methods that explicitly finetune a model to infer similarity between (observed) example and (unobserved) target *programs* (i.e., Target Similarity Tuning from 83) could offer clearer performance advantages.

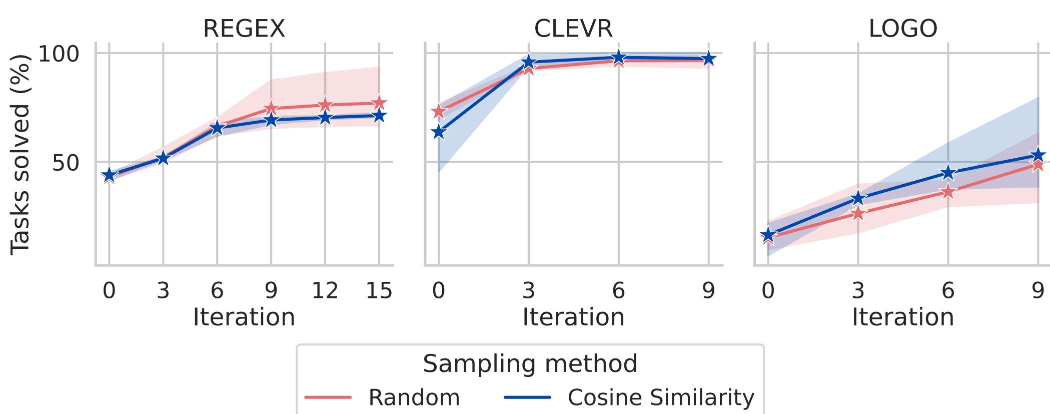

Figure 13: Head-to-head comparison between task example selection methods for the main LILO model.

	Tasks solved (%)								
	REGEX			CLEVR			LOGO		
Model	*max*	*mean*	*std*	*max*	*mean*	*std*	*max*	*mean*	*std*
LILO (Random)	93.20	77.07	14.14	99.03	96.76	3.12	73.87	48.95	22.15
LILO (Similarity)	72.60	71.33	1.10	100.00	97.41	2.24	79.28	53.15	22.67

Table 5: Final performance of task example selection methods for the main LILO model.

C.6 Computational Efficiency Analysis

Given that program search is the most computationally expensive component of synthesis, we would like to be able to quantify and compare the compute costs of LLM-based and enumerative search. However, performing an apples-to-apples comparison is non-trivial because the source of these costs is different between the two cases. As discussed in Apx. B.4, enumerative search requires a high degree of CPU parallelism, so the primary cost associated with running DreamCoder in our experiments is the on-demand CPU-hour cost of renting suitably large machines from AWS. In contrast, LLM search is GPU-intensive, and (in our implementation) is performed on external servers for which we do not have access to exact specifications or cost metrics. In practice, "LLM-as-a-service" models, such as OpenAI's API, charge a fixed price per text token, so the primary costs of LILO-style program search arise from the number of LLM queries, the length of the prompts, and the desired completion length.

In this section, we compare the computational efficiency of the two search approaches across three fronts. First, we consider *wall clock time*, which—in addition to being an informative metric in its own right—also allows us to compute a cost basis for enumerative search. Next, we consider *token usage*, which allows us to compute a cost basis for LLM search methods. These analysis culminate in a *dollar-to-dollar comparison* that, while dependent on pricing schemes of third-parties and the markets more generally, nevertheless offers the closest means of direct comparison.

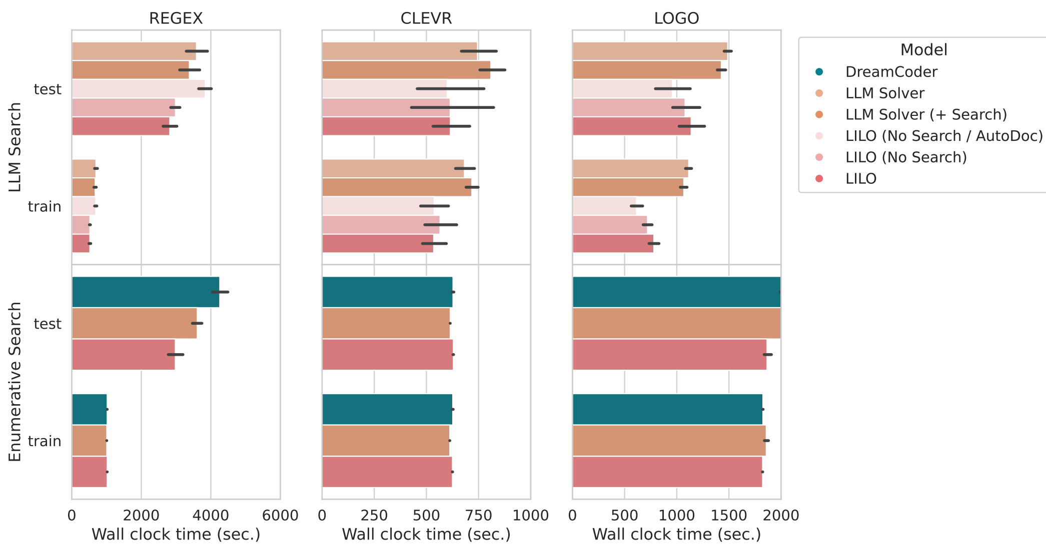

Figure 14: **Comparison of wall clock runtimes across search procedures and domains.** Each bar shows average runtime for a single iteration of train/test program search (error bars indicate 95% confidence intervals). Even with network latency from interfacing with OpenAI servers, LLM search (top row), typically requires less execution time than enumerative search (bottom row), which runs locally on a 96-CPU machine.

We start by analyzing observed (a.k.a. "wall clock") runtimes of our different models. Fig. 14 breaks these down by domain, where the x-axis corresponds to the average time to perform a single search iteration during training and test.[2] Overall, we observe that even with network latency from interfacing with OpenAI servers, a round of LLM search typically runs more quickly than an equivalent round of enumerative search. This difference is especially pronounced on LOGO, which requires longer search times (the enumeration budget for the DreamCoder baseline is set on a per-domain basis using the timeouts from [22]; see Apx. B.4 for more details). We do not observe major differences in runtimes within the different LLM Search conditions, though it is worth noting that the LILO and LLM Solver (+ Search) conditions require approximately 2x more total runtime than the other models because they perform both LLM-based and enumerative search on each iteration.

[2]Note that in Fig. 14, despite appearances, for a given model on a given domain, the per-task search times between train and test splits are approximately equal. Any apparent within-condition discrepancies between train and test are due to the fact that during training, we search on minibatches of 96 tasks, whereas during test, we search on the entire test set. Thus, for domains where the number of tasks is many multiples of the batch size (e.g., REGEX), there is a larger discrepancy betweeen train and test search times.

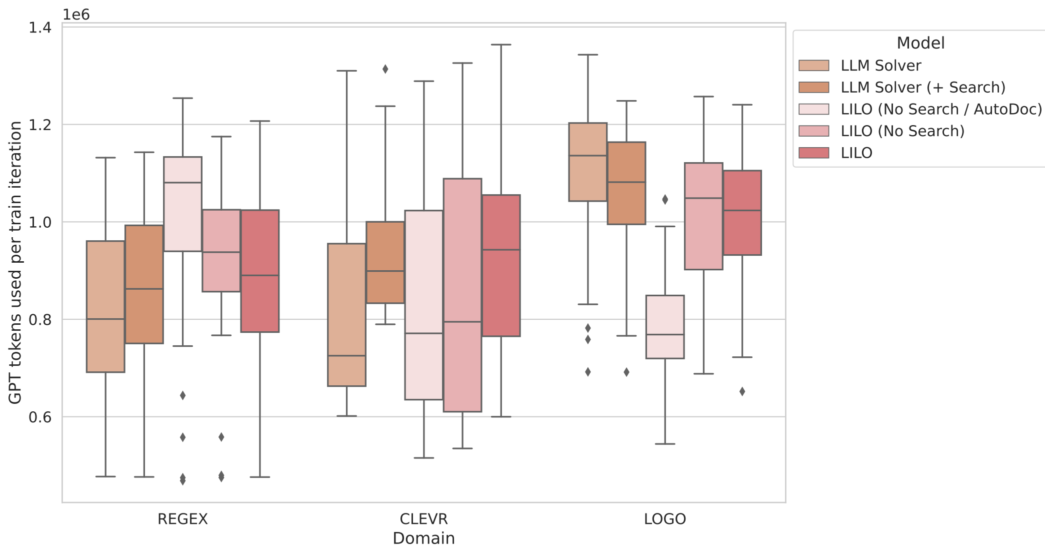

Figure 15: **GPT token usage per training iteration.** Token usage provides a useful metric for assessing the computational costs of LLM-based program search. A typical training iteration uses on the order of 0.8M-1.2M GPT tokens between the prompt and the completion. (Note the y-axis measures millions of tokens.) Boxes indicate quartiles of the distribution and whiskers extend to 1.5 inter-quartile ranges, with outliers shown as individual points.

Next, we consider the token usage of the LLM solver conditions. Fig. 15 breaks these down by domain and model. A typical training iteration uses on the order of 0.8M-1.2M GPT tokens between the prompt and the completion. For completeness, all models are shown separately, but we do not note any clear trends in token usage by model; all models empirically use similar token counts. This may be because token usage is influenced by a complex interplay of several factors. Better-performing models will require fewer queries per task to discover a solution, so they should use fewer tokens. (In practice, however, we cap $n_{\text{prompts\_per\_task}} = 4$, and all conditions must make at least one query per task, so the number of queries is bounded fairly tightly.) Models that use Stitch for compression (i.e., everything except the LLM Solver models) will also tend to benefit from shorter program description lengths per task. In particular, the LILO (✂ Search / AutoDoc) condition, which uses anonymous function names (e.g., `fn_42`), tends to use the fewest tokens per task. However, because we "pack the prompt" with as many examples as can fit, per-task description length does not directly influence token usage; though, as we discuss throughout, too much compression could affect token usage indirectly by obfuscating program semantics, therefore making the LLM solver require more queries to solve new tasks.

		REGEX		CLEVR		LOGO	
		mean	*std*	*mean*	*std*	*mean*	*std*
LLM Search	LLM Solver	\$1.65	\$0.35	\$1.66	\$0.44	\$2.19	\$0.32
	LLM Solver (+ Search)	\$1.70	\$0.33	\$1.88	\$0.29	\$2.13	\$0.30
	LILO (✂ Search / AutoDoc)	\$2.04	\$0.39	\$1.66	\$0.47	\$1.59	\$0.24
	LILO (✂ Search)	\$1.86	\$0.30	\$1.70	\$0.52	\$2.03	\$0.31
	LILO	\$1.77	\$0.38	\$1.87	\$0.42	\$2.01	\$0.30
Enumerative Search	DreamCoder	\$1.16	\$0.01	\$0.71	\$0.01	\$2.07	\$0.01
	LLM Solver (+ Search)	\$1.14	\$0.00	\$0.69	\$0.00	\$2.11	\$0.06
	LILO	\$1.16	\$0.00	\$0.71	\$0.00	\$2.07	\$0.00

Table 6: **Dollar cost comparison between LLM-based and enumerative search.** Each entry is the cost of running one training iteration of search, estimated based on measured wall-clock time (for enumerative search) or token usage (for LLM search). As a rough heuristic, we find that one iteration of LILO's LLM-amortized search scheme is approximately equivalent to an 1800-second enumerative search on 96 CPUs—or, about 48 CPU-hours—in terms of compute cost.

Finally, in the spirit of providing an apples-to-apples compute cost comparison, we combine our time cost and token cost analyses to estimate a dollar cost for each model per training iteration. For conditions that perform enumerative search, we compute CPU cost using the on-demand AWS EC2 instance price for a `c5.24xlarge` machine in `us-east-2`, currently priced at $4.08 / hr. Meanwhile, for conditions that involve LLM search (everything except DreamCoder), we compute LLM inference cost using OpenAI's current API pricing. As discussed in §2, our experiments took advantage of OpenAI's `Codex` model beta for researchers—in other words, they were effectively free. Accordingly, we estimate the cost of our queries using OpenAI's more recent `gpt-3.5-turbo` model, which is available to the public and priced at $0.002 per 1K tokens (at the time of writing). For the LLM solver cost analysis, we choose not to factor in the cost of running a "head node" to issue API queries, as this machine is an order of magnitude cheaper than the `c5.24xlarge`, has no specific spec requirements, and could be arbitrarily downscaled or even replaced with a laptop.

Tab. 6 summarizes the results of this analysis. Remarkably, despite the fact that LLM-based and enumerative searches use very different compute platforms with prices set by two different third-party companies, the dollar costs per training iteration come out to within the same order of magnitude—indeed, they are approximately comparable. In general, we find the tradeoff between LLM and enumerative search to be closely tied to the search time budget: domains with shorter enumeration timeouts (e.g., CLEVR) cost 2-2.5x less than LLM search, while domains with longer enumeration timeouts (e.g., LOGO) cost about the same. Therefore, as a rough heuristic, we can say that one iteration of LILO's LLM-amortized search scheme is approximately equivalent to an 1800-second enumerative search on 96 CPUs—or, about 48 CPU-hours—in terms of compute cost.

Of course, this cost analysis is heavily tied to market factors that are subject to change—in particular, the hardware, electricity, and logistics costs that AWS and OpenAI face in operating their compute platforms, as well as the profit margins that their pricing schemes bake in. Nevertheless, we find it noteworthy that it is currently possible to implement a search scheme like LILO—which requires thousands of LLM queries over millions of tokens per training iteration—while generally achieving better solution rates, faster wall clock runtimes, and comparable dollar costs to enumerative search. Moreover, we note that general-purpose cloud compute platforms like AWS have been available for many years; especially as Moore's Law is believed to be reaching its tail end [84], we are unlikely to see significant reductions in the cost of large-scale CPU compute. In constrast, the LLM-as-a-service model is a recent innovation; with increased scale, hardware optimizations, product maturation, and growing market competition, we are likely to see the costs of LLM inference decrease dramatically in the coming years. We are particularly excited about the growing diversity of open source LLM packages, which should make it possible to implement LILO in an even more cost efficient manner and with increased control over cost-performance tradeoffs.

Copyright Notice