Discovering Abstractions from Language via Neurosymbolic Program Synthesis

By

Gabriel J. Grand

A.B., Harvard University (2019)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Gabriel J. Grand
Electrical Engineering and Computer Science
May 19, 2023

Certified by: Jacob D. Andreas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: Joshua B. Tenenbaum
Professor of Brain and Cognitive Sciences
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Discovering Abstractions from Language via Neurosymbolic Program Synthesis

by

Gabriel J. Grand

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Large language models (LLMs) are growing highly adept at *language-guided program synthesis*: translating natural language specifications into code to solve programming tasks. Nevertheless, current approaches require searching through a vast space of strings, often needing thousands of guesses to discover solutions to difficult tasks at inference time. In contrast, human programmers learn to solve problems on-the-fly by building up hierarchical libraries of *abstractions*: symbolic expressions that encapsulate reusable functionality. In this work, we draw on models of *library learning* from the programming languages (PL) literature, enriching them with the ability to perform search and abstraction learning with LLMs. We introduce **Lilo**, a neurosymbolic framework for **Library Induction from Language Observations**, which consists of three components: an LLM synthesizer, a symbolic compression module, and an auto-documentation (AutoDoc) procedure. Drawing on human language as a source of commonsense knowledge, Lilo learns abstractions that would be intractable to discover with traditional enumerative search. In our evaluations against DreamCoder, a state-of-the-art library learning algorithm, we find that Lilo solves more tasks while achieving faster search times and comparable computational costs. A central aspect of Lilo is a neurosymbolic integration between the LLM synthesizer and Stitch, a high-performance program compression algorithm that identifies useful abstractions in lambda calculus expressions. Lilo augments Stitch with AutoDoc, which generates human-readable names and docstrings for abstractions using an LLM. In addition to improving interpretability, we find that AutoDoc crucially assists Lilo's synthesizer to infer the semantics of abstractions. In sum, Lilo offers an optimistic "better together" vision where human programmers work in tandem with LLMs and PL tools, building up shared libraries of abstractions to enable creative solutions to complex software problems.

Code for this work is available at: github.com/gabegrand/lilo.

Acknowledgments

This work very much is the product of the weird, wonderful, and vibrant research communities at MIT in which I have been so lucky to encompass myself over the past two years. The synthesis of ideas from natural language processing, programming languages, and cognitive science reflects the remarkable multidisciplinary ecosystem here and the many researchers who sustain and nurture it.

I am deeply grateful to work with not one, but two, academic powerhouses: Jacob Andreas and Josh Tenenbaum. Their co-advising style is both dynamic and complementary: Jacob never misses a single detail, while Josh always makes sure that I don't miss the big picture. While Jacob has a knack for finding creative angles within the fast-paced currents of modern machine learning, Josh gives careful thought to situating research within decades-long academic traditions. And, at the end of the day, both are thoughtful, caring, and spirited individuals whose mentorship has helped me flourish.

In turn, my life is deeply enriched by the two lab communities that Jacob and Josh have cultivated at MIT: the Language and Intelligence Group (LINGO) and Computational Cognitive Science (CoCoSci) group. I have energetically embraced my role as a cross-pollinator between LINGO and CoCoSci, spending my workdays flitting back and forth between the Stata Center and the Brain and Cognitive Sciences building. It is a testament to the strong culture of both of these groups that after long days of research, meetings, and presentations, everyone is eager to get together for evenings and weekends of cooking, hiking, and skiing.

Both intellectually and implementationally, this work builds on one of CoCoSci's largest-scale achievements of recent memory. The DreamCoder project (Ellis et al., 2020, 2021) has served as impetus and inspiration to pursue my PhD. However, as I quickly discovered, DreamCoder presents a steep learning curve—years later, it has turned out to be one of the most technically complex systems I have worked with.

Fortunately, I was lucky enough to be desk neighbors with one of the world's foremost Bayesian Program Learning experts: Lionel Wong. I am eternally indebted to Lio for shepherding me through the world of DreamCoder and the esoteric lambda calculus languages that serve as its substrate. In addition to being a devoted collaborator, natural polymath, and one of the hardest-working researchers I know, Lio is a bon vivant whose boisterous personality makes them a terribly fun friend and travel partner.

In addition to Lio, I have been fortunate to commiserate with a small group of graduate students who are also DreamCoder acolytes. In Spring 2022, Matt Bowers and Theo Olausson helped me to kickstart the research that eventually became this thesis via our final project for Jacob's delightful seminar, 6.884: Doing Things with Words. I am grateful to Matt, in particular, for serving as principal architect, steadfast advocate, and dedicated tech support for Stitch—and for insisting that this work be named Lilo accordingly. I am also grateful to Theo for helping to whiteboard through some of the key equations that appear in this work. In addition

to Matt, Lio, and Theo, I have been thankful for the many regular opportunities to workshop ideas with the other members of our STITCH working group: Megan Tjandrasuwita, Sam Acquaviva, Kavi Gupta, and Kevin Ellis. I especially thank Kevin for the encouragement and feedback on an early version of LILO and for serving as a senior mentor to this group.

Notably, this line of work now spans at least three academic generations: I am grateful to Armando Solar-Lezama for teaching me foundational PL knowledge and skills that have been critical to this research. Armando's twin courses 6.820: Program Analysis and 6.S981: Program Synthesis helped me to understand how analysis and synthesis are two sides of the same coin. Within Armando's group, I am deeply grateful to Jack Feser, who TA'ed Analysis and whose OCaml wizardry helped overcome a significant technical roadblock for this project. Likewise, I am grateful to Leonardo Hernandez-Cano, who TA'ed Synthesis and is an enthusiastic and gracious dialogue partner in both research and our improvisational jazz guitar duo.

Outside the immediate LILO orbit, there are many individuals whose ideas, support, and friendship have meaningfully impacted other areas of my research life. Within MIT LINGO: Ekin Akyurek, Evan Hernandez, Athul Paul Jacob, Belinda Li, Alexis Ross, and Pratyusha Sharma. Within CoCoSci: Kartik Chandra, Cédric Colas, Yoni Friedman, Jenny LeClair, Sydney Levine, Ben Lipkin, Max Nye, Tom Silver, Kevin Smith (and pup Tucker), Ced Zhang, and other members of The Long Room—you know who you are. Within the MIT Probabilistic Computing Group: Alex Lew, Nishad Gothoskar, Tan Zhi Xuan (Xuan), and Vikash Mansinghka. Within MIT BCS: Evelina Fedorenko, Ted Gibson, and Roger Levy; and the many inspiring individuals in their language and cognition groups, including Jon Gauthier, Matthias Hofer, Jennifer Hu, Anna Ivanova, and Hope Kean. And outside MIT, in places like UC San Diego and Stanford: Judy Fan, Robert Hawkins, Will McCarthy, Noah Goodman, and many others.

Undertaking this kind of research is an intensive process along many dimensions—one of which is measured in terms of dollars. I am grateful to the LINGO lab for providing AWS cloud credits that funded the thousands of CPU hours needed to run these experiments. I thank The Infrastructure Group (TIG) at MIT CSAIL for maintaining both the LINGO DGX and CAP Sketch clusters, and for keeping the wi-fi connected, email working, and lights on in Stata. I also thank Juston Forte and Lama Ahmad at OpenAI for their responsive technical assistance, and the OpenAI team more generally for their academic beta program that provided free access to their Codex model for these experiments. Finally, I am deeply grateful for the overarching support of the MIT Presidential Fellowship and the National Science Foundation Graduate Research Fellowship that has made this research possible.

Finally, I thank my fiancée and longtime partner, Nora O'Neill, who is quickly becoming a trusted medical doctor and an accomplished academic in her own right. Nora was there for the last thesis, was here for this one, and will be there for the next one. Thank you for taking this journey with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A longstanding goal of AI is to build systems that are *guided by natural language*: following instructions (Branavan et al., 2009; Artzi and Zettlemoyer, 2013; Fried et al., 2018; Sharma et al., 2021), answering queries (Johnson et al., 2017; Andreas et al., 2020; Zellers et al., 2021), engaging in dialogue (Lewis et al., 2017; Paranjape and Manning, 2021), and interacting with existing data and software (Nakano et al., 2022; Schick et al., 2023; OpenAI, 2023). At the heart of these efforts is a translation problem: how can we convert human language (*i.e.,* words) into machine language (*i.e.,* symbolic programs) to communicate our intentions and facilitate downstream computations?

Recent years have seen a renaissance in language-to-code translation sparked by advances in large language models (LLMs). Trained on vast corpora of internet text—which typically include terabytes of computer code from sources like GitHub—these transformer-based neural architectures (Vaswani et al., 2017) are capable of auto-completing function implementations, generating documentation, and engaging in back-and-forth dialogues with human programmers (Chen et al., 2021; Austin et al., 2021; Wang et al., 2021; Nijkamp et al., 2022; Fried et al., 2022; Chowdhery et al., 2022). Code-trained LLMs are even beginning to solve competition-level programming puzzles (Li et al., 2022; Chen et al., 2021; Hendrycks et al., 2021; Haluptzok et al., 2022) that are considered challenging for human experts.

Despite these advances, the general task of *language-guided program synthesis*

poses a daunting search problem. Current state-of-the-art approaches achieve solution rates in the 10-40% accuracy range (Li et al., 2022; Haluptzok et al., 2022) (though in many cases, humans also struggle with these problems). Moreover, these LLM-based approaches often require sampling thousands to millions of candidate programs, discovering solutions only when allowed to make many guesses (i.e., pass@k for k-large; Kulal et al., 2019; Chen et al., 2021). Where LLMs achieve stronger few-shot performance, it is often the case that significant portions of the benchmarks were leaked into their training data (OpenAI, 2023).

How do human programmers navigate this combinatorial search space, quickly becoming language and domain experts who only need a few guesses—as opposed to thousands—to develop a working solution? In part, the answer lies in our ability to write and use *abstractions*: functions that factor out common program logic and facilitate compositional reuse (Ellis et al., 2021; Wong et al., 2022; Bowers et al., 2023). These abstractions form the fabric of *libraries*: hierarchical collections of abstractions that build on one another to provide simplified interfaces to complex computations. Entire communities of human programmers form around treasured libraries, like the Unix kernel (Lions, 1977) or `NumPy` (Harris et al., 2020), building out new functionality while painstakingly trying to maintain clean and well-documented code. Pioneering programmers staking out a new research direction typically undergird their efforts by writing new libraries; this is how we have tools like TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019), which have pivotally enabled contemporary breakthroughs in LLMs.

Compared to the dynamic processes that characterize human software library development, current LLM approaches to program synthesis are quite static. Models are typically pre-trained once; at inference time, preset prompts (perhaps, with some amount of templating) are fed in to produce downstream outputs. How might we draw insights from human software development—in particular, the process of composing abstractions to form libraries—in order to bootstrap learning?

A line of work in inductive program synthesis proposes models that learn to write programs from examples of inputs and outputs (Gulwani et al., 2017; Balog et al.,

2016; Devlin et al., 2017; Polozov and Gulwani, 2015; Parisotto et al., 2016; Nye et al., 2019). Within this space, a common approach is to gradually build up an explicit library of symbolic abstractions that capture shared structure in the domain (Hinton et al., 1995; Liang et al., 2010; Dechter et al., 2013; Lake et al., 2015; Shin et al., 2019; Ellis et al., 2020, 2021; Wong et al., 2021). Such models are capable of making stronger compositional generalizations than neural models while training on far less data than contemporary LLMs.

In this work, we draw particular inspiration from DreamCoder (Ellis et al., 2020, 2021), a wake-sleep Bayesian program learning algorithm that learns to solve programming tasks by iteratively searching for solutions (*wake* phase) and refactoring shared abstractions into a library (*sleep* phase) that in turn helps to bootstrap search.[1] One of DreamCoder's strengths is its generality: starting with a minimal set of primitives in a domain-specific language (DSL), DreamCoder discovers useful libraries of abstractions in diverse domains ranging from classical list processing tasks to inverse graphics to scientific equation discovery. Unlike the internal representations of large neural models, which are difficult to decode even with specialized tooling, libraries learned by DreamCoder afford a high degree of interpretability, allowing anyone with domain familiarity to inspect the concepts learned by the model.

While library learning is attractive for its data efficiency and interpretability, current approaches often face a "chicken-and-egg" bootstrapping problem: solving new tasks requires having the right library abstractions, but these abstractions may not be discoverable from data alone. For instance, suppose a user interfacing with a program synthesis system for string editing inputs the following instruction: `If there is a vowel, replace that with 's'`. Unless the system has been pre-programmed with the concept of a *vowel*, such an instruction poses a combinatorially intractable search problem. For instance, in a regular expression DSL (§ 4.1.1), the concept of a *vowel* might be expressed as `(or 'a' (or 'e' (or 'i' (or 'o' 'u'))))`. For an enumerative search algorithm that has no prior knowledge of which letter primitives belong together, *vowel* is a needle in a haystack that contains roughly 26^5 possible

[1] We describe DreamCoder in more depth in §2.2 and provide additional technical details in §3.1.

combinations.[2] Moreover, discovering the concept of *vowel* is a critical stepping stone to solving more complex problems in the domain, like checking if the *first letter* is a vowel, or replacing *consonants*.

Another challenge for library learning systems like DreamCoder concerns the computational cost of program search. DreamCoder follows in a line of neurally-guided program synthesis models (Gulwani et al., 2017; Balog et al., 2016; Devlin et al., 2017) that train a neural search policy in-the-loop to steer search towards expressions that are more likely to solve a particular task. However, because undiscovered abstractions may differ structurally from existing program solutions, even with neural guidance, such enumerative search procedures are extremely computationally expensive: a typical DreamCoder experiment takes more than two CPU-*months* just to learn a single domain (Ellis et al., 2021; and see §3.3). Much of this search time is spent "getting off the ground": 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 domains.

How can we imbue program synthesis systems with this kind of commonsense knowledge—like the concept of a *vowel*—without having to encode a laundry list of primitives for each domain? In this work, we integrate large language models and library learning, demonstrating how the two approaches complement one another. We propose a general framework for Library Induction from Language Observations (Lilo), in which LLMs play multiple interconnected roles in facilitating language-guided program synthesis.

First, in Lilo, we use LLMs as a modern, fast, and generalizable alternative to traditional enumerative search. We find that LLMs are surprisingly adept at composing novel expressions in esoteric lambda calculus DSLs when prompted with just a handful of few-shot examples. We observe that LLMs are able to leveraging rich priors learned from pretraining in order make zero-shot generalizations to task descriptions that invoke new constructs, such as the vowel concept.

[2] The size of the search space is actually more like 42^5, if we consider the other primitives like `or`, `if`, `append`, etc. that are needed to sequence programs in our string editing domain.

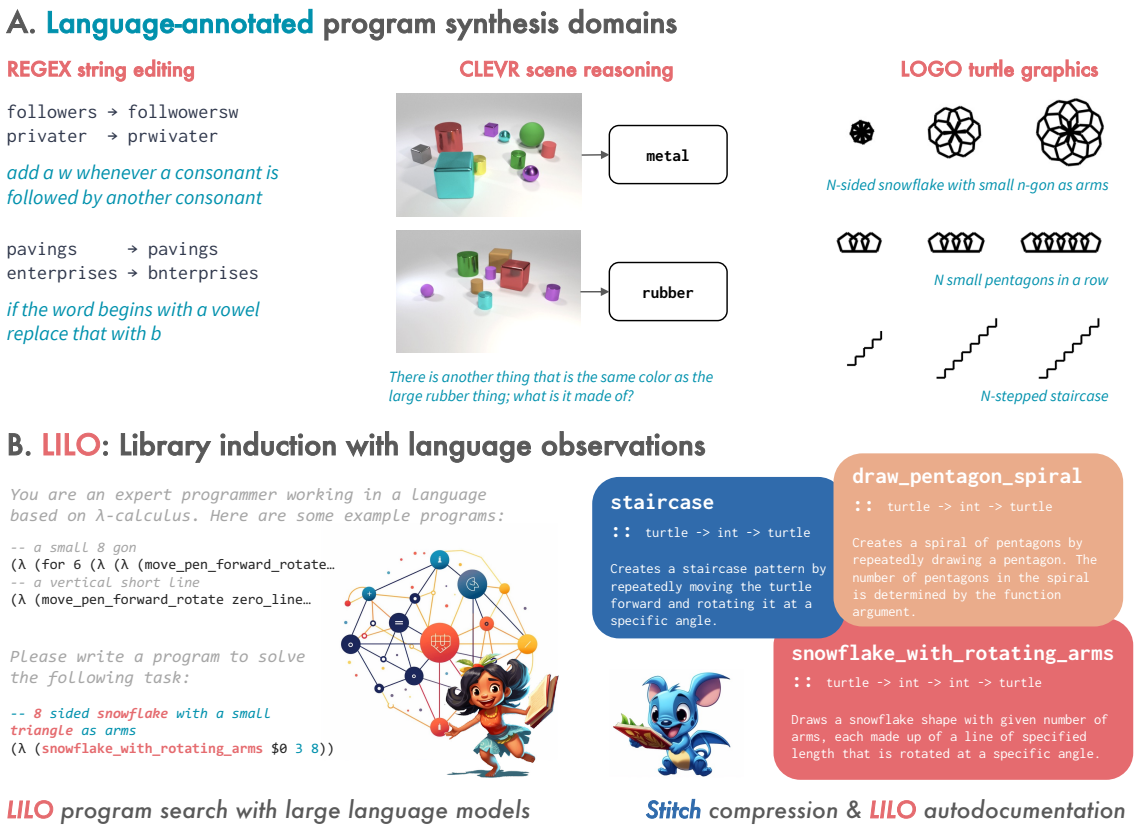

Figure 1-1: **Introducing Lilo: Library induction with language observations.** (A) We evaluate Lilo on three *language-annotated* program synthesis domains: *string editing* with regular expressions, *scene reasoning* on the CLEVR dataset, and *graphics composition* in the 2D Logo turtle graphics language. (B, left) Lilo performs language-guided program search with LLMs, which provide a modern, fast, and generalizable alternative to traditional enumerative search. (B, right) Lilo integrates a symbolic compression algorithm, Stitch, with LLM-generated autodocumentation to produce interpretable library abstractions that facilitate synthesis.

While we show that LLMs perform well in the role of a classic programming languages search algorithm, we also explore whether techniques from PL—in particular, the concept of library learning—can assist in LLM-guided program synthesis. In this spirit, in addition to using LLMs as solvers, we also use them to build up interpretable libraries of abstractions. To facilitate library learning, we put Lilo in dialogue with Stitch (Bowers et al., 2023), a state-of-the-art abstraction learning algorithm for lambda calculus programs. There are many advantages to using an external symbolic module to perform compression: Stitch is purpose-built to solve the difficult optimization problem of factoring out recurring subtrees in a corpus of lambda cal-

culus programs, and has been performance-engineered to achieve approximately 3-4 orders of magnitude faster runtimes and smaller memory footprints as compared to the original compression algorithm from DreamCoder (Bowers et al., 2023).

Because STITCH compresses chunks of useful code into a short string expression, we hypothesize that providing an LLM with access to these abstractions should improve its ability to synthesize complex programs. However, a key challenge with this setup is getting the LILO solver to correctly infer the semantics of the abstractions so as to be able to deploy them effectively during generation. In particular, traditional program compression algorithms produce anonymous function names, like `fn_42` (for the 42nd function in the DSL), that have the effect of obfuscating the underlying semantics. Consequently, as part of LILO, we introduce an LLM-backed *auto-documentation* (AutoDoc) module that takes as input in-context example usages of anonymous abstractions and generates human-readable function names and docstrings. In addition to producing learned libraries that are significantly more interpretable, we show that AutoDoc improves LILO's overall synthesis performance.

We evaluate LILO against DreamCoder 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 (see §4.1 for an overview of the domains). We find that LILO solves more tasks than a language-guided DreamCoder variant on all three domains—in particular, achieving much higher solve rates on string editing—thanks to its ability to learn abstractions that encode human-like commonsense knowledge.

Contrary to conventional wisdom about the insurmountable speed of enumerative search, we find that we are able to achieve faster overall wall clock runtimes with LLM-based search due to orders of magnitude better sample efficiency. Indeed, in terms of dollar cost, we estimate that one round of LILO search is equivalent to 48 CPU-hours of traditional enumerative search (see § 4.2.5). Of course, LLM-based and enumerative search are not mutually-exclusive: we present a LILO variant that integrates both of these procedures and achieves the highest solve rates of all conditions we evaluated. As software and hardware ecosystems around LLMs continue to mature, our findings

make a strong case that LLMs can effectively *amortize* search in many program synthesis domains, reducing the need for exhaustive traditional search.

In contrast to predictions that LLMs will subsume all formal accounts of programming languages, LILO offers an optimistic "better together" vision for integrating LLMs and traditional program synthesis techniques. For search, we find that LLMs can broadly amortize a computationally intensive step of classical synthesis, while also benefiting from additional symbolic search. Meanwhile, for library learning, we find that integrating LLMs with symbolic PL tools like STITCH can lead to more interpretable and useful abstractions. Ultimately, we hope this research will serve as a first step towards a world where AI and human programmers work in tandem, building up libraries of shared abstractions and conventions to enable creative solutions to complex software problems.

In the rest of this thesis, we will present formalisms, data, and qualitative analyses in an effort to make good on this narrative. In §2, we give a purpose-directed background primer, building up key concepts to arrive at language-guided program synthesis. In §3, we delve further into the relevant technical components of DreamCoder and GPT language models and how they are integrated into the LLM Solver and AutoDoc modules that form the basis of LILO. §4 begins with an overview of our synthesis domains, followed by a presentation of our experimental results, which are grouped into three areas: empirical results on synthesis performance, qualitative analysis of learned libraries, and a computational efficiency comparison. Finally, in §5, we present several future avenues for research based on the many ideas developed though our work on LILO.

Chapter 2

Background and Related Work

2.1 Inductive program synthesis

In inductive program synthesis, we are given a specification of a task t in the form of a set of input-output pairs $\{(I_i, O_i)\}_i$. The goal is to find a program ρ in a language $\mathcal{L}$ such that ρ is consistent with the specification of t, denoted $\rho \vdash t$. One way of formalizing this computation is to frame it as solving the optimization problem

$$\hat{\rho} = \arg\max_{\rho \in \mathcal{L}^*} \mathrm{P}[\rho \mid \mathcal{L}, t] \propto \mathrm{P}[t \mid \rho] \cdot \mathrm{P}[\rho \mid \mathcal{L}] \tag{2.1}$$

where the likelihood $\mathrm{P}[t \mid \rho] \triangleq \mathbb{1}_{\rho \vdash t}$ checks whether the program correctly maps all inputs to outputs. This formulation makes obvious the key challenge involved: $\mathcal{L}^*$, being the set of all strings of symbols over the language $\mathcal{L}$, is intractable to compute for nontrivial languages.

Historically, program synthesis has dealt with this challenge in one of three ways:

- By defining a restricted space S (with $|S| \ll |\mathcal{L}^*|$) of programs to consider; e.g., using type-driven synthesis (Polikarpova et al., 2016).
- By solving the optimization problem approximately; e.g., using Monte Carlo methods (Liang et al., 2010; Shin et al., 2019).
- By exhaustively enumerating all possible programs in descending order of prob-

ability and hoping that one within a reasonable amount of time encounters a program that is consistent with the task.

The last of these approaches is typically referred to as *enumerative* search, and is traditionally accomplished by defining a program prior $\mathrm{P}[\rho \mid \mathcal{L}]$, where each w_i below is a primitive in the language $\mathcal{L}$:

$$\mathrm{P}[\rho \mid \mathcal{L}] = \mathrm{P}[w_1, w_2, ..., w_n \mid \mathcal{L}] = \prod_{i=1}^{n} \mathrm{P}[w_i | w_1, w_2, ..., w_{i-1}, \mathcal{L}] \tag{2.2}$$

If $\mathcal{L}$ is a (probabilistic) context-free grammar (Chomsky, 1956; Johnson, 1998; Hopcroft et al., 2001), then it is easy to enumerate programs in decreasing order of likelihood simply by considering longer and longer programs in accordance with the probabilities of the grammar's production rules. However, this blind search procedure does not make any use of the specification of the task. In more recent years, it has therefore become popular to use a neural network to guide the search by training it to approximate the posterior $\mathrm{P}[\rho \mid \mathcal{L}, t]$ (Gulwani et al., 2015; Balog et al., 2016; Parisotto et al., 2016; Devlin et al., 2017; Nye et al., 2019; Ellis et al., 2021). Either way, combinatorially many programs must typically be considered before one that is consistent with the specification is found.

One issue with enumerative approaches is that the longer a program is, the harder it is to find it during search. This is evident in Eq. (2.2), since each factor $w_j \leq 1$ in the right hand side. This issue is often dealt with by performing search in a favorable *domain-specific language* (DSL) which can express the desired computation succinctly. Unfortunately, designing a DSL is a laborious, potentially error-prone process. Failing to provide a necessary component will place complex tasks out of reach of the search procedure. Moreover, providing spurious components can also hinder performance, since increasing the number of possible primitive tokens in the DSL increases the width of the combinatorial search space. Thus, in program synthesis systems based on enumerative approaches, the DSL forms the primary source of inductive bias for the learning procedure. However, the optimal inductive bias may vary by task, requiring the DSL author to anticipate the structure and common patterns within all tasks.

2.2 Library learning

Recently, *library learning* has emerged as a powerful alternative to the problem of DSL design. In library learning, the language $\mathcal{L}$ is learned alongside the search mechanism. From the point of view of developing agents with human-like adaptability and intelligence, this is an attractive approach since it allows us to explicitly encode acquisition of skills and expertise (Rule et al., 2020).

One way to put this problem into probabilistic terms is to re-frame it as performing joint inference on both the programs and the DSL itself over a hierarchical Bayesian model (Liang et al., 2010; Lake et al., 2015). Defining the set of all tasks as $\mathcal{T} \triangleq \{t_i\}_{i=1}^{N}$ and the set of all programs as $\pi = \{\rho_j\}_{j=1}^{N}$ yields the following joint objective:

$$\hat{\mathcal{L}}, \hat{\pi} = \arg\max_{\mathcal{L}, \pi \subseteq \hat{\mathcal{L}}^*} \mathrm{P}[\pi, \mathcal{L} \mid \mathcal{T}] \propto \mathrm{P}[\mathcal{T} \mid \pi, \mathcal{L}] \cdot \mathrm{P}[\pi, \mathcal{L}] \tag{2.3}$$

This optimization objective is, of course, even less friendly than the one from Eq. (2.1). The state-of-the-art library learning system DreamCoder (Ellis et al., 2020, 2021) tackles the intractability of the problem by making two simplifications. Firstly, they decouple the optimization of the program and that of the library, yielding the two optimization problems:

$$\begin{aligned} \hat{\pi} &= \arg\max_{\pi \subseteq \hat{\mathcal{L}}^*} \mathrm{P}[\pi \mid \mathcal{T}, \hat{\mathcal{L}}] \\ &\propto \mathrm{P}[\mathcal{T} \mid \pi] \cdot \mathrm{P}[\pi \mid \hat{\mathcal{L}}] \\ &\propto \prod_{t \in \mathcal{T}} \prod_{\rho \in \pi \vdash t} \mathrm{P}[t \mid \rho] \cdot \mathrm{P}[\rho \mid \hat{\mathcal{L}}] \end{aligned} \tag{2.4}$$

$$\begin{aligned} \hat{\mathcal{L}} &= \arg\max_{\mathcal{L}} \mathrm{P}[\mathcal{L} \mid \mathcal{T}, \hat{\pi}] \\ &\propto \mathrm{P}[\mathcal{L}] \cdot \mathrm{P}[\mathcal{T} \mid \hat{\pi}] \cdot \mathrm{P}[\hat{\pi} \mid \mathcal{L}] \\ &\propto \mathrm{P}[\mathcal{L}] \prod_{t \in \mathcal{T}} \prod_{\rho \in \pi \vdash t} \mathrm{P}[t \mid \hat{\rho}] \cdot \mathrm{P}[\hat{\rho} \mid \mathcal{L}] \end{aligned} \tag{2.5}$$

DreamCoder alternates between optimizing these two objectives through a cyclical *Wake-Sleep* learning paradigm. During the *Wake* phase, a guided search is performed under a fixed library $\hat{\mathcal{L}}$ to find programs that jointly solve tasks and also achieve high probability under the library. This $\mathrm{P}[\pi \mid \hat{\mathcal{L}}]$ term (Eq. 2.4) is computed via *description length*: the number of primitives, lambda-abstractions and applications necessary to describe the program. In turn, during the *Sleep* phase, the set of program solutions found during search $\hat{\pi}$ is held fixed and the *library* itself is optimized. Again, description length plays a key role: the library is chosen to maximize $\mathrm{P}[\hat{\pi} \mid \mathcal{L}]$ (Eq. 2.5); in other words, the library that minimizes the length of the existing solutions. We now turn to the problem of finding such optimal libraries, approximating this search via *compression*.

2.3 Lambda calculus program compression

A program compression algorithm implements two key operations. The first operation is rewrite, which re-expresses a set of programs $\{p_t\}$ more concisely in terms of new abstraction α:

$$\{\hat{p}_t\} = \textsc{rewrite}(\{p_t\}, \alpha) \tag{2.6}$$

The second operation is compress, which chooses the *optimal* abstraction to add to the library, such that the extended library allows for expressing a set of programs as concisely as possible, where $\mathrm{size}(\cdot)$ returns the description length of a set of programs:

$$\hat{\alpha} = \textsc{compress}(\{p_t\}) = \arg\min_{\alpha} \ \mathrm{size}(\textsc{rewrite}(\{p_t\}, \alpha)) \tag{2.7}$$

$$\hat{\mathcal{L}} \leftarrow \mathcal{L} \cup \{\hat{\alpha}\} \tag{2.8}$$

Intuitively, compress finds an abstraction that captures the largest and most frequent shared structure in the set of programs. The processes of compressing-and-rewriting can be repeated to add multiple abstractions to the library.

In this work we use an open-source implementation of the Stitch program compression algorithm from Bowers et al. (2023), which offers approximately 3-4 orders

of magnitude time and memory improvements over the original DreamCoder algorithm while producing abstractions of matching or better compressivity (see § 3.1.3 for more details). However, even with a highly-efficient abstraction learning algorithm, discovering programs to compress still requires searching in a combinatorially large space of possible programs, which motivates our use of *language-guidance*.

2.4 Language-guided library learning

A common theme in the preceding sections is the difficulty of the underlying optimization problems and the need to approximate them by means of search. These search procedures often rely on heuristics, leading to suboptimality both in the solutions obtained and in the time spent obtaining them. One promising approach that has recently gained some traction is to use annotations written in natural language to guide synthesis. This approach is motivated by the insight that natural language captures structure and compositionality amongst objects and actions, which correlates well with the structure and compositionality present in programs. Recent work by Wong et al. (2021) extends the DreamCoder framework to use natural language annotations both during the search phase (*Wake*) as well as the library learning phase during (*Sleep*). Their method, LAPS (Language for Abstractions and Program Search), yields both improved downstream performance and qualitatively richer libraries.

Nevertheless, because LAPS requires learning a language-to-program translation model from scratch, it faces a version of the bootstrapping issue discussed in the previous sections: early in training, this translation model must induce alignments between language and programs from only a handful of paired examples of programs and their descriptions. In LAPS, this data scarcity constraint necessitates the use of a class of simple statistical machine translation models (Brown et al., 1993) that make strict token-to-token decomposition assumptions. In our work, the use of *pre-trained* models allows us to overcome these issues. Rather than attempt to learn a translation model from small data, we start out with a model that already has strong priors over the joint distribution of language and programs and further condition that model on

small numbers of available examples, drawing on recent advances in large language models.

2.5 Large language models for code

In recent years, large language models (LLMs) have become an important tool in program synthesis research. These methods, which originated in the natural language processing (NLP) literature (Vaswani et al., 2017; Devlin et al., 2018; Radford et al., 2018), have opened up the doors for a more liberal use of natural language annotations within program synthesis. Since these models are typically trained on large corpora of text from the internet, and because source code is prevalent within these corpora, pre-trained LLMs often demonstrate fluency in various programming languages.

A key development for LLM-driven program synthesis was the introduction of `Codex` (Chen et al., 2021), an LLM based on GPT-3 (Brown et al., 2020) but fine-tuned on GitHub. Chen et al. find that Codex is capable of solving 4-25% of introductory code problems from the APPS dataset (Hendrycks et al., 2021) depending on how many samples are drawn. Excitingly, this was achieved without any fine-tuning; instead, the authors relied on *few-shot learning*, where the pre-trained model is fed a prompt consisting of a handful of solutions to task-specific problems and then asked to complete the solution for a new one. In our work, we make similar use of the few-shot capabilities of code LLMs, showing that they are able to produce syntactically-valid and—in many cases, semantically-correct—lambda calculus programs, given just a few in-context examples.

In the next chapter (§3), we lay out a general blueprint for slotting code LLMs into a program synthesis framework. Starting with a technical overview of DreamCoder and GPT language models, we demonstrate how they are integrated into the LLM Solver and AutoDoc methods that form the basis of Lilo.

Chapter 3

Methods

3.1 DreamCoder

The framework we propose in this work offers several general patterns for augmenting inductive program synthesis systems with LLMs. However, our work is motivated by and builds directly on the DreamCoder algorithm (Ellis et al., 2020, 2021). Here, we present a brief technical overview.[1]

DreamCoder is an iterative algorithm that jointly learns to solve inductive program synthesis tasks (§2.1) and to express these solutions in terms of a learned library of abstractions (§2.2). DreamCoder is initialized with a base library $\mathcal{L}_0$ of starting primitives and a dataset of training tasks $\mathcal{T}$. It returns a *learned* final library $\mathcal{L}_f$ augmented with program abstractions and a learned neural search model $Q(\rho \mid t, \mathcal{L})$ that predicts high probability programs conditioned on the task examples. As discussed in §2.2, DreamCoder is a *wake-sleep* algorithm (Hinton et al., 1995), meaning that it alternately searches for solution programs to the training tasks (given a current library $\mathcal{L}_i$ and search model Q_i) and updates the library and search model based on new solved tasks.

Though the overall system is complex, DreamCoder can be broken down into three main modules (Fig. 3-1): a *probabilistic context-free grammar (PCFG)*, a *neural search*

[1]Portions of the DreamCoder overview in this section are adapted with permission from Wong et al. (2021).

module, and a *symbolic compression algorithm*. We describe each of these components briefly in order to delineate the role that LLMs will play in amortizing and augmenting the basic framework.

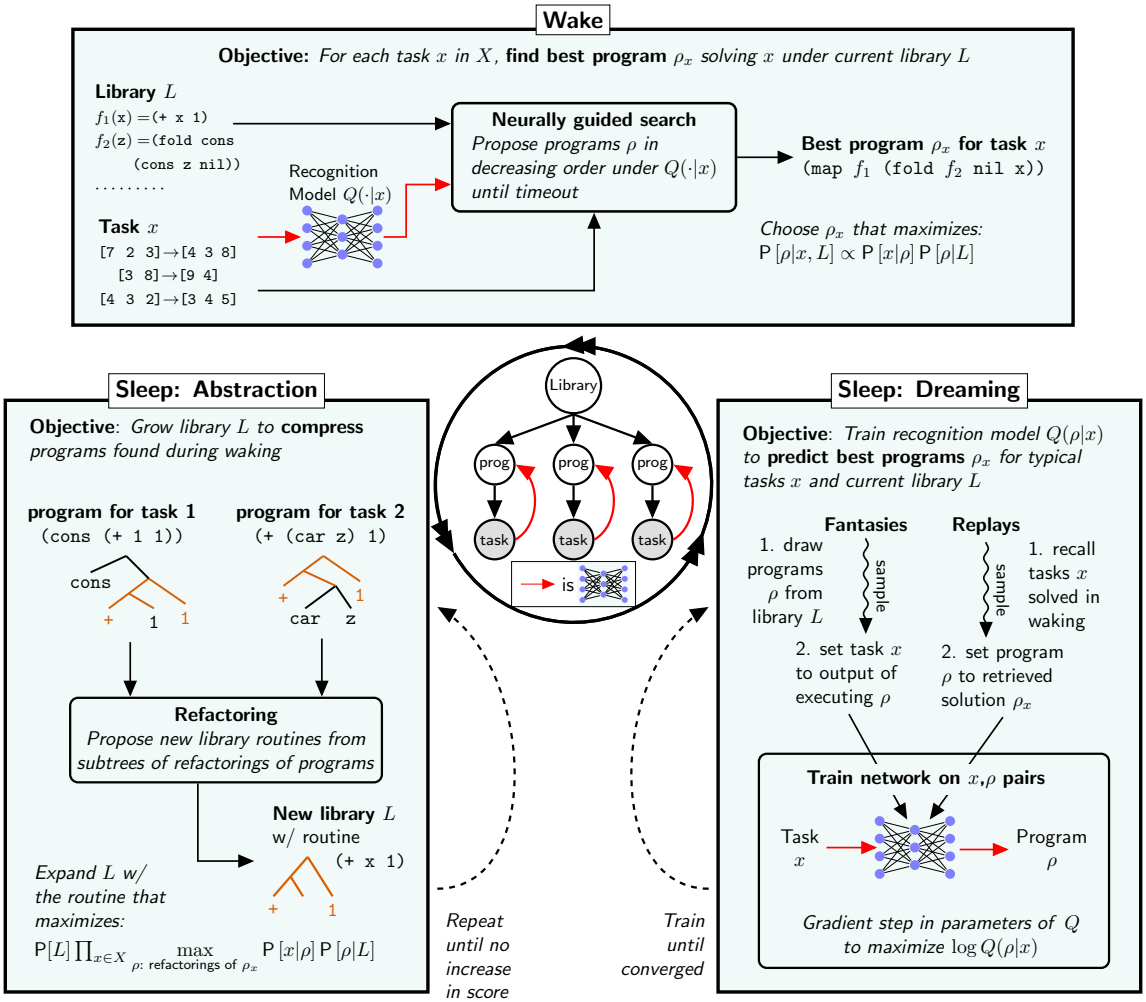

Figure 3-1: **Overview of DreamCoder architecutre.** DreamCoder performs approximate Bayesian inference for the graphical model in the **middle**: inputting synthesis tasks, which it explains with latent programs, and infers a latent library capturing cross-program regularities. A neural net, called the *recognition model* (red arrows) is trained to infer programs with high posterior probability. Waking (**top**) infers programs while holding the library and recognition model fixed. Abstraction (**left**) updates the library while holding the programs fixed by refactoring programs found during waking and abstracting out common components (highlighted in orange). Dreaming (**right**) trains the recognition model on 'Fantasies' (programs sampled from library) & 'Replays' (programs found during waking). (Figure and caption reproduced from Ellis et al., 2021).

3.1.1 Probabilistic context-free grammars

DreamCoder defines the prior over programs as a probabilistic context free grammar (PFCG; Johnson 1998) for programs generated as productions from a library $\mathcal{L}$ of functions $l \in \mathcal{L}$.[2] Formally, DreamCoder assigns a real-valued weight $\theta_{\mathcal{L}i}$ to each library function, which is normalized to yield a production probability $\mathrm{P}[l \mid \mathcal{L}, \theta_{\mathcal{L}}]$. The prior probability of a program ρ is given by the weighted product of probabilities of all of its constituent library functions:

$$\mathrm{P}[\rho \mid \mathcal{L}, \theta_{\mathcal{L}}] = \prod_{l \in \rho} \mathrm{P}[l \mid \mathcal{L}, \theta_{\mathcal{L}}] \tag{3.1}$$

As all $\mathrm{P}[l \mid \mathcal{L}, \theta_{\mathcal{L}}] < 1$, this is equivalent to a *description length* prior over programs: longer programs (with more constitutent elements) will have lower prior probability under Eq. (3.1) since $\mathrm{P}[l \mid \mathcal{L}, \theta_{\mathcal{L}}]$ monotonically decreases as $|\rho| = |\{l \in \rho\}|$ increases.

3.1.2 Neurally-guided program search

The prior in Eq. (3.1) defines a generative model of programs that can be efficiently sampled in order of its prior probability, by enumerating programs in length order ranked by the weight vector $\theta_{\mathcal{L}}$. During learning, unconditional forward search from the prior plays an important role in the distant supervision setting—in the absence of ground truth programs or a pre-defined search heuristic, the model acquires learning signal by enumerating programs until it finds ones that solve at least a few training tasks from which it can learn. At subsequent iterations, as the model updates both the library $\mathcal{L}_i$ and its parameters $\theta_{\mathcal{L}}$ that define the prior, forward sampling should ideally only become more effective at discovering additional learning signal.

To guide the search procedure to areas of program space that are more likely to solve tasks, DreamCoder trains a neural search heuristic $Q_i(\rho \mid t, \mathcal{L}_i)$ at each iteration to approximate the inverse conditional model. The heuristic uses a neural

[2]Note that a *library* is a natural extension of what we termed a *language* in §2: the library defines the set of functions in the DSL. For this reason, we use the same notation $\mathcal{L}$ to refer to libraries going forward.

model trained to predict programs written in the current library $\mathcal{L}_i$ according to the posterior:

$$Q_i(\rho \mid t, \mathcal{L}_i) \approx \mathrm{P}[\rho \mid t, (\mathcal{L}_i, \theta_{\mathcal{L}i})] \propto \mathrm{P}[t \mid \rho] \cdot \mathrm{P}[\rho \mid (\mathcal{L}_i, \theta_{\mathcal{L}i})] \tag{3.2}$$

conditioned on an encoding of the training examples. This model is trained in the distant supervision setting (which begins with no supervised program data) by leveraging the forward generative model: sampling programs from the prior, executing them to produce observed tasks, and then minimizing $Q(\rho \mid t, \mathcal{L})$ in Eq. (3.2) on the sampled programs, conditioned on their executions. This generative training procedure is generally applicable to any neural implementation of $Q(\rho \mid t, \mathcal{L})$.

In the original framing, the neural recognition model is conditioned on modality-dependent task embeddings (images are encoded by a convolutional neural network, while text-based tasks are encoded by a recurrent neural network). Wong et al. (2021) extend this framework to the *language-guided program synthesis* (§2.4) setting, where each task is accompanied by a language description d_t that defines a *joint* prior $\mathrm{P}[\rho, d_t \mid J, \theta_J]$. In turn, they extend the neural recognition model to condition on the description using a *translation model* T as follows:

$$\begin{aligned} Q(\rho \mid t, d_t, J_i) &\approx \mathrm{P}[\rho \mid t, d_t, J, \theta_J] \\ &\propto \mathrm{P}[t \mid \rho] \cdot \mathrm{P}[\rho, d_t \mid J, \theta_J] \\ &\propto \mathrm{P}[t \mid \rho]\mathrm{P}[d_t \mid \rho] \cdot \mathrm{P}[\rho \mid \mathcal{L}, \theta_{\mathcal{L}}] \\ &\approx \mathrm{P}[t \mid \rho] \cdot T(d_t \mid \rho, \mathcal{L}) \cdot \mathrm{P}[\rho \mid \mathcal{L}, \theta_{\mathcal{L}}] \end{aligned} \tag{3.3}$$

In practice, d_t is encoded using a RNN trained on task descriptions to condition the neural recognition model jointly with the task encoding. We use this *language-conditional recognition model* in our DreamCoder baseline condition in our experiments (§4).

3.1.3 Abstraction learning via program compression

At each iteration, DreamCoder updates the library $(\mathcal{L}_i, \theta_{\mathcal{L}_i})$ to approximately optimize the likelihood of the inferred latent programs:

$$\hat{\mathcal{L}}, \theta_{\hat{\mathcal{L}}} = \arg\max_{\mathcal{L}, \theta_{\mathcal{L}}} \mathrm{P}(\mathcal{L}, \theta_{\mathcal{L}}) \tag{3.4}$$

Ellis et al. (2021) leverage equivalence to a *compression* problem defined over programs and the library. As discussed in § 3.1.1, the PCFG program prior is equivalent to a description length prior over programs. Ellis et al. (2021) place an additional Dirichlet prior over the library description length:

$$\mathrm{P}\left[\mathcal{L}\right] \propto \exp\left(-\lambda \sum_{\rho \in \mathcal{L}} \mathrm{size}(\rho)\right) \tag{3.5}$$

Estimating the optimal library then becomes the problem of inferring new library abstractions which can jointly compress the latent training programs (rewritten under the new library $\hat{\mathcal{L}}$) and the description length $|\hat{\mathcal{L}}|$ of the updated library (to optimize for shared abstractions across programs).

DreamCoder's program compression objective requires inference over all possible ways of refactoring the latent programs under the updated library, which is naively intractable. To approximate this inference, a lambda calculus compression algorithm based on version space algebras and E-graph matching is introduced (see Ellis et al., 2021, for a technical review). Because this compression scheme can consider millions of possible rewrites, it balloons to daunting memory and search requirements as the corpus scales in size and complexity. For instance, Bowers et al. (2023) find that DreamCoder was unable to discover even a single abstraction when run directly on any of the datasets from Wong et al. (2022), despite being given hours of runtime and 256GB of RAM.

To address these tractability issues and improve performance, we replace the original DreamCoder compression algorithm with Stitch: a top-down search algorithm that was designed and performance-engineered to improve efficiency of abstraction

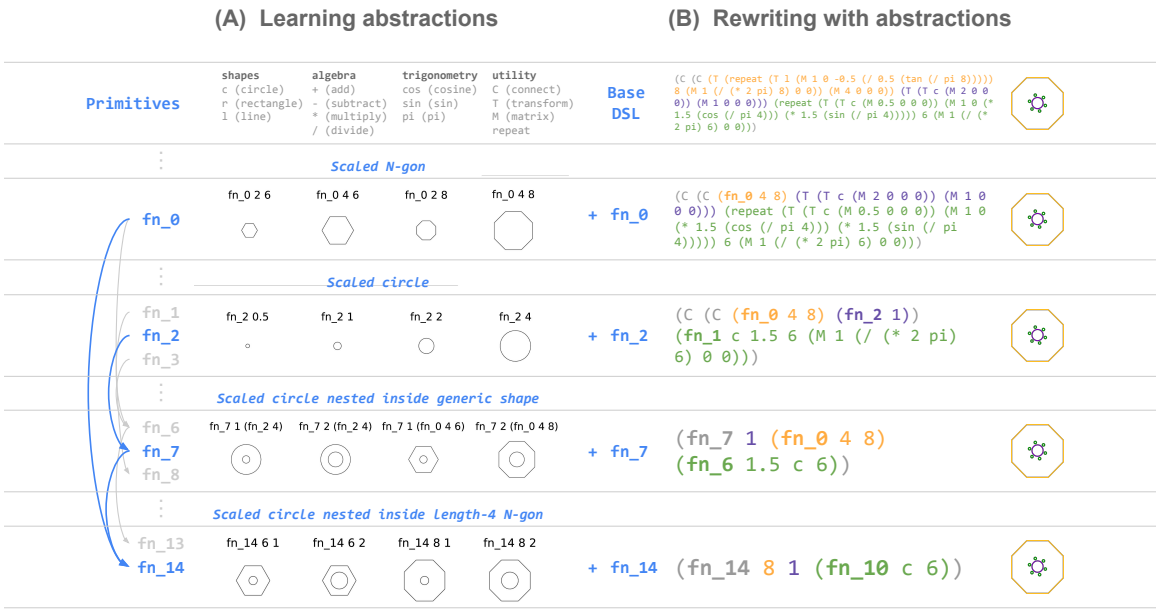

Figure 3-2: Visualization of STITCH library learning on a compositional graphics domain (Wong et al., 2022). (**A**) From the base DSL primitives (top row), STITCH iteratively discovers a series of abstractions that compress programs in the domain. Arrows demonstrate how abstractions from selected iterations build on one another to achieve increasingly higher-level behaviors. (**B**) Rewriting a single item from the domain with the cumulative benefit of discovered abstractions yields increasingly compact expressions. Colors indicate correspondence between object parts and program fragments: **orange = outer octagon**, **green = ring of six circles**, **purple = inner circle**. (Figure reproduced from Bowers et al. 2023.)

learning over datasets of lambda calculus programs (Bowers et al., 2023). For a representative subset of program synthesis benchmarks, Stitch is 3-4 orders of magnitude faster and uses 2-3 orders of magnitude less memory than DreamCoder compression (Fig. 3-3).[3] Moreover, as Bowers et al. (2023) demonstrate, STITCH discovers abstractions that match or achieve better compression ratios than those produced by DreamCoder compression. Nevertheless, prior analyses were limited to standalone compression experiments; in this work, we integrate STITCH into the DreamCoder loop and report on the first experiments using STITCH in a full program synthesis setting. Though not the main focus of our analysis, we find that STITCH dramatically reduces the computational cost of abstraction learning, running in seconds on a single CPU on a typical iteration. These performance improvements reduce a key

[3] https://stitch-bindings.readthedocs.io/en/stable/

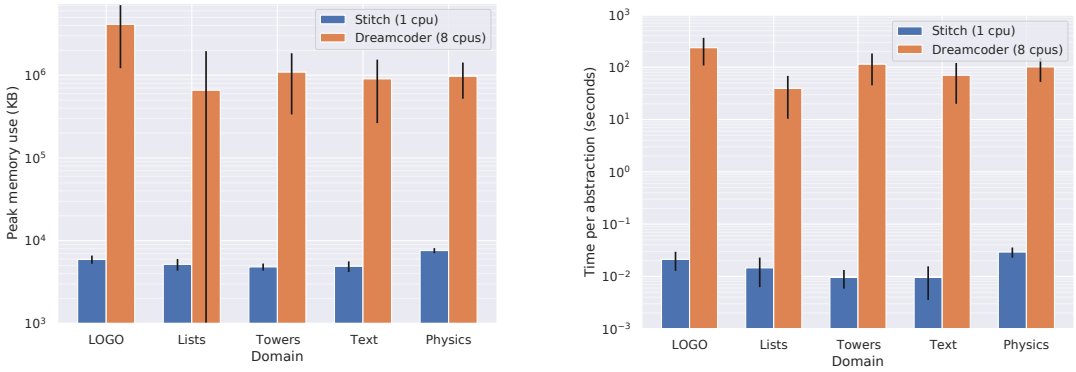

Figure 3-3: **Program compression performance comparison.** STITCH is orders of magnitude more efficient than the program compression algorithm used in (Ellis et al., 2021). Left: Peak RAM usage of STITCH and DreamCoder while running on the five domains considered in Bowers et al. (2023). Right: Wall-clock time required to find (and rewrite under) one abstraction, averaged over all benchmarks. Lower is better; black lines indicate ± one standard deviation. Note the logarithmic y-axis. (Figure reproduced from Bowers et al. 2023.)

computational bottleneck in the DreamCoder pipeline, enabling us to iterate more rapidly in our experiments.

3.2 GPT language models

In this work, we explore multiple distinct roles that LLMs might play in an inductive program synthesis framework: synthesizing solutions to tasks based on language descriptions (§3.3), and writing human-readable names and documentation for function abstractions (§3.4).

For the experiments presented in this work, we leverage the GPT family of language models from OpenAI (Brown et al., 2020; Chen et al., 2021; OpenAI, 2023). Our decision to use GPT in this work was motivated by several technical and practical factors. LLMs are notoriously expensive to train from scratch, which is why much research in the field over the past several years has used off-the-shelf models. Starting in late 2021, when we started these experiments, OpenAI began offering Codex model access to academic researchers via a free private beta program.[4] Codex (Chen

[4] https://openai.com/blog/openai-codex. At the time of writing, the Codex private beta was still active. However, OpenAI have announced plans to discontinue this program in 2023 and

et al., 2021) is based on the GPT-3 architecture (Brown et al., 2020) and is finetuned on a vast corpus of code from GitHub and other sources. At the time of its release, Codex represented a major advance in code-trained language models—and remains one of the strongest-performing off-the-shelf code LLMs to date. Since we began this work, OpenAI has released several other LLMs with coding capabilities—most notably, ChatGPT[5] and GPT-4 (OpenAI, 2023). Conveniently, all of these models are accessible via OpenAI's API, meaning that it has been relatively straightforward to integrate the latest advances in LLMs into our experimental framework.

In this work, we experiment with three different GPT-family models: Codex (`code-davinci-002`), ChatGPT (`gpt-3.5-turbo-0301`), and GPT-4 (`gpt-4-0314`). Where possible, we use snapshot versions of these models (i.e., `gpt-3.5-turbo-0301` is a snapshot of ChatGPT from March 1, 2023) to ensure consistency in our experiments and avoid the possibility that behind-the-scenes updates affect our results. Nevertheless, many aspects of OpenAI's LLMs have not been made public, and the black box nature of this technology has only increased as OpenAI have moved towards productizing their offerings and providing LLMs as a paid service (OpenAI, 2023).

One key aspect of ChatGPT and GPT-4 that remains murky concerns the use of instruction tuning. In response to the many concerns surrounding AI safety and alignment, various methods have been developed to *align LLM outputs to human preferences* (Ouyang et al., 2022; Wei et al., 2022; Bakker et al., 2022). These methods have proven effective at encouraging LLMs to follow instructions provided in prompts and avoiding outputs that are considered harmful. However, the use of instruction-tuning means that—unlike Codex—these newer GPT iterations can no longer be considered "pure" models of joint distributions of language and code. For this reason (and because queries to Codex were free under the research private beta), we tend to prefer Codex for tasks that involve generating a large number of samples from this joint distribution (§3.3). Meanwhile, for tasks that require closely following a particular instruction set or output format, such as abstraction naming (§3.4), we

transition future Codex access to Microsoft's Azure platform, which powers GitHub Copilot.

[5] https://openai.com/blog/chatgpt

find that ChatGPT and GPT-4 produce higher quality generations.

Of course, the methods we introduce in the rest of this section are general techniques that can be implemented with any number of off-the-shelf LLMs. Excitingly, since we began this work, a number of code LLMs have been released as open source software (Fried et al., 2022; Nijkamp et al., 2022; Wang et al., 2021; Taori et al., 2023)—many in just the last few weeks.[6] While a comparative analysis of the capabilities of this growing zoo of LLMs is out of scope for this work, we are optimistic about the roles that open-source code-trained models might play in our program synthesis framework.

3.3 LILO: Amortized synthesis

The search procedure of DreamCoder relies heavily on exhaustive enumeration to discover program solutions. Even with neural guidance, this search is extremely computationally intensive: Ellis et al. run experiments on 20, 64, and 96 CPU machines and report typical runtimes of one day (see Ellis et al., 2021, Appendix J), with the longest experiment running for *five days* on a 64-CPU machine.[7] While Wong et al. (2021) introduce language-guided program search techniques that are shown to improve search efficiency, they similarly report thousands of CPU hours consumed in their experiments. Though proposed as a model of human-like learning, at its core, this exhaustive search is one aspect of the DreamCoder algorithm that deviates significantly from intuitions about how people approach inductive reasoning tasks given limited time and working memory.

In this work, we consider LLMs as an alternative model of program search—one that leverages strong inductive biases learned through large-scale pre-training so as to evaluate many orders of magnitude fewer programs during search. In terms of

[6] E.g., StarCoder (https://huggingface.co/blog/starcoder), replit-code-v1-3b (https://huggingface.co/replit/replit-code-v1-3b), Dolly 2.0 (https://huggingface.co/databricks/dolly-v2-12b) and others have all been released post-April 2023.

[7] In the five-day experiment, DreamCoder was tasked with solving classical functional programming problems, starting just from basic Lisp primitives. It was shown to re-discover "origami programming" — first reinventing fold, then unfold, and then defining a library of functional programming idioms including map, zip, length, etc. in terms of folding and unfolding.

computational efficiency, the use of LLMs as "foundation models" trades off expensive upfront pre-training for fast and efficient inference (Bommasani et al., 2022). Nevertheless, unlike traditional program search in a PCFG, where all possible candidates are constrained to be valid programs, LLMs generate outputs in *string space* and are not guaranteed to produce syntactically-valid programs. Moreover, programs in DreamCoder domains are expressed in bespoke domain specific languages (DSLs) that are not likely to be represented in standard LLM pre-training corpora. While constrained decoding methods for LLMs do exist (Poesia et al., 2022), these approaches typically require access to output logits, which are not currently available via API for models like ChatGPT and GPT-4, and necessitate multiple LLM queries per generation, thus significantly increasing the cost and time to generate programs.

Here, we utilize GPT LLMs in an off-the-shelf manner to ask whether they can effectively "amortize" tranditional enumerative synthesis approaches in lambda calculus DSLs. Our approach takes advantage of the in-context learning capabilities of modern LLMs (Brown et al., 2020), which we briefly formalize here. The standard formal treatment of (unidirectional) language models considers them to approximate the distribution of the next word in a fixed-length sequence (Eisenstein, 2018).

$$\text{LLM}(w_{N+1}) \approx \text{P}[w_{N+1} \mid w_1, \ldots, w_N] \tag{3.6}$$

A growing line of work in NLP demonstrates that neural language models accurately capture hierarchical structure (Wilcox et al., 2019; Tenney et al., 2019; Belinkov and Glass, 2019); and that their ability to represent higher-order structure improves with model scale (Brown et al., 2020; Chowdhery et al., 2022; Zhang et al., 2022). In this work, we therefore "lift" our formal treatment of LLMs to consider them as conditional models over a distribution of *entire programs*:

$$\text{LLM}(\rho_{N+1}) \approx \text{P}[\rho_{N+1} \mid \{\rho_i\}_{i=1}^{N}] \tag{3.7}$$

As discussed, in this work, we operate in the *language-guided* program synthesis setting, in which programs are accompanied by natural language descriptions. Ac-

cordingly, we extend Eq. (3.7) to condition on program-description pairs $\{\rho_i, d_i\}_{i=1}^N$, where the goal is to produce a program for some unsolved description d_{N+1}:

$$\text{LLM}(\rho_{N+1}) \approx \text{P}[\rho_{N+1} \mid d_{N+1}, \{\rho_i, d_i\}_{i=1}^N] \tag{3.8}$$

One advantage of using LLMs as synthesizers is that very little extra work is required to approximate the conditional distribution in Eq. (3.8). We take advantage of LLM's strong in-context learning abilities to procedurally construct few-shot prompts designed to elicit programs that solve target tasks. Fig. 3-4 illustrates the contents of a typical prompt in the LOGO domain. The prompt consists of a static header describing the domain at a high level and the available library functions in the DSL. The header is followed by a dynamically populated list of (d_t, ρ_t) pairs randomly sampled from the set of solved tasks. The final line of the prompt contains a task description for an unsolved task. Because the set of few-shot examples is randomly-sampled, some prompts may contain more task-relevant information than others. Accordingly, for each target task, we construct $n_{\text{prompts\_per\_task}} = 4$ prompts; in turn, for each prompt, we query the LLM for $n_{\text{samples\_per\_prompt}} = 4$ completions, for a total of 16 candidate completions per task. For each completion, we perform a series of syntax and type-checking procedures. Programs that form syntactically valid programs with type signatures that adhere to the task specification are then evaluated on the I/O examples. Those that satisfy the specification are then added to the solution set.

One of the main challenges of inductive synthesis with library learning is "getting off the ground"; i.e., finding an initial set of program solutions to bootstrap downstream library learning and synthesis. In prior work (Ellis et al., 2021; Wong et al., 2021), this is accomplished through brute-force search in a uniform grammar until enough task solutions have been encountered to train the neural recognition model. In this work, we reproduce a DreamCoder baseline for each domain and initialize the LLM solver with the set of programs that was discovered during the initial search (typically on the order of tens of task solutions). In our experiments, we run 3 random seeds for each condition, including the baseline; accordingly, to facilitate fair compar-

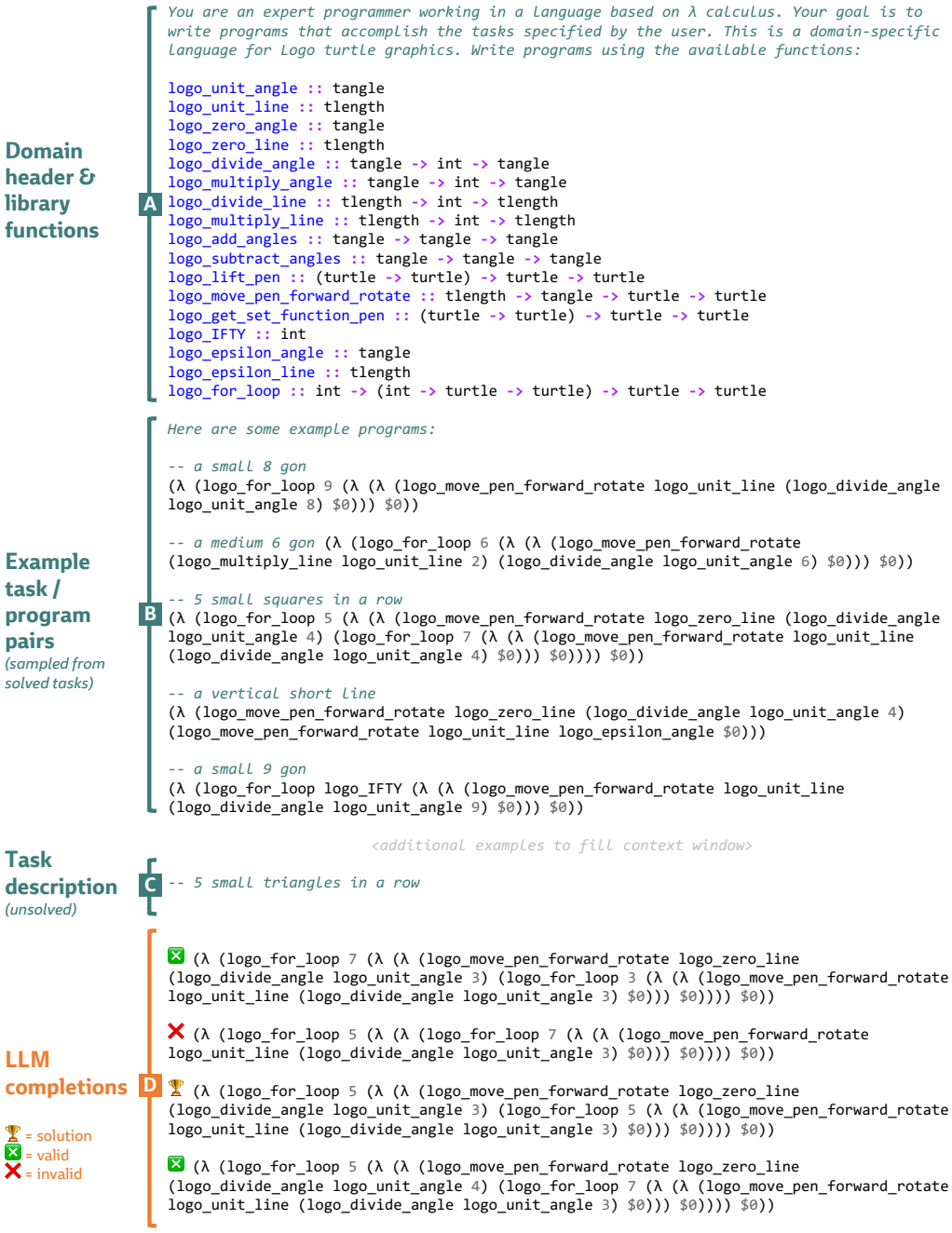

Figure 3-4: **Anatomy of an LLM solver prompt.** (A) Each prompt begins with a short domain description followed by an autogenerated list of the DSL primitives and their type signatures. (B) We randomly sample task solutions and their language descriptions to construct the prompt body. (C) The final 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.

isons, we match each run of an LLM solver condition to the DreamCoder baseline run of the corresponding seed. We view this initialization method as the closest comparison to the experimental settings of the prior work, which can be thought of as testing a model's ability to learn a new DSL from scratch. An alternative evaluation setting would be to define for each domain some minimal set of pedagogically-relevant example solutions. As it seems ecologically implausible that one might encounter a novel programming language in the wild for which there are no known examples of written programs, such an approach might make for a more naturalistic evaluation setting for future inductive program synthesis work.

3.4 LILO: Library auto-documentation

Beyond facilitating search, how might LLMs be integrated into a program synthesis framework? One of the main lessons of DreamCoder is that *compression* can bootstrap learning by constraining the search space. Might the same approach be applicable in an LLM-guided program search?

Indeed, even though beam search in LLMs and enumerative search in DreamCoder traverse vastly different spaces, program compression techniques like STITCH can be thought of as general compression algorithms, reducing not only the size of the program AST, but also the *description length* in string space. Because library abstractions are effectively entire blocks of code that can be deployed in a single token, it is natural to hypothesize that providing an LLM with access to these functions should improve its ability to synthesize complex programs.

In our experiments, we ask exactly this question: how does putting an LLM in "dialogue" with a compression algorithm like STITCH affect its performance on synthesis tasks? And can LLMs in turn contribute to the efficacy of library learning, by introducing useful semantic constructs that form new library functions?

A critical observation from these experiments—which informs our methodological approach—is that unlike traditional methods from PL, language models *care* what functions are called: having trained on terabytes of text data, they have learned to

draw meaningful inferences about the semantics of functions from their names and documentation. Unfortunately, automatic abstraction learning algorithms are typically not equipped to write human-readable function names—indeed, algorithms like STITCH are a broad swath of PL methods that do not reason at all about the semantics of programs, deriving all their power and utility through sensitivity to syntax. Recently, the idea of using LLMs for *code deobfuscation* has received a small, but notable, amount of interest. A core observation is that LLMs appear to be especially well-suited to writing descriptive, human-readable function names (Lachaux et al., 2021; Sharma et al., 2021; Cambronero et al., 2023). Conversely, neural language models have been shown to be susceptible to adversarial attacks that obfuscate naming schemes while preserving functionality (Srikant et al., 2021; Zeng et al., 2022).

In our experiments, we observed that naively providing LLMs access to anonymous lambda abstractions from STITCH did not improve synthesis efficacy. In fact, introducing new abstractions and rewriting the few-shot program examples in terms of anonymous function names like `fn_42` worked as a form of code obfuscation and *significantly hurt* performance.

Motivated by these findings, as part of LILO, we introduce a *library auto-documentation* (AutoDoc) procedure that writes descriptive names and docstrings for library functions proposed by STITCH. Fig. 3-5 gives an overview of this pipeline (the full prompting scheme is reproduced in Appendix A.3). In this prototypical example in the REGEX domain, the LLM has solved some problems that require vowel substitutions by writing the expression `(regex_or 'a' (regex_or 'e' (regex_or 'i' (regex_or 'o' 'u'))))`. Subsequently, STITCH has pulled out this expression for occurring regularly in the solution set (indeed, in the library shown, this function is the most compressive one) and has defined it as an arity-0 function (i.e., a constant). The AutoDoc procedure is prompted to write a human-readable name and docstring for this expression based on examples of its usage in the solution set, and produces the following piece of documentation:

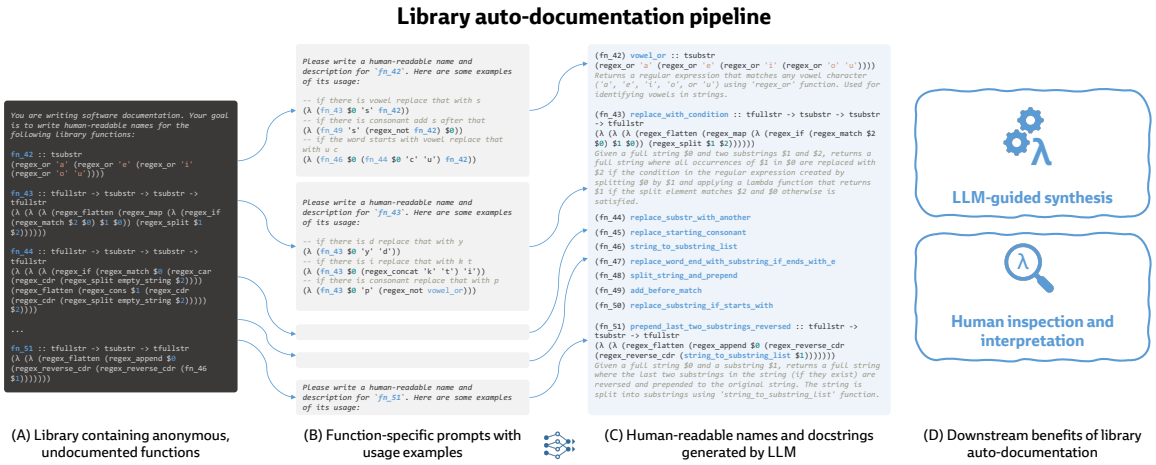

Figure 3-5: **Overview of Lilo library auto-documentation pipeline with REGEX as an example domain.** The figure illustrates our AutoDoc prompting workflow that writes human-readable names and docstrings for abstractions. (A) Library proposed by Stitch containing anonymous lambda abstractions, serving as the header for a prompt. (B) For each abstraction, we query an instruction-tuned LLM to produce a human-readable name and description, given examples of usage from solved tasks. As abstractions are named in serial, names are *inlined* into subsequent prompts (e.g., after being named, `vowel_or` is invoked in one of the usage examples for `fn_43`). (C) After auto-documentation, the abstraction library is both (D) more easily interpretable by humans and more amenable to LLM-guided program synthesis. The full AutoDoc prompt text is reproduced in Appendix A.3.

```
(fn_42) vowel_or :: tsubstr
(regex_or 'a' (regex_or 'e' (regex_or 'i' (regex_or 'o' 'u'))))
{- Returns a regular expression that matches any vowel character ('a', 'e', 'i', 'o',
or 'u') using 'regex_or' function. Used for identifying vowels in strings. -}

{- Example usages -}
--if the word ends with vowel consonant add g after that
(λ (regex_if (regex_match vowel_or (regex_tail (regex_reverse_cdr
(string_to_substring_list $0)))) (regex_flatten (regex_append 'g'
(string_to_substring_list $0))) $0))
--if there is consonant consonant add d before that
(λ (add_before_match (regex_split (regex_concat (regex_not vowel_or) (regex_not
vowel_or)) $0) 'd' (regex_concat (regex_not vowel_or) (regex_not vowel_or))))
--if there is vowel add h before that
(λ (add_before_match (string_to_substring_list $0) 'h' vowel_or))
```

In LILO, just as in DreamCoder, the various modules contribute recursively to learning by *talking to one another*. In this case, existing programs that invoke `fn_42` are rewritten to refer to `vowel_or`. Moreover, the above documentation is added to the LLM Solver prompt on the next learning iteration, allowing the solver to deploy an entire chunk of useful code by invoking a single, readable abstraction.

For AutoDoc, we encountered difficulties in coaxing Codex to perform the 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 find adhere to the desired output JSON schema 100% of the time. Moreover, as we explore in § 4.2.4, they tend to produce plausible, descriptive abstraction names and docstrings. All of this behavior is *zero-shot*; unlike for search, we do not provide any few-shot examples of the desired transformations, making AutoDoc an extremely domain-general technique that is easy to implement across a variety of settings.

In our analysis in § 4.2.1, we explore how AutoDoc benefits both downstream synthesis performance, leading to better quality and more human-interpretable libraries.

3.5 Implementation

We provide a brief summary of key implementation details relevant to the experiments reported in §4. We ran all experiments on AWS EC2 instances with machine specs tailored to suit the computational workload of each experiment.

For experiments involving DreamCoder 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 DreamCoder 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 Wong et al. (2021) (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 Wong et al. (2021) 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.

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 8-CPU `c5.2xlarge` machines. (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.

For experiments involving compression, we make use of the STITCH Python bindings, which interface with a fast backend written in Rust. 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 the performance of an LLM solver.

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

Chapter 4

Experiments

4.1 Domains

We evaluate our approach on three inductive program synthesis domains: REGEX string editing, CLEVR scene reasoning, and LOGO compositional graphics. These three domains were introduced in LAPS (Wong et al., 2021) as more complex extensions of the kinds of tasks evaluated in DreamCoder (Ellis et al., 2020, 2021). Each domain is split into train and test tasks; we reproduce the exact task splits used in Wong et al. (2021). Table 4.1 contains a summary of the domains showing the relative numbers of tasks in each split and summary statistics that indicate the complexity of the underlying programs, as measured by description and string length.

	#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 4.1: **Summary statistics for the domains used in this work.** 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.

4.1.1 REGEX: String editing

As an entree into this work, we evaluate on a domain of *structured string transformation problems*–a classic task in inductive program synthesis (Lau and Weld, 1998). The dataset, originally introduced in Andreas et al. (2017), 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 (Ellis et al., 2021; Wong et al., 2021), 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 (Andreas et al., 2017).

4.1.2 CLEVR: Scene reasoning

We extend our approach to a *visual question answering* (VQA) task based on the CLEVR dataset (Johnson et al., 2017). Following successful efforts in modeling VQA as program synthesis (Andreas et al., 2016; Hu et al., 2017), each synthesis problem 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 Wong et al. (2021), 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 in order to solve. 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 are generated based on the original high-level templates in Johnson et al. (2017).

4.1.3 LOGO: Compositional graphics

Following in a long tradition of modeling *vision as inverse graphics*, (Kersten and Yuille, 1996; Kersten et al., 2004; Yuille and Kersten, 2006; Lee and Mumford, 2003; Wu et al., 2015; Yildirim et al., 2020; Wu et al., 2017; Yi et al., 2018; Gothoskar et al., 2021) we evaluate on a domain of *compositional drawing problems*. The dataset, originally introduced in (Wong et al., 2021) and based on a simpler dataset from (Ellis et al., 2021), contains programs that generate shapes and designs in a vector graphics language. The DSL is based on Logo Turtle graphics (Abelson and diSessa, 1986), 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 (Ellis et al., 2021). Synthetic language annotations (Wong et al., 2021) are generated with high-level templates over the objects and relations in each task.

4.2 Results

4.2.1 Synthesis efficacy

LLMs facilitate effective search over lambda calculus programs. Our first question is whether the LLM-based search procedure introduced in §3.3 can effectively amortize enumerative DreamCoder search. In terms of the overall percentage of test tasks solved per domain, we find that the LLM Solver performs comparably to the DreamCoder baseline on LOGO ($\mu_{\text{LLM}} = 32.13\%$ vs. $\mu_{\text{DC}} = 28.53\%$), slightly worse

than DreamCoder on CLEVR ($\mu_{\text{LLM}} = 88.67\%$ vs. $\mu_{\text{DC}} = 94.50\%$), and significantly better than DreamCoder on REGEX ($\mu_{\text{LLM}} = 76.13\%$ vs. $\mu_{\text{DC}} = 43.93\%$) (see Table 4.2 for a full summary of task solution rates). The improvements on REGEX are primarily attributable to LLMs' ability to generate expressions for concepts like `vowel` and `consonant` that invoke human commonsense prior knowledge.

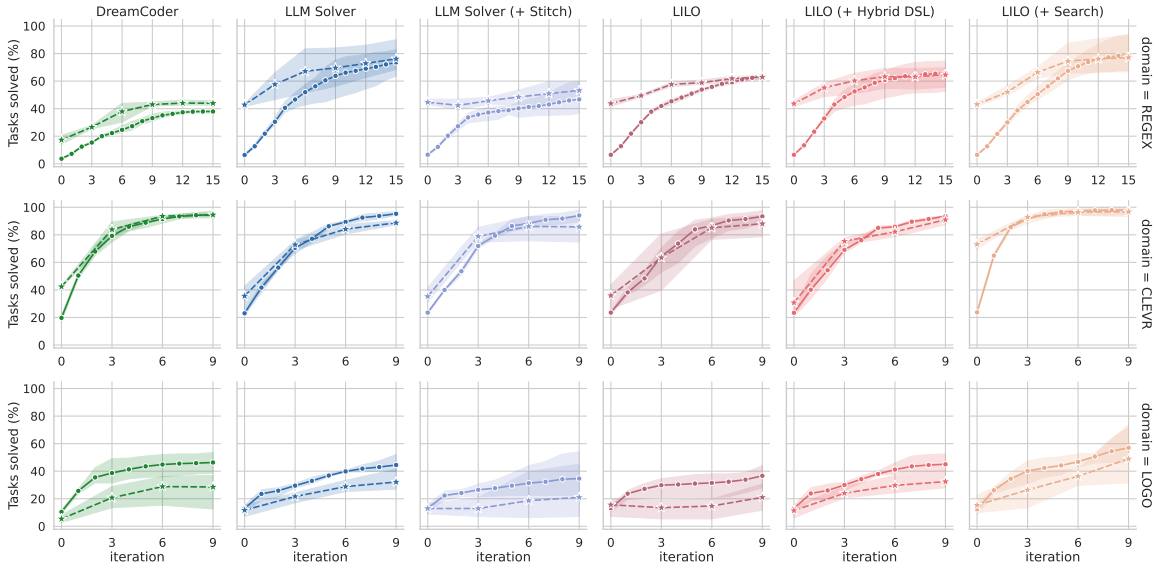

Figure 4-1: **Comparison of learning curves across synthesis experiments.** Results are organized by domain (row) and model (column). Within each plot, the x-axis is the experiment iteration and the y-axis shows the percent of tasks solved. Each plot contains two lines: **train** (•) and **test** (⋆), where test is evaluated every 3 iterations. Error bars show standard deviation across 3 randomly-seeded runs. Each run starts at `iteration = 0`, which corresponds to searching in the base DSL.

Naive integration of LLM Solver and compression hurts performance. Early experiments interfacing the LLM Solver with Stitch (Table 4.2, `LLM Solver (+ Stitch)`) revealed a puzzling finding: providing the LLM with higher-level abstractions from Stitch, which are designed by construction to provide useful functionality, did not help—and in some cases, actually *hurt*—downstream synthesis performance. Relative to the `LLM Solver` baseline, we observed mean solution rate decreases of −30.60% (REGEX), −2.91% (CLEVR), and −11.11% (LOGO) after introducing Stitch compression. Qualitative inspection found that GPT was struggling to de-

	Tasks solved in language-guided synthesis (%)								
	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 (+ Stitch)	59.40	53.20	5.38	93.20	85.76	9.72	45.05	21.02	20.88
LILO	63.80	62.93	1.50	94.17	88.03	8.26	30.63	21.02	9.46
LILO (+ Hybrid DSL)	69.60	64.53	7.76	94.17	90.94	3.41	40.54	32.43	7.04
LILO (+ Search)	**93.20**	**77.07**	14.14	**99.03**	**96.76**	3.12	**73.87**	**48.95**	22.15

Table 4.2: **Task solution rates for primary synthesis experiments.** We report final solve rates for the best (*max*), average (*mean*), and standard deviation (*std*) across the runs in each condition. Performance on these domains is directly comparable to results from Wong et al. (2021) (see Appendix A.4).

ploy the abstractions and compose them in contextually-appropriate ways. One key insight was that Stitch returns abstractions using anonymous function names (i.e., `fn_42`, where the number refers to the ordering of the function in the PCFG, and where abstractions are always numbered after the existing DSL primitives). As we explore further in § 4.2.4, there are certain cases in which high-level abstractions can *obfuscate* the underlying DSL semantics. These observations motivate the introduction of auto-documentation procedure in Lilo (§3.4), which we evaluate below.

Lilo aids the LLM Solver in contextual deployment of library abstractions. After introducing AutoDoc, we see mean improvements of +9.73% (REGEX) and +2.27% (CLEVR) over the `LLM Solver (+ Stitch)` condition (Table 4.2, `LILO`; we do not observe any change in mean performance on LOGO). We also introduce a variant, `LILO (+ Hybrid DSL)`, where we start by prompting the LLM with the full library as in the base `LILO` condition. If the task is not solved after some heuristic fraction of the budgeted $n_{\text{prompts\_per\_task}}$ (here, we use 50%), we remove the abstractions and revert to prompting in the Base DSL. With this hybrid DSL prompting technique, we observe mean improvements of +10.20% (REGEX), +5.18% (CLEVR), and +11.41% (LOGO) absolute accuracy over `LLM Solver (+ Stitch)`. Note that this variation does not consume any more resources than base Lilo; both are given the same total prompt budget.

Integrating LILO and search achieves the strongest performance on all domains. We tested a variant of LILO that combines LLM prompting with DreamCoder enumerative search: at each iteration, we run the LLM Solver, followed by a round of DreamCoder search. Across all domains, we observe that this ensembled search model achieves the best performance of the conditions we tested. This model improves on both the DreamCoder baseline by +33.14% (REGEX), +2.26% (CLEVR), +20.42% (LOGO) and the LLM Solver baseline by +3.20% (REGEX), +8.09% (CLEVR), +16.82% (LOGO). However, unlike the `LILO (+ Hybrid DSL)` condition, `LILO (+ Search)` *does* consume more resources than base LILO as it performs an enumerative search in addition to running the LLM Solver. A further analysis of computational tradeoffs is included in § 4.2.5.

4.2.2 Benchmark comparison to prior work

The results from our three domains are directly comparable to those from Wong et al. (2021). The primary results from that work are reproduced in Appendix A.4, 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 Wong et al., is comparable to the "LAPS in neural search" condition in Table A.1, 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 (93.2% best vs. 57.00% best) than this prior work, even without explicitly computing language/program alignment. 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). Adding in LILO (+ Search) 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. Moreover, even matching the 1800s search time, we were unable to obtain a DreamCoder run that matches their equivalent baseline on

this domain (36.94% 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. A-3, *LAPS in neural search*), so a comparison of average results (28.93% LILO mean vs. 52.93% LAPS mean) may be more appropriate. Only in the `LILO (+ Search)` condition are our results (73.87% best, 48.95% mean) comparable to the LOGO baseline reported in 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 Wong et al.. It is also worth noting that only the `DreamCoder` and `LILO (+ Search)` 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 four conditions that rely exclusively on LLM-guided search must infer what to draw solely based on the task descriptions; an exceedingly difficult generalization task.

4.2.3 Quantitative evaluation of library abstractions

The goal of library learning is not just to solve synthesis tasks, but also to discover useful, generalizable abstractions. One way to measure the quality of a library is to evaluate its performance in *unconditional search*. In other words, if a library is useful, then we should be able to use it to solve many tasks relatively quickly. An advantage of this evaluation is that it is entirely self-contained; holding the parameters of the search fixed, downstream performance depends entirely on the utility of the abstractions in the library.

To operationalize this evaluation, we initialize a PCFG with the final library $\mathcal{L}_f$ from each model run. We then perform *unconditional search* in this PCFG for a fixed test time budget equal to the search time allocated to each test iteration on a per-domain basis (c.f. §3.5). We choose unconditional search, as opposed to neurally-guided search, for this evaluation because this provides a more direct comparison of libraries: whereas neurally-guided search requires training a recognition network on a run-dependent set of tasks and descriptions, unconditional search depends only on the primitives and abstractions in $\mathcal{L}_f$ and the prior probabilities they are assigned under $\theta_{\mathcal{L}_f}$.

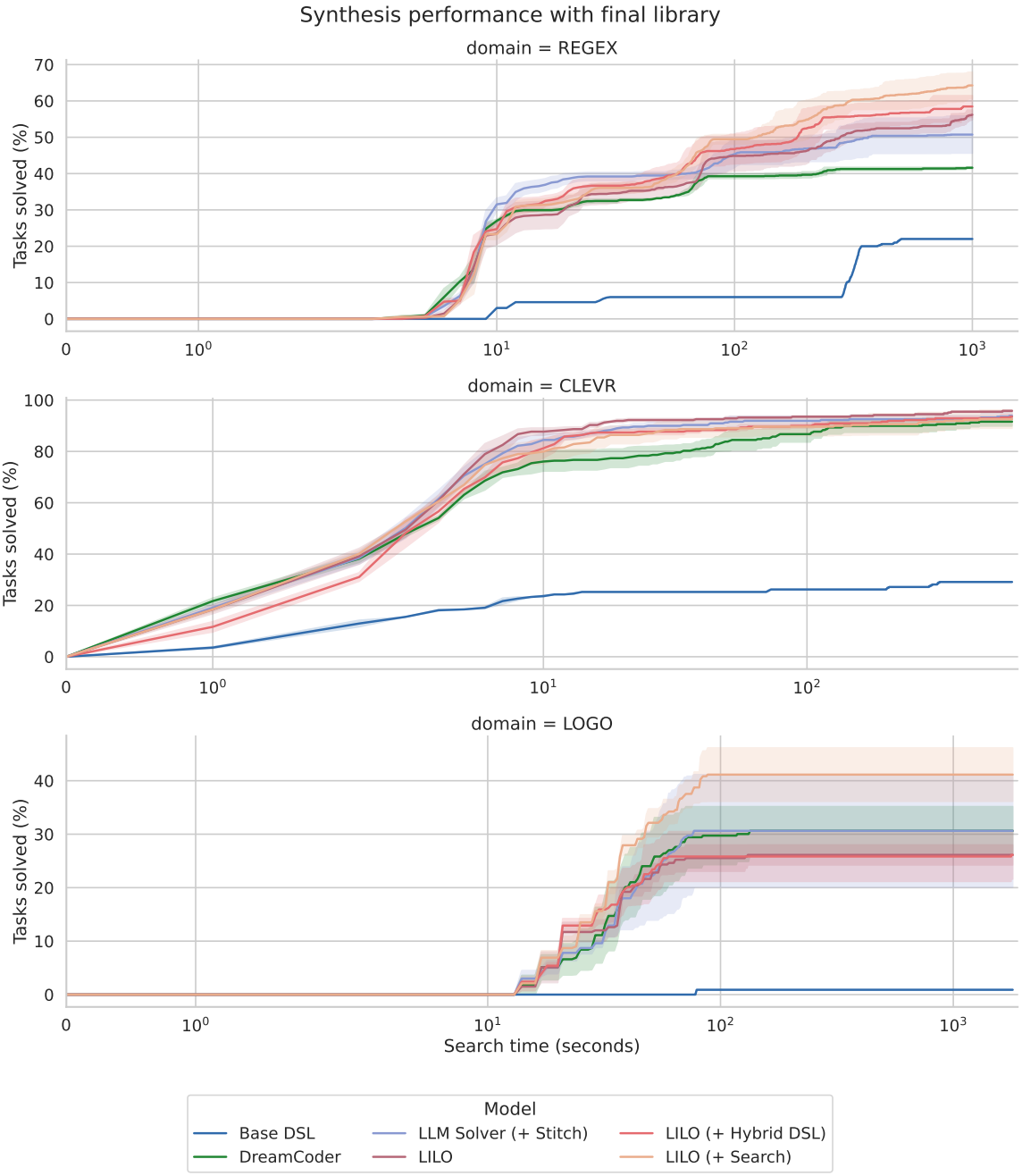

Figure 4-2: **Evaluating library quality by synthesizing with the final library.** We initialize a weighted PCFG with the final library $\mathcal{L}_f$ from each model run. We perform unconditional search (no neural guidance) in this PCFG for a per-domain fixed test time budget (note the log-scale on the x-axis). With search parameters identical across conditions except for the initial PCFG, higher performance (y-axis) indicates more useful, generalizable libraries.

	Tasks solved in unconditional synthesis (%)								
	REGEX			CLEVR			LOGO		
Model	*max*	*mean*	*std*	*max*	*mean*	*std*	*max*	*mean*	*std*
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 (+ Stitch)	60.80	50.73	8.85	95.15	93.85	2.24	51.35	30.63	18.22
LILO	57.60	56.20	2.25	96.12	95.79	0.56	28.83	26.13	3.25
LILO (+ Hybrid DSL)	62.40	58.47	5.18	95.15	93.53	1.48	35.14	26.13	7.85
LILO (+ Search)	**71.40**	**64.27**	6.31	**96.12**	**92.56**	6.17	**50.45**	**41.14**	8.66

Table 4.3: **Task solution rates for unconditional synthesis experiments with final libraries.** We report final solve rates for the best (*max*), average (*mean*), and standard deviation (*std*) across the runs in each condition.

Fig. 4-2 and Table 4.3 show the results of these unconditional search evaluations. In each domain, we measure synthesis performance in the base DSL (dark blue line), which serves as a baseline for this experiment. As expected, we can significantly outperform this baseline using library learning: DreamCoder (green) improves absolutely on the Base DSL solve rates by +19.6% (REGEX), +62.46% (CLEVR), and +29.73% (LOGO). Moreover, in each domain, the best Lilo conditions improve further on DreamCoder, showing, on average, absolute solution rate gains of +42.27% (REGEX), +63.43% (CLEVR), and +43.24% (LOGO) over the base DSL performance. As these quantitative results demonstrate, Lilo learns high-quality libraries that generalize well to downstream synthesis tasks, outperforming DreamCoder on all three domains.

4.2.4 Qualitative inspection of library abstractions

We generated graphical visualizations of the libraries learned by the best Lilo model for each domain (`LILO (+ Search)`). Each graph includes the DSL primitives, the 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.

Fig. 4-3 shows the library map for the CLEVR library (the maps for REGEX and LOGO libraries are included in Appendix A.1). We choose CLEVR as a showcase

because LILO solves $> 99\%$ of the domain's test tasks using this library (Table 4.2), so it is a useful success case study. The final library contains two layers of abstractions. The lower layer implements `filter` operations over size, color, shape, and material attributes, which constitute the main axes of variation in the domain. These abstractions form the building blocks for a layer of higher-level abstractions that implement more specialized operations, like `count_remaining_objects_by_color_and_shape` and `filter_objects_by_rubber_material`. The library auto-documentation (reproduced in full in Appendix A.2.2) provides userful insight into the mechanics of these functions. For instance:

```
(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))
```

Libraries in the other domains also exhibit examples of compositional and hierarchical reuse. For instance, in REGEX (Fig. A-1), the most compressive library abstracion is the `vowel_or` expression highlighted in §3.4. This abstraction is invoked by the higher-level `replace_starting_consonant` abstraction, which makes use of the fact that *consonant* is the set-complement of *vowel*:

```
(fn_45) replace_starting_consonant :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_if (regex_match (regex_not vowel_or) (regex_car (regex_split
empty_string $2))) (regex_flatten (regex_cons $0 (regex_cons $1 (regex_cdr
(regex_split empty_string $2))))) $2))))
{- Given a full string $0 and two substrings $1 and $2, returns a full string where
the first letter of each word starting with a consonant in $0 is replaced with $1$2.
Vowels are determined using `$vowel_or` regular expression. -}

{- Example usages -}
--if the word starts with consonant replace that with r p
(λ (replace_starting_consonant $0 'p' 'r'))
--if the word starts with consonant replace that with q
(λ (replace_starting_consonant $0 'q' empty_string))
```

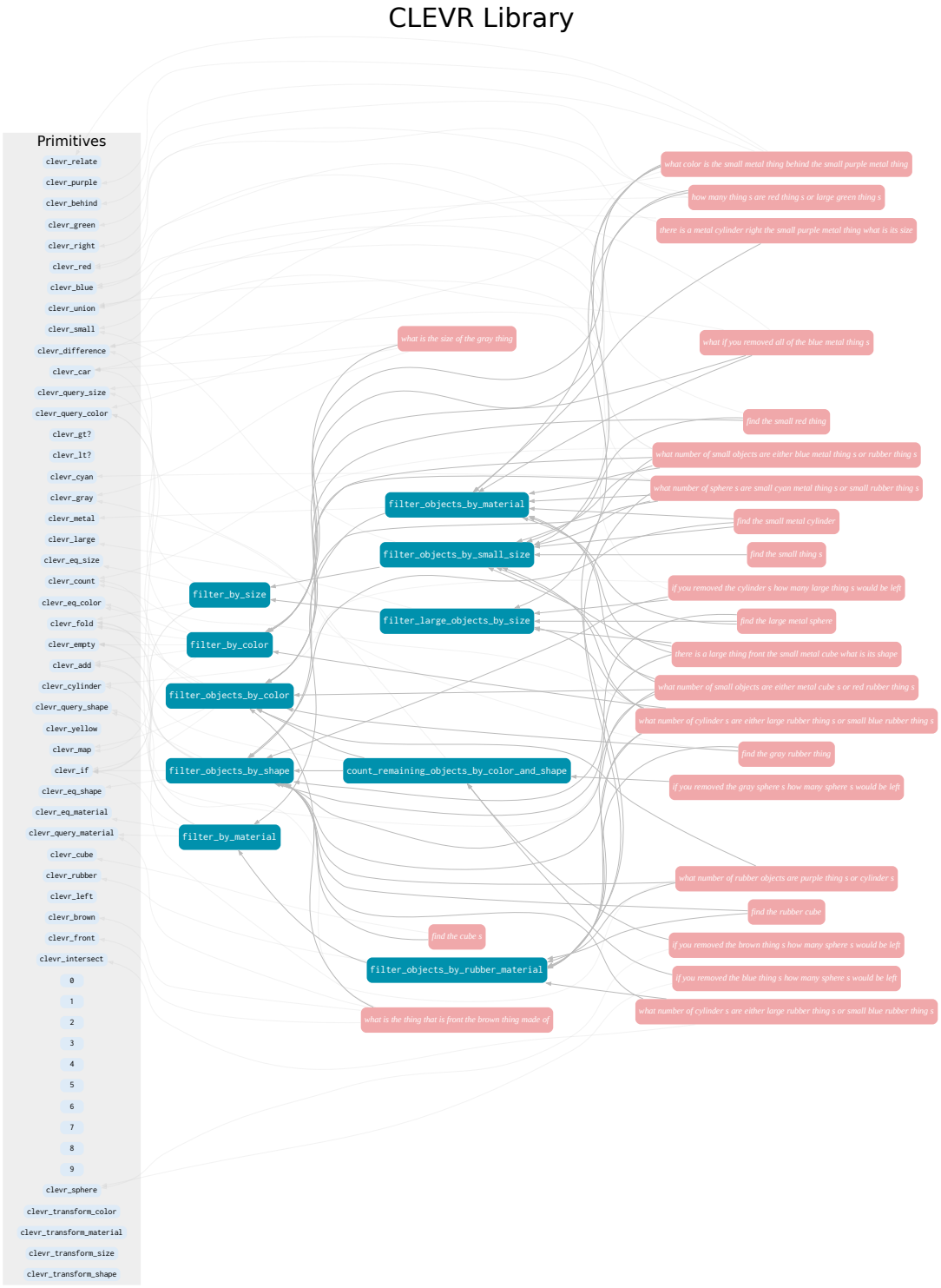

Figure 4-3: **Graphical map of CLEVR library learned by Lilo.** Named abstractions (turquoise) are hierarchically composed of other abstractions and ground out in the base DSL primitives (gray box). For instance, `filter_objects_by_rubber_material` invokes the lower-level abstraction `filter_by_material`, as well as the primitive constant `clevr_rubber`. Solved tasks (red) are shown with their language descriptions and the set of library functions utilized in the solution. Graphical maps for REGEX and LOGO libraries are included in Appendix A.1.

```
--if the word starts with consonant replace that with e n
(λ (replace_starting_consonant $0 'n' 'e'))
```

Similarly, in LOGO, the most compressive abstraction, `turtle_loop_move_rotate`, is a general method for drawing n-gons that is invoked by several higher-level library abstractions; e.g., for drawing snowflakes and for making n-gons with double-length sides. The ability to bootstrap hierarchies of learned concepts while jointly learning to solve programming tasks has traditionally been one of the selling points of DreamCoder. The examples above showcase how LILO builds on these hallmarks, facilitating discovery of abstractions like `vowel_or` that require human-like priors knowledge, while also improving the interpretability of the learned libraries through auto-documentation.

Having highlighted some of the success cases of LILO, we now turn to some failure modes. One common issue is the presence of *semantic errors* in abstraction naming and documentation. While instruct-tuned GPT models do a remarkable job at inferring program semantics from the prompts, we observe various cases where they produce unhelpful or even misleading outputs. For instance, in the LOGO domain (Fig. 4-4), `turtle_loop_move_rotate (fn_27)` is a rather uninformative name for what is actually a function that generates polygons. Because LILO names libraries iteratively, passing in the existing names at each step, there is a tendency for naming schemes to propagate to higher-level abstractions. While naming consistency is generally a desirable property, iterative naming can also create situations where the LLM "doubles down" on prior issues, causing semantic errors to compound. For instance, `double_length_loop_move_rotate (fn_34)` is a decidedly ambiguous name for a function that produces n-gons of fixed side length 2. Moreover, the documentation (Fig. 4-4) makes clear that GPT appears to misunderstand the function's semantics, incorrectly stating that each iteration doubles the length of the turtle's movement. Such a misinterpretation may have been influenced by a similar semantic error that the LLM had previously made for `(fn_31)`, where the documentation falsely states that "the angle by which the lines are rotated increases with each iteration of the loop." In LILO, this documentation becomes part of the library and is passed to the LLM

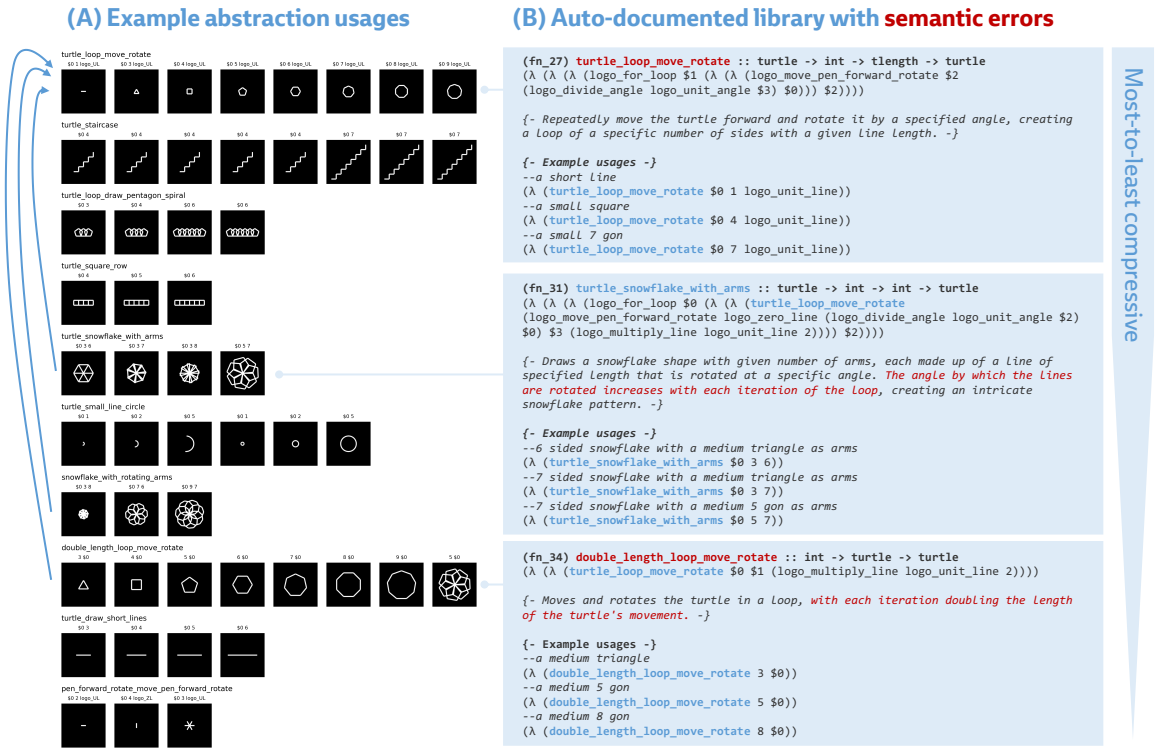

Figure 4-4: **Qualitative inspection of LOGO library.** (A) Rendered examples of usages of each abstraction in the final library learned by LILO. Above each render is the set of arguments that the abstraction was invoked with, where `$0` refers to the canvas object. Abstractions are ordered top-down from most-to-least compressive. For instance, the top abstraction, `turtle_loop_move_rotate (fn_27)`, is a general method for drawing n-gons that is invoked by several other library abstractions (arrows, left side). (B) Excerpts from the auto-documentation of three selected abstraction, with semantic errors highlighted in red. For instance, the documentation for `double_length_loop_move_rotate (fn_34)` incorrectly states that each iteration doubles the length of the turtle's movement. In actuality, `fn_34` invokes `fn_27` to produce n-gons of fixed side length 2. AutoDoc errors such as these may unintentionally obfuscate library semantics, inversely affecting the LLM Solver's ability to deploy abstractions in context.

Solver at search time. Therefore, these kinds of semantic errors do not merely affect human interpretability; they also have the potential to adversely impact downstream solution rates.

4.2.5 Computational efficiency

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 traditional 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 §3.5, 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.

We start by analyzing observed (a.k.a. "wall clock") runtimes of our different models. Fig. 4-5 breaks these down by domain, where the x-axis corresponds to the average time to perform a single search iteration during training and test.[1] Overall, we observe that even with network latency from interfacing with OpenAI servers, a round

[1]Note that in Fig. 4-5, 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.

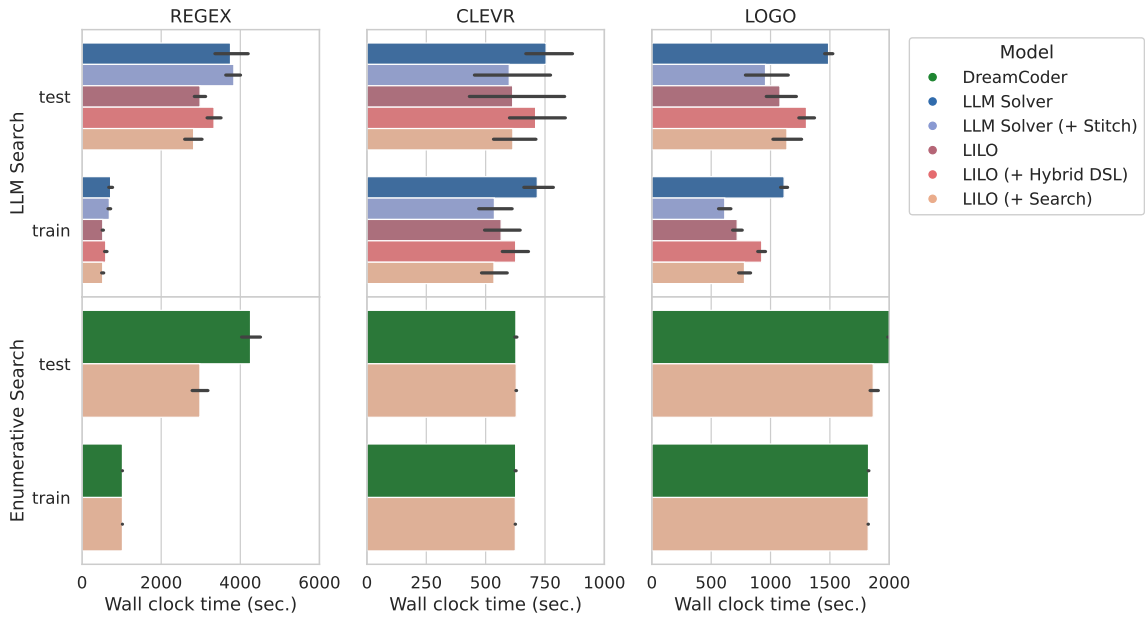

Figure 4-5: **Comparison of wall clock runtimes across search procedures and domains.** Each bar shows average runtime for a single iteration of train/test program search (error bars indicate 95% confidence intervals). Even with network latency from interfacing with OpenAI servers, LLM search (top row), typically requires less execution time than enumerative search (bottom row), which runs locally on a 96-CPU machine.

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 Wong et al. (2021); see §3.5 for more details). We do not observe major differences in runtimes within the different LLM Solver conditions, though it is worth noting that the `LILO (+ Search)` model requires approximately 2x more total runtime than the other LILO models because it performs both LLM-based and enumerative search on each iteration.

Next, we consider the token usage of the LLM solver conditions. Fig. 4-6 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

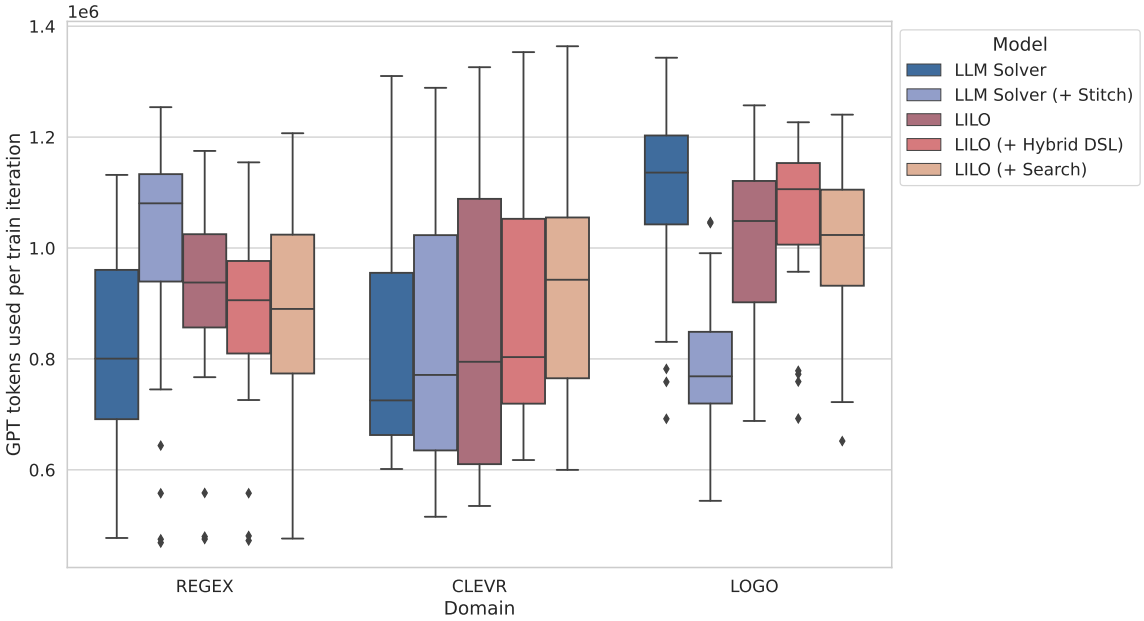

Figure 4-6: **GPT token usage per training iteration.** Token usage provides a useful metric for assessing the computational costs of LLM-based program search. A typical training iteration uses on the order of 0.8M-1.2M GPT tokens between the prompt and the completion. (Note the y-axis measures millions of tokens.) Boxes indicate quartiles of the distribution and whiskers extend to 1.5 inter-quartile ranges, with outliers shown as individual points.

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 `LLM Solver`) will also tend to benefit from shorter program description lengths per task. In particular, the `LLM Solver (+ Stitch)` 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.

Finally, in the spirit of providing an apples-to-apples compute cost comparison, we

		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 (+ Stitch)	$2.04	$0.39	$1.66	$0.47	$1.59	$0.24
	LILO	$1.86	$0.30	$1.70	$0.52	$2.03	$0.31
	LILO (+ Hybrid DSL)	$1.78	$0.30	$1.78	$0.43	$2.12	$0.29
	LILO (+ Search)	$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
	LILO (+ Search)	$1.16	$0.00	$0.71	$0.00	$2.07	$0.00

Table 4.4: **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.

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 §3.3, the LLM Solver experiments reported here took advantage of OpenAI's `Codex` model private beta for academic 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. 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.

Table 4.4 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 Amazon 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 (Theis and Wong, 2017), 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. Moreover, as discussed in §3.2, 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 visibility into performance considerations.

4.3 Discussion

We conclude the presentation of our experiments with several discussion points situating our findings in the broader landscape of program synthesis research.

4.3.1 Transformer self-attention as implicit library learning

While we set out in this work to tell a story about *library learning*, one of the most striking findings from our experiments is the strong performance of the LLM-only baseline relative to both DreamCoder and LILO. Our experiments show that even without the explicit ability to compress solutions into symbolic abstractions, LLMs nevertheless appear able to identify, reuse, and repurpose existing program structures to solve new tasks. Partly, this is due to the synthetic nature of our domains, which often have stereotyped correspondences between task descriptions and solved programs: given solutions to `find the small metal cylinder` and `find the large metal sphere`, even an RNN encoder-decoder networks can generalize to `find the large metal cylinder` (Andreas et al., 2016, 2017).

What makes these generalizations notable in transformers is that—unlike prior RNN-based approaches—there is no updating of network parameters. In this sense, LLMs can be seen as implementing a form of *implicit library learning*, where the solution set is itself the library.

How far can we scale this idea? In theory, transformer self-attention is limited both by model depth (number of layers) and witdh (size of context window) (Hahn, 2020). Nevertheless, since the invention of the transformer architecture, researchers have been investigating ways to achieve longer-range attention (see Tay et al., 2022, 2021 for a review). Various approaches have drawn on truncated backpropagation through time (Dai et al., 2019; Rae et al., 2020) and continuous-space attention (Martins et al., 2022) to achieve compressive memory with longer or even infinite context width. Other approaches use retrieval from external knowledge sources (Guu et al., 2020; Lewis et al., 2020; Fan et al., 2021), augmenting transformers with vast and typically lossless memory capacity. Recent work in this line explicitly envisions LLMs capable of ingesting and immediately acquiring new knowledge at inference time through memorization (Wu et al., 2022). In the near term, leveraging these advances, we can start to imagine systems that can both solve programming tasks on-the-fly and cache them for later retrieval; using analogy to prior experience to

inform the solution of new problems.

4.3.2 Measuring and trading off costs of compression

Sooner or later, any library learning system must confront the following question: *How many abstractions should be in the library?* In this work, we offload this question to STITCH, which internally computes a utility for each abstraction based on the amount it compresses the corpus (Bowers et al., 2023). To avoid the computationally thorny problem of jointly estimating this utility for all library abstractions, STITCH instead builds up libraries in a greedy manner, adding in abstractions one-by-one and stopping when no further compression can be achieved.

However, for any moderately-size program corpus, there is a long tail of marginally-compressive abstractions. Naively adding all of these to the library will have an adverse effect on program search, both in the enumerative and LLM-driven settings. For enumerative algorithms, the library size $|\mathcal{L}|$ defines the branching factor of the search, so each additional abstraction causes the search space to scale accordingly. Meanwhile, for LILO-style LLM search, each additional abstraction definition consumes space in the prompt (Fig. 3-4 A), reducing the space available for in-context examples. Thus, there is an imperative to keep the library relatively modest in size—but how can we quantify these tradeoffs?

This is one area where DreamCoder offers a more principled answer than STITCH—albeit, one that comes with performance costs. Because DreamCoder compression represents many possible rewrites of candidate programs simultaneously via version space algebras, it is able to perform a beam search to approximately recover a large number of candidate libraries (in practice, the top 10^6 libraries are considered). This beam search is one factor that makes DreamCoder many orders of magnitude more time and memory intensive than STITCH. However, in the absence of such a search over libraries, it is difficult to know *when to stop* growing the library. In practice, we use a relatively low heuristic cutoff: each library can have a maximum of 10 abstractions. However, even with this cutoff, we observe some instances of overfitting: our final libraries contain abstractions like `filter_objects_by_rubber_material` (Ap-

pendix A.2.2) and `replace_word_end_with_substring_if_ends_with_e` (Appendix A.2.1) that are clearly overspecified. For this reason, an important next step will be to dynamically adapt the library size based on some more formal notion of utility.

Returning to insights from prior work, one approach is to explicitly formalize the tradeoff between the description length of a candidate library and the description length of the corpus of programs under that library. For instance, Wong et al. (2022) compute a *combined representational cost* $C_{\mathcal{L}_i} = |\mathcal{L}_i| + \frac{1}{N}\sum_{\rho}|\rho_{\mathcal{L}_i}|$, where $\mathcal{L}_i$ is the candidate library and N is the cardinality of the set of programs to be expressed under $\mathcal{L}_i$. (DreamCoder also computes a version of this cost via its library prior in Eq. (3.5).) Such a notion of combined representational cost may offer a cognitively-grounded way of approaching the library size problem. Indeed, human language usage empirically supports the hypothesis that people favor a lexicon that allows concise item-wise descriptions, while also minimizing the size of the lexicon itself (Wong et al., 2022). Integrating such a utility function into STITCH therefore might offer a more principled way of determining the ideal library size throughout the course of learning.

4.3.3 Connections to dual-system accounts of problem-solving

How can we think about the relationship between LLM-based and enumerative search in a broader cognitive framework? Dual-process theories of cognition posit that reasoning emerges from an interplay between a more intuitive and associative "System 1" and a more deliberative and logical "System 2." (Sloman, 1996; Stanovich, 1999; Evans, 2003; Kahneman, 2011) Under the dual-process framework, System 1 drives automatic responses based on pattern-recognition, while System 2 is engaged for more deliberative, multi-step reasoning tasks. Recent work suggests that adopting such a dual-process framing for LLMs—integrating them with symbolic reasoning modules—has the potential to improve coherence and consistency in LLM generations (Nye et al., 2021; Huang et al., 2022).

Inspired by this work, two aspects of LILO can be framed in terms of dual-system theories. First, the integration with STITCH, an external, symbolic compressor, is comparable to interfaces with other symbolic modules, like logic engines. As we

argued in §1, STITCH is purpose-built to solve a difficult refactoring problem that falls squarely into traditional System 2 territory (though in §5.3, we explore the possibility LLMs might productively amortize compression as well). Moreover, as we saw in our experiments, LLMs and enumerative search are not mututally exclusive. Indeed, the `LILO (+ Search)` model—which was explicitly designed with a dual-system hypothesis in mind—achieves the best solve rates on all domains (though, as currently implemented, it comes with higher computational costs from performing both forms of program search).

An operative challenge for future work, then, is to build more tightly integrated neurosymbolic systems—specifically, to allow an LLM in the role of System 1 to determine when to call out to a symbolic compression or search routine in the role of System 2. A growing line of work in the LLM literature explores precisely this question: interfacing LLMs with external computational tools designed for more systematic reasoning, including calculators (Cobbe et al., 2021), logic engines (Weir and Van Durme, 2022), external databases (Borgeaud et al., 2022; Thoppilan et al., 2022; Alon et al., 2022; Izacard et al., 2022), web browsers (Nakano et al., 2022), AI planners (Collins et al., 2022), physics simulators (Liu et al., 2022), or bundled APIs over several of these modules (Karpas et al., 2022; Schick et al., 2023; OpenAI, 2023). As LLMs begin to interact with increasingly large libraries of tools, we believe that these models will benefit from automatic procedures—such as the ones introduced in LILO—for growing these libraries with useful, interpretable, and well-documented abstractions.

Chapter 5

Future Directions

In this work, we have explored how LLMs can productively be combined with ideas from program synthesis, focusing concretely on two main areas: guiding search via language and improving library quality and interpretability. However, these experiments only scratch the surface of a much broader set of research opportunities and questions. Here, we outline several further directions in which this line of work might be extended.

5.1 Example-guided synthesis with LLMs

Classically, inductive program synthesis techniques make heavy use of input-output examples, not just to compute program likelihoods (i.e., in Eq. 2.1), but also to constrain the search (Winston, 1970; Summers, 1976; Lau and Weld, 1999; Singh and Gulwani, 2012; Solar-Lezama et al., 2006; Solar-Lezama, 2008; Solar-Lezama, 2013; Ellis et al., 2015, 2018). One notable limitation of the LLM-amortized search we introduced in §3.3 is that it does not directly condition on these I/O examples, instead focusing on inferring patterns in description-solution pairs (d_t, ρ).

Conditioning on task descriptions makes sense as a starting point for LLM-guided search, as we desire a relatively concise specification modality that can fit in the prompt window. In contrast, I/O examples may not necessarily be easily serializable into an LLM prompt. For instance, in the LOGO domain, examples are specified

by pixel arrays specifying a rendered target image that the program should produce. Even in non-visual domains, the I/O examples may be too unwieldy to include in the prompt: in our CLEVR domain, the inputs for each task consist of multiple scene graphs, each containing arrays of JSON objects specifying various attributes of the objects in the scene. While it might be possible to fit a few such scene graphs in the prompt, doing so would significantly limit the total number of examples that can be included.

Recent developments in LLMs offer an exciting and practical avenue for extending our amortized synthesis to condition on examples. Promisingly, we are starting to see numerous *multimodal large language models* (MLLMs) (Driess et al., 2023; Huang et al., 2023; Girdhar et al., 2023) that are able to condition jointly on text, images, and often include capabilities for other modalities such as audio, depth images, video, and robotic control. Such MMLMs offer a unified way to encode task specifications across different domains. However, problem-solving in these models is typically performed in unstructured text space (i.e., via chain-of-thought prompting; Wei et al. 2023; Kojima et al. 2022). As the complexity of problems scales, adapting a more explicit program synthesis may unlock new forms of multi-step problem solving in MMLMs. Moreover, taking a *library learning* approach, where models write reusable functions in the service of solving tasks, could help to make such models more generalizable and robust.

5.2 Resource-rational inferences

Humans are incredibly flexible thinkers. How might we incorporate understanding of human problem-solving into program synthesis models? Indeed, DreamCoder is sometimes positioned as a model of human-like concept learning in its ability to bootstrap new concepts during learning. However, as discussed in §3.3, the enumerative search that forms the core of this approach is computationally expensive to a degree that calls into question the cognitive plausibility of such a learning procedure.

One way of understanding cognitive plausibility is to view human cognition as

making the optimal use of limited computational resources (Lieder and Griffiths, 2020; Gershman et al., 2015). In this work, the use of LLMs to amortize program search is, in part, intended as a step in the direction of a more *resource rational* allocation of computational costs. In particular, rather than enumerating and checking millions of programs, LILO leverages LLMs as fast, neural pattern-recognizers to perform a kind of *syntactic bootstrapping* (Gleitman, 1990)—inferring the meaning of novel program expressions from a mixture of context that includes language descriptions, descriptive naming, and documentation.

Nevertheless, in practice, there is still much ground to cover before LLMs can be considered approximations of human learning from a resource rational perspective. In § 4.2.5, we discussed some of the technical tradeoffs that our current implementation faces in terms of computational efficiency. Motivated by theories of resource rationality, there are several explorations that are applicable and practicable with current language models.

We might begin by taking a resource rational perspective on *memory* (Gershman and Goodman, 2014; Dasgupta and Gershman, 2021), using the LLM context window as a loose model of human working memory. In this work, to construct prompts for the LLM Solver, we cram as many examples of solved tasks as we can into the context window, with the goal of maximizing the solution rate. Yet, as LLMs continue to scale to larger context windows—the largest GPT-4 model currently accommodates 32K tokens, or approximately 23,000 words (OpenAI, 2023), while Anthropic's Claude model now accommodates 100K tokens[1]—they become increasingly *implausible* as models of human working memory. Accordingly, one interesting experimental direction is to scale in the opposite direction, by trying to maximize the number of program solutions one can derive through a limited number of in-context examples. Comparing task solution rates at progressively restricted token budgets would offer a direct means of evaluating LILO-style search in terms of resource efficiency. Given that compression algorithms like STITCH directly reduce the string length of program expressions, making it easier to do more with a finite token budget, we expect that

[1] https://www.anthropic.com/index/100k-context-windows

such an evaluation would tend to favor library learning over LLM-only approaches.

A resource rational evaluation also suggests new modeling directions for LILO. Under a strict token budget, *selecting the right examples* to put in the prompt becomes critical. In this work, we adopt a naive strategy of randomly sampling examples from the set of solved tasks. However, it would be straightfoward to use one of various example retrieval strategies (Poesia et al., 2022) to select task-relevant examples for the LLM Solver prompt. Similar approaches might also benefit LILO's auto-documentation procedure, addressing some of the semantic errors observed in § 4.2.4 and leading to more accurate and readable function names and docstrings. Because the costs of LLM inference scale with prompt length, adopting a resource-rational might body dramatically reduce the computational costs associated with LILO, allowing learning to run for more iterations and potentially boosting overall library quality.

5.3 Program compression with LLMs

A key part of our approach in LILO was to explore the consequences of amortizing a particular module of DreamCoder—in this case, program search—with LLMs. Counter to our initial expectations when we began the project (though perhaps less surprisingly in light of the recent explosion in demonstrations of LLM capabilities), we found that LLMs are highly adept at solving tasks in niche domain-specific languages expressed in esoteric variations of lambda calculus. A core part of the story is that such substitution of rigid PCFG search for a softer, pre-trained string search allows us to incorporate commonsense priors learned through natural language—for instance, the concepts of "vowel" and "consonant"—that would otherwise be combinatorially difficult to discover.

In light of these findings, a natural next step is to ask whether we can similarly *amortize compression* via LLMs. Concretely, we might devise a variant of the AutoDoc scheme from §3.4 that prompts an LLM to *write the abstractions themselves* in addition to producing names and documentation. Analogous to search, an LLM

compressor could bring in prior knowledge learned from pre-training to implement functions that it believes could be relevant to the domain. Unlike symbolic compressors like STITCH, which are limited to pulling out common expressions in the input data, an LLM compressor would be free to propose abstractions that are underrepresented or even completely missing from the current set of solved programs. As in the search case, natural language, such as descriptions of unsolved tasks, or even meta-descriptions of the search domain, could be used to condition abstraction learning.

Chapter 6

Conclusion: The Ship of Synthesis

In the Introduction (§1), we began our journey with DreamCoder. We set out to apply contemporary advancements to this system, swapping out architectural elements piece-by-piece. First, we replaced DreamCoder's compressor with STITCH in order to attain multiple orders of magnitude improvements in efficiency while maintaining the ability to learn high-quality abstractions. Next, we asked whether it is possible to amortize DreamCoder's search algorithm with a large language model and found the answer to be "yes." We showed that when prompted appropriately with a handful of seed programs and task descriptions, LLMs can bootstrap syntactically-valid, semantically-accurate lambda calculus programs, matching DreamCoder's solve rates—and in some cases, significantly exceeding them—on real program synthesis domains. Finally, in introducing LILO's AutoDoc procedure, we took initial steps towards LLM-driven abstraction learning, which led us to hypothesize above that LLM's might productively replace compression as well.

The odyssey we have just completed is reminiscent of an Ancient Greek thought experiment recorded by Plutarch and debated by philosophers over the centuries (Dryden et al., 1859; Hobbes, 1656; Brown, 2005). The "Ship of Theseus" paradox goes as follows: Imagine a battle-worn Athenian ship that has been sailing for many years. Over time, as parts of the ship become worn out or damaged, they are replaced with new parts. Eventually, a day comes when every single part of the ship has been interchanged. The question then arises: Is it still the same ship? If we consider the

ship as a whole, it appears to be the same ship because it retains its original form and function. However, if we focus on the individual parts, none of the original elements remain. So, is it the arrangement of parts that defines the identity of the ship, or is it something else?

We hold that, even after deconstructing and reconstructing every piece of our "Ship of Synthesis," the *spirit* of DreamCoder still stands. The essence of this spirit can be summarized as an *architectural imperative for modularity*. Even with all the symbolic pieces replaced by neural machinery, there is something fundamental about the interplay between search and compression—or *wake* and *sleep*—that the DreamCoder algorithm captures, and that we should seek to imbue in future systems. In particular, even if transformer context windows can be scaled to epic proportions, models that cannot form new abstractions will face strict limitations that arise from having to redo many computations—re-deriving semantics and re-inferring prior inferences—every time they are invoked.

In this light, abstraction learning is an essential piece in our understanding of how people program. Since the inception of the automatic computer, human programmers have formed collaborative communities that have prioritized well-factored code and clear documentation. Accordingly, we believe the near-term future of program synthesis is an evolution following naturally from these longstanding traditions, in which human and AI programmers work in tandem to build up shared libraries of abstractions, enabling creative solutions to new generations of software problems.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abelson, H. and diSessa, A. (1986). *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. The MIT Press.

Alon, U., Xu, F. F., He, J., Sengupta, S., Roth, D., and Neubig, G. (2022). Neuro-Symbolic Language Modeling with Automaton-augmented Retrieval. *undefined*.

Andreas, J., Bufe, J., Burkett, D., Chen, C., Clausman, J., Crawford, J., Crim, K., DeLoach, J., Dorner, L., Eisner, J., Fang, H., Guo, A., Hall, D., Hayes, K., Hill, K., Ho, D., Iwaszuk, W., Jha, S., Klein, D., Krishnamurthy, J., Lanman, T., Liang, P., Lin, C. H., Lintsbakh, I., McGovern, A., Nisnevich, A., Pauls, A., Petters, D., Read, B., Roth, D., Roy, S., Rusak, J., Short, B., Slomin, D., Snyder, B., Striplin, S., Su, Y., Tellman, Z., Thomson, S., Vorobev, A., Witoszko, I., Wolfe, J., Wray, A., Zhang, Y., and Zotov, A. (2020). Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Andreas, J., Klein, D., and Levine, S. (2017). Learning with latent language. *arXiv preprint arXiv:1711.00482*.

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2016). Neural Module Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 39–48, Las Vegas, NV, USA. IEEE.

Artzi, Y. and Zettlemoyer, L. (2013). Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bakker, M. A., Chadwick, M. J., Sheahan, H. R., Tessler, M. H., Campbell-Gillingham, L., Balaguer, J., McAleese, N., Glaese, A., Aslanides, J., Botvinick, M. M., and Summerfield, C. (2022). Fine-tuning language models to find agreement among humans with diverse preferences.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2016). Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Belinkov, Y. and Glass, J. (2019). Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M., Krishna, R., Kuditipudi, R., Kumar, A., Ladhak, F., Lee, M., Lee, T., Leskovec, J., Levent, I., Li, X. L., Li, X., Ma, T., Malik, A., Manning, C. D., Mirchandani, S., Mitchell, E., Munyikwa, Z., Nair, S., Narayan, A., Narayanan, D., Newman, B., Nie, A., Niebles, J. C., Nilforoshan, H., Nyarko, J., Ogut, G., Orr, L., Papadimitriou, I., Park, J. S., Piech, C., Portelance, E., Potts, C., Raghunathan, A., Reich, R., Ren, H., Rong, F., Roohani, Y., Ruiz, C., Ryan, J., Ré, C., Sadigh, D., Sagawa, S., Santhanam, K., Shih, A., Srinivasan, K., Tamkin, A., Taori, R., Thomas, A. W., Tramèr, F., Wang, R. E., Wang, W., Wu, B., Wu, J., Wu, Y., Xie, S. M., Yasunaga, M., You, J., Zaharia, M., Zhang, M., Zhang, T., Zhang, X., Zhang, Y., Zheng, L., Zhou, K., and Liang, P. (2022). On the Opportunities and Risks of Foundation Models.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J.-B., Damoc, B., Clark, A., Casas, D. d. L., Guy, A., Menick, J., Ring, R., Hennigan, T., Huang, S., Maggiore, L., Jones, C., Cassirer, A., Brock, A., Paganini, M., Irving, G., Vinyals, O., Osindero, S., Simonyan, K., Rae, J. W., Elsen, E., and Sifre, L. (2022). Improving language models by retrieving from trillions of tokens.

Bowers, M., Olausson, T. X., Wong, L., Grand, G., Tenenbaum, J. B., Ellis, K., and Solar-Lezama, A. (2023). Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL).

Branavan, S. R., Chen, H., Zettlemoyer, L., and Barzilay, R. (2009). Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 82–90.

Brown, C. (2005). *Aquinas and the ship of theseus: Solving puzzles about material objects*. A&C Black.

Brown, P. F., Della Pietra, S. A., Della Pietra, V. J., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Cambronero, J., Gulwani, S., Le, V., Perelman, D., Radhakrishna, A., Simon, C., and Tiwari, A. (2023). FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proceedings of the ACM on Programming Languages*, 7(POPL):952–981.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2022). Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Cobbe, K., Kosaraju, V., Bavarian, M., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. (2021). Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Collins, K. M., Wong, C., Feng, J., Wei, M., and Tenenbaum, J. B. (2022). Structured, flexible, and robust: Benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *ArXiv*, abs/1901.02860.

Dasgupta, I. and Gershman, S. J. (2021). Memory as a computational resource. *Trends in Cognitive Sciences*, 25(3):240–251.

Dechter, E., Malmaud, J., Adams, R. P., and Tenenbaum, J. B. (2013). Bootstrap learning via modular concept discovery. In *Proceedings of the International Joint Conference on Artificial Intelligence*. AAAI Press/International Joint Conferences on Artificial Intelligence.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. (2017). Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR.

Driess, D., Xia, F., Sajjadi, M. S. M., Lynch, C., Chowdhery, A., Ichter, B., Wahid, A., Tompson, J., Vuong, Q., Yu, T., Huang, W., Chebotar, Y., Sermanet, P., Duckworth, D., Levine, S., Vanhoucke, V., Hausman, K., Toussaint, M., Greff, K., Zeng, A., Mordatch, I., and Florence, P. (2023). PaLM-E: An Embodied Multimodal Language Model.

Dryden, J. et al. (1859). *Plutarch's Lives: Life of Theseus*, volume 2. S. Low.

Eisenstein, J. (2018). Natural language processing.

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. (2018). Learning to infer graphics programs from hand-drawn images. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31.

Ellis, K., Solar-Lezama, A., and Tenenbaum, J. B. (2015). Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 973–981.

Ellis, K., Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. (2021). 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.

Evans, J. S. (2003). In two minds: dual-process accounts of reasoning. *Trends in Cognitive Sciences*, 7(10):454–459.

Fan, A., Gardent, C., Braud, C., and Bordes, A. (2021). Augmenting transformers with knn-based composite memory for dialog. *Transactions of the Association for Computational Linguistics*, 9:82–99.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. (2022). Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Fried, D., Andreas, J., and Klein, D. (2018). Unified pragmatic models for generating and following instructions. 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 1951–1963, New Orleans, Louisiana. Association for Computational Linguistics.

Gershman, S. and Goodman, N. (2014). Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, volume 36.

Gershman, S. J., Horvitz, E. J., and Tenenbaum, J. B. (2015). Computational rationality: A converging paradigm for intelligence in brains, minds, and machines. *Science*, 349(6245):273–278.

Girdhar, R., El-Nouby, A., Liu, Z., Singh, M., Alwala, K. V., Joulin, A., and Misra, I. (2023). ImageBind: One Embedding Space To Bind Them All.

Gleitman, L. (1990). The structural sources of verb meanings. *Language acquisition*, 1(1):3–55.

Gothoskar, N., Cusumano-Towner, M., Zinberg, B., Ghavamizadeh, M., Pollok, F., Garrett, A., Tenenbaum, J., Gutfreund, D., and Mansinghka, V. (2021). 3dp3: 3d scene perception via probabilistic programming. *Advances in Neural Information Processing Systems*, 34:9600–9612.

Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. (2015). Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99.

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

Guu, K., Lee, K., Tung, Z., Pasupat, P., and Chang, M. (2020). Retrieval augmented language model pre-training. In *International Conference on Machine Learning*, pages 3929–3938. PMLR.

Hahn, M. (2020). Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171.

Haluptzok, P., Bowers, M., and Kalai, A. T. (2022). Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. (2021). Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. M. (1995). The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161.

Hobbes, T. (1656). *Elements of Philosophy the First Section, Concerning Body*. Printed by R. & W. Leybourn for Andrew Crooke.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

Hu, R., Andreas, J., Rohrbach, M., Darrell, T., and Saenko, K. (2017). Learning to Reason: End-to-End Module Networks for Visual Question Answering. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 804–813, Venice. IEEE.

Huang, S., Dong, L., Wang, W., Hao, Y., Singhal, S., Ma, S., Lv, T., Cui, L., Mohammed, O. K., Patra, B., Liu, Q., Aggarwal, K., Chi, Z., Bjorck, J., Chaudhary, V., Som, S., Song, X., and Wei, F. (2023). Language Is Not All You Need: Aligning Perception with Language Models.

Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., Zeng, A., Tompson, J., Mordatch, I., Chebotar, Y., et al. (2022). Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*.

Izacard, G., Lewis, P., Lomeli, M., Hosseini, L., Petroni, F., Schick, T., Yu, J. A., Joulin, A., Riedel, S., and Grave, E. (2022). Few-shot Learning with Retrieval Augmented Language Models. *undefined*.

Johnson, J., Hariharan, B., Van Der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., and Girshick, R. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2901–2910.

Johnson, M. (1998). Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632.

Kahneman, D. (2011). *Thinking, fast and slow*. macmillan.

Karpas, E., Abend, O., Belinkov, Y., Lenz, B., Lieber, O., Ratner, N., Shoham, Y., Bata, H., Levine, Y., Leyton-Brown, K., Muhlgay, D., Rozen, N., Schwartz, E., Shachaf, G., Shalev-Shwartz, S., Shashua, A., and Tenenholtz, M. (2022). MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning.

Kersten, D., Mamassian, P., and Yuille, A. (2004). Object perception as bayesian inference. *Annu. Rev. Psychol.*, 55:271–304.

Kersten, D. K. D. and Yuille, A. (1996). Introduction: A bayesian formulation of visual perception. *Perception as Bayesian inference*, pages 1–21.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large Language Models are Zero-Shot Reasoners.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. (2019). Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Lachaux, M.-A., Roziere, B., Szafraniec, M., and Lample, G. (2021). DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *Advances in Neural Information Processing Systems*, volume 34, pages 14967–14979. Curran Associates, Inc.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338.

Lau, T. A. and Weld, D. S. (1998). 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. ACM.

Lau, T. A. and Weld, D. S. (1999). Programming by demonstration: An inductive learning formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, IUI '99, pages 145–152, New York, NY, USA. ACM.

Lee, T. S. and Mumford, D. (2003). Hierarchical bayesian inference in the visual cortex. *JOSA A*, 20(7):1434–1448.

Lewis, M., Yarats, D., Dauphin, Y. N., Parikh, D., and Batra, D. (2017). Deal or no deal? end-to-end learning for negotiation dialogues. *arXiv preprint arXiv:1706.05125*.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. (2022). Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Liang, P., Jordan, M. I., and Klein, D. (2010). Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646.

Lieder, F. and Griffiths, T. L. (2020). Resource-rational analysis: Understanding human cognition as the optimal use of limited computational resources. *Behavioral and brain sciences*, 43:e1.

Lions, J. (1977). A commentary on the sixth edition unix operating system. *Department of Computer Science, The University of New South Wales*.

Liu, R., Wei, J., Gu, S. S., Wu, T.-Y., Vosoughi, S., Cui, C., Zhou, D., and Dai, A. M. (2022). Mind's eye: Grounded language model reasoning through simulation. *arXiv preprint arXiv:2210.05359*.

Martins, P. H., Marinho, Z., and Martins, A. F. T. (2022). ∞-former: Infinite memory transformer. In *ACL*.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., Jiang, X., Cobbe, K., Eloundou, T., Krueger, G., Button, K., Knight, M., Chess, B., and Schulman, J. (2022). WebGPT: Browser-assisted question-answering with human feedback.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. (2022). A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. (2019). Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR.

Nye, M., Tessler, M., Tenenbaum, J., and Lake, B. M. (2021). Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. *Advances in Neural Information Processing Systems*, 34:25192–25204.

OpenAI (2023). GPT-4 Technical Report.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback.

Paranjape, A. and Manning, C. D. (2021). Human-like informative conversations: Better acknowledgements using conditional mutual information. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 768–781.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. (2016). Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Poesia, G., Polozov, O., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. (2022). Synchromesh: Reliable code generation from pre-trained language models.

Polikarpova, N., Kuraj, I., and Solar-Lezama, A. (2016). Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538.

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

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

Rae, J. W., Potapenko, A., Jayakumar, S. M., and Lillicrap, T. P. (2020). Compressive transformers for long-range sequence modelling. *ArXiv*, abs/1911.05507.

Rule, J. S., Tenenbaum, J. B., and Piantadosi, S. T. (2020). The child as hacker. *Trends in cognitive sciences*, 24(11):900–915.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Sharma, P., Torralba, A., and Andreas, J. (2021). Skill induction and planning with latent language. *arXiv preprint arXiv:2110.01517*.

Shin, R., Allamanis, M., Brockschmidt, M., and Polozov, A. (2019). Program synthesis and semantic parsing with learned code idioms. In *NeurIPS 2019*.

Singh, R. and Gulwani, S. (2012). Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751.

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

Solar-Lezama, A. (2008). *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley.

Solar-Lezama, A. (2013). Program sketching. *STTT*, 15(5-6):475–495.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. (2006). Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415.

Srikant, S., Liu, S., Mitrovska, T., Chang, S., Fan, Q., Zhang, G., and O'Reilly, U.-M. (2021). Generating Adversarial Computer Programs using Optimized Obfuscations.

Stanovich, K. E. (1999). *Who is rational?: Studies of individual differences in reasoning*. Psychology Press.

Summers, P. D. (1976). A methodology for lisp program construction from examples. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 68–76, New York, NY, USA. ACM.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., and Hashimoto, T. B. (2023). Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., and Metzler, D. (2021). Long range arena: A benchmark for efficient transformers. *ArXiv*, abs/2011.04006.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. (2022). Efficient transformers: A survey. *ACM Computing Surveys (CSUR)*.

Tenney, I., Das, D., and Pavlick, E. (2019). Bert rediscovers the classical nlp pipeline. *arXiv preprint arXiv:1905.05950*.

Theis, T. N. and Wong, H.-S. P. (2017). The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50.

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., Li, Y., Lee, H., Zheng, H. S., Ghafouri, A., Menegali, M., Huang, Y., Krikun, M., Lepikhin, D., Qin, J., Chen, D., Xu, Y., Chen, Z., Roberts, A., Bosma, M., Zhao, V., Zhou, Y., Chang, C.-C., Krivokon, I., Rusch, W., Pickett, M., Srinivasan, P., Man, L., Meier-Hellstern, K., Morris, M. R., Doshi, T., Santos, R. D., Duke, T., Soraker, J., Zevenbergen, B., Prabhakaran, V., Diaz, M., Hutchinson, B., Olson, K., Molina, A., Hoffman-John, E., Lee, J., Aroyo, L., Rajakumar, R., Butryna, A., Lamm, M., Kuzmina, V., Fenton, J., Cohen, A., Bernstein, R., Kurzweil, R., Aguera-Arcas, B., Cui, C., Croak, M., Chi, E., and Le, Q. (2022). LaMDA: Language Models for Dialog Applications.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. (2022). Finetuned Language Models Are Zero-Shot Learners.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2023). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.

Weir, N. and Van Durme, B. (2022). Dynamic Generation of Interpretable Inference Rules in a Neuro-Symbolic Expert System.

Wilcox, E., Levy, R., and Futrell, R. (2019). Hierarchical representation in neural language models: Suppression and recovery of expectations. *arXiv preprint arXiv:1906.04068*.

Winston, P. H. (1970). Learning structural descriptions from examples. Technical report, Cambridge, MA, USA.

Wong, C., Ellis, K. M., Tenenbaum, J., and Andreas, J. (2021). Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pages 11193–11204. PMLR.

Wong, C., McCarthy, W., Grand, G., Andreas, J., Tenenbaum, J. B., Hawkins, R., and Fan, J. (2022). Identifying concept libraries from language about object structure. In *Proceedings of the 44th Annual Meeting of the Cognitive Science Society*.

Wu, J., Tenenbaum, J. B., and Kohli, P. (2017). Neural scene de-rendering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 699–707.

Wu, J., Yildirim, I., Lim, J. J., Freeman, B., and Tenenbaum, J. (2015). Galileo: Perceiving Physical Object Properties by Integrating a Physics Engine with Deep Learning. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Wu, Y., Rabe, M. N., Hutchins, D. S., and Szegedy, C. (2022). Memorizing transformers. *ArXiv*, abs/2203.08913.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. B. (2018). Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *arXiv preprint arXiv:1810.02338*, 31:1031–1042.

Yildirim, I., Belledonne, M., Freiwald, W., and Tenenbaum, J. (2020). Efficient inverse graphics in biological face processing. *Science Advances*, 6(10):eaax5979.

Yuille, A. and Kersten, D. (2006). Vision as bayesian inference: analysis by synthesis? *Trends in cognitive sciences*, 10(7):301–308.

Zellers, R., Lu, X., Hessel, J., Yu, Y., Park, J. S., Cao, J., Farhadi, A., and Choi, Y. (2021). Merlot: Multimodal neural script knowledge models. *Advances in Neural Information Processing Systems*, 34.

Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., and Zhang, L. (2022). 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. ACM.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. (2022). Opt: Open pre-trained transformer language models.

Appendix

A.1 Graphical maps of learned libraries

We generated graphical visualizations of the libraries learned by the best LILO model for each domain (`LILO (+ Search)`; c.f., § 4.2.1). 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. Here, we include graphical maps for the REGEX and LOGO domains; the map for the CLEVR library is provided in Fig. 4-3.

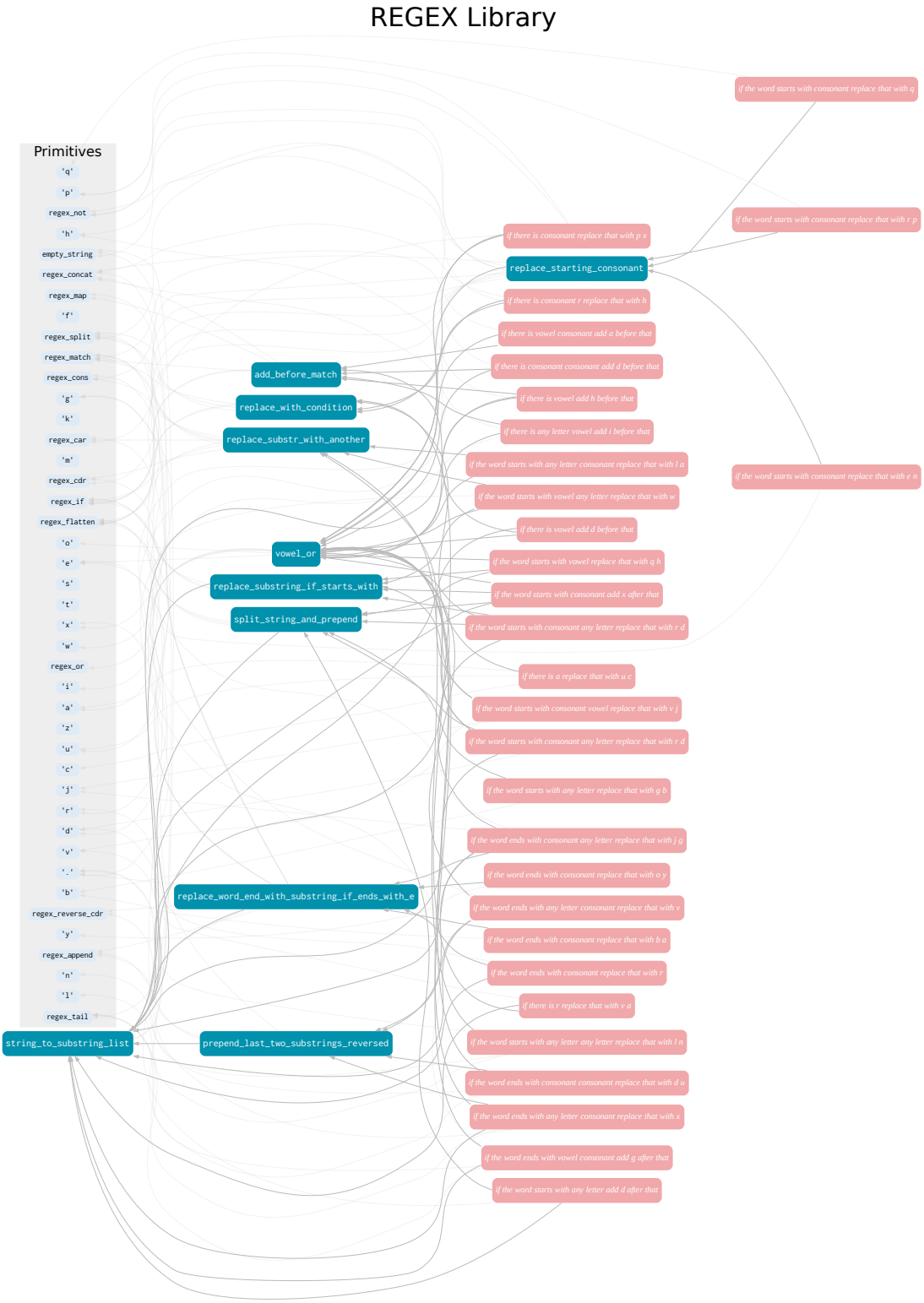

Figure A-1: **Graphical map of REGEX library learned by Lilo.** Named abstractions (turquoise) are hierarchically composed of other abstractions and ground out in the base DSL primitives (gray box).

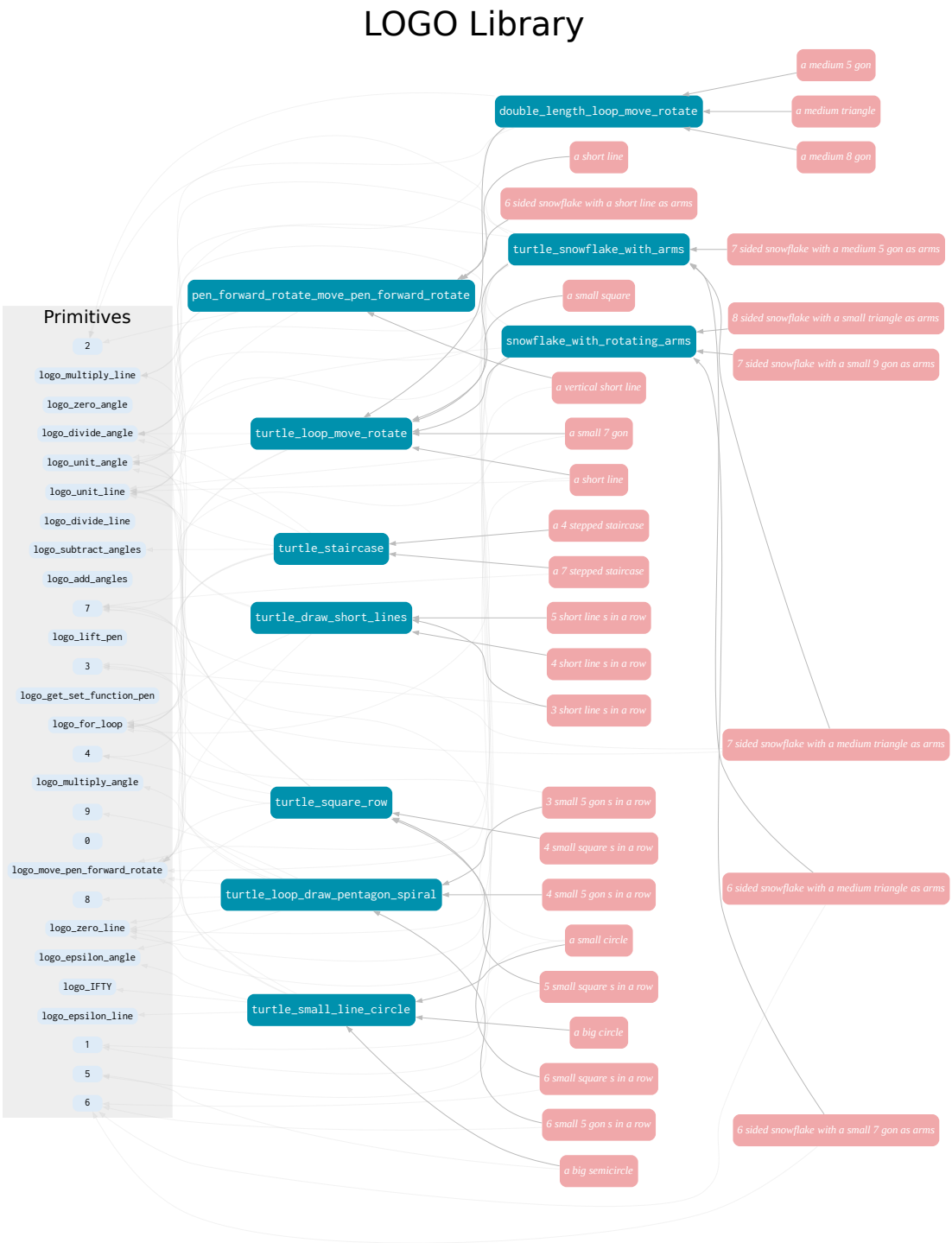

Figure A-2: **Graphical map of REGEX library learned by Lilo.** Named abstractions (turquoise) are hierarchically composed of other abstractions and ground out in the base DSL primitives (gray box).

A.2 Auto-documentation of learned libraries

A.2.1 Library for REGEX

```
(fn_42) vowel_or :: tsubstr
(regex_or 'a' (regex_or 'e' (regex_or 'i' (regex_or 'o' 'u'))))
{- Returns a regular expression that matches any vowel character ('a', 'e', 'i', 'o',
or 'u') using 'regex_or' function. Used for identifying vowels in strings. -}

{- Example usages -}
--if the word ends with vowel consonant add g after that
(λ (regex_if (regex_match vowel_or (regex_tail (regex_reverse_cdr
(string_to_substring_list $0)))) (regex_flatten (regex_append 'g'
(string_to_substring_list $0))) $0))
--if there is consonant consonant add d before that
(λ (add_before_match (regex_split (regex_concat (regex_not vowel_or) (regex_not
vowel_or)) $0) 'd' (regex_concat (regex_not vowel_or) (regex_not vowel_or))))
--if there is vowel add h before that
(λ (add_before_match (string_to_substring_list $0) 'h' vowel_or))

(fn_43) replace_with_condition :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_flatten (regex_map (λ (regex_if (regex_match $2 $0) $1 $0))
(regex_split $1 $2))))))
{- Given a full string $0 and two substrings $1 and $2, returns a full string where
all occurrences of $1 in $0 are replaced with $2 if the condition in the regular
expression created by splitting $0 by $1 and applying a lambda function that returns
$1 if the split element matches $2 and $0 otherwise is satisfied. -}

{- Example usages -}
--if there is a replace that with u c
(λ (replace_with_condition $0 'a' (regex_concat 'u' 'c')))
--if there is consonant r replace that with h
(λ (replace_with_condition $0 (regex_concat (regex_not vowel_or) 'r') 'h'))
--if there is consonant replace that with p x
(λ (replace_with_condition $0 (regex_not vowel_or) (regex_concat 'p' 'x')))

(fn_44) replace_substr_with_another :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_if (regex_match $0 (regex_car (regex_cdr (regex_split empty_string
$2)))) (regex_flatten (regex_cons $1 (regex_cdr (regex_cdr (regex_split empty_string
$2))))) $2))))
{- Given a full string $0 and two substrings $1 and $2, returns a full string where
all occurrences of $1 in $0 are replaced with $2 if $1 appears as a word. The match is
performed using regular expressions. -}

{- Example usages -}
--if the word starts with any letter any letter replace that with l n
(λ (replace_substr_with_another $0 (regex_concat 'l' 'n') '.'))
--if the word starts with consonant vowel replace that with v j
(λ (replace_substr_with_another $0 (regex_concat 'v' 'j') vowel_or))
--if the word starts with any letter consonant replace that with l a
```

```
(λ (replace_substr_with_another $0 (regex_concat 'l' 'a') (regex_not vowel_or)))

(fn_45) replace_starting_consonant :: tfullstr -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_if (regex_match (regex_not vowel_or) (regex_car (regex_split
empty_string $2))) (regex_flatten (regex_cons $0 (regex_cons $1 (regex_cdr
(regex_split empty_string $2))))) $2))))
{- Given a full string $0 and two substrings $1 and $2, returns a full string where
the first letter of each word starting with a consonant in $0 is replaced with $1$2.
Vowels are determined using `$vowel_or` regular expression. -}

{- Example usages -}
--if the word starts with consonant replace that with r p
(λ (replace_starting_consonant $0 'p' 'r'))
--if the word starts with consonant replace that with q
(λ (replace_starting_consonant $0 'q' empty_string))
--if the word starts with consonant replace that with e n
(λ (replace_starting_consonant $0 'n' 'e'))

(fn_46) string_to_substring_list :: tfullstr -> list(tsubstr)
(λ (regex_split empty_string $0))
{- Given a full string, returns a list of substrings where each substring is a word in
the original string. -}

{- Example usages -}
--if there is r replace that with v a
(λ (replace_with_condition (regex_flatten (string_to_substring_list $0)) 'r'
(regex_concat 'v' 'a')))
--if the word ends with consonant replace that with r
(λ (replace_word_end_with_substring_if_ends_with_e (regex_flatten
(string_to_substring_list $0)) empty_string 'r'))
--if the word starts with any letter add d after that
(λ (split_string_and_prepend $0 'd' (regex_car (string_to_substring_list $0))))

(fn_47) replace_word_end_with_substring_if_ends_with_e :: tfullstr -> tsubstr ->
tsubstr -> tfullstr
(λ (λ (λ (regex_if (regex_match 'e' (regex_tail (string_to_substring_list $2))) $2
(regex_flatten (regex_append (regex_concat $0 $1) (regex_reverse_cdr (regex_split '.'
$2)))))))
{- Given a full string $0, a substring $1 and another substring $2, returns a new full
string where the last character of each word in $0 is replaced with $2 if the word
ends with the character 'e' or concatenated with $1 if it doesn't. -}

{- Example usages -}
--if the word ends with consonant replace that with o y
(λ (replace_word_end_with_substring_if_ends_with_e $0 'y' 'o'))
--if the word ends with consonant replace that with b a
(λ (replace_word_end_with_substring_if_ends_with_e $0 'a' 'b'))
--if the word ends with consonant any letter replace that with j g
(λ (regex_if (regex_match vowel_or (regex_tail (regex_reverse_cdr
(string_to_substring_list $0)))) $0 (replace_word_end_with_substring_if_ends_with_e
(prepend_last_two_substrings_reversed $0 'j') 'g' 'j')))

(fn_48) split_string_and_prepend :: tfullstr -> tsubstr -> tsubstr -> tfullstr
```

```
(λ (λ (λ (regex_flatten (regex_cons $0 (regex_cons $1 (regex_cdr
(string_to_substring_list $2))))))))
{- Given a full string $0 and two substrings $1 and $2, splits $0 into a list of
substrings and prepends $1 and $2 to the first two substrings respectively, then
flattens the resulting list into a new full string. -}

{- Example usages -}
--if the word starts with any letter replace that with g b
(λ (split_string_and_prepend $0 'b' 'g'))
--if the word starts with vowel replace that with q h
(λ (replace_substring_if_starts_with $0 (split_string_and_prepend $0 'h' 'q')
vowel_or))
--if the word starts with consonant any letter replace that with r d
(λ (replace_substring_if_starts_with $0 (split_string_and_prepend (regex_flatten
(regex_cdr (string_to_substring_list $0))) 'd' 'r') (regex_not vowel_or)))

(fn_49) add_before_match :: list(tsubstr) -> tsubstr -> tsubstr -> tfullstr
(λ (λ (λ (regex_flatten (regex_map (λ (regex_if (regex_match $1 $0) (regex_concat $2
$0) $0)) $2)))))
{- Given a list of substrings $0 and two substrings $1 and $2, returns a new full
string where $1 is added before any substring in $0 that matches $2. The match is
performed using regular expressions with 'regex_match', and the new substring is
obtained by concatenating $1 and the matched substring with 'regex_concat' -}

{- Example usages -}
--if there is vowel consonant add a before that
(λ (add_before_match (regex_split (regex_concat vowel_or (regex_not vowel_or)) $0)
'a' (regex_concat vowel_or (regex_not vowel_or))))
--if there is any letter vowel add i before that
(λ (add_before_match (regex_split (regex_concat '.' vowel_or) $0) 'i' (regex_concat
'.' vowel_or)))
--if there is vowel add d before that
(λ (add_before_match (string_to_substring_list $0) 'd' vowel_or))

(fn_50) replace_substring_if_starts_with :: tfullstr -> tfullstr -> tsubstr ->
tfullstr
(λ (λ (λ (regex_if (regex_match $0 (regex_car (string_to_substring_list $2))) $1
$2))))
{- Given a full string $0, a replacement string $1, and a regular expression substring
$2, returns a new full string where any substring in $0 that matches $2 is replaced
with $1 at the beginning of the substring. -}

{- Example usages -}
--if the word starts with consonant add x after that
(λ (replace_substring_if_starts_with (split_string_and_prepend $0 'x' (regex_car
(string_to_substring_list $0))) $0 vowel_or))
--if the word starts with consonant any letter replace that with r d
(λ (replace_substring_if_starts_with $0 (split_string_and_prepend (regex_flatten
(regex_cdr (string_to_substring_list $0))) 'd' 'r') (regex_not vowel_or)))
--if the word starts with vowel any letter replace that with w
(λ (replace_substring_if_starts_with $0 (replace_substr_with_another $0 'w' '.')
vowel_or))
```

```
(fn_51) prepend_last_two_substrings_reversed :: tfullstr -> tsubstr -> tfullstr
(λ (λ (regex_flatten (regex_append $0 (regex_reverse_cdr (regex_reverse_cdr
(string_to_substring_list $1)))))))
{- Given a full string $0 and a substring $1, returns a full string where the last two
substrings in the string (if they exist) are reversed and prepended to the original
string. The string is split into substrings using 'string_to_substring_list' function.
-}

{- Example usages -}
--if the word ends with any letter consonant replace that with x
(λ (regex_if (regex_match vowel_or (regex_tail (string_to_substring_list $0))) $0
(prepend_last_two_substrings_reversed $0 'x')))
--if the word ends with consonant consonant replace that with d u
(λ (regex_if (regex_match '.' (regex_tail (regex_split vowel_or $0))) $0
(prepend_last_two_substrings_reversed $0 (regex_concat 'd' 'u'))))
--if the word ends with any letter consonant replace that with v
(λ (regex_if (regex_match (regex_not vowel_or) (regex_tail (string_to_substring_list
$0))) (prepend_last_two_substrings_reversed $0 'v') $0))
```

A.2.2 Library for CLEVR

```
(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. -}

{- Example usages -}

(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. -}
```

```
{- Example usages -}

(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)))))))
```

A.2.3 Library for LOGO

```
(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))
```

A.3 Auto-documentation prompt

To encourage reputability, 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
}
```

In practice, we find that OpenAI's instruction-tuned ChatGPT models adhered to this JSON specification 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 §4.

A.4 Results from LAPS experiments

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 A.1: **Percent held-out test-tasks solved for LAPS.** textitBest reports the best model across replications; *Mean* averages across replications. (Reproduced from Wong et al., 2021.)

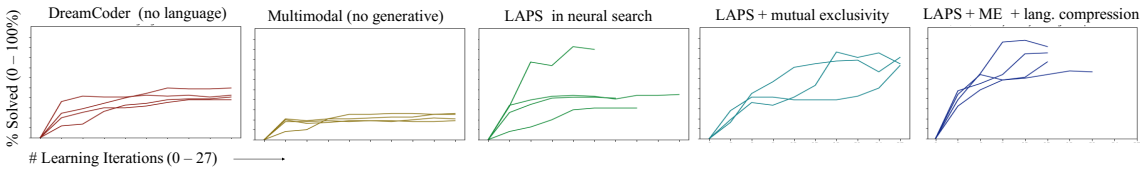

Figure A-3: Learning curves comparing baselines and LAPS models in Table A.1, showing % heldout tasks solved on the graphics domain over random training task orderings. (Reproduced from Wong et al., 2021.)

A.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

Base DSL prompts: 50% (LILO + Hybrid DSL only)

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

Some of the graphical assets that appear in this work were generated by Midjourney, a third-party service provider. Subject to the Midjourney Terms of Service for Paid Users, all assets created with the services are owned by the author to the extent possible under current law. The author acknowledges and respects the copyrights and trademarks held by the Walt Disney Company. Any likeness to characters or properties trademarked by Disney is considered Fair Use under US Transformative Use laws, which provide broad protections for commentary, criticism, parody, satire, education, research, and other forms of creative expression.