The child as hacker: Building more human-like models of learning

by

Joshua S. Rule

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Cognitive Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Brain and Cognitive Sciences
September 5, 2020

Certified by .....
Joshua B. Tenenbaum
Professor of Computational Cognitive Science
Thesis Supervisor

Accepted by .....
Rebecca Saxe
John W. Jarve (1978) Professor of Brain and Cognitive Sciences
Associate Head, Department of Brain and Cognitive Sciences
Affiliate, McGovern Institute for Brain Science
Chairman, Department Committee on Graduate Theses

The child as hacker: Building more human-like models of learning

by

Joshua S. Rule

Submitted to the Department of Brain and Cognitive Sciences
on September 5, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Cognitive Science

Abstract

Cognitive science faces a radical challenge in explaining the richness of human learning and cognitive development. This thesis proposes that developmental theories can address the challenge by adopting perspectives from computer science. Many of our best models treat learning as analogous to computer programming because symbolic programs provide the most compelling account of sophisticated mental representations. We specifically propose that learning from childhood onward is analogous to a style of programming called hacking—making code better along many dimensions through an open-ended and internally-motivated set of diverse values and activities. This thesis also develops a first attempt to formalize and assess the child as hacker view through an in-depth empirical study of human and machine concept learning. It introduces list functions as a domain for psychological investigation, demonstrating how they subsume many classic concept learning tasks while opening new avenues for exploring algorithmic thinking over complex structures. It also presents HL, a computational learning model whose representations, objectives, and mechanisms reflect core principles of hacking. Existing work on concept learning shows that learners both prefer simple explanations of data and find them easier to learn than complex ones. The child as hacker, by contrast, suggests that learners use mechanisms that dissociate hypothesis complexity and learning difficulty for certain problem classes. We thus conduct a large-scale experiment exploring list functions that vary widely in difficulty and algorithmic content to help identify structural sources of learning difficulty. We find that while description length alone predicts learning, predictions are much better when accounting for concepts' semantic features. These include the use of internal arguments, counting knowledge, case-based and recursive reasoning, and visibility—a measure we introduce to modify description length based on the complexity of inferring each symbol in a description. We further show that HL's hacker-like design uses these semantic features to better predict human performance than several alternative models of learning as programming. These results lay groundwork for a new generation of computational models and demonstrate how the child as hacker hypothesis can productively contribute to our understanding of learning.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor of Computational Cognitive Science

Acknowledgments

First thanks go to Josh Tenenbaum and Steve Piantadosi. Chapters 1–2 contain material taken from a paper we wrote together (Rule et al., in press), but every page profits from their influence. They are the reason the rest of this thesis uses *we* rather than *I*. They not only contributed their effort and ideas, but they taught me how to contribute my own. Who I am as a scientist deeply reflects what I have learned from them. Thank you both for believing in the importance of our work and for guiding me to make it far better than it would have been otherwise. Josh, thank you for teaching me to use nagging doubts as stepping stones to new insights and for being the first to invite me into the awe-inspiring and life-changing community that is MIT. Steve, thank you for teaching me to try the simplest possible thing first, ideally in the next half-hour, and for repeatedly helping me distill the sea of my confusion into a clear hypothesis.

I cannot imagine better people to shape this work than my thesis committee. Susan Carey, you transformed my thinking about the depths to be mined in understanding the young mind. Thank you for teaching me that what is actual is possible, that computational models must make contact with empirical facts (the more, the better), and for demonstrating how a deep understanding of learning builds on all the fields I love most: mathematics, psychology, philosophy, and computer science. Laura Schulz, thank you for challenging me to take seriously the richness, inventiveness, and raw power of children's learning and for telling me flat out that not every contribution I made had to come in the form of a model. I needed to hear that.

I am also immensely grateful to the broader intellectual community that has introduced me to cognitive science. Cocosci and Colala have simultaneously sharpened and broadened my thinking about the mind. Special thanks to the program inductors—especially Eyal Dechter, Eric Schulz, Lucas Morales, Kevin Ellis, Max Nye, Zenna Taveres, Luc Cary, and Andrés Campero. I am also grateful for many good conversations with Liz Spelke, Tomer Ullman, John McCoy, Andrew Cropper, Josh Hartshorne, Max Kleiman-Weiner, Tim O'Donnell, and Brenden Lake. My time in BCS owes much to the dedicated work of Denise Heintze, Kris Kipp, Sierra Vallin, and Julianne Gale Ormerod and the generous funding of

the NSF Graduate Fellowship, the Stark Graduate Fellowship, and the Leventhal Fellowship.

Many others have deeply shaped my own cognitive development, including: Bob Wengert, Gerald DeJong, Dan Roth, Paul Drelles, Gordon Brown, Keith Hamilton, Bob Parker, and Ted Malt. Special thanks to Max Riesenhuber for his role in my decision to pursue academic research, his support and friendship over the years, and his challenging me to apply to MIT in the first place. Michel Selva, thank you for wise advice and friendly counsel.

I am nearly overwhelmed when I think of the debt of gratitude I owe to my dearest friends. Thanks to the Champaign crew—Alyssa Combs, Micah & Priscilla Putman, Josh & Julie Birky, and Josh & Jess Kober—your friendship has often been felt deepest when needed most. To Aidan & Sarah McCarthy, Austin & Becca Ward, Caleb & Lauren Hintz, Bre & Wes Rieth, Matt & Lynnae Terrill, Kevin & Kelsey McKenzie, and Meg & Adam Norris: thank you for loving me well and for making Aletheia Church my spiritual family in Cambridge. Donny Fisher, thank you for keeping me sane, too, through the many miles we have walked together. I hope there are many more.

My family has been a constant and inexhaustible source of support. Adam, I am so glad we share the field of cognitive science, as well as a love of walking wild places fast and far. Oresta, thank you for catching my throw-away jokes on family calls and ensuring I had sufficient ice cream to study productively. AJ & Ariel were a constant source of encouragement during our time in Boston. Shelley, Daniel, Kim, & Matt have loved me and my family even as my own attention has been consistently devoted to model runs, writing, and analysis. Mom, Dad, Jim, & Laura: I would not be writing these words without your help. Mom & Dad, I thank you especially for your support, prayers, and willingness to do anything that might help me. Ana and Forest, you are by far the most successful models of intelligence I have collaborated on to date. You bring me so much joy. Thank you for teaching me how little I really understand about the mind. Amy, I cannot even begin to describe how grateful I am for you or this thesis would be at least twice as long as it is. You have helped me to be more than I could be on my own and to do what, at times, I did not think possible. You are more than I have ever hoped for. You take my breath away.

Soli Deo gloria

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Human cognitive development is qualitatively unique. Though humans are born unusually helpless, they quickly transform a limited set of core cognitive abilities into an unparalleled cognitive repertoire including: intuitive theories for domains like physics and biology, formal theories in mathematics and science, language comprehension and production, complex perceptual and motor skills, the ability to introspect on their own emotional and cognitive processes, and the capacity for creative expression. People also learn to learn, producing higher-order knowledge that enriches existing concepts and enhances future learning. Human-like performance in any one of these domains seems substantially beyond current efforts in artificial intelligence. Yet, people essentially acquire these abilities simultaneously and universally. Given the right kinds of practice, humans can even develop world-class expertise and discover new ideas that radically alter humanity's understanding of the world. The foundational fields of cognitive science—including philosophy, psychology, neuroscience, and computer science—face a radical challenge in explaining the richness of human development.

Early discussions of cognition often polarized on the role of learning (Macnamara, 1999). Empiricists like Aristotle (*de Anima*), Locke (1690), and others argued that people are born with an empty conceptual repertoire and learn entirely from experience. Nativists like Plato (*Meno*, *Phaedo*, *Apology*) argued that true learning is impossible, a view prominently defended in recent times by Fodor (1980; cf. Carey, 2015). A long line of experimental work suggests both views are wrong. Humans instead start their lives with a small but

surprisingly rich body of conceptual and high-level perceptual capabilities collectively called *core cognition* or *core knowledge* (Barner & Baron, 2016; Carey, 2009; Spelke & Kinzler, 2007; Kinzler & Spelke, 2007). Core cognition supports reasoning about magnitudes, distances and geometries, objects and physical interactions, and agents and social interactions. While it is unclear how much of what has been attributed to core cognition is available at birth and how much develops quickly through experience, it is clear that the initial starting state interacts with the structure of the world in such a way that these core cognitive abilities reliably emerge over the first years of life. Still, the question remains: what exactly is learning?

To help answer this question, this thesis develops a framework for understanding learning and cognitive development through the lens of computer science. The resulting view has the potential to support a unified and computationally precise account of a broad variety of learning phenomena while simultaneously doing justice to the richness and variety of goals and processes which learners bring to bear during development. This is a tall order, and we fill it in the following way.

First, we introduce the *child as hacker* as a hypothesis about the representations, objectives, and processes of distinctively human-like learning. Like the *child as scientist* view (Piaget, 1955; Carey, 1985, 2009; Gopnik, 2012; Schulz, 2012b), the child as hacker is both a fertile metaphor and a source of concrete hypotheses about cognitive development. It also suggests a roadmap to what could be a unifying formal account of major phenomena in development. A key part of the child as hacker is the idea of *learning as programming*, which holds that symbolic programs—i.e. code—provide the best formal knowledge representation we have. Learning therefore becomes a process of creating new mental programs. The overall approach is a modern formulation of Fodor's (1975) language of thought (LOT). The rest of **Chapter 1** reviews support for learning as programming and argues that while on increasingly solid ground as a computational-level theory (Marr, 1982), it remains underspecified. **Chapter 2** extends the idea of learning as programming by drawing inspiration from *hacking*, an internally-driven approach to programming emphasizing the diverse goals and means humans use to make code better. Our core claim is that the specific representations, motivations, values, and techniques of hacking form a rich set of largely untested hypotheses about learning.

This rest of this thesis develops a first attempt to formalize and assess the child as hacker hypothesis through an in-depth empirical study of human and machine concept learning. While we are deeply inspired specifically by children's learning, many of the issues at stake are hypothesized to be relevant across a learner's lifespan. Because it is often easier to collect rich data from adults than from children, the focus of this study is adult concept learning. The findings are likely to apply to child learners, too, but we mark specifically developmental applications as future directions.

In **Chapter 3**, we present list functions as a domain for studying human concept learning. The premise of the domain is simple: concepts consist of computable functions from lists to lists. We focus specifically on concepts concerning lists of natural numbers. This domain has a long history in artificial intelligence (Green et al., 1974; Shaw et al., 1975; Biermann, 1978; Green, 1981; Smith, 1984; Feser et al., 2015; Osera & Zdancewic, 2015; Polikarpova et al., 2016; Cropper et al., 2019) but is relatively unstudied in cognitive psychology. We show how it nonetheless combines the best features of many classic concept learning tasks. Inductive learning can be framed as inference in a hierarchical generative model, and this model supports a variety of related tasks. The domain is familiar to participants and allows them to bring rich background knowledge to bear during learning. It contains complex objects with intricate internal structure. Crucially, it contains natural numbers and makes use of them as symbolic, ordinal, and cardinal values. It supports algorithmically sophisticated concepts—under the right assumptions, the domain is computationally universal. Despite all this richness, lists and numbers are well-studied objects with simple formal descriptions and well-established libraries of related concepts. This makes it straightforward to construct model LOTs whose concepts vary smoothly from trivially easy to formidably difficult. We further show that list functions actually subsume many classic tasks and capture the dynamics of key developmental achievements while making it possible to explore richer kinds of algorithmic thinking over complex structures. We argue that its long computational history and potential as a domain for psychological investigation make it a prime candidate for investigating hacking as a metaphor for learning and for studying cognitive development more broadly.

Next, we introduce HL (Hacker-Like), a computational model of inductive concept learn-

ing which takes important steps toward applying the lessons of the child as hacker. **Chapter 4** describes the model in detail. Each aspects of the model's architecture—including its representations, objectives, and learning mechanisms—is inspired by a phenomenon in human learning interpreted through the lens of hacking. Briefly, HL models learning as the iterative development of an entire programming language, capturing the dynamics of developing a domain-specific language similar to the way that humans develop conceptual systems governed by inferential role semantics. It uses complex objective functions that vary based on the task at hand, favoring accuracy, well-formedness, discoverability, and simplicity. These objectives allow it to avoid premature optimization, exploring suboptimal, even wrong hypotheses during search, while favoring the best hypotheses during later decision-making. It also permits a sort of queried-guided search (Chu et al., 2019). Finally, by using a diverse set of structured learning mechanisms, HL reparameterizes search. Rather than searching directly for a specific language, it searches a space of meta-programs describing compact generative processes for creating programming languages, and is able to rapidly discover certain kinds of complex languages. It thus performs a sort of hypothesis-and-goal-driven search related to the iterative revision and refactoring practiced by hackers. These features are all illustrated using list functions.

There are strong normative arguments (Solomonoff, 1964a; Jefferys & Berger, 1992; Chater & Vitányi, 2003; Baum, 2004; Grünwald, 2007) suggesting that, all else being equal, people prefer a simple explanation of data over a complex one. Recent empirical work has also demonstrated that, in many cases, the difficulty of learning is strongly related to a formal measure of simplicity, such as the size of the hypothesis space or the description length of the concept in a domain-appropriate model LOT (Feldman, 2000; Tenenbaum, 1999, 2000; Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2012, 2016). The child as hacker, by contrast, emphasizes the richess of learning and predicts that learners make use of a diversity of structure-sensitive values and learning mechanisms, some of which may dissociate the complexity of learning a hypothesis from its description length in a target LOT. As a result, the complexity of learning may be more strongly related to other structural features of a concept than to its description length or the size of its hypothesis space.

To investigate these ideas empirically, **Chapter 5** introduces a benchmark set of 250

list functions and assesses human learning on each. We encode each task as a program in a rich, domain-appropriate model LOT. We analyze these programs for structural features describing a variety of syntactic and semantic properties. Among these are multiple measures of description length and multiple measures of visibility, a concept we introduce to describe how transparently each symbol in a program can be inferred from observed input/output pairs. Other measures relate to recursive, conditional, and numerical reasoning, including the use of counting knowledge. We then report a large scale online behavioral experiment testing human learning for the 250 concepts. We find that while our description length measures are predictive, the predictions becomes significantly more accurate when taking semantic features into account. Finally, we show that computational models of learning whose behavior is intrinsically connected to description length, specifically grammar-based enumeration and random sampling, do a much poorer job than HL, a model sensitive to semantic structure, in describing human performance across the entire dataset. This work suggests that humans—unlike simple learning algorithms based on sampling or exhaustive enumeration—are strongly sensitive to structural features in data that can be used to infer semantic content. They may be able to navigate their hypothesis spaces using learning mechanisms which dissociate the complexity of the concept from the complexity of search. As a result, they are able to quickly learn complex concepts, finding the simplest explanation of the data in a way that can be largely disconnected from the complexity of the explanation itself. While humans strongly prefer simple explanations, and while simplicity is sometimes the primary predictor of learning difficulty, there is often much more to learning than simplicity alone.

Chapter 6 investigates these ideas computationally. It compares HL with several benchmark program-induction-based models of concept learning in terms of their ability to predict human learning performance for a 100-concept subset of the 250 list functions. These concepts were selected to represent a broad spectrum of the features analyzed in Chapter 5 while remaining tractable for current learning models. HL implements the idea of learning through a diversity of structure-sensitive learning mechanisms. The remaining models each exemplify paradigmatic approaches to learning as programming: exhaustive search, stochastic search, proof-guided search, and neural program synthesis. HL does a better job of explaining participants' patterns of success and failures across these tasks than the competing models and

captures important aspects of human learning currently absent in the other models. HL does not close the gap entirely, so **Chapter 7** describes a series of critical next steps for hacker-like models of learning as well as the child as hacker hypothesis more broadly.

1.1 Three observations about learning

The puzzle of learning and cognitive development—how humans get so much from so little so quickly—lies at the heart of cognitive science. There are many kinds of knowledge and forms of learning, and the mind appears to be equipped with varying levels of domain-specific support for each (Carey, 2009; Fodor, 1983). Some conclude after a matter of seconds, leaving the sensation of blinding insight (Sternberg & Davidson, 1995), while others require arduous years of seemingly piecemeal effort (Davidson et al., 2012; Barner, 2017). No matter what is learned or over what timespan, however, resolving the puzzle requires describing four things: the state of a mind before learning, the state of that mind after learning, how the mind transitions between these states, and the objective which this transition serves.

Three general observations suggest a way to begin applying this framework. Consider the concept c_1. It represents a way of transforming a list of numbers into a single number, so read $c_1(x) = y$ as c_1 *transforms* x *into* y. We describe c_1 in more detail shortly, but take a moment now to determine what it does before continuing:

$$
\begin{aligned}
c_1([2,\ 5,\ 1,\ 5,\ 2,\ 6]) &= 1 \\
c_1([6]) &= 0 \\
c_1([5,\ 5,\ 5]) &= 2 \\
c_1([\ ]) &= 0 \\
c_1([3,\ 9,\ 5,\ 3,\ 3,\ 4,\ 3]) &= 3 \\
c_1([6,\ 2,\ 6,\ 2,\ 6,\ 2,\ 2]) &= 2
\end{aligned}
$$

c_1 might eventually be learnable, but it is difficult with so few examples and such little context. This highlights the first observation: there are concepts and conceptual systems which

humans can learn but only with difficulty. Category theory, particle physics, and gourmet cooking, for example, are each enormously complex systems of concepts that thousands of people have acquired, but only with extended effort. Now, consider the concepts c_2, c_3, and c_4 (which takes two arguments) below, and try to determine what they do:

$c_2([1,\ 2,\ 3]) = 1$	$c_3([5,\ 2,\ 6]) = [2,\ 6]$	$c_4(3\ \ [2,\ 4,\ 3]) = 1$
$c_2([0,\ 8,\ 6,\ 2]) = 0$	$c_3([3]) = [\ ]$	$c_4(0\ \ [1,\ 4,\ 0,\ 0]) = 2$
$c_2([6,\ 1]) = 6$	$c_3([1,\ 6,\ 4,\ 6]) = [6,\ 4,\ 6]$	$c_4(9\ \ [7,\ 2,\ 5,\ 8]) = 0$
$c_2([9,\ 2,\ 7]) = 9$	$c_3([3,\ 3,\ 0,\ 1,\ 9]) = [3,\ 0,\ 1,\ 9]$	$c_4(7\ \ [7,\ 3,\ 8,\ 7,\ 7]) = 3$

These three concepts lead to the second observation: some concepts are easy to learn. For many people, c_2, c_3, and c_4 are much easier to learn than c_1. c_2 takes a list and provides the first item in it, the item at the head of the list. c_3 takes a list and returns the same list after removing its head, what we might call the tail of the list. c_4 takes a number and a list and counts how many times the number appears in the list. These concepts may have been immediately obvious or required a moment of thought, but they built directly on things you probably already knew.

This leads to the third observation: it is much easier to learn difficult concepts given the right curriculum of antecedent concepts. Such a curriculum should teach a series of antecedents making the difficult concepts simpler to express but should also avoid concepts unnecessary for expressing the difficult concepts. The curriculum should be tailored to the conceptual system in question. Something as complex and abstract as category theory becomes much easier after learning about arithmetic, sets, and algebra, but is helped comparatively little by learning about north Indian geography or bicycle maintenance. c_2, c_3, and c_4 were easy only because most readers already have the antecedent concepts of lists and counting. A reader lacking them would probably need a curriculum teaching those things first (instead of mountaineering or accordion). Just as c_2–c_4 depend on counting and lists, so c_1 depends on c_2–c_4; they are a curriculum for c_1. Briefly reconsider the examples of c_1 on the previous page in light of them.

Together, these three concepts provide a compact way to explain c_1, which counts the number of times the head of a list appears in its tail:

$$c_1(xs) \equiv c_4(c_2(xs)\ \ c_3(xs)).$$

For example:

$$\begin{aligned} c_1([6,\ 8,\ 1,\ 6,\ 2,\ 4,\ 6]) &= c_4(c_2([6,\ 8,\ 1,\ 6,\ 2,\ 4,\ 6])\ \ c_3([6,\ 8,\ 1,\ 6,\ 2,\ 4,\ 6])) \\ &= c_4(6\ \ c_3([6,\ 8,\ 1,\ 6,\ 2,\ 4,\ 6])) \\ &= c_4(6\ \ [8,\ 1,\ 6,\ 2,\ 4,\ 6]) \\ &= 2 \end{aligned}$$

The overall intuition these observations give is that concepts are built compositionally from other concepts, and having the right concepts at the right time is what makes learning possible. On this view, the initial state is some initial set of concepts, and the final state is some other set of concepts. The mechanism of change iteratively invents and combines concepts according to some curriculum, evolving the concepts of the initial state toward those of the final state. The objective is uncovering a better description of the data, in all the richness of what we could mean by *better*. This may not be the only kind of learning, but it does seem to describe at least some of what they do.

Several other lines of reasoning support these intuitions and suggest that cognition occurs, or can be modeled as occurring, in something more structured than a chaotic soup of concepts: a mental language or language of thought (LOT; Turing, 1936; Fodor, 1975; Fodor & Pylyshyn, 1988; Fodor, 2008). Each concept is an expression in this language. Explaining learning and development then becomes a problem of specifying the trajectory of the LOT and any accompanying machinery from infancy onward. To avoid a form of radical nativism in which every possible concept is either known from birth or no more than a short search away (Fodor, 1975; Fodor, 1980), specifying change includes not only changes in how the LOT combines concepts for ad hoc use, but also in how the LOT itself changes (Gopnik, 1983; Carey, 1985, 2015).

Logic	first-order, modal, deontic logic
Mathematics	number systems, geometry, calculus
Natural language	morphology, syntax, number grammars
Sense data	audio, images, video, haptics
Computer languages	C, Lisp, Haskell, Prolog, LaTeX
Scientific theories	relativity, game theory, natural selection
Operating procedures	Robert's Rules, bylaws, checklists
Games & sports	Go, football, 8 queens, juggling, Lego
Norms & mores	class systems, social cliques, taboos
Legal codes	constitutions, contracts, tax law
Religious systems	monastic orders, vows, rites & rituals
Kinship	genealogies, clans/moieties, family trees
Mundane chores	knotting ties, making beds, mowing lawns
Intuitive theories	physics, biology, theory of mind
Domain theories	cooking, lockpicking, architecture
Art	music, dance, origami, color spaces

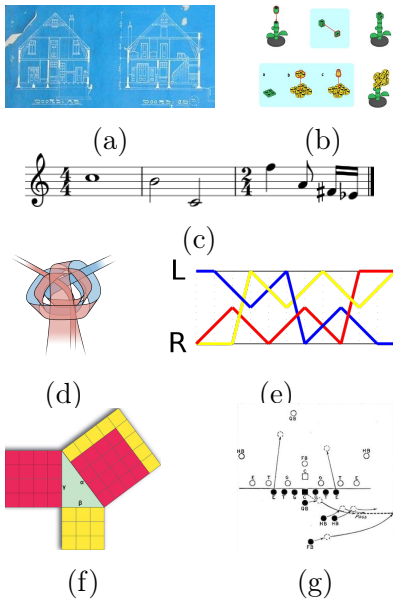


Table 1.1 & Figure 1-2: Table: A sampling of domains requiring algorithmic knowledge formalizable as programs, with motivating examples. Figure: Several classes of programs expressed as symbolic images: (a) blueprints; (b) assembly instructions; (c) musical notation; (d) knotting diagrams (e) juggling patterns; (f) graphical proofs; and (g) football plays.

1.2 Knowledge as code

To explain how the LOT changes requires describing how the LOT is represented, how learning proposes changes to the LOT, and on what basis proposals are accepted or rejected. A critical mass of work throughout cognitive science has converged on the hypothesis that human learning operates over structured, probabilistic, program-like representations (Chater & Oaksford, 2013; Zylberberg et al., 2011; Calvo & Symons, 2014; Lake et al., 2017; Goodman et al., 2015; Piantadosi & Jacobs, 2016; Goodman, Tenenbaum, Feldman, et al., 2008; Depeweg et al., 2018; Rothe et al., 2017; Erdogan et al., 2015; Yildirim & Jacobs, 2015; Amalric et al., 2017; Romano et al., 2018; Wang et al., 2019; cf. Lupyan & Bergen, 2016). This modern formulation treats the LOT as something like a computer programming language. Learning in the LOT consists of forming expressions to encode knowledge—for instance, composing computational primitives like CAUSE, GO, and UP to form LIFT (Siskind, 1996). This work argues that a compositional mental language is needed to explain systematicity, productivity, and compositionality in thought (Fodor, 1975; Fodor & Pylyshyn, 1988; Calvo & Symons, 2014); a probabilistic language capable of maintaining distributions over structures

is needed to explain variation and gradation in thinking (Goodman & Lassiter, 2015; Goodman, Mansinghka, et al., 2008; Lake et al., 2017; Goodman et al., 2015); and an expressive language—capable of essentially any computation—is needed to explain the scope of thought (Turing, 1936; Baum, 2004). Despite comparative and cross-cultural work seeking semantic primitives for mental languages (Wierzbicka, 1996), other work suggests that learners add and remove primitives, effectively building entirely new languages (Barner & Baron, 2016; Carey, 2009, 1985; Gopnik, 1983).

There are good reasons beyond empirical success to model mental representations as programs and the LOT as a programming language. Humans possess broad algorithmic knowledge, manipulating complex data in structured ways across many domains (Table 1.1). Symbolic programs—i.e. computer code—form a universal formal representation for algorithmic knowledge (Turing, 1936; Baum, 2004), and so may be the best model of mental representations currently available. While there have been many other proposals for modeling conceptual representations, only programs arguably capture the full breadth and depth of people's algorithmic abilities (Goodman et al., 2015). The rapid rise of programs as tools for manipulating information—from obviously symbolic domains like mathematics and logic to seemingly non-symbolic domains like video, audio, and neural processing—further identifies programs as a capable knowledge representation (e.g. Andreessen, 2011). Programs can also be communicated in many forms, including not only formal programming languages but a wide variety of forms familiar in all cultures, including natural language (e.g. *If you want to X, first you need to Y, then try to Z, but if that fails...*; Lupyan & Bergen, 2016) and symbolic images (Figure 1-1).

Code can model both procedural (Section 2.2) and declarative information (Section 2.3) and allow them to interact seamlessly (Figures 2-3 and 2-4). It operates at multiple levels, including: individual symbols, expressions, statements, data structures & functions, libraries, and even entire programming languages. Each level can interact with the others: libraries can embed one language inside another, statements can define data structures, and higher-order functions can take functions as arguments and return functions as outputs. This leads to one of the fundamental insights that makes code so successful. By writing computations as code, they become data that can be formally manipulated and analyzed (Abelson et al.,

1996). Programming languages thus become programs which take code as input and return code as output. Because all programming languages are programs, any knowledge that can be expressed in any program can be integrated into a single formal knowledge representation.

Programs are expressed according to a formal syntax, and their semantics specify computations by means of the relationships between programs according to some well-defined model, for example a Turing machine, the x86 processor, or the rules of lambda calculus. The meaning of a program is thus constrained by the set of relationships in which it participates, a form of procedural semantics (Johnson-Laird, 1977; Woods, 1981), also known as conceptual role semantics or inferential role semantics (Field, 1977; Harman, 1975, 1987, 1982; Loar, 1982; Block, 1987, 1997). Procedural semantics claims that programs get their meaning through their relations to other programs. In a simple arithmetic language, for example, the meaning of `two` is based on the fact that the language intepreter defines an *evaluates to* relation, `=`, such that `successor(one) = two` and `predecessor(three) = two`, `prime(two) = true`, `even(two) = true`, and so on. These and other relationships constrain the meaning of `two` and distinguish it from `three` or `predecessor`, which participate in different relationships under `=`. Procedural semantics is often criticized as falling prey to the specter of conceptual holism (Fodor & Lepore, 1992), the idea that if expressions are defined in terms of their relations with other expressions, then any small change could have dire ramifications throughout the entire system. Programming languages, however, have introduced techniques for encapsulating information—modularized code, objects, namespaces, and explicit dependency structures, among others (Harper, 2016)—which allow code written by thousands, even millions, of individuals to interact as a unified language.

1.3 Learning as programming

If knowledge is code, learning is then program induction: discovering programs that explain how observed data were generated (Kitzelmann, 2009; Flener & Schmid, 2008; Gulwani et al., 2017). The theory thus draws on inductive programming literature stretching back to the birth of cognitive science (Newell et al., 1958; Newell et al., 1959), and includes subsequent developments in recursive program synthesis (Smith, 1984), structure & heuristic

discovery (Lenat, 1976; Lenat, 1983), meta-programming (Sussman, 1973; Schmidhuber, 1987), genetic programming (Holland, 1975; Koza, 1989) and inductive logic programming (Shapiro, 1983; Muggleton & De Raedt, 1994). The approach also makes use of insights from other formalizations of learning, e.g. deep learning (LeCun et al., 2015), connectionism (Rumelhart et al., 1987), reinforcement learning (Sutton & Barto, 2018), probabilistic graphical models (Koller & Friedman, 2009), and production systems (Lovett & Anderson, 2005). These can be viewed as exploring specific subclasses of programs or possible implementation theories. The learning as programming approach, however, is importantly different in providing learners the full expressive power of symbolic programs both theoretically (i.e. Turing completeness) and practically (i.e. freedom to adopt any formal syntax).

This approach applies broadly to developmental phenomena—including counting (Piantadosi et al., 2012), concept learning (Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2016), function words (Piantadosi, 2011), kinship (Mollica & Piantadosi, 2019), theory learning (Kemp et al., 2010; Ullman et al., 2012), verb learning (Siskind, 1996; Abend et al., 2017; Gauthier et al., 2019), question answering (Rothe et al., 2017), semantics & pragmatics (Goodman & Lassiter, 2015; Goodman & Frank, 2016; Frank & Goodman, 2012), recursive reasoning (Lake & Piantadosi, 2020), sequence manipulation (Rule et al., 2018), sequence prediction (Amalric et al., 2017; Cheyette & Piantadosi, 2017), structure learning (Kemp & Tenenbaum, 2008; Stuhlmuller et al., 2010), action concepts (Lake et al., 2015), perceptual understanding (Depeweg et al., 2018; Overlan et al., 2017), causality (Goodman et al., 2011), and physical inference (Ullman et al., 2018). These applications build on a tradition of studying agents who understand the world by inferring computational processes that could have generated observed data, which is optimal in a certain sense (Solomonoff, 1964a; Hutter, 2005), and aligns with rational constructivist models of development (Xu, 2019; Gopnik & Wellman, 2012; Xu & Griffiths, 2011; Gopnik & Tenenbaum, 2007).

While these ideas appear to be on increasingly solid empirical and theoretical ground, much work remains to formalize them into robust and precise descriptions of children's learning. Most recent LOT work has argued that learners seek short (simple) programs explaining observed data, a version of Occam's Razor. A bias for simplicity favors generalization over memorization, while a bias for fit favors representations that match the world.

Mathematically, these two can be balanced in a principled way using Bayes' theorem or minimum description length formalisms to favor simple, explanatory programs (Goodman, Tenenbaum, Feldman, et al., 2008; Feldman, 2000), a principled approach (Kolmogorov, 1963; Solomonoff, 1964a; Jefferys & Berger, 1992; Chater & Vitányi, 2003; Baum, 2004; Grünwald, 2007; Grünwald & Vitányi, 2008) that fits human data well (Tenenbaum, 1999, 2000; Feldman, 2000; Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2016). Bayesian LOT models have often hypothesized that learners stochastically propose candidates by sampling from a posterior distribution over programs, a process that empirically resembles children's apparently piecemeal, stop-start development (Ullman et al., 2012).

Modeling knowledge as code and learning as programming has worked well for many individual domains. We need, however, a general formalization of mental representations and how they develop. The child as hacker suggests such an account, emphasizing how information can be encoded, assessed, and manipulated as code regardless of domain, with the intention of developing formal tools applicable to broad classes of human learning phenomena. Chapter 2 discusses the child as hacker in detail and illustrates how it might be applied to explain several well-known case studies in developmental psychology.

Chapter 2

The child as hacker

This chapter introduces the *child as hacker* as a hypothesis about the representations, processes, and objectives of distinctively human-like learning. Like the child as scientist view (Piaget, 1955; Carey, 1985, 2009; Gopnik, 2012; Schulz, 2012b), the child as hacker is both a fertile metaphor and a source of concrete hypotheses about cognitive development. It also suggests a roadmap to what could be a unifying formal account of major phenomena in development. We extend the idea of learning as programming discussed in Chapter 1 by drawing inspiration from *hacking*, an internally-driven approach to programming emphasizing the diverse goals and means humans use to make code better. Our core claim is that the specific representations, motivations, values, and techniques of hacking form a rich set of largely untested hypotheses about learning.

2.1 From programming to hacking

Though the idea of learning as programming has been important in formalizing LOT-based learning, views based entirely on simplicity, fit, and stochastic search are likely to be incomplete. Most real-world problems requiring program-like solutions are complex enough that there is no single metric of utility nor unified process of development (Figure 2-1). Even so, modern computational approaches to learning—whether standard learning algorithms or more recent LOT models—use far fewer techniques and values than human programmers. For any task of significance, software engineering means iteratively accumulating many changes

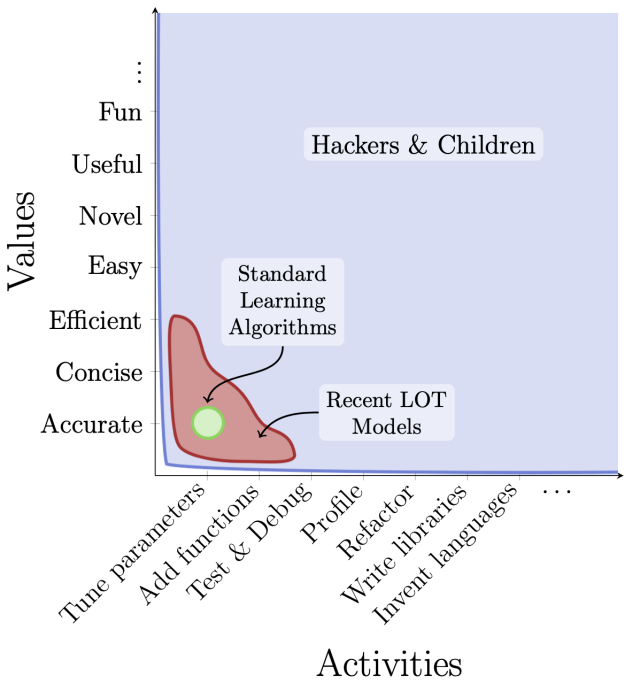


Figure 2-1: Overview of the child as hacker hypothesis. Code can be changed using many techniques (x-axis) and assessed according to many values (y-axis). Standard learning models in machine learning and psychology (green region) tend to focus solely on tuning the parameters of statistical models to improve accuracy. Recent LOT models (red region) expand this scope, writing functions in program-like representations and evaluating them for conciseness and sometimes efficiency. Yet, the set of values and techniques used by actual hackers (and by hypothesis, children; blue region) remains much larger.

to code using many techniques across many scales (Figure 2-2).

In what follows, we enrich learning as programming with a distinctly human style of programming called hacking. Today, the term *hacking* has many connotations: nefarious, kludgy, positive, ethical, and cultural. Rather than directly importing these modern connotations, we draw on earlier ideas about hacking from the origins of modern computing culture (Levy, 1984). Hacking, as used here, is about exploring the limits of a complex system, starting with whatever is at hand and iteratively pushing the system far beyond what initially seemed possible. We thus begin with a notion of hacking as making code better. But the essence of hacking goes deeper. It lies in all the values that count as "better," all the techniques people use to improve code, and a profound sense of internal motivation.

Accurate	demonstrates mastery of the problem; inaccurate solutions hardly count as solutions at all
Concise	reduces the chance of implementation errors and the cost to discover and store a solution
Easy	optimizes the effort of producing a solution, enabling the hacker to solve more problems
Fast	produces results quickly, allowing more problems to be solved per unit time
Efficient	respects limits in time, computation, storage space, and programmer energy
Modular	decomposes a system at its semantic joints; parts can be optimized and reused independently
General	solves many problems with one solution, eliminating the cost of storing distinct solutions
Portable	avoids idiosyncrasies of the machine on which it was implemented and can be easily shared
Robust	degrades gracefully, recovers from errors, and accepts many input formats
Novel	solves a problem unlike previously solved problems, introducing new abilities to the codebase
Useful	solves a problem of high utility
Fun	optimizes for the pleasure of producing a solution
Clever	solves a problem in an unexpected way
Minimal	reduces available resources to better understand some limit of the problem space
Elegant	emphasizes symmetry and minimalism common among mature solutions
Clear	reveals code's core structure to suggest further improvements; is easier to learn and explain

Table 2.1: Learners and hackers share similar values. Hackers want to make their code better, and listed here are some features of good code. They are also features of useful conceptual systems.

2.1.1 The many values of hacking

There are many dimensions along which a hacker might seek to improve her code, making it not only more accurate, but perhaps faster, clearer, more modular, more memory-efficient, more reusable, cleverer, and so on (Table 2.1). The simplest program is unlikely to be the most general; the fastest is usually not the easiest to write; the most elegant typically is not the most easily extensible. Importantly, real world systems do not focus exclusively on the metrics that have come to the forefront of current LOT-learning paradigms. In addition to accuracy and simplicity, hackers are also concerned with values related to resource use, intrinsic reward, aesthetic concerns, and a host of values specific to constructing complex representation. As a result, they often maintain multiple solutions to the same problem, tuned for different sets of values that would otherwise be in tension with one another. Moreover, effective systems in the real world care more about managing complexity than about being short, simple, or terse—though these are sometimes useful tools for managing complexity. Indeed, many foundational ideas in computer science are less about computation per se and

more about managing the inevitable complexity that arises when putting computation to use (Abelson et al., 1996; Fowler, 2018).

2.1.2 The many activities of hacking

To pursue these diverse objectives, hackers have developed many process-level mechanisms for improving their representations (Fowler, 2018), including adding new functions and data structures, debugging faulty code, refactoring code, and even inventing new languages (Table 2.2). Hackers understand dozens, even hundreds, of these mechanisms and their potential impacts on various values. Some make small, systematic, and predictable changes, while others are dramatic and transformative; most are specially tailored to specific kinds of problems. For instance, a hacker might care about speed and so cache the output of subcomputations in an algorithm. She might seek modularity and so define data structures that encapsulate information and make it accessible only through specific interfaces. Or, she might seek reusable parts and so abstract common computations into named functions. This diversity of techniques makes hacking different from both common learning algorithms and recent LOT models. Typically, these other models explore a small set of techniques for improving programs, based on relatively simple (even dumb) local methods like gradient descent, random sampling, or exhaustive enumeration.

2.1.3 The intrinsic motivation of hacking

Hacking is intrinsically motivated. Though a hacker may often be motivated in part by an extrinsic goal, she always generates her own goals—choosing specific dimensions she wants to improve—and pursues them at least as much for the intrinsic reward of better code as for any instrumental purpose. Sometimes, her goal is difficult to assess objectively and so unlikely to arise extrinsically. Other times, her goal can be measured objectively, but she chooses it regardless of—perhaps in opposition to—external goals (e.g. making code faster, even though outstanding extrinsic requests explicitly target higher accuracy). Whatever their origins, her choice of goals often appears spontaneous, even stochastic. Her specific goals and values may change nearly as often as the code itself—constantly updated in light of recent

Tune parameters	adjust constants in code to optimize an objective function.
Add functions	write new procedures for the codebase, increasing its overall abilities by making new computations available for reuse.
Extract functions	move existing code into its own named procedure to centrally define an already common computation.
Test & debug	execute code to verify that it behaves as expected and fix problems that arise. Accumulating tests over time increases code's trustworthiness.
Handle errors	recognize and recover from errors rather than failing before completion, thereby increasing robustness.
Profile	observe a program's resource use as it runs to identify inefficiencies for further scrutiny.
Refactor	restructure code without changing the semantics of the computations performed (e.g. remove dead code, reorder statements).
Add types	add code explicitly describing a program's semantics, so syntax better reflects semantics and supports automated reasoning about behavior.
Write libraries	create a collection of related representations and procedures which serve as a toolkit for solving a entire family of problems.
Invent languages	create new languages tuned to particular domains (e.g. HTML, SQL, LaTeX) or approaches to problem solving (e.g. Prolog, C, Scheme).

Table 2.2: Learners and hackers share similar techniques. Hackers have many techniques for changing and improving code; some are listed here. The child as hacker suggests that the techniques of hackers are a rich source of hypotheses for understanding the epistemic practices of learners.

changes. She is deeply interested in achieving each goal, but she frequently adopts new goals before reaching her current goal for any number of reasons: getting bored, deciding her progress is "good enough," getting stuck, or pursuing other projects. Rather than randomly walking from goal to goal, however, she learns to maintain a network of goals: abandoning bad goals, identifying subgoals, narrowing, broadening, and setting goals aside to revisit later. Even if she eventually achieves her initial goal, the path she follows may not be the most direct available. Her goals are thus primarily a means to improve her code rather than ends in themselves.

The fundamental role of intrinsic motivation and active goal management in hacking suggests deep connections with curiosity and play (Oudeyer, 2018; Gottlieb et al., 2013; Kidd & Hayden, 2015; Haber et al., 2018), which have also been posited to play central roles in children's active learning. We do not speculate on those connections here except to say that in thinking about intrinsic motivation in hacking, we have been inspired by Chu and

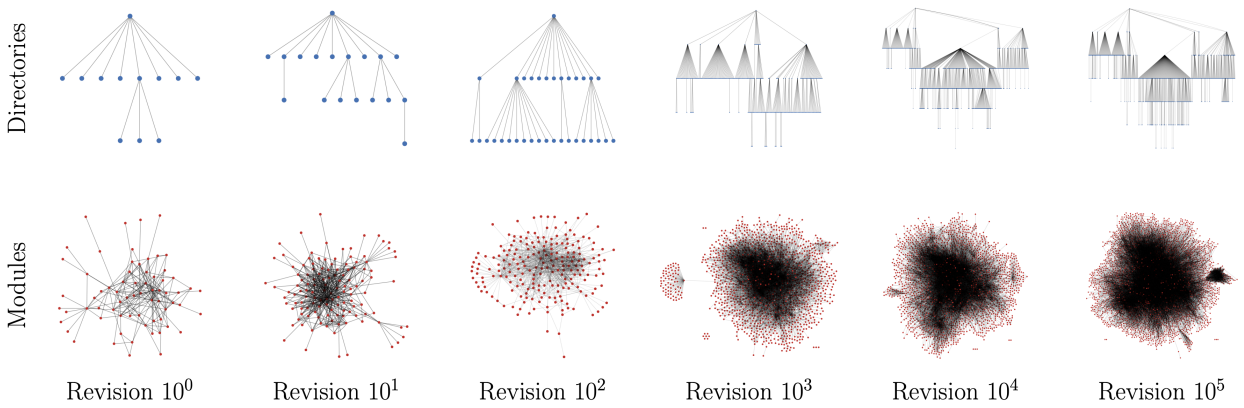


Figure 2-2: Hacking accumulates many changes at multiple scales. The figure shows six snapshots of Linux, after 1, 10, 100, 1,000, 10,000, and 100,000 revisions, including the hierarchical organization of code into directories (blue dots), and the network organization of code into interdependent logical modules (red dots).

Schulz's work exploring the role of goals and problem-solving in play (Chu & Schulz, 2020). Further understanding of this aspect of both learning and hacking could be informed by our search for better accounts of play and curiosity.

2.1.4 Putting it all together

All these components of hacking—diverse values, a toolkit of techniques for changing code, and deep intrinsic motivation—combine to make hacking both a highly successful and emotionally engaging approach to programming. The ability to select appropriate values, goals, and changes to code transforms seemingly stochastic behavior into reliably better code. The combination of internal motivation, uncertain outcomes, and iterative improvement makes hacking a creative and rewarding experience. Consider the following quotes from three experienced hackers (two come from books about specific programming languages, Forth and Lisp, but the lessons apply more broadly):

> A common sequence is: Read the code, gain some insight, and use refactoring to move that insight from your head back into the code. The clearer code then makes it easier to understand it, leading to deeper insights and a beneficial positive feedback loop. There are still some improvements I could make, but I feel I've done enough to pass my test of leaving the code significantly better than how I found it... The key to effective [hacking] is recognizing that you go faster when you take tiny steps, the code is never broken, and you can compose those tiny steps into substantial changes. (Fowler, 2018)

> [Your code] then become[s] the language for describing the data structures and algorithms

> of components written at a a higher level... Now Forth's methodology becomes clear. Forth programming consists of extending the root language toward the application, providing new commands that can be used to describe the problem at hand... In fact, you shouldn't write any serious application in Forth; as a language it's simply not powerful enough. What you should do is write your own language in Forth... to model your understanding of the problem, in which you can elegantly describe its solution. (Brodie, 2004)

> In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient. (Graham, 1993)

This is how hacking develops software. Consider, for example, the growth of the Linux kernel, the basis for several popular operating systems and a long-running open source project. Linux started as a personal project aimed at helping then-undergraduate Linus Torvalds learn about operating systems. It has grown to contain over 750,000 recorded revisions contributed by over 25,000 hackers (Ellerman, 2020). Its developers have diverse interests including: efficiency, accuracy, robustness to crashes, security, portability across hardware platforms, domain-specific utility (e.g. scientific or embedded computing), and modularity. Development occurs in multiple programming languages and relies on many tools, some developed specifically for Linux. The consistent and repeated application of hacking values and techniques transformed this hobby project into a large ($>$ fifteen million lines of code), capable, and reliable codebase still in active use and development.

Figure 2-2 illustrates the dramatic transformation of Linux over time in two ways. First, the source code for Linux is stored in files hierarchically organized into directories. Semantically related files are stored together, such that growth in the directory structure (see "Directories") represents growth in the kinds of things which Linux knows how to do. For example, one directory might contain various ways of representing file systems, another might contain code for talking to external hardware like monitors and keyboards, etc. Changes to this taxonomy might add entirely new kinds of things (e.g. networking), or they might divide old things into multiple categories (e.g. adding a directory for each file system Linux can use). Second, most of the individual files in the Linux kernel describe modules. The creation

of new modules represents the creation of new libraries of code, whether by writing entirely new code or by reorganizing what had previously been a single library into several related but meaningfully distinct libraries. While these modules are physically organized using the hierarchical directory structure, the code in any given module might depend on other code written in other modules. Changes in this dependency structure (see "Modules"), rather than representing taxonomic growth, primarily represent changes in the kinds of representations that are relevant for understanding a given domain. Figure 2-2 thus shows growth both in the kinds of things which Linux knows how to do and in the interconnectedness of the representations which support those abilities.

2.2 Hacking early arithmetic

It is helpful to look through the hacker's lens at a concrete example of algorithmic revision from cognitive development: how preschoolers and early grade-schoolers learn to solve simple addition problems like $2 + 3$. In this section, we demonstrate how the child as hacker can be used to explain key findings in arithmetic learning as natural consequences of changing code-like representations according to hacker-like values and techniques.

We focus on the well-known *sum* to *min* transition (Ashcraft, 1982, 1987; Groen & Resnick, 1977; Kaye et al., 1986; Siegler & Jenkins, 1989; Svenson, 1975), in which children spontaneously move from counting out each addend separately and then recounting the entire set (*sum* strategy) to counting out the smaller addend starting from the larger addend (*min* strategy). Small number addition has been modeled many times (Siegler & Shrager, 1984; Siegler & Jenkins, 1989; Shrager & Siegler, 1998; Jones & Van Lehn, 1994; Neches, 1987; Resnick & Neches, 1984), but even as this case is well known, its significance for understanding learning generally (Siegler, 1996) is not appreciated. This domain is notable because children learn procedures and, in doing so, display many hallmarks of hackers.

Throughout this transition and beyond, children do not discard previous strategies when acquiring new ones but instead maintain multiple strategies (Baroody, 1984; Carpenter & Moser, 1984; Geary & Burlingham-Dubree, 1989; Goldman et al., 1989). The work of Siegler and colleagues, in particular, explicitly grapples with the complexity of both the many values

Algorithm Pseudocode	Trace (2+5)	Resources		
		Operations	Fingers	Memory

`def sum(a1,a2):`
`  raise(a1, LeftHand)`
`  raise(a2, RightHand)`
`  y = count(LeftHand, 0)`
`  sum = count(RightHand, y)`
`  return sum`
	1 2 . . .
	1 2
	7!
	12345 . . .
	34567 . . .
		$2(a_1 + a_2) + 1$	$a_1 + a_2$	1

`def shortcutSum(a1, a2):`
`  y = raiseCount(a1, LeftHand, 0)`
`  sum = raiseCount(a2, RightHand, y)`
`  return sum`
	1 2
	7!
	34567 . . .
		$a_1 + a_2 + 1$	$a_1 + a_2$	2

`def countFromFirst(a1, a2):`
`  sum = raiseCount(a2, LeftHand, a1)`
`  return sum`
	2
	7!
	34567 . . .
		$a_2 + 1$	a_2	2

`def min(a1, a2):`
`  if a1 > a2:`
`  return countFromFirst(a2, a1)`
`  else`
`  return countFromFirst(a1, a2)`
	5
	7!
	6 7 . . .
		$\min(a_1, a_2) + 1$	$\min(a_1, a_2)$	2

`def retrieval(a1,a2):`
`  if (a1, a2) not in seen:`
`  seen[(a1, a2)] = add(a1, a2)`
`  return seen[(a1, a2)]`
	7!	2	0	$[0, \infty)$

Table 2.3: Small number addition algorithms. Each entry lists: code (Algorithm Pseudocode); what a child might do and say (Trace); the number of operations (Operations); how many fingers (or other objects) are needed (Fingers); and how many numbers the child must remember simultaneously (Memory). `raise(N, hand)` holds up `N` fingers on `hand` by counting from 1. `Y = count(hand, X)` counts fingers held up on `hand` starting from `X` to return `Y`. `Y = raiseCount(N, hand, X)` combines `raise` and `count`, counting from `X` while holding up `N` fingers on `hand`. Resource counts for `retrieval` assume a previously seen problem; the values otherwise grow to accommodate a call to `add`, a generic adding algorithm which selects a specific addition algorithm appropriate to the addends. a_1 and a_2 denote the first and second addend, respectively.

which learners might adopt and the need to select among many strategies for solving the same problem. They have shown that children appropriately choose different strategies trial-by-trial based on features like speed, memory demands, and robustness to error (Siegler & Jenkins, 1989; Siegler, 1996).

Table 2.3 implements several early addition strategies as code. For the sake of space, we highlight five strategies (cf. Jones & Van Lehn, 1994; Siegler & Shipley, 1995). Children

acquire the *sum* strategy through informal interactions with parents or at the onset of formal education (Saxe et al., 1987; Siegler & Jenkins, 1989; Baroody & Gannon, 1984) (`sum`; Table 2.3). `sum` appears optimized for instruction and learning. It is simple, uses familiar count routines, requires little rote memorization, and respects children's limited working memory. It also computes any sum in the child's count list, making `sum` an accurate and concise strategy for addition. Most recent LOT models would consider the problem well-solved. `sum` is slow and repetitive, however, counting every object twice.

Restructuring `sum` to simultaneously track both counts, updating the sum while creating each addend, counts each object only once and explicitly represents a strong generalization: here, that the two counts are not coincidental but used for closely connected purposes. The result is `shortcutSum` (Table 2.3): count out each addend, reciting the total count rather than the current addend count. `shortcutSum` tracks both counts using a newly implemented function, `raiseCount`. Maintaining simultaneous counts increases working memory load and the potential for error and is, unsurprisingly, a late-developing counting skill (Fuson et al., 1982; cf. Steffe et al., 1983). Addition strategies incorporating simultaneous counts naturally appear during early grade-school (Fuson et al., 1982) but can be discovered earlier given practice (Siegler & Jenkins, 1989).

Many techniques for improving code are sensitive to execution traces recording a program's step-by-step behavior. In `shortcutSum`, for example, the first call to `raiseCount` is redundant: it counts out `a1`, the first addend, to produce `y`, meaning `y` is equal to `a1`. Removing the first count and replacing `y` with `a1` produces `countFromFirst` (Table 2.3). It is on average twice as fast as `shortcutSum` while reducing finger and working memory demands. These changes, however, are not based on code alone; they require sensitivity to the behavior of code via something like an execution trace. While reported in children and common in theoretical accounts (Neches, 1987; Secada et al., 1983; Resnick & Neches, 1984), there is debate about how frequently `countFromFirst` appears in practice (Siegler & Jenkins, 1989; Shrager & Siegler, 1998).

Changes in a hacker's basic understanding of a problem provide another source of revisions. New understanding often comes from playing with code in the manner of "bricolage" (Turkle & Papert, 1992) rather than formal instruction. For example, she might

notice that addition is commutative—changing the addend order never affects the final sum. `shortcutSum` helps explain why: every raised finger increments the sum exactly once. The principle of commutativity is formally introduced as early as first grade (National Governors Association Center for Best Practices, Council of Chief State School Officers, 2010), but can be independently discovered earlier (Baroody & Gannon, 1984). Commutative strategies are also common before children understand that addition is commutative, suggesting an incomplete or incorrect understanding of addition (Baroody & Gannon, 1984; Steffe et al., 1983).

These discoveries justify swapping addend order when the first addend is smaller than the second. This gives the well-studied *min* strategy: count out the smaller addend from the larger addend (`min`; Table 2.3). *min* is perhaps the best attested small number addition strategy, common from first-grade through adulthood (Groen & Parkman, 1972; Groen & Resnick, 1977; Ashcraft, 1982, 1987; Kaye et al., 1986; Svenson, 1975) but spontaneously developed earlier given extensive practice (Siegler & Jenkins, 1989). On average, `min` removes half the counting necessary for `countFromFirst` and further reduces finger and working memory demand. `min`, however, requires the ability to rapidly compare numbers—the hacking approach naturally draws on libraries of interacting, and often simultaneously developing, cognitive abilities.

Finally, a hacker given certain addition problems multiple times might realize that she could save time by memorizing and retrieving answers after computing them the first time (`retrieval`; Table 2.3), as in dynamic programming algorithms (Cormen et al., 2009). Indeed, as children age they rely decreasingly on strategies requiring external cues (e.g. fingers, verbal counting), and increasingly on memorization (Siegler & Jenkins, 1989), a transition humans formally teach (National Governors Association Center for Best Practices, Council of Chief State School Officers, 2010) and also discover independently (Saxe, 1988b, 1988a).

Much of we know about the development of small number addition is thus well-aligned with the child as hacker, which naturally accommodates and unifies many seemingly disparate phenomena. The child as hacker also suggests several next steps for work on addition and related domains.

First, we need models of learning that formalize knowledge as code modified using hacker-

```
fCounts :: (Set -> Word) -> Bool

def fCounts(f):
  usesCountList(f) and touchesAllItems(f)
```

(a)

```
count :: (Set -> Word)

def count(set, start="one"):
  say(start)
  if not isSingleton(set):
    count(pop(set), next(start))
```

(b)

```
count :: (x: Set) ->
  (y: Word | y = name(cardinality(x)))
def count(set, start="one"):
  say(start)
  if not isSingleton(set):
      count(pop(set), next(start))
```

(c)

Figure 2-3: Three steps toward a mature understanding of counting: (a) a predicate for identifying counting procedures; (b) a procedure for counting; and (c) a more nuanced type for counting.

```
1  data Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2  data Number = Simple Digit | Compound Number Digit
3
4  succDigit 0 = 1
5  -- ...
6  succDigit 8 = 9
7
8  succ (Simple 9) = Compound (Simple 1) 0
9  succ (Simple x) = Simple (succDigit x)
10 succ (Compound n 9) = Compound (succ n) 0
11 succ (Compound n d) = Compound n (succDigit d)
```

```
12 succ :: (x : int) -> (y : int | greater(y x))
13
14 greatestNumber ::
15   (x : int | not(exists(y greater(y x))))
16
17
18 -- Type Inference Fails:
19 --
20 -- greater(succ(greatestNumber) greatestNumber)
21 --
22 -- greatestNumber and succ are incompatible!
```

Figure 2-4: Learning about infinity through number grammars and the successor function: (1–11) data structures and functions for computing base-10 successors; and (12–22) discovering via type inference that the successor function is incompatible with a greatest number.

like values, goals, and techniques. Explicitly situating arithmetic learning within the context of the child as hacker will likely suggest useful and novel hypotheses (e.g. specific hacking techniques (Fowler, 2018) might explain specific chains of strategy introduction; differences in values might explain differences in performance (Siegler & Jenkins, 1989)).

Second, mathematical learning extends far beyond small number addition, including both early sensitivities to number and the development of counting, and the later development of compositional grammars for large numbers, a concept of infinity, more complex arithmetic, and so on. Many of these require deep interactions between procedural and declarative knowledge. The child as hacker suggests ways to integrate these phenomena into a general account of mathematical development.

For example, developing a procedural mastery of the simpler skill of counting and a conceptual grasp on its declarative meaning is an iterative and protracted process that unfolds over many years. After carving out an initial semantic field (Wagner et al., 2016;

Bloom & Wynn, 1997; Wynn, 1992b), 18-month-olds recognize features of correct counting by others (e.g. uses number words, touches all items) before counting themselves (Slaughter et al., 2011). Hackers develop similar sensitivities by writing specification predicates for behavior they want to replicate. Figure 2-3a shows such a predicate which takes as input a function `f` from **`Set`**s to count **`Word`**s and returns a **`Bool`** indicating whether the function meets the specification of using the count list and touching all the items in the set[1]. Children then acquire their own count list (Briars & Siegler, 1984; Fuson et al., 1982; Fuson, 1988) and build on core sensitivities to small set cardinalities (Wynn, 1992a) to learn a series of partial, *subset* counting algorithms before acquiring a general counting procedure (Wynn, 1990a, 1992b; Carey, 2009; Piantadosi et al., 2012). Children at this stage count correctly but typically fail to connect counting with cardinality (Davidson et al., 2012; Barner, 2017; Jara-Ettinger et al., 2017). Hackers often write working functions for which they have only a rudimentary type signature. Types provide semantic summaries of code, allowing for fast and automated reasoning about certain properties of a program. More complex type systems support richer inferences. Figure 2-3b shows `count`, which counts by removing items from the set while pronouncing count words (Piantadosi et al., 2012). The type, however, describes `count` merely as a function from **`Set`**s to count **`Word`**s—many incorrect functions also fit this description. The semantics of counting mature over the following months and years (Davidson et al., 2012; Barner, 2017; Jara-Ettinger et al., 2017), and hackers similarly revise types repeatedly to encode more detailed information. In Figure 2-3c, the definition of `count` is unchanged, but the type now encodes that the count **`Word`** signifies the **`Set`**'s cardinality. This marks a more complete understanding of counting as establishing and naming set cardinalities.

From there, learning a base system, number grammar, and procedures for manipulating large numbers (Fuson, 1992; Fuson, 1988), eventually leads to the successor principle (every number has a successor) and the concept of infinity (there is no largest number) (Cheung et al., 2017; Hartnett & Gelman, 1998; Hartnett, 1991; Yang, 2016). Again, this requires significant interplay between declarative knowledge and procedural knowledge. A hacker

[1]This is not a complete specification of counting but is certainly a plausible early specification for children just beginning to recognize counting behavior.

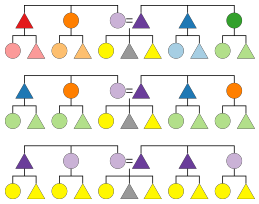
might similarly: develop numerical data structures, write a successor function to manipulate them, infer a type for successor showing that the output is greater than the input, and prove that successor is incompatible with a greatest number. In Figure 2-4, lines 1–2, `Digit` and `Number` describe possible values that a digit or number can take, respectively. Lines 4–11 implement successor for these data structures. In line 12, the type of `succ` encodes the successor principle as *a function from an integer* `x` *to an integer* `y` *where* `y` *is greater than* `x`. Similarly, lines 14–15 describe `greatestNumber` as *an integer* `x` *where no integer is greater than* `x`. Lines 18-22 are the output of a failed type inference algorithm, saying that these types are incompatible because `succ greatestNumber` is greater than `greatestNumber`. Either there is no greatest number or the type of `succ` is incorrect. The same kinds of thinking could also be applied to other mathematical concepts like: shapes, patterns, multiplication, subtraction, integers, division, rational numbers, decimals, real numbers, algebra, geometry, logic, and even calculus and higher mathematics.

Third and finally, the child as hacker should also provide paths to algorithmic theories for qualitatively different kinds of knowledge acquisition—e.g. intuitive theories of the physical and social world—as described next.

2.3 Hacking intuitive theories

While the small number addition example examines procedural learning in mathematics, the child as hacker equally applies to other domains and kinds of knowledge. This section briefly considers two: kinship systems (Figure 2-5a) can be seen as logical and declarative theories of social relatedness, and Mendelian inheritance (Figure 2-6a) as a probabilistic and causal theory of biological relatedness. As with early arithmetic, the key point is not that these exact processes are the ones actually occurring in children's minds during development. It is instead that the tools and techniques of hacking provide a rich language for describing what might be happening and offer a rich source of formal hypotheses for further investigation.

A hacker might implement both kinship and Mendelian inheritance by compressing a set of observations into more reusable, generalizable, and modular code. In both cases,



(a)

```
lives(jim, chicago).
mother(jenny, ben).
eye_color(dan, blue).
father(jim, hacker).
uncle(dan, ben).
fashionable(may).
aunt(may, hacker).
% ...
```

(b)

```
father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(Y, Y), female(X).
husband(X, Y) :- male(X), spouse(X, Y).
sister(X, Y) :-
  parent(P, X), parent(P, Y),
  female(X).
% ...
```

(c)

```
parent(jim, hacker).
parent(jenny, ben).
female(hacker).
male(ben).
male(jim).
female(jenny).
lives(jim, chicago).
% ...
```

(d)

Figure 2-5: Mapping kinship to code: (a) a family tree labeled by three kinship systems (circle = female, colors are different terms, child generation ignores gender); (b–d) kinship in Prolog: (b) initial kinship data; (c) rules for inferring kinship relations, including new primitives `parent`, `spouse`, `male`, and `female`; and (d) a small set of rules such that (c) & (d) implies all of (b).

she iteratively improves her program—adding, deleting, and revising code—and occasionally adds entirely new structures simply by defining and using them. Some changes help, others are rejected, and she eventually produces compact theories of both domains.

In learning kinship, she can frame the task as refactoring a long list of relations about individuals (Figure 2-5b) into rules for high-level kinship terms (Figure 2-5c) and a small set of basic facts (Figure 2-5d) from which all relations can be easily derived. As she develops her theory, she ultimately decides to implement it in a logic programming language called Prolog. Prolog expresses computations as Horn clauses called rules, `Head :- Body.`. `Head` is true if each term in `Body` is true. Empty bodies are considered true. This logical representation cleanly matches the logical structure of kinship that she is discovering. Along the way, our hacker hypothesizes four latent relationships—`parent(X, Y)`, `spouse(X, Y)`, `male(X)`, and `female(X)`. While these do not appear in the original data, adding them to the system ends up dramatically simplifying her explanations. The process of adding them is straightforward. She simply names each symbol and begins using it. Each takes on meaning as she defines the facts and rules in which it participates. She discovers that the terms she is most interested in—`brother(X, Y)`, `sister(X, Y)`, `mother(X, Y)`, etc.—all depend on these new latent predicates, rather than on others like `lives(X, Y)` or `eye_color(X, Y)`. Given this theory, the Prolog interpreter then allows her to draw inferences using deductive proof to learn, e.g. who her uncles are.

In learning Mendelian inheritance, she can frame the task as refactoring a long list of

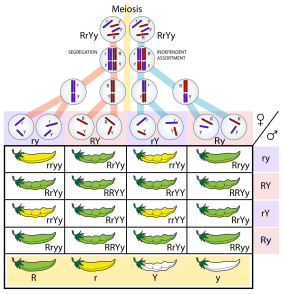
phenotypes and parentage records (Figure 2-6b) into a causal theory of biological inheritance relating phenotypes to genotypes via the three laws of inheritance (Figure 2-6c). Because patterns of inheritance are not strictly logical but require distributional reasoning, and because she is looking for a causal explanation, our hacker implements her theory as a generative model in a probabilistic programming language called Church (Goodman, Mansinghka, et al., 2008). Church expresses computations as parenthesis-delimited trees that change according to certain built-in rules. These rules can be modified by using `def` to define specific changes for a given leaf or subtree. For example, Figure 2-6b says that the leaf `data` should be replaced by the subtree `((a1 NA NA (YW WR)) (a2 NA`.... This generative model specifies a uniform `prior` distribution over unknown parents (i.e. each gene in their genotype is drawn randomly), and a causal process by which parents `breed` to produce children based on the laws of `segregation` and `separation`. Together, these can be used to sample the possible `genotype` for a given individual, from which it is then possible to derive their visible `phenotype` given the law of `dominance` and a list of `traits`. To invert this generative process to perform inference, she queries her theory using Church's built-in tools for sampling-based inference to learn, e.g. likely genotypes for `a1` and `a2` given the observed phenotypes of all the individuals listed in `data`.

2.4 Hacking and other metaphors

The child as hacker builds on several other key developmental metaphors. All these views are valuable and have significantly improved our understanding of learning. Here, we explain how the child as hacker extends these accounts, highlighting its potential contributions. See Table 2.4 for a summary of the major claims of each view.

2.4.1 The child as scientist

The child as scientist metaphor is one of the strongest influences on the child as hacker. With roots in the work of Piaget (Piaget, 1955) and since extensively developed (Carey, 1985; Gopnik, 1996; Carey, 2009; Schulz, 2012b; Gopnik, 2012), this view emphasizes how children structure their foundational knowledge in terms of intuitive theories analogous in



(a)

```
(def data
  '((a1 NA NA (YW WR))
    (a2 NA NA (GN SM))
    (b1 a1 a2 (YW SM))
    (b2 a1 a2 (YW SM))
    (c1 b1 b2 (YW WR))
    (c2 b1 b2 (GN SM))
    (c3 b1 b2 (YW SM))
    (c4 b1 b2 (GN WR))
    (c5 b1 b2 (YW SM))
    ;; ...
))
```

(b)

```
(def traits (list (pair YW GN) (pair SM WR)))
(def (prior) (repeat (len traits) random-gene))
(def (breed genotype1 genotype2)
 (map pair (segregation genotype1) (segregation genotype2)))
(def (genotype id) (if (eq? id 'NA)
  (prior)
  (breed (genotype (parent1 id)) (genotype (parent2 id)))))
(def (phenotype genotype) (map dominance genotype traits))
(def (segregation parent) (map separation parent))
(def (separation gene) (uniform-draw gene))
(def (dominance gene trait)
  (if (dominant? gene) (dominant trait) (recessive trait)))
```

(c)

Figure 2-6: Mapping Mendelian inheritance to code: (a) an overview of Mendelian inheritance; (b–c) Mendelian inheritance in Church. (b) a list of individuals (`a1`, `a2`, `b1`,...), their parents, and phenotypes (`YW` = Yellow; `GN` = Green; `SM` = Smooth; `WR` = Wrinkly); (c) a list of traits (dominant followed by recessive) and part of a generative theory using Mendel's laws and a uniform prior over unknown parents (i.e. `random-gene` draws a pair of alleles uniformly at random).

important ways to scientific theories (Murphy & Medin, 1985; Gopnik & Meltzoff, 1997; Wellman & Gelman, 1992, 1998), and build knowledge via epistemic values and practices (Schulz, 2012b; Gopnik et al., 2004; Gopnik & Tenenbaum, 2007; Gopnik & Wellman, 2012; Xu & Griffiths, 2011) similar to the ways scientists collect and analyze evidence and modify theories in response to evidence, constructing theories which are accurate, general, and simple. The related view of rational constructivism (Xu, 2019) emphasizes the sophisticated mechanisms children use in theory-building—Bayesian statistical inference, constructive thinking processes such as analogy, mental simulation, and other forms of "learning by thinking" (Lombrozo, 2019), and active, curiosity driven exploration—and the importance of formalizing these mechanisms in rational computational models.

The child as scientist and the child as hacker are best seen not as competitors but as natural companions, with overlapping but complementary notions of knowledge representation, epistemic values and practices, and constraints on learning, which together paint a more complete picture of cognitive development. The child as scientist emphasizes children's learning as centrally focused on building causal models of the world and the conceptual sys-

Metaphor	Knowledge	Objectives	Mechanisms
Scientist	Theories: causal models and conceptual systems of varying abstraction	Accuracy, generality, simplicity of models	Inductive inference, experimentation, analogy, bootstrapping
Workshop/ Evolution	Cognitive tools: rules, strategies, networks, hypotheses, schemas	Expected utility (reward), intrinsic motivation	Darwinian variation and selection with goal-sketch constraints
Hacker	Programs: functions, data structures, type systems, libraries, languages (Section 1.2)	The above plus many other dimensions of value in good code (Table 2.1)	The above plus many other methods for revising & improving code (Table 2.2)

Table 2.4: A comparison of three families of developmental metaphors discussed in this chapter—the child as scientist, the workshop and evolutionary metaphors, and the child as hacker—along three dimensions: the kinds of knowledge learners acquire, the primary objectives of learning, and the mechanisms used in learning.

tems (intuitive theories) supporting these models. It asks questions about how theories are represented, what makes for good theories, and what mechanisms support theory learning, drawing inspiration from how scientists have approached these questions and implementing its proposals computationally as approximations to Bayesian inference over spaces of causal networks, probabilistic first-order logic, and probabilistic programs (Gopnik et al., 2004; Gopnik & Tenenbaum, 2007; Gopnik & Wellman, 2012; Goodman et al., 2015; Ullman et al., 2012; Goodman et al., 2011; Kemp & Tenenbaum, 2008; Kemp et al., 2010).

The child as hacker extends these ideas with its broader view of what kinds of representations are worth learning, what values set goals for learning, and what practices are useful for accomplishing these goals: programs may go beyond the purely causal, there are many values for good programs beyond those traditionally used to assess scientific theories (accuracy, generality, simplicity), and learning draws on many algorithmic-level processes across multiple timescales, not just the stochastic sampling or search mechanisms that have traditionally been used in Bayesian models of theory learning. This view could enrich both the computational and algorithmic-level claims of child as scientist models in many specific ways. For example, intuitive theories could benefit from being formalized as domain-specific libraries or languages for writing generative probabilistic programs (e.g. Section 2.3), and the construction of more radically new kinds of concepts could be captured as the construc-

tion of new function and data types, not just new functions or data structures of existing types. The many values of good code in Table 1 could also have analogs in the goals that guide children in constructing their intuitive theories, and the processes of improving code in Table 2 could all have analogs in how children build their intuitive theories; perhaps these could help formalize some of the mechanisms of analogy, bootstrapping, explanation-driven and goal-driven search proposed in the child-as-scientist and rational constructivism views (Carey, 2009; Schulz, 2012b; Lombrozo, 2019; Xu, 2019) which have not been fully captured by previous algorithmic-level learning models.

It is perhaps fitting that scientists recognize highly familiar scientific practices and values in development, but in addition to an evocative metaphor, the child as scientist is a fruitful hypothesis. It has sparked numerous Child-As-X theories in cognitive psychology positing specific modes of scientific thinking as key throughout development. Children can be seen as: linguists determining the structure of language (Gleitman et al., 1977; Karmiloff-Smith, 1992; Labov, 1989); anthropologists systematically studying behavior (Harris, 2012); statisticians inferring latent world structure (Gigerenzer & Murray, 1987; Peterson & Beach, 1967); econometricians discovering preferences (Lucas et al., 2014); and philosophers refining understanding through reflection and analysis (Kohlberg, 1968; Selman, 1981). We hope the child-as-hacker view will further grow this productive tradition. Efforts to formalize the child-as-scientist metaphor have also played key roles in its fruitfulness (Gopnik et al., 2004; Gopnik & Schulz, 2004; Gopnik & Tenenbaum, 2007; Gopnik & Wellman, 2012; Xu & Griffiths, 2011). Indeed, many of the LOT models discussed earlier were explicitly developed to formalize aspects of theory learning and the broader scientific process. Formalizing the child as hacker may seem like a daunting challenge, but this process took decades of sustained interdisciplinary effort for the child as scientist. A long-term investment in computational models for the child as hacker could prove similarly fruitful.

2.4.2 Resource rationality & novelty search

The idea of resource rationality (Lieder & Griffiths, 2020; Griffiths et al., 2015; Lewis et al., 2014) argues that theories must account for cognition as realized in finite computational devices. Time, memory, and energy are limited. Learners can thus more quickly find practical

hypotheses by evaluating resource use alongside simplicity and fit. Stanley and colleagues have developed the idea of novelty search (Lehman & Stanley, 2011b, 2011a) around the observation that many learning problems require navigating large hypothesis spaces in finite time. Comparing trivially different hypotheses is unlikely to be helpful. They demonstrate for many classes of problems that agents sensitive to novelty learn more effectively than agents using other objectives.

Both resource rationality and novelty search are important ways of thinking about objectives in learning. The child as hacker embraces these insights, but makes claims about learners' objectives beyond either view. First, it encourages considering both efficiency and novelty, rather than arguing for either alone. Second, it argues for a radically larger set of possible influences on the objective function, including engineering and aesthetic concerns and perhaps more (Table 2.1). Third, it suggests that learners' objectives constantly change in complex and as-yet poorly understood ways, identifying a key area for future research. Rather than searching for the right human-like objective function, the child as hacker suggests that cognitive scientists seek to understand an entire space of possible objectives and the ways that learners move between them.

2.4.3 Workshop and evolutionary metaphors

The child as hacker is also closely related to a pair of metaphors from Siegler and colleagues emphasizing the dynamics of learning: the workshop metaphor (Siegler & Jenkins, 1989) and the evolutionary metaphor (Siegler, 1996). The workshop metaphor emphasizes the diversity of knowledge (raw materials) and learning processes (tools) available to children when producing mental representations (products) to meet the demands of daily life (work orders), and the importance of selecting appropriate materials and tools for a given product. The evolutionary metaphor recasts these ideas in light of biological evolution, highlighting the essential role of variability, selection, and adaptation in learning. These metaphors work together to tell a broader story about learning. Both argue that we maintain multiple strategies for solving any given problem and adaptively choose among them, learning about their context-specific usefulness over time. In contrast to "staircase" theories suggesting long periods of relatively uniform thinking punctuated by brief and dramatic transitions, they

suggest that children navigate "overlapping waves" as new strategies appear and others fade.

The child as hacker shares much with these metaphors. They all emphasize the importance of bringing a diverse collection of mental representations to bear during learning, as well as selecting representations, values, and learning strategies most relevant to the specific task at hand. Each view also highlights the way knowledge is iteratively revised; the outcomes of learning are themselves frequently the raw inputs for future learning. Each makes variability, selection, and adaptation central features of learning.

The mind, however, operates on representations which bear a closer resemblance to software than hardware, looking more like programs than tables or chairs. We could think of the child as hacker as updating the workshop metaphor for the software era and focusing on the tools needed to build a rich computational model of a richly computational mind: all the ways we have come to represent knowledge with programs and programming constructs, and all the values and activities of hacking for making programs better, which seem more directly tied to the goals and mechanisms of learning and more amenable to computational formalization than those of carpentry or metalwork.

The child as hacker may also be better aligned with children's goal-orientedness during learning. Evolution is an intentionless process whose primary change mechanisms act at random. In the workshop and evolutionary metaphors, goal schemas can constrain this random search process (Shrager & Siegler, 1998) but that is different from directly and deeply guiding it. As with other forms of stochastic search or reinforcement learning, learning under an evolutionary mechanism would thus require tremendous amounts of computation and time (Baum, 2004). Children's learning, by contrast, is remarkably efficient (Tenenbaum et al., 2011), in part because it is strongly goal-directed (Schulz, 2012b). Children's behavior may sometimes look random, but there is almost always an underlying goal driving that behavior. Where the apparent randomness comes from, the evolutionary character, is perhaps a dynamically changing set of goals: initially X, then Y, then Z, then back to X until it is achieved, and so on. This dynamic is more consistent with the intrinsic nature of goals in hacking, where children's goals might then address different values such that each improves representations in different ways. Externally, without access to those goals or their internal logic, both learning and hacking may look random, piecemeal, and

non-monotonic—sometimes progressing, sometimes regressing. Internally, however, each is intensely goal-driven, resulting in profound, long-term growth.

In sum, the child as hacker helps to refine and advance the workshop and evolutionary views, by giving a less metaphorical take on the workshop metaphor and a better fit for the goal-driven behavior of children than the intentionless randomness of evolution. Moreover, the child as hacker makes specific suggestions beyond either metaphor, including a strong emphasis on program-like representations and the specific values and processes that guide how programs get better (Tables 2.1 and 2.2), which we hope can serve as the basis for a new generation of modeling in cognitive development.

2.5 Prospects for a computational account of learning

Hacking represents a collection of epistemic values and practices adapted to organizing knowledge using programs, and there is growing evidence that programs are a good model of mental representations. The child as hacker combines these ideas into a roadmap toward a computational account of learning and cognitive development. It makes testable claims about a general class of inductive biases humans ought to have—namely those related to synthesizing, executing, and analyzing information as programs. It also concretely identifies the representations, objectives, and processes supporting learning with those of human hackers. Finally, it makes a unifying claim about how these three might be implemented as code, procedures for assessing code, and procedures for revising code, respectively.

To explain learning in light of these claims, we must systematically use code as a lens on learning. Doing so produces testable hypotheses that differ from common alternatives. For instance, the child as hacker predicts that children frequently change beliefs in the absence of external data. It predicts that children might learn representations which are less accurate or more complex than alternatives so long as they win on, e.g. modularity or cleverness. It also predicts that, while dramatic, global changes are possible, learning typically occurs through many simple, structured changes, similar to the way code tends to be refactored.

Both machine learning and psychology would benefit from a united effort to pursue this roadmap in developing a computational account of human learning. Machine learning

would benefit greatly from the growth of empirical programs in psychology to understand how children hack their own representations (see Outstanding Questions), how real hackers assess and improve their code in practice, and how children adopt and pursue goals. Effectively searching large hypothesis spaces is a fundamental problem in machine learning, so one crucial question for this second program is how humans effectively search the space of Turing-complete computations. Psychologists and cognitive scientists would benefit greatly from a sophisticated framework for program induction. Such a framework would bring together existing knowledge about theoretical computer science, programming languages, compilers, program synthesis, and software engineering to provide tools capturing human-like approaches to solving problems in these domains.

The child as hacker offers a path toward addressing central challenges of human learning and development that both reframes classic questions and helps us ask new questions. Recent efforts in cognitive science on constructive thinking (Lombrozo, 2019), the neuroscience of programming (Fedorenko et al., 2019; Ikutani et al., 2020; Ivanova et al., 2020), and modeling the development of intuitive physics using game engines (Ullman et al., 2017; Smith et al., 2019) represent promising complementary steps. Recent developments in program synthesis are also beginning to operationalize aspects of specific hacking techniques, including work on backward chaining of goals and subgoals (Osera & Zdancewic, 2015; Polikarpova et al., 2016; Polozov & Gulwani, 2015), neurally-guided synthesis (Balog et al., 2017; Devlin et al., 2017), iterative refactoring (Dechter et al., 2013; Ellis et al., 2018; Lin et al., 2014; Cropper et al., 2019), incremental programming (Solar-Lezama, 2008; Nye et al., 2019; Ellis et al., 2019), learning generative probabilistic models (Hewitt et al., 2020; Ellis et al., 2020), and learners sensitive to resource use (Knoth et al., 2019; Cropper & Muggleton, 2019). These efforts have the potential to move the child as hacker beyond just another metaphor, or just a hypothesis, to a working and testable computational account of cognitive development. But they are just first steps. We look forward to all the work that remains to be done to understand how it is that children hack their own mental representations to build yet-unparalleled tools for thinking. The rest of this thesis takes another step in this direction by conducting a detailed empirical investigation of concept learning focused on the domain of list functions. In Chapter 3, we describe this family of concepts in detail.

Chapter 3

List functions as a domain for psychological investigation

The remainder of this thesis develops a first attempt to formalize learning as hacking in a computational model, and explores this model along with several alternative approaches in one in-depth empirical study of human and machine concept learning. The goal of this chapter is to introduce the family of concepts that we will be studying for the rest of thesis, the domain of *list functions*.

The variability of human concept learning is part of what makes it so intriguing. Sometimes, learning is swift and powerful, recovering abstract, generalizable conceptual systems after brief interactions with sparse data (Tenenbaum et al., 2011). The answer might appear immediately, almost without conscious effort. Word meanings, for example, can often be inferred from just a single informal encounter (Carey & Bartlett, 1978). If not that, it might be that a learner can tailor problem-solving strategies on a task-by-task basis in sophisticated ways, even for novel tasks, reliably reaching correct answers and understanding why the strategy worked (Siegler, 1996). Other times, even after accounting for immediate restrictions on energy, time, and computational power (Lieder & Griffiths, 2020; Griffiths et al., 2015), inductive learning is slow, piecemeal, and labored (e.g. for natural number: Carey, 2009; Barner, 2017). People have the sensation of struggling for a long time through a difficult search, trying multiple options and only slowly narrowing in on the solution after many false starts and frequent backtracking. The problem might even remain stubbornly

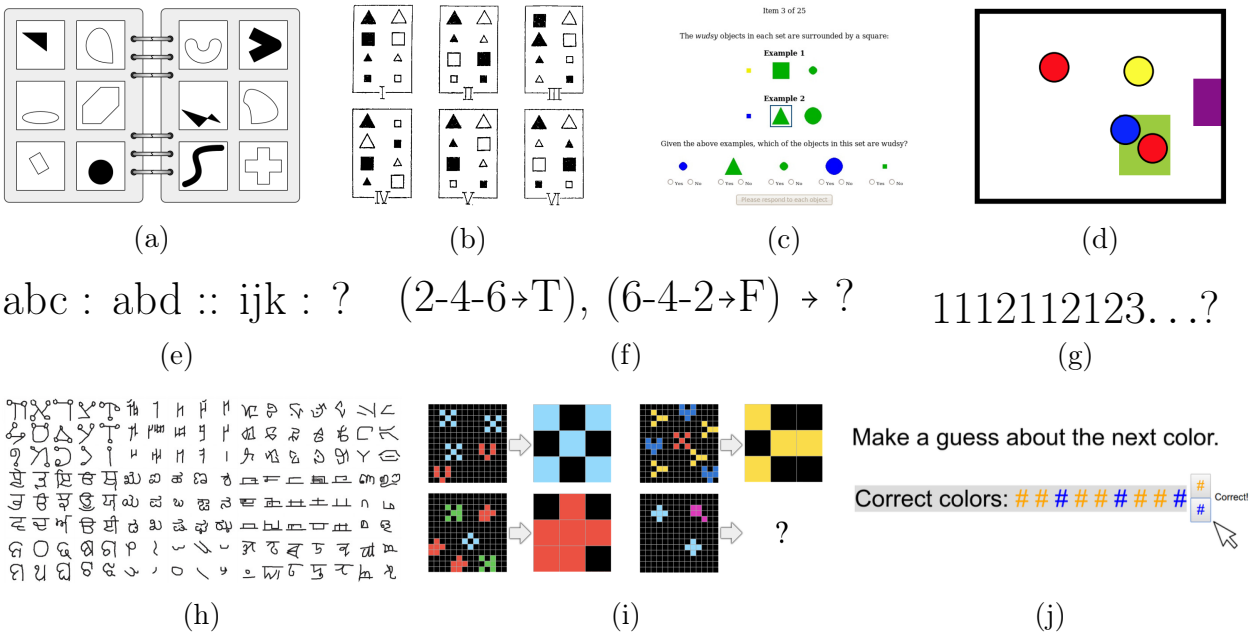


Figure 3-1: Stimuli from key experimental domains for inductive concept learning. (a): a Bongard problem; (b): the six Boolean concept types studied in Shepard et al. (1961), with exemplars on the left and non-exemplars on the right; (c): the Boolean reasoning task from Piantadosi et al. (2016); (d): an artificial world from Ullman et al. (2018). (e): a text-based analogy, *abc is to abd as ijk is to. . .?*; (f): examples from the 2-4-6 task; (g): a numerical sequence prediction task; (h): characters from Omniglot; (i): stimuli for a concept from the ARC challenge; (j): a binary sequence prediction task from Cheyette and Piantadosi (2017).

opaque until the solution is explicitly provided—sometimes even after it has been provided.

One difficulty cognitive scientists face in studying concept learning is finding problem classes which expose this variance to empirical study. Several domains have emerged as key areas of investigation (Figure 3-1). Bongard problems (Bongard, 1967; Hofstadter, 1979; Foundalis, 2006) are visual classification tasks. Participants are shown 12 images divided into two gorups of six. Each group is governed by a rule and, typically, the rules between the two groups are related. The task is to infer both rules. Boolean and n-ary function learning (Bruner et al., 1956; Shepard et al., 1961; Nosofsky, Gluck, et al., 1994; Nosofsky, Palmeri, et al., 1994; Feldman, 2000; Feldman, 2003; Goodman, Tenenbaum, Feldman, et al., 2008; Goodwin & Johnson-Laird, 2013; Piantadosi et al., 2016) provide a more constrained sort of classification problem. They present images varying on multiple Boolean or n-ary dimensions such as color, size, shape, or number. Participants are asked to classify images based on some logical rule. In Shepard et al. (1961), stimuli vary on three Boolean dimensions, providing

6 structurally distinct types of propositional concepts. Exemplars and non-exemplars are assigned different labels, and the task is to learn correct labels for each object. By contrast, Piantadosi et al. (2016) presented subjects with entire sets of stimuli, and asked them to classify objects in each set as either exemplars or non-exemplars of a first-order rule.

This sort of set inclusion task has also been explored in rule learning. In Wason's (1960) 2-4-6 task, participants are given an initial triple in accordance with a rule, e.g. *2-4-6* might be given for the rule *numbers in ascending order*, and asked to propose new triples. They receive feedback after each proposal. When ready, they guess the rule. In Tenenbaum (2000), subjects are shown random samples from a concept specifying a subset of the numbers 0–100 according to some rule. They are then asked to label every remaining number as either belonging to or being excluded from the set. Numerical reasoning has also been studied in function learning tasks (see Schulz et al., 2017, for review). In these tasks, participants are presented with a real number or continuous magnitude given as input to a mathematical function; they are asked to predict the function's output magnitude. Sequence prediction tasks also investigate numerical and Boolean cognition (Bartlett, 1958; Mahabal, 2010; Amalric et al., 2017; Cheyette & Piantadosi, 2017). Here, subjects are presented with the beginning of a sequence, typically numerical or Boolean, and asked to continue it.

Analogy-based domains present a different sort of task in which participants view a series of input/output relations and are asked to infer some rule relating inputs and outputs. Learning is tested by asking people to predict outputs for novel inputs by applying their candidate rule. Text-based analogies as studied in Seek-Whence, Jumbo, and CopyCat (Meredith, 1986; Mitchell, 1992; Hofstadter, 1995) define input/output relations over character strings, while visual analogy domains such as ARC (Chollet, 2019) define input/output relations over images. These analogies can be quite sophisticated, relying on knowledge about language, number, shape, color, symmetry, and other aspects of visual cognition.

Several domains have recently been used to investigate participants' ability to infer probabilistic generative models for objects such as stochastic tree structures (Stuhlmuller et al., 2010), hand-written characters in Omniglot (Lake et al., 2015; Lake et al., 2019), and the fundamental physical parameters of simulated worlds (Ullman et al., 2018). These causal domains can be very complex. In the simulated world experiments, participants view 5s

videoclips of scenes filled with various objects and surfaces, marked in different colors. They must then infer properties for each surface and object—such as mass or friction coefficients—as well as global parameters like gravity. Being defined by a hierarchical generative model, these domains support a variety of tasks. The work of Lake and colleagues on hand-written characters, for example, includes: classifying characters, parsing characters into strokes, generating new examples of a character, generating novel characters for an alphabet, and generating new alphabets.

These tasks share many common traits. Not all domains excel in all areas, but most excel on most dimensions. They are both familiar and engaging to learners, ideally drawing on meaningful background knowledge. They frequently support multiple tasks requiring different kinds of reasoning and operating at multiple levels of abstraction. They contain objects with complex internal structure over which many relations can be defined. They support a broad variety of problem difficulties: some are trivially easy and some resist nearly all attempts at understanding, but most can be reliably learned with a little bit of effort. They naturally support the full range of algorithmic thinking, including key features like recursive, pattern-based, and conditional reasoning. Finally, despite everything else, they remain formally tractable and interpretable, amenable to decomposition into meaningful primitives that could reside in an LOT.

This chapter presents list functions as a domain for studying human concept learning. The premise of the domain is simple: concepts consist of computable functions from lists to lists. We focus specifically on lists of natural numbers. This domain has a long history in artificial intelligence (Green et al., 1974; Shaw et al., 1975; Biermann, 1978; Green, 1981; Smith, 1984; Feser et al., 2015; Osera & Zdancewic, 2015; Polikarpova et al., 2016; Cropper et al., 2019) but is basically unstudied in cognitive psychology. It nonetheless combines many of the best features of classic concept learning tasks listed above. It also both subsumes many of these classic tasks and provides tasks similar to several key developmental case studies. At the same time, list functions open new avenues for exploring algorithmic thinking over complex structures. We argue that its long computational history and rich potential as a domain for studying human learning make it a prime candidate for investigating the child as hacker and cognitive development more broadly.

3.1 The domain of list functions

The domain of list functions is simple to describe. Every concept in the domain defines a rule for creating an output list from an output list. In its most general form, these can be lists of arbitrary structures, including lists of lists, and the input and output lists do not need to contain the same kinds of elements. For example:

1. *return the input unchanged*
2. *remove all but the third element*
3. *reverse the list*
4. *sum the unique elements less than 50*
5. *count the elements equal to 5*
6. *count the elements equal to today's date*
7. *replace each number with its name in Hindi*
8. *return a list of all halting Turing machines*

In this general form, the domain is exceptionally broad. Formal explanation requires nothing less than a complete theory of intelligence. For the sake of simplicity and tractability, we introduce three restrictions. First, we require that the rules be computable. This excludes item 8 in the list above. Second, we focus on cases in which the input and output list contain only natural numbers, which also excludes item 7. Third, we assume that the rules focus on the unchanging symbolic and structural properties of the inputs rather than requiring significant context or world knowledge, which excludes item 6 but includes the very similar item 5[1]. Items 1–5 are in fact all examples of list functions meeting these restrictions. Additionally, c_1 and c_3 from Chapter 1 are examples of concepts from this domain. c_2 is not, because it returns a lone natural number, rather than a list, though it could easily be adapted to return a list containing a single element. c_4 is similarly not a member of this domain because it takes two arguments rather than one, one of those arguments is not a list, and it returns a lone natural number. It, too, could be easily adapted by appending the first input to the second input and returning the output as a singleton list. These

[1]Figuring out which rules meet these conditions out is itself a list function: given a list of list functions, remove any which are not computable functions over lists of natural numbers.

sorts of manipulations might seem unnatural upon first reading, but our empirical studies in Chapter 5 show that people can reliably learn and accurately describe functions making use of these sorts of internal arguments.

As explored in the rest of this thesis, then, the set of list functions consists of computable functions over lists of natural numbers. These concepts are learned inductively from input/output pairs, written $i \rightarrow o$ for input i and output o. One might be given, for example:

$$
\begin{array}{ll}
[98,\ 15,\ 1,\ 7,\ 23,\ 76] & \rightarrow [76,\ 23,\ 7,\ 1,\ 15,\ 98] \\
[63] & \rightarrow [63] \\
[13,\ 8,\ 92] & \rightarrow [92,\ 8,\ 13] \\
[\ ] & \rightarrow [\ ] \\
[36,\ 19,\ 54,\ 82,\ 13,\ 12,\ 2] & \rightarrow [2,\ 12,\ 13,\ 82,\ 54,\ 19,\ 36]
\end{array}
$$

From these, the task would be to infer that the concept is item 3 above, reversing a list.

List functions share many of the best features of classic concept learning domains. First, list functions, like recent work on hand-written characters, novel environments, or stochastic trees, can be viewed as part of a hierarchical generative model (Figure 3-2). The highest level of this model is a type system describing the kinds of objects the world can contain. Given this, the model can sample the type of the functions in the domain—that is, the basic rule of the domain itself: concepts takes lists of natural numbers as input and return lists of natural numbers as output. This means that the restriction to use lists of natural numbers could be removed or changed to include functions over other kinds of lists. Given a type system, a model can also sample a language describing the rules for constructing various objects and the ways in which they behave. Conditioned on these, individual concepts can be sampled. Inputs can then be sampled conditioned on the concept and the language. These inputs may or may not be sampled pedagogically, but at the very least, they must satisfy any constraints imposed by the concept itself. For example, if the concept uses the first element of the list as an index into the rest of the list, the first element must be no greater than the length of the list. Finally, outputs can be computed conditioned on an input and the concept. If the concept happens to be non-deterministic, this last step is also non-deterministic, meaning that there is interesting distributional structure throughout the entire model. Not all samples from this model will necessarily include all levels. For example, the basic rule might be to

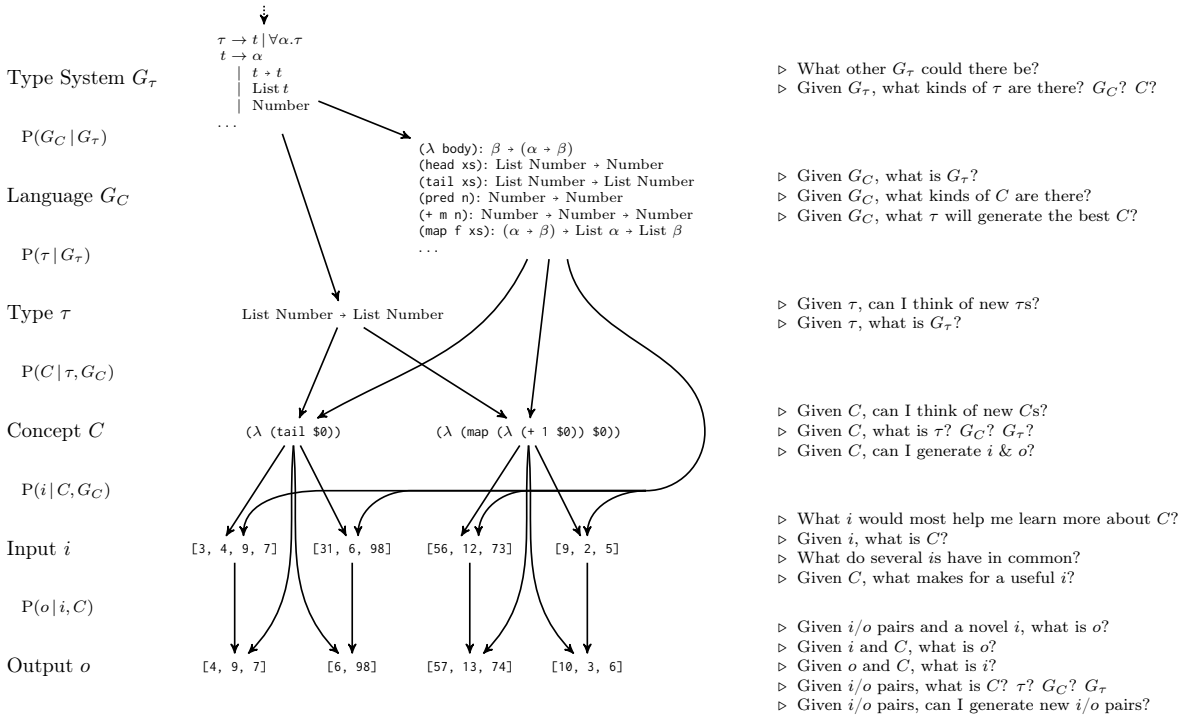


Figure 3-2: A hierarchical Bayesian model of list functions. The model include: a type system G_τ describing the kinds of possible objects; a language G_C defining how those objects are built and how they behave; a type τ describing what kind of object the target concept is; a concept C which is an expression in G_C of type τ; the input i to the concept; and the output o. The center shows how these various pieces are connected. The right lists representative questions that can be asked about each level of the hierarchy.

guess a specific number or list of numbers rather than a function from lists to lists. In that case, there would be no inputs or outputs, and the hierarchy would terminate at C.

While we focus on inferring concepts from input/output pairs, list functions, like other domains based on probabilistic generative models, support multiple tasks representing inference about different aspects of the model's hierarchy. For example, rather than providing outputs for novel inputs, subjects could be asked to provide verbal descriptions of a concept, or to write down a program computing the input/output relation. Rather than being given inputs for which to predict outputs, they could provide inputs for which they are interested in seeing outputs, similar to the 2-4-6 task. They could also be asked to provide entire input/output examples, in response to seeing either input/output examples, a verbal description of the concept, or a program. Other tasks might ask participants to describe what makes for a good input, or what a series of inputs have in common. Given some experience

in the domain, participants could also be asked to generate new concepts, and new examples for those concepts. They could even be asked to modify the basic rule of the domain about using lists of natural numbers, devising variants such as using a single number as input or output, or operating over lists of lists, and so on. Understanding list functions in terms of a hierarchical model exposes a range of possible tasks for investigating the full extent of what humans can learn in this domain.

Second, list functions are familiar. Numerate societies provide extensive experience with sequences, numbers, and sequences of numbers. Sequences have long been proposed to be a fundamental feature of cognition (Bartlett, 1958), and many people's early experiences with number explicitly come in the form of numerical sequences. For example, counting and skip-counting (e.g. counting by 2 or 10) are simple forms of numerical list manipulation (Fuson et al., 1982; Fuson, 1988). Other kinds of functions of sequences of numbers are also common. One must frequently sum a series of numbers, order a series of options, select top performers based on some quantitative measure, and so on.

As a result, many people have developed extensive background knowledge about the kinds of things one can do with numbers, with lists, and specifically with lists of numbers. They know how to perform arithmetic, compare magnitudes, extract digits at various positions in a number, and test for properties such as whether a number is even or odd, prime or composite. They know what it means to reverse a sequence, to arrange its elements in some sort of ascending or descending order, or to select members meeting certain criteria. They know how to select the first few or the last few, how to remove the first or last few, and how to extract a subsequence from the middle of a longer sequence. They can iterate over sequences to find maximums, minimums, means, and modes, or to count the number of elements. The list goes on: people have dozens of basic skills relevant to processing numerical sequences.

Like any concept learning task, familiarity and naturalness should not be confused with the task being identical to one that people perform in the world. Very few people naturally encounter stimuli that look quite like those used in Boolean concept learning, the number game, CopyCat, or ARC. Nonetheless, they do form rules for classification, and they do reason analogically. In the same way, people rarely sit down simply to puzzle out the

relationship between pairs of lists[2]. They do, however, interact regularly with sequences of numbers and the relationships that link them together. All these tasks differ from truly naturalistic stimuli in ways that make them simpler to study, but they also all capture important aspects of learning in daily life.

Third, despite their familiarity and naturalness, list functions remain engaging. In our pilot work, we frequently received comments about the tasks being enjoyable; many people called them puzzles or games. We conducted these experiments online using Amazon Mechanical Turk, where participants routinely complete behavioral experiments. Several people reported our task being the most interesting they had completed in weeks. Because these tasks provide a well-defined challenge which is typically just beyond their current skill level and provide regular feedback, list functions—and most classic concept learning domains—share much in common with the motivational states of both deliberate practice (Ericsson, 2006) and flow (Csikszentmihalyi, 1990). They present tasks in which participants both improve rapidly and are motivated to succeed.

Fourth, the stimuli of list functions have complex internal structure. Each list can have any non-negative number of elements, and the order and repetition of these elements is significant in a way that it would not be if the stimuli were bags (i.e. multi-sets) or sets. This structure has a starting point, an end point, and natural procedures for enumerating elements from either end. Moreover, each element can be one of unboundedly many numbers, each represented by a string of digits whose order and repetition conveys additional meaning according to a specific compositional structure. [2, 1, 3] is different from [1, 2, 3] despite identical elements, and [1, 2, 3] is itself different from [323, 324, 325], even though they both contain three successive numbers.

Other concept learning domains vary widely on this front. Boolean and n-ary stimuli, for example, are typically highly constrained such that each stimulus varies on only a few dimensions of just two or three levels each. Even when these stimuli have been grouped together (Piantadosi et al., 2016), they have been grouped in sets where order and repetition are insignificant. Numerical function learning and the number game both involve numbers but, again, there is at best a minor role for repetition or order. Text-based analogy problems

[2]Although, this probably does occur more often than chance encounters with Bongard problems.

make order and repetition important, but use alphabetic characters. There are only 26 characters, and they do not contain any internal structure. The stimuli for Bongard problems and Omniglot, on the other hand, consist of line drawings and can thus contain seemingly arbitrary internal structure. ARC problems occupy an interesting middle ground. They use two-dimensional grids of up to 30×30, a sort of list of lists, and the order and repetition of elements in this grid are important. As with text-based analogies, however, there are a small number of elements (i.e. 10) lacking internal structure. The 2-4-6 task and sequence prediction tasks are perhaps closest to list functions, typically using a sequence of numbers whose order and repetition play a significant role.

One particular strength of list functions stimuli is that they include natural numbers. Number, particularly natural number, is one of the richest formal conceptual systems which people possess. Its development builds on core cognitive systems, interacts with natural language, and supports the development of later conceptual systems like integers, rational numbers, and real numbers (see Carey, 2009, for detailed discussion). It is deeply integrated into daily life in numerate societies, such that people learn many interrelated numerical systems for counting objects, dealing with currency, and reasoning about time. This perhaps explains why it is a central feature of several classic domains including Wason's 2-4-6 problem, Tenenbaum's number game, function learning, and sequence prediction. It also features peripherally in Boolean concept and analogy domains via small number counting (e.g. How many stripes on the shape? How many repetitions of the first letter?). People can bring all this knowledge about the structure and interrelations of numbers to bear in learning list functions. Moreover, arranging numbers in sequences allows them to play many roles. They can be bare symbols, as in *reverse the order of the unique elements*, cardinal values, as in *sum the elements of the list*, or ordinal values, as in *remove the element indexed by the first element if it is less than 10*. This last example shows how the same concept might require taking multiple perspectives, in this case using numbers as ordinals, symbols, and cardinals. In this way, list functions expose interactions between different perspectives on the same object and how they influence learning. They make it possible to see how these various perspectives compete for a learner's attention and what sorts of cues suggest one role versus another to a given learner. For the other domains discussed here which feature

number centrally, this is not the case; they primarily feature numbers as cardinals.

Fifth, list functions vary widely in their learnability. Some are exceptionally simple, such as:

$$
\begin{array}{ll}
[1,\ 4,\ 23,\ 21] & \rightarrow [15,\ 8,\ 34,\ 6] \\
[60,\ 7] & \rightarrow [15,\ 8,\ 34,\ 6] \\
[8,\ 70,\ 3,\ 67,\ 54,\ 54,\ 6,\ 97,\ 7] & \rightarrow [15,\ 8,\ 34,\ 6] \\
[3,\ 3,\ 6,\ 55,\ 63,\ 7] & \rightarrow [15,\ 8,\ 34,\ 6] \\
[29] & \rightarrow [15,\ 8,\ 34,\ 6]
\end{array}
$$

which encodes a simple constant function: every input produces the output [15, 8, 34, 6]. Even after considering just a single example, the output seem sufficiently unlikely to be a function of the input to make the correct hypothesis a likely candidate. Other functions, by contrast are remarkably difficult, such as:

$$
\begin{array}{ll}
[29,\ 88,\ 44,\ 75,\ 5,\ 17,\ 36,\ 0,\ 89,\ 31] & \rightarrow [29,\ 88,\ 17,\ 36] \\
[54,\ 4,\ 7,\ 43,\ 8,\ 97,\ 25,\ 5,\ 0] & \rightarrow [54,\ 43,\ 5] \\
[24,\ 41,\ 96,\ 14,\ 93,\ 47] & \rightarrow [41,\ 96] \\
[19,\ 81,\ 1,\ 53,\ 85,\ 3,\ 97] & \rightarrow [\ ] \\
[3,\ 76,\ 20,\ 11,\ 86,\ 8,\ 5,\ 94] & \rightarrow [3,\ 76,\ 11,\ 86,\ 5]
\end{array}
$$

The concept here is to keep only elements followed by an even element. Learning this dependency structure has proven very difficult for people. Even harder rules are possible. One could, for example, create multiple conditions based on list length, or introduce long range dependencies between elements that are several items apart in the list, or select elements using large numbers that must first be interpreted modulo the length of the list. Many of these are likely to be sufficiently difficult to be practically unlearnable for a novice subject in any reasonable timeframe. They might, however, be excellent families of concepts to use in exploring curriculum-based learning. As with c_1 in Chapter 1, the right series of antecedent concepts might make any one of these difficult concepts significantly easier to learn.

While there are many concepts likely to sit either near the floor or the ceiling of human performance for typical learners, there is also a large set which smoothly connects these two points. Here are three, roughly in order of increasing difficulty:

[7, 51, 94, 72, 88, 19] → [7, 19, 51, 72, 88, 94]
[2, 0, 92, 21, 33] → [0, 2, 21, 33, 92]
[75, 32, 46, 71, 49, 60] → [32, 46, 49, 60, 71, 75]
[10, 12, 11, 8, 9, 7] → [7, 8, 9, 10, 11, 12]
[52, 87, 27, 25] → [25, 27, 52, 87]

[2, 0, 92, 21, 33] → [3, 2, 95, 25, 38]
[7, 51, 94, 72, 88, 19] → [8, 53, 97, 76, 93, 25]
[75, 32, 46, 71, 49, 60] → [76, 34, 49, 75, 54, 66]
[10, 12, 11, 8, 9, 7] → [11, 14, 14, 12, 14, 13]
[52, 87, 27, 25] → [53, 89, 30, 29]

[36, 45, 15, 70, 85, 2] → [45]
[90, 54, 16, 3] → [16]
[69, 63, 50, 8, 86, 17, 0, 80, 19, 7] → [0]
[62, 0, 1] → [62]
[95, 51, 9, 93, 6, 5, 31, 47, 4] → [93]

The first of these sorts the list and was likely recognizable after considering just the first example or two. The second adds the index of each element to the value of the element itself and might have required a few examples to learn or perhaps very careful examination of a single example. The third removes all but the element whose position is indicated by the last element in the list. Learning this typically requires carefully studying multiple examples. List functions thus hit a sweet spot that both exposes the abilities and the limitations of human learners. Many classic domains are similar, such as the general class of Boolean and n-ary concepts over a set of stimuli, visual and textual analogies, and sequence prediction tasks. Function learning is also a fairly large class of problems, though the relatively simpler stimuli (i.e. inputs and outputs which are both a single number) might limit the number of concepts which remain easily learnable. Nearly all the structure in complex concepts remains latent. By contrast, some domains were explicitly designed with a definite ceiling on difficulty. Several studies of Boolean concepts, for example, limit the complexity or form of the formulae studied (Bruner et al., 1956; Shepard et al., 1961; Feldman, 2000). Similarly,

the number game selects from a finite set of hypotheses chosen to be relatively straightforward to induce, and the characters in Omniglot have been used in actual alphabets and are thus likely bounded in total complexity.

Sixth, list functions make it possible to investigate the full range of algorithmic thinking. For example, they support humans' excellent abilities for detecting structure-based patterns, as in:

$$[0, 5, 2, 4, 3, 1, 6, 8, 7, 9] \rightarrow [2, 5, 0, 4, 3, 6]$$
$$[9, 8, 1, 2, 7, 4, 5, 6, 3, 0] \rightarrow [1, 8, 9, 4, 7, 5]$$
$$[7, 9, 0, 2, 6, 8, 3, 5, 1, 2] \rightarrow [0, 9, 7, 4, 6, 3]$$
$$[6, 7, 1, 3, 2, 0, 8, 9, 4, 5] \rightarrow [1, 7, 6, 4, 2, 8]$$
$$[5, 3, 9, 8, 0, 7, 2, 1, 4, 6] \rightarrow [9, 3, 5, 4, 0, 2]$$

This concept involves multiple steps, but each is transparently reflected in the final output, which contains element 3, element 2, element 1, the number 4, element 5, and element 7, in that order. This concept can be understood as a form of something studied in computer science as *pattern matching*. An abstract template or pattern is used to extract various pieces of the input (i.e. elements 1, 2, 3, 5, and 7), and these pieces are then used to construct the final output. This sort of pattern-based reasoning is common in many programming languages and has been a key part of artificial intelligence systems for decades (Sussman, 1973). List functions also support case-based and conditional reasoning, as in the following:

$$[4,\ 8,\ 3,\ 7,\ 8] \rightarrow [4,\ 8,\ 3,\ 7,\ 8,\ 3]$$
$$[5,\ 8,\ 2,\ 9,\ 0,\ 0] \rightarrow [5,\ 8,\ 2,\ 9,\ 0,\ 0,\ 9]$$
$$[7,\ 0] \rightarrow [7,\ 0]$$
$$[9,\ 1,\ 1,\ 5,\ 1,\ 6,\ 5,\ 6] \rightarrow [9,\ 1,\ 1,\ 5,\ 1,\ 6,\ 5,\ 6,\ 9]$$
$$[4,\ 7,\ 4,\ 6] \rightarrow [4,\ 7,\ 4,\ 6]$$
$$[3,\ 8,\ 5] \rightarrow [3,\ 8,\ 5,\ 3]$$
$$[4,\ 2,\ 2,\ 6,\ 8,\ 5,\ 4] \rightarrow [4,\ 2,\ 2,\ 6,\ 8,\ 5,\ 4]$$
$$[4,\ 8,\ 9,\ 2,\ 5,\ 7,\ 1,\ 0,\ 6,\ 2] \rightarrow [4,\ 8,\ 9,\ 2,\ 5,\ 7,\ 1,\ 0,\ 6,\ 2,\ 9]$$
$$[5,\ 7,\ 7,\ 3,\ 0,\ 2,\ 0,\ 6,\ 1] \rightarrow [5,\ 7,\ 7,\ 3,\ 0,\ 2,\ 0,\ 6,\ 1,\ 3]$$

This concept appends a 3 if the input contains 3, otherwise appending a 9 if the input contains a 9, otherwise appending nothing. These conditions can be nested and made arbitrarily

complex (e.g. *if the first element is prime or even, xor the second element is greater than the first or equal to 0, then reverse the list.*). List functions can also rely on recursive and iterative reasoning. This is evident in some of the concepts already considered, such as reversing a list, or scanning a list to see if it contains certain elements. Here are examples from two more explicitly iterative/recursive concepts:

[87, 58, 78, 1, 51] → [78, 85, 87, 10, 15]
[9, 76, 3, 4, 35, 77, 73, 91] → [90, 67, 30, 40, 53, 77, 37, 19]
[32, 13, 52] → [23, 31, 25]
[65, 36, 28, 7, 41, 39, 93] → [56, 63, 82, 70, 14, 93, 39]
[5, 61, 72, 8, 6, 98, 22, 0, 50] → [50, 16, 27, 80, 60, 89, 22, 0, 5]

[68, 40, 6, 1, 5, 17, 0, 85, 82, 9] → [68, 40, 6, 0, 82]
[7, 43, 66, 79, 68, 33, 8] → [66, 68, 8]
[99, 65, 46, 2, 6, 23, 78, 1, 58] → [46, 2, 6, 78, 58]
[4, 75, 8, 63, 18, 3, 51, 7] → [4, 8, 18]
[54, 97, 49, 5, 6, 35, 2, 1, 70] → [54, 6, 2, 70]

The first concept iterates over the list and swaps the digits of each element. It is an example of an algorithmic pattern known as a *map*. In a map, a function is applied to each item. In this case, the function being mapped reverses the digits of a number. The second concept combines iteration and conditional reasoning, removing items if they are odd and keeping them if they are even. This pattern is called a *filter*. It iteratively tests each element and filters out any which fail the test. In this case, the test is simply whether an element is even. Here are examples of two other common iterative/recursive patterns:

[12, 6, 6, 7, 12, 6] → [12, 18, 25]
[4, 6, 1, 6, 1, 0, 9] → [4, 10, 11, 11, 20]
[20, 1, 20, 1, 1, 20] → [20, 21]
[7, 5, 1, 5, 0, 7, 5] → [7, 12, 13, 13]
[2, 1, 1, 8, 15, 9, 8, 15, 2] → [2, 3, 11, 26, 35]

[6, 13, 1] → [6, 7, 8, 9, 10, 11, 12, 13]
[34, 67, 3] → [34, 37, 40, 43, 46, 49, 52, 55, 58, 61, 64, 67]
[0, 50, 5] → [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
[6, 18, 2] → [6, 8, 10, 12, 14, 16, 18]
[22, 62, 10] → [22, 32, 42, 52, 62]

The first concept transforms an input into the cumulative sum of its unique elements, in order of first appearance. Computing the unique elements and computing a cumulative sum are both examples of a pattern known as a *fold*. When mapping over a list, the function is applied directly to each element. There is no hidden state on which the function can rely. Folding, however, allows iteration with state. It maintains a data structure that can encode information about the work that has been done during past iterations. When the iteration is complete, this structure is returned. The entire list is folded into the structure. In the case of computing unique elements, the state tracks the set of elements seen so far. In the case of cumulative sums, it tracks the sequence of partial sums computed so far. Other common example include computing the length of the list, the maximum, and the minimum. Each of these folds the list into a single value: the number of elements, the largest element, or the smallest element, respectively. The second concept skip-counts from the first element to the second element, incrementing according to the third element. This concept is an example of a pattern known as an *unfold*. If a fold consumes a list to produce a single value (which could itself be a list), an unfold uses a single value (which could also be a list) to iteratively generate a list. In this case, it takes a triple describing the parameters of the skip-count, and then iteratively constructs each element of the output. Folding and unfolding are very powerful patterns that, when combined, can describe an immense range of iterative/recursive concepts (Gibbons, 2003), including mapping and filtering. They also both demonstrate ways for list functions to rely on internal state, maintaining latent structures which are necessary to explain the overall function.

We could continue with demonstrations of tasks requiring any number of other core algorithmic concepts—hashing, non-determinism, parameter use, and so on—because list functions are in fact computationally universal. The Turing machine (Turing, 1936) is a formal model of computation built around the rule-based manipulation of a list of symbols.

It is one of the standard models of computation, such that it is strongly suspected that Turing machines can represent every possible computable function over the natural numbers. They do so by representing a computation as a finite set of rules for manipulating the list of symbols. Because there must be finitely many rules, and the format of the rules strongly limits their complexity, any given Turing machine can deal with at most finitely many symbols. If we treat natural numbers as symbols, list functions provide infinitely many symbols and so can certainly represent any finite number of symbols. Given the right set of rules, then, the list function domain over natural numbers can be used to model any Turing Machine and so is computationally universal. Moreover, there are a class of Turing machines which are themselves *universal* in that they treat their input as a description of another Turing machine and its input. These universal machines can simulate any other possible Turing machine. This implies that there are individual list functions which are themselves Turing complete, capable of representing any possible computation.

Some of the classic concept learning domains share this computational richness. Analogy problems, sequence prediction, and the 2-4-6 task can all specify algorithmically sophisticated problems in natural ways. Numerical functions can also include complex algorithms[3], though the way this is typically done—encoding complex structures and functions as numbers—would likely feel unnatural to typical learners. Many other domains strongly limit their overall computational complexity. Omniglot, simulated worlds, and the number game, for instance, were all designed to capture learning in which algorithmic complexity is not a prominent feature. Boolean and n-ary concepts, particularly first-order concepts, can be Turing-complete, as demonstrated by the logic programming paradigm used in languages like Prolog (Bratko, 2001). As studied in cognitive psychology, however, the domain supports limited algorithmic content, mostly focused on conditional reasoning and basic forms of quantification.

Seventh, despite all their computational power, operations over lists of numbers are easily formalized. Much of the earliest work on automated programming and program synthesis revolved around manipulating lists of data (Smith, 1984; Biermann, 1978; Green, 1981;

[3]Early formal definitions of computatability focused explicitly on computations over natural numbers, so the set of computable functions from integers to integers is a paradigmatic example of Turing-completeness.

Shaw et al., 1975; Green et al., 1974). They continue to be a tractable domain of interest in artificial intelligence and machine learning today (Feser et al., 2015; Osera & Zdancewic, 2015; Polikarpova et al., 2016; Cropper et al., 2019). At first glance, this appears to be mere historical accident. Much of the early work on program synthesis used the lisp programming language introduced by McCarthy (1960). Lisp uses lists as a model of both programs and the data those programs manipulate. It treats programs as nested lists of symbols defining rules for manipulating other lists of symbols[4]. Learning lisp programs then largely consists of learning list functions, making list functions one of the early target domains for program synthesis.

Looking more deeply, however, the reason list functions appear so early, and continue to be studied today in artificial intelligence, is perhaps because numbers and lists are among the simplest possible recursive structures. A list containing objects of type `X` can be defined, for example, as either an empty list or a list containing an `X` prepended to a list of `X`s. This prepending operation is called `Cons` for historical reasons, so this definition can be formalized as:

```
List X = Empty | Cons X (List X).
```

Natural numbers can be defined using an even simpler unary encoding where numbers are either `Zero` or the `Successor` of a number. That is, numbers are like lists that cannot carry data. They can be written as:

```
Number = Zero | Successor Number.
```

These four symbols—`Empty`, `Cons`, `Zero`, and `Successor`—define two recursive structures. Using them according to the two rules given above makes it possible to construct any list of

[4]Technically, the basic syntax of lisp consists of expressions that are either atomic symbols, such as `A` and `B`, or ordered pairs of expressions. The pair of `A` and `B`, for example, is written `(A . B)`. These expressions can be thought of as trees. Each atom is a tree consisting of a single node, while an ordered pair represents a binary tree whose left and right children are the first and second elements of the pair, respectively. Most of the time, however, these trees are interpreted as lists, so much so that lisp stands for LISt Processor. In fact, the first thing McCarthy (1960) does after describing these pair-based expressions is define how they can be used to represent lists. Pairs are nested such that the first element represents an element of the list, and the second element represents the rest of the list. A special symbol `NIL` represents the empty list. For example, the list [1, 2, 3, 4, 5] is written as `(1 . (2 . (3 . (4 . (5 . NIL)))))`.

numbers, any list of lists of numbers, and so on. The set of lists we can build using these four symbols is unboundedly large and varied.

If `Empty`, `Cons`, `Zero`, and `Successor` act as *constructors*, i.e. the symbols by which lists and numbers are built, it is also useful to define *destructors*, i.e. symbols by which lists and numbers are decomposed into their basic parts. For lists, the fundamental destructors are `Head`, which returns the first element of a list, and `Tail`, which returns all but the first element. `Head` is undefined over `Empty`, while `Tail` returns `Empty` again. For numbers, the basic destructor is `Predecessor`, which removes a single `Successor`; it returns `Zero` for input `Zero`. That is, the destructors obey these rules:

```
Head (Cons X Y) = X
Tail (Cons X Y) = Y
Tail Empty = Empty
Predecessor (Successor X) = X
Predecessor Zero = Zero
```

Formalizing the entire domain of list functions requires not only defining constructors and destructors, but also all possible operations over those structures. Since the space is Turing-complete, that means embedding lists and numbers into a Turing-complete language. The lambda calculus is a natural choice (Church, 1932; Barendregt et al., 1984). It is a simple Turing-complete formalism which represents computation through function abstraction and application. Function abstraction introduces variable bindings, so an expression, e, composes variables, abstractions, and applications as follows:

$$
\begin{array}{lll}
e \to x & & \text{(variable)} \\
\quad | & \lambda\, x.\, e & \text{(abstraction)} \\
\quad | & e\; e & \text{(application)}
\end{array}
$$

Expressions are considered equal up to variable renaming, and parentheses are used freely to clarify grouping. They are evaluated using β-reduction, $X \to Y$, which formalizes the notion of applying a function to an argument. When a function abstraction is applied to an argument, $((\lambda x.M)\ N)$, β-reduction maps the variable, x, to the argument, N, written

Name	Description	Definition
`True`	Boolean true	$\lambda x.\lambda y.x$
`False`	Boolean false	$\lambda x.\lambda y.y$
`Pair`	ordered pair (x, y)	$\lambda x.\lambda y.\lambda f.f\ x\ y$
`First`	x in the pair (x, y)	$\lambda p.p$ `True`
`Second`	y in the pair (x, y)	$\lambda p.p$ `False`
`Empty`	empty list	$\lambda x.$`True`
`Cons`	prepend element to list	`Pair`
`Head`	first element in list	`First`
`Tail`	all but first element in list	`Second`
`Zero`	natural number 0	$\lambda f.\lambda x.x$
`Successor`	successor of n	$\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
`ShiftInc`	$(m, n) \to (n, n+1)$	$\lambda x.$`Pair` (`Second` x) (`Successor` (`Second` x))
`Predecessor`	predecessor of n	$\lambda n.$`First` (n `ShiftInc` (`Pair` 0 0))
`Y`	recursive operator	$\lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$

Table 3.1: Church encodings for Booleans, lists, numbers, and Y, a fixpoint operator used to implement recursion. In this encoding, lists are nested pairs, so the encoding for `Pair`, `First`, and `Second` are reused for `Cons`, `Head`, and `Tail`, respectively.

$\sigma \equiv \{x \mapsto N\}$. This substitution, σ, is then applied to M, written σM, which replaces occurrences of x with N throughout M. That is:

$$(\lambda x.M)\ N\ \to \{x \mapsto N\}M$$

For example:

$$\begin{aligned}(\lambda v.\lambda w.v\ w\ v)\ (\lambda x.\lambda y.x)\ (\lambda z.z) &\to (\lambda v.(\lambda x.\lambda y.x)\ w\ (\lambda x.\lambda y.x))\ (\lambda z.z)\\ &\to (\lambda x.\lambda y.x)\ (\lambda z.z)\ (\lambda x.\lambda y.x)\\ &\to (\lambda y.\lambda z.z)\ (\lambda x.\lambda y.x)\\ &\to (\lambda z.z)\end{aligned}$$

There are several well-known ways to encode numbers and lists into lambda calculus (Koopman et al., 2014), and definitions for Boolean data structures and basic iterative/recursive procedures are also well known. One standard method known as the Church encoding[5] is

[5]Alonzo Church introduced both the lambda calculus and this specific method of encoding data types.

shown in Table 3.1. The basic idea is to define a function for each constructor and destructor, such that the behavior of each function satisfies the rules given above. Because the lambda calculus is Turing-complete, these embeddings then allow lambda calculus to represent any computation taking a list as input and returning a list as output and can thus represent any list function. As a simple example, consider a function, `StartSeven`, which takes a list and replaces the first element with 7. Here is a definition and its application to the list $[1,\ 2,\ 3]$[6]:

$$\texttt{StartSeven} \equiv (\lambda x.\ \texttt{Cons}\ 7\ (\texttt{Tail}\ x))$$

$(\texttt{StartSeven}\ [1,\ 2,\ 3]) \equiv ((\lambda x.\texttt{Cons}\ 7\ (\texttt{Tail}\ x))\ [1,\ 2,\ 3])$		Definition of `StartSeven`
	$\rightarrow(\texttt{Cons}\ 7\ (\texttt{Tail}\ [1,\ 2,\ 3]))$	Reduction
	$\equiv ((\lambda x.\lambda y.\lambda f.f\ x\ y)\ 7\ (\texttt{Tail}\ [1,\ 2,\ 3]))$	Definition of `Cons`
	$\rightarrow((\lambda y.\lambda f.f\ 7\ y)\ (\texttt{Tail}\ [1,\ 2,\ 3]))$	Reduction
	$\rightarrow(\lambda f.f\ 7\ (\texttt{Tail}\ [1,\ 2,\ 3]))$	Reduction
	$\equiv (\lambda f.f\ 7\ ((\lambda p.p\ \texttt{False})\ [1,\ 2,\ 3]))$	Definition of `Tail`
	$\rightarrow(\lambda f.f\ 7\ ([1,\ 2,\ 3]\ \texttt{False}))$	Reduction
	$\equiv (\lambda f.f\ 7\ ((\texttt{Cons}\ 1\ [2,\ 3])\ \texttt{False}))$	Definition of a list
	$\equiv (\lambda f.f\ 7\ (((\lambda x.\lambda y.\lambda f.f\ x\ y)\ 1\ [2,\ 3])\ \texttt{False}))$	Definition of `Cons`
	$\rightarrow(\lambda f.f\ 7\ (((\lambda y.\lambda f.f\ 1\ y)\ [2,\ 3])\ \texttt{False}))$	Reduction
	$\rightarrow(\lambda f.f\ 7\ ((\lambda f.f\ 1\ [2,\ 3])\ \texttt{False}))$	Reduction
	$\rightarrow(\lambda f.f\ 7\ (\texttt{False}\ 1\ [2,\ 3]))$	Reduction
	$\equiv (\lambda f.f\ 7\ ((\lambda x.\lambda y.y)\ 1\ [2,\ 3]))$	Definition of `False`
	$\rightarrow(\lambda f.f\ 7\ ((\lambda y.y)\ [2,\ 3]))$	Reduction
	$\rightarrow(\lambda f.f\ 7\ [2,\ 3])$	Reduction
	$\equiv [7,\ 2,\ 3])$	Definition of `Cons`

This is not to say that people represent list functions either using lambda calculus or the particular structures given here. They almost certainly use more complex representations building on ancient core cognitive representations and an LOT with a far richer set of primitives. The discussion here merely demonstrates that lists and list functions can be formally defined extremely simply and without extensive hand-engineering. This simple definition provides a minimal starting point for formal investigation. In practice, numbers and lists are

[6]For legibility, we use $0 \equiv$ `Zero`, $1 \equiv$ `Successor Zero`,..., and bracket notation for lists, where $[\ ] \equiv$ `Empty`, $[1] \equiv$ `(Cons 1 Empty)`, $[1,\ 2] \equiv$ `(Cons 1 (Cons 2 Empty))`,...

such fundamental data structures that they have both been deeply explored in computer science and software engineering. Programming languages often permit representing numbers in a variety of bases beyond the unary representation discussed here. Binary, decimal, and hexadecimal are common choices, and each can be encoded using a variety of representations. Similarly, the list structure we discuss here is called a singly-linked list: any given list provides access to its first element and a *link* pointing to the rest. Computer scientists have developed many other implementations with different properties, including: resizable vectors, doubly-linked lists, zipper lists, skip lists, difference lists, and functional random-access lists (Okasaki, 1999; Cormen et al., 2009). Most general-purpose programming languages contains mature libraries for working with lists and with numbers; these libraries define dozens of operations over these structures.

Other domains which rely primarily on discrete data structures are similarly straightforward to implement, such as: Boolean and n-ary concepts, numerical and Boolean rule learning tasks, function learning, sequence prediction, text-based analogies, and ARC. Those which rely on fine visual detail, such as Bongard problems and Omniglot's hand-written characters, require a great deal of computational machinery and hand-engineering to formalize (e.g. Lake et al., 2015). The inference of probabilistic programs specifying, for example, simulated worlds or hand-written characters occupies a sort of middle ground. It too, requires a significant amount of computational machinery, though less than for arbitrary images, and the objects to be learned (i.e. the probabilistic programs) are themselves simpler than the full set of weights for, e.g. a deep network.

3.2 Capturing classic domains and developmental case studies

List functions compare well with classic concept learning tasks on a broad spectrum of features and are well situated as a tool for studying human concept learning. In this section, we show how five of these classic tasks can actually be seen as natural subclasses of list functions. At the same time, list functions naturally encode problems which are unlikely to appear in

any of these classic tasks. We also show how list functions can be used to capture the essential structure of several key tasks from developmental psychology, including *Give-a-Number* and *How-Many* from the counting literature (Wynn, 1992b), simple arithmetic problems (e.g. Siegler & Jenkins, 1989), and seriation tasks (Piaget, 1952; Inhelder & Piaget, 1964). List functions thus provide a unifying way to examine a very general class of concepts spanning multiple kinds of inductive learning studied in cognitive and developmental psychology.

One of the simplest domains to translate to the list domain is function learning. In their most basic form, these problems can be modeled using input and output lists that each contain a single element, such as:

$$\begin{aligned}[12] &\rightarrow [20] \\ [6] &\rightarrow [8] \\ [30] &\rightarrow [56]\end{aligned}$$

The function here is $f(x) = 2x - 4$. Most numerical function learning tasks, however, rely on rational numbers. These could be encoded using natural numbers by giving inputs and outputs of two elements. The number 35.5, for example, could either be represented as the numerator and denominator of the rational, [71, 2], or the integral and fractional parts, $[35, 5]$. The latter is relatively similar to how numbers are written in many parts of the world, though conceptually cumbersome. A better alternative might simply be to extend the domain to allow rational numbers. Doing so would provide a close match to the original task. List functions also provide interesting options for generalizing the task, such as providing multiple inputs at a time, changing the function based on the position of a number in the list, or creating multivariate functions which require multiple elements as input (e.g. $ax_1 + bx_2 + cx_3 + d$).

Sequence prediction tasks, where participants are given a partial sequence and asked to produce the next element, are also straightforward to interpret as list functions. Given some input list, the task is to provide as output a singleton list containing the next element, as in:

$$\begin{aligned}[2,\ 4,\ 8] &\rightarrow [16] \\ [6,\ 12,\ 24] &\rightarrow [48] \\ [5,\ 10,\ 20,\ 40] &\rightarrow [80]\end{aligned}$$

Alternatively, one could append the new element to the input, reproducing the input sequence with one additional member. At their core, these tasks ask individuals to infer and extend some generative process from examples. They are, in that sense, a particular kind of unfolding[7]. List functions provide a way to explore these and other kinds of unfolding, such as generating a list from arguments rather than examples as was done in the previous section, within a single framework.

Rule learning tasks are similarly easy to translate. The 2-4-6 task, for example, could be encoded by providing an three-element list as input and returning a single-element list containing 1 if the input is included in the target concept, and 0 otherwise. The following, for example, provides examples based on the original task, *numbers in ascending order*:

$$\begin{aligned}[2,\ 4,\ 6] &\rightarrow [1]\\ [3,\ 10,\ 19] &\rightarrow [1]\\ [6,\ 4,\ 2] &\rightarrow [0]\end{aligned}$$

The number game has a slightly different structure. Participants are given elements taken from the target concept and must then generalize to predict which additional elements are in the set defined by the concept. This can be framed as a list function in which each input contains a single number, and the output is either 1 or 0 to signify inclusion or exclusion from the set, respectively. For example, the following are examples for the concept *divisible by 10*

$$\begin{aligned}[10] &\rightarrow [1]\\ [15] &\rightarrow [0]\\ [70] &\rightarrow [1]\end{aligned}$$

In an experiment, learners would first be given only positive examples. They would then be asked to make predictions about the remaining elements. Filtering problems, like those discussed in the previous section, provide another sort of rule learning task similar in many ways to the Boolean concept learning paradigm in Piantadosi et al. (2016). List functions

[7]They also present a sort of meta-induction problem. Each input list provides examples of a slightly different generative process unfolding. When observing several input/output pairs, learners must infer an abstract process that they can fit to each example.

provide an interesting generalization of this task: it supports concepts where individual elements—e.g. the first element, or the first few elements—can be interpreted as arguments which parameterize how to process the set described by the remainder of the list.

Analogy problems are perhaps most directly translated from text-based domains like seek-whence, jumbo, and CopyCat. In that case, one simply substitutes numbers for characters, most obviously using $\mathrm{A} = 1, \mathrm{B} = 2, \ldots$ Then, the introductory example of *replace the last element with its successor* would have examples like:

$$
\begin{aligned}
&[1,\ 2,\ 3] && \rightarrow [1,\ 2,\ 4] \\
&[5,\ 9,\ 3] && \rightarrow [5,\ 9,\ 4] \\
&[28,\ 16,\ 55,\ 78,\ 0,\ 41] && \rightarrow [28,\ 16,\ 55,\ 78,\ 0,\ 42]
\end{aligned}
$$

Given the number of relations defined over number which are not available for alphabetic characters, this encoding makes it possible to test a much wider variety of analogies. ARC problems can be seen as a special way of presenting functions over lists of lists of the numbers 0–9, a way that takes advantage of our visual intelligence. At their core, however, they are also another form of list processing problem, and many of the dynamics observed in ARC problems can be studied in some form even in simple one-dimensional lists.

Finally, there are multiple ways to encode Boolean and n-nary formulae as list functions. One option is to focus on formulae defined over properties of numbers themselves (e.g. number in the tens digit, number in the ones digit, even or odd, prime or composite), though these properties might not be as obvious as visual stimuli varying in shape, number, color, or size. In this case, the input would be a single arbitrary number and the output would be a single list of either 1 or 0 encoding of whether the formulae was true or false, respectively. Another option would be to allow each position in the list to encode a different feature. If each feature were Boolean, then the input would be a list of 0s and 1s, one for each feature, and the output would again be a single 0 or 1, depending on whether the input satisfied the formula. This might look like the following for the concept $(x_1 \wedge x_3) \vee (x_2 \wedge \neg x_4)$:

$$[0,\ 1,\ 0,\ 0] \rightarrow [1]$$
$$[1,\ 0,\ 1,\ 0] \rightarrow [1]$$
$$[1,\ 1,\ 0,\ 0] \rightarrow [0]$$
$$[1,\ 1,\ 0,\ 1] \rightarrow [0]$$

This encoding separates the Boolean concept itself from the more naturalistic object categorization setting in which it is typically studied. This is potentially problematic, in that it divorces explicit feature-based logical reasoning from any sort of implicit reasoning that might be supported by visual cognition or systems for creating and reasoning about kinds. It is potentially beneficial for the same reason: it helps to disentangle the roles of multiple systems that might contribute to category learning overall. List functions extend these sorts of problems in multiple ways. Each element need not be Boolean or n-ary. They can be arbitrary numbers. These numbers can then be combined using arithmetical as well as logical operations. This blurs the line between Boolean concept learning and multivariate function learning. Perhaps more importantly, though, lists permit a variety of algorithmic tools beyond either logical connectives or arithmetic operators. Boolean concepts can be defined which make use of iteration, recursion, and pattern-based reasoning and thereby push studies of human concept learning into new territory.

In addition to classic concept learning tasks from cognitive psychology, list functions can be used to model key tasks from developmental psychology. These encodings simplify the learning tasks to make formal modeling feasible. In doing so, they allow modelers to grapple with important psychological questions about the dynamics of learning and the representational resources that support those dynamics. We specifically consider four examples: seriation, the *Give-a-Number* and *How-Many* tasks from the counting literature, and small number arithmetic.

Seriation is a basic developmental task that has been studied since Piaget (1952): given a set of orderable objects, arrange them so that they are in order. The task has been modeled several times and basic empirical phenomena are well known (Young, 1976; Mareschal & Shultz, 1999; Schultz & Vogel, 2004; McGonigle-Chalmers & Kusel, 2019). This task is straightforward to express as a list function. Given an array of evenly spaced numbers in random order, rearrange the elements so they are in either ascending or descending order:

$$[25,\ 20,\ 30,\ 15,\ 10] \rightarrow [10,\ 15,\ 20,\ 25,\ 30]$$
$$[15,\ 19,\ 11,\ 17,\ 13] \rightarrow [11,\ 13,\ 15,\ 17,\ 19]$$
$$[9,\ 5,\ 2,\ 6,\ 4,\ 0] \rightarrow [0,\ 2,\ 4,\ 5,\ 6,\ 9]$$

A model working on this task could be given a relatively minimal LOT, such that it would have to develop notions of size, comparison, and order, much as children are hypothesized to do (Greenfield et al., 1972). They might also need to develop the notion that list elements can themselves be lists—i.e. that there is such a thing as a list of lists—would could be useful in learning to seriate and combine sub-arrays, as children frequently do. Finally, there are many sorting algorithms; studying the development of sorting in such a constrained environment might provide insight into what learning mechanisms support the development of the algorithms seen in children as opposed to those favored in computer science for their formal properties (Cormen et al., 2009).

Wynn (1990b) describes two number learning tasks—*Give-a-Number* and *How-Many*—which have since become standard parts of the counting and number learning literature. Both test a child's ability to reason about set cardinalities through counting. In *Give-a-Number*, children are given a large collection of objects and asked to provide a subset containing a specific number of objects. This can be modeled as a list function which takes as input a long list and selects a sublist. To encode both the set and the query, the first item in the list can be used to encode the number of items to retain from the remainder of the list:

$$[3,\ 0,\ 0,\ 0,\ 0,\ 0] \rightarrow [0,\ 0,\ 0]$$
$$[6,\ 0,\ 2,\ 4,\ 9,\ 5,\ 0,\ 1,\ 4] \rightarrow [0,\ 0,\ 2,\ 1,\ 5,\ 4]$$
$$[4,\ 6,\ 1,\ 0,\ 2,\ 3] \rightarrow [6,\ 1,\ 0,\ 2]$$

As shown in the examples, the objects to be given can either be treated as identical or as distinct, and the order in which objects are given can be ignored so long as the correct number of items is provided. In *How-many*, children are given a set of items and asked to name its cardinality. This can be modeled simply as computing the length of a list of objects:

$$[3,\ 3,\ 3,\ 3,\ 3] \rightarrow [5]$$
$$[4,\ 1,\ 2] \rightarrow [3]$$
$$[4,\ 0,\ 9,\ 2] \rightarrow [4]$$

One potential limitation of both these encodings is that numbers play roles as both symbolic objects and as cardinalities. While this is a benefit for the domain overall, it is in this case unlike the experimental tasks. For children, the number words used to label sets are clearly different from the toys and small objects composing the sets themselves. It is straightforward to extend the domain to include lists with multiple kinds of objects, including bare symbols. In a modeling setting, one would presumably limit a model's LOT so that it did not extend to treat large, multi-digit numbers as numbers, since the task is to model early number knowledge. In that case, using large numbers to represent objects would be functionally similar to introducing entirely new kinds of symbols.

Finally, small number addition problems of the kind discussed in Chapter 2 are also straightforward to encode as list functions. In this case, the input is a pair of numbers, and the output is a singleton containing the sum:

$$[5,\ 2] \rightarrow [7]$$
$$[1,\ 8] \rightarrow [9]$$
$$[3,\ 3] \rightarrow [6]$$

All these tasks differ in important ways from the actual tasks which children face. Casting them as list functions is valuable, however, for at least three reasons. First, it isolates the development of a symbolic, language-like conceptual system for describing the dynamics of the task from, for example, perceptual and motor components. This is useful because these dynamics are, even in isolation, likely to be computationally sophisticated and challenging to model. Second, the detailed empirical studies which already exist for these domains can be used to help guide the kinds of learning mechanisms and LOTs which might be explored in computational models of these and other tasks. Third, it suggests that a detailed exploration of list functions is likely to shed light on aspects of learning that go far beyond the domain itself, extending even to central questions in cognitive development.

3.3 Conclusion

The domain of list functions is very simple to describe but incredibly rich in its scope, providing structured stimuli and a large number of natural problems that can be scaled to contain arbitrary computational complexity. Concepts describe functions which take a list as input and return a list as output, where both lists contain only natural numbers. It compares well with classic concept learning domains on a broad set of features and is well situated as a tool for studying human concept learning. It is psychologically natural and engaging, drawing on significant background knowledge. It allows for many kinds of algorithmic reasoning and features numbers variously as symbols, ordinals, and cardinals. The domain is also formally tractable, and seems likely to capture a wide range of variance in learning mechanisms and abilities. It also has a long history in artificial intelligence, continuing to the present day. Moreover, list functions include natural subsets related to many classic concept learning domains, suggest generalizations of these tasks, and provide tasks that go beyond any classic domain. They also provide a way to isolate aspects of conceptual development involved in several prominent case studies from developmental psychology.

Despite all their advantages, list functions have not been used in any general way to study human learning except our own preliminary work (Rule et al., 2018). The combination of psychological familiarity, rich structure, algorithmic sophistication, and formal tractability, however, make them a prime candidate for exploring the child as hacker both computationally and empirically. In Chapter 4, we examine HL, a computational model of inductive learning designed to provide a hacker-like approach to learning in rich domains like list functions. The rest of the thesis then investigates human learning of list functions and HL's ability to explain human performance.

Chapter 4

HL: A hacker-like model of learning

One of the most stunning features of human learning is how rich it is not merely in content but also in form and objective. Learners, particularly children, are constantly shifting not only what they are trying to learn, but the reasons for which they are learning, the kinds of changes they are trying to make to their mental representations, and the means by which they accomplish those changes. As highlighted in Chapter 2, they might: remove repetition from an addition algorithm to make it shorter, faster, and less memory intensive (Siegler & Jenkins, 1989); search for a general theory explaining a large body of kinship facts to reduce memory load and improve generalization (Mollica & Piantadosi, 2019); or discover that the concept of a largest number and the concept of the successor function are logically incompatible (Cheung et al., 2017; Chu et al., 2020). People learn entirely new systems of interdefined concepts (Carey, 1985; Block, 1987) and do so in a way that is driven by hypotheses and goals rather than blind search (Carey, 2009; Chu et al., 2019), with a sensitivity to what is good in a hypothesis and what is suboptimal or even wrong (Schulz, 2012a; Chu & Schulz, 2020). Such richness is interesting because there are extremely simple algorithms guaranteed to discover optimal hypotheses, e.g. enumerating every possible hypothesis (Gold, 1967; Solomonoff, 1964a). That such a proposal sounds alien as a model of cognition, despite the extreme computational power of the human brain (Gallistel, 2017; Baum, 2004), reflects something not only about the complexity of the world that learners are trying to explain but also their reasons for learning and the sophistication of the machinery they bring to bear in doing so.

This thesis builds on the successful learning as programming paradigm. Learning as

programming hypothesizes that cognition occurs in a language of thought (LOT) which behaves like a programming language. Learning thus becomes programming, and formal models treat learning as program induction, the construction of programs which explain observed data. We extend these ideas by treating learning as analogous to a particular style of programming called hacking, which focuses on the ways in which humans make code better. Many learning as programming models of learning in the LOT focus on providing an extremely simple account of learning. They use basic local search mechanisms to discover simple and accurate programs in a fixed, minimal LOT. By contrast, hacking emphasizes the richness and diversity of the representations, values, and activities people use to improve code. It hypothesizes that simple accounts are unlikely to accurately explain the power and vast scope of human learning.

This chapter introduces HL (Hacker-Like), a computational model of inductive concept learning in the LOT. HL uses the child as hacker to take steps toward a richer and more realistic computational account of learning. Each major aspect of its design—its conceptual representations, learning mechanisms, and objectives—formalizes a well-known pattern from hacking which is itself chosen to help address some of the cognitive richness of learning missing in more general models of learning as programming. While not attaining to nearly the flexibility nor the scope of human learning, HL defines an extensible framework which can be adapted over time to support the interaction of multiple learning mechanisms, changing goals, and complex LOTs.

Regarding mental representations, part of the richness of human learning is the development of distinct conceptual systems whose basic constituents gain meaning through their inferential role. These include systems for talking about number, space, time, color, and kinship. The way people develop and use these conceptual systems is similar to the way hackers develop and use domain-specific languages, programming languages tailored to particular problem classes. These ideas are implemented in HL through the use of term rewriting systems. Rather than searching for individual programs in a fixed grammar, the use of term rewriting allows HL to learn entire LOTs, adding and removing primitives and adapting their meaning to fit observed data.

Unlike models that rely on a single form of local search, human learning makes use of

many constructive learning mechanisms. This gives much of learning the flavor of *learning by thinking* (Lombrozo, 2019). Hackers similarly make use of well-defined techniques for revising code that preserve, and sometimes even enhance, the semantics of the code by applying hypothesis-driven changes. Rather than searching over millions or billions of programs using local changes blind to semantics, HL incorporates these ideas by modeling learning as the iterative application of a toolkit of structured revisions—essentially program-changing programs—to a comparatively small set of existing hypotheses. In this way, it actually searches for meta-programs describing how an LOT is constructed rather than searching directly for the LOT itself. Rather than organizing the search for meta-programs around a memoryless process like Markov Chain Monte Carlo or neural synthesis, HL uses Monte Carlo tree search to construct a longterm memory which helps it balance exploration of unknown meta-programs against exploitation of known meta-programs, all while never proposing the same meta-program twice.

Finally, people are sensitive to useful structure in suboptimal, even wrong, hypotheses. They are thus willing to entertain ideas they know are bad overall if they are good in the right ways and bear faults that can be corrected by future learning. We relate this to the hacker maxim of avoiding premature optimization. HL implements these ideas by scoring hypotheses differently when building the search tree than when deciding which, of all the hypotheses it finds, to use for making predictions. It specifically relaxes its objective during search to make it more likely to explore seemingly suboptimal revisions that have the right overall structure. To help with this, HL's objectives include terms sensitive to the complexity of the meta-program required to construct a candidate LOT and the well-formedness of the LOT's predictions on novel inputs. These terms are used in addition to more standard terms reflecting the complexity of the LOT itself and its accuracy in predicting known input/output pairs. HL also naturally supports online learning: it is capable of adapting its current hypotheses as new information becomes available.

Subsequent sections detail the way HL blends insights about human learning and hacking in its aspect of its design, including: representation (Section 4.1), learning mechanisms (Section 4.2), and learning objectives (Section 4.3).

4.1 Representation: Term rewriting as a model of mental representations

This section describes a basic feature of human mental representations, an analogous technique hackers use for the development of better code, and how HL brings these ideas together in the representations it uses to model learning. Briefly, humans develop conceptual systems using symbols which are defined in terms of one another, taking on meaning through their inferential role. Hackers often do a similar thing through the development of domain-specific languages, a process HL formalizes by learning term rewriting systems.

4.1.1 Meaning through conceptual role

Humans develop conceptual systems and intuitive theories for reasoning about many domains, including time, space, color, kinship, number, and intuitive biology (Gopnik, 1983; Murphy & Medin, 1985; Carey, 1985, 2009; Karmiloff-Smith, 1988; Wellman & Gelman, 1992; Gopnik & Meltzoff, 1997; Gopnik & Wellman, 2012; Barner & Baron, 2016). One interesting feature of these systems is that the key concepts seem to be best defined in terms of one another rather than in terms of an additional set of more basic parts. *Uncle* is easy to describe with respect to *parent* and *sibling*, but cumbersome to explain in terms of lambda functions, S & K combinators, or the function words of English.

This observation has led to a way of thinking about meaning called procedural semantics (Johnson-Laird, 1977; Woods, 1981), also known as conceptual role semantics (CRS) or inferential role semantics (Field, 1977; Harman, 1975, 1987, 1982; Loar, 1982; Block, 1987, 1997). The basic idea is that mental representations gain meaning through the relationships in which they participate with one another. This approach argues that defining a domain's concepts in terms of one another, rather than leading to circularity, is actually central to a coherent theory of meaning. *Uncle*, for example, gains meaning precisely because of the way it interacts with other concepts, particularly *parent* and *sibling*. Those other concepts also gain meaning through their relationship to *Uncle*. It is by learning the network of these relationships that any one of the concepts comes to mean anything. Block (1987) elaborates

on this idea in a discussion of learning physics:

> One way to see what the CRS approach comes to is to reflect on how one learned the concepts of elementary physics, or anyway, how I did. When I took my first physics course, I was confronted with quite a bit of new terminology all at once: 'energy', 'momentum', 'acceleration', 'mass', and the like... I never learned any definitions of these new terms in terms I already knew. Rather, what I learned was how to use the new terminology—I learned certain relations among the new terms themselves (e.g., the relation between force and mass, neither of which can be defined in old terms), some relations between the new terms and the old terms, and, most importantly, how to generate the right numbers in answers to questions posed in the new terminology.

Terms like mass, acceleration, force, energy, and velocity are all defined in terms of one another. What is mass? Mass is the thing, m, such that $f = ma$, $e = mv^2/2$, $p = mv$, and so on. The same is true for any of these terms: they are defined by these relationships. This specific set of symbols and their relationships captures a particular way of carving up the world, with its own vocabulary and internal logic.

The argument is not necessarily that it is impossible to define a system of concepts by composing them out of more basic parts or translating them into the language of some other conceptual system. It is, instead, that doing so is unlikely to be explanatory. Fodor (1975), for example, begins his philosophical defense of the LOT by developing an argument for the role of cognitive science apart from neuroscience or physics. The argument centers on the idea that each discipline has a distinctly domain-specific repertoire of concepts best defined by their domain-specific interactions rather than by reducing them to some set of "more primitive" concepts. The concepts of cognitive science, he argues, are best described in terms of one another, rather than by reduction to neuroscience or physics. Reducing them to physics, for example, is unlikely to add anything to a good theory of cognitive science. It is instead much more likely to confuse and complicate our understanding, because the concepts which are important in cognitive science are unlikely to map neatly onto categories which are important in physics. The two are instead likely to be at odds with one another, which is precisely why we have two domains of study. The unique ways in which each discipline carves up the world is what sets it apart from the others. Considered in context with later arguments supporting the core of the LOT hypothesis, this argument suggests the need for a variety of semi-independent LOTs tuned to specific domains of thought. That is, humans develop conceptual systems and intuitive theories as tools for organizing thought, because

they better capture the essential dynamics of a domain than would be possible with a set of domain-general concepts[1]

The use of conceptual systems whose constituents are defined by conceptual role helps to explain why certain concepts seem to be learned together as systems. These systems identify concepts which are tightly interrelated. They are most naturally expressed in terms of one another rather than in terms of other concepts. For any one part of the system to work, all the concepts and their interrelations that define the system are necessary. The concepts and interrelations of other systems, however, are largely irrelevant. This also help to explain why translating concepts from one domain to another, for example in transfer learning, can be so difficult (Gick & Holyoak, 1980, 1983). If cognition used a single general-purpose LOT rather than a set of LOTs tuned to specific domains, transfer learning should be easy. All concepts would be built from the same basic parts, so transfer learning would be the default. That things are otherwise suggests that some significant effort is required to make the ideas of one conceptual system available in another, perhaps by translating them from one language to another.

4.1.2 The value of domain-specific languages

Hacking poses a fundamental tension. On one hand, a hacker wants a strong inductive bias—i.e. she wants her language to make it as easy as possible for her to write the program she is currently working on. On the other hand, her goals are constantly changing. The program she is working on now may look completely different tomorrow, or in a week, or a month. She wants as much freedom as possible to let her program evolve as she learns, discovers new structure, and improves her code. In sum, she needs to balance the effort required to improve her code now against the effort required to improve her code in the future. She needs to balance the inductive bias she wants today against the inductive bias she will want tomorrow.

Hackers frequently resolve this tension in the following way. First, they recognize that every programming language is itself just a program. It is a program that takes code as

[1]Others have similarly used the notion of multiple cognitive languages to help separate sensation, perception, and cognition (Macnamara & Reyes, 1994; Burge, 2010).

input and produces other[2] code as output. In order to have the language she wants today and the language she wants tomorrow, a hacker often first develops a special language which makes it trivial to write code achieving her current goals. As her goals change, she iteratively adapts this specialized language, taking care that the language always makes it as easy as possible to describe the solution to her current problem. To do so, she might implement her specialized language in a second language which is slightly less specialized but nonetheless somewhat tuned to the domains she thinks are most relevant to her problem. This second language might itself be implemented in an even less specialized third language and so on. In essence, she hierarchically factors the problem until the solution is nearly trivial to express.

Computer science is full of these domain-specific languages (DSLs; Fowler, 2010). HTML, for example, is a DSL for describing web pages (W3C, 2017), STAN is a DSL for describing and performing inference in probabilistic models (Carpenter et al., 2017), and SQL is a DSL for interacting with relational databases (ISO, 2016). Library systems in domain-general languages are also often used to provide domain-specific extensions to the language. In each case, the syntax and semantics of the language are tailored to make certain kinds of tasks easy, while other kinds of tasks may be impossible. HTML is excellent for describing the content of webpages, for example, but bad for describing how they should look, much less for expressing probabilistic models or database queries.

The use of DSLs in computer science is similar to the development of conceptual systems in psychology. DSLs, like conceptual systems, are tailored so that their primitives directly express the dynamics of their target domain and frequently expose only the relevant concepts, rather than appearing to ground out in any sort of external primitives. DSLs are also like conceptual systems in that the programs of a DSL gain meaning through the way in which the program evaluates when executed. Essentially, the language defines a set of relationships between programs, and evaluation iteratively transforms an input program according to these relationships to produce an output. DSLs are thus organized similarly to conceptual systems, defining a set of composable primitives that gain meaning through the complex relationships in which they participate.

[2]Unless, of course, the program is a quine, a special class of programs which return their own source code as output (Hofstadter, 1979).

```
# c1 (heads-in-tail)
(c1 [7, 2, 7, 7]) = 2
(c1 [1]) = 0
(c1 []) = 0
(c1 [2, 5, 9, 2]) = 1
(c1 [5, 5, 5]) = 2
```

```
# c2 (fst/head)
(c2 [7, 2, 7, 7]) = 7
(c2 [1]) = 1
(c2 [1, 2, 3]) = 1
(c2 [2, 5, 9, 2]) = 2
(c2 [5, 5, 5]) = 5
```

```
# c3, (snd/tail)
(c3 [7, 2, 7, 7]) = [2, 7, 7]
(c3 [1]) = []
(c3 [1, 2, 3]) = [2, 3]
(c3 [2, 5, 9, 2]) = [5, 9, 2]
(c3 [5, 5, 5]) = [5, 5]
```

```
# c4, (count)
(c4 7 [7, 2, 7, 7]) = 3
(c4 1 [1]) = 1
(c4 8 []) = 0
(c4 2 [2, 5, 9, 2]) = 2
(c4 5 [5, 5, 5]) = 3
```

Figure 4-1: Example data for a list function problem. `c2` takes a list and returns the first element. `c3` takes a list and returns all but the first element. `c4` takes an element and a list and returns the number of times the element appears in the list. `c1` operates like `c4` but counts the number of times the first item in a list appears in the rest of the list.

To understand how valuable the DSL approach can be, consider again the concepts c_1–c_4 from Chapter 1. How might one represent that system of concepts? One way to do so is example by example (Figure 4-1). While accurately capturing past experience, that approach is problematic, because it has no ability to generalize. Without any ability to predict how the concepts behave for novel inputs, every single example needs to be memorized. Because the set of numerical lists is infinite, there is no possible way to store all the necessary examples in a finite mind. Even very simple concepts like c_2 cannot be learned.

Instead of memorizing, it would be better to find general descriptions of c_1–c_4. Perhaps the simplest way to do this is to frame concept learning as a search for expressions in an LOT. To learn a concept like c_1 means finding some expression in this LOT which behaves like c_1, giving the same inputs and outputs for all the provided examples. Repeating the process for each concept eventually provides a definition for each new concept. Assume the LOT is computationally universal, so that it can encode not only c_1–c_4 but arbitrary knowledge[3]. We use a fairly minimal language here, the SK combinatory calculus, extended with a symbol, `E` for computing syntactic equality. The exact dynamics of `S` and `K` are unimportant here save that they are Turing-complete. We could adopt more complex primitives, but given enough to learn, any fixed set of primitives will seem similarly minimal. Describing c_1–c_4 in this way is possible[4], but tedious (Figure 4-2).

[3]This discussion considers several different representations for learning. Because it assumes from the outset that each is Turing-complete, any one of them could learn to model the dynamics of any other by discovering the necessary computational machinery and making it available as part of the LOT. The point at issue here, however, is how various these pieces of computational machinery change the dynamics of learning and make certain kinds of learning easier or harder.

[4]Because each of the representations in this example are computationally universal, there are ways to encode natural numbers and basic arithmetic (e.g. $+$, $-$, $\times$, $/$) in them using, for example, Church numerals (Barendregt et al., 1984). Showing how to do so is outside the scope of the example, so assume that these abilities are provided.

```
# Fixed primitives
(S a b c) = ((a c) (b c))
(K a b) = a
(E x x a b) = a
(E x y a b) = b

# Target concepts
Empty = (((S ((S (K ((S (K S)) K))) S)) (K K)) ((((S ((S (K ((S (K S)) K))) S)) (K K)) ((S K) K)) K) K)
Cons  = (((S (K S)) K) (((S (K S)) K) (((S ((S (K ((S (K S)) K))) S)) (K K)) ((((S ((S (K ((S (K S)) K))) S)) (K K))
↪  ((S K) K)) (K ((S K) K))))) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)) (((S ((S (K ((S (K S)) K))) S)) (K
↪  K)) ((S K) K))))
c2 = (((S (K S)) K) (S ((S K) K) (K K)) (S ((S K) K) (K (S K))))
c3 = (((S (K S)) K) (S ((S K) K) (K (S K))) (S ((S K) K) (K (S K))))
c4 = (((S (K ((S (K ((S (S ((S ((S K) K)) K))) (S ((S K) K))))) S))) K) (((((S (K S)) K) ((S (K S)) K)) ((((S (K S)) K)
↪  ((S (K S)) K)) S)) ((((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))) E (((S ((S (K ((S (K S)) K))) S)) (K K))
↪  ((((S ((S (K ((S (K S)) K))) S)) (K K)) ((S K) K)) K) K) 0) (((((S (K S)) K) ((S (K S)) K)) ((((S (K S)) K) S) (((S
↪  (K S)) K) S))) (((((S (K S)) K) (((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))))) (((S (K S))
↪  K) (((S (K S)) K) (((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))))) ((((S (K S)) K) (((S (K
↪  S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)))) (((S (K S)) K) (((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S))
↪  K))) S)) (K K))))) ((((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)))) ((S (K S)) K))))) E (((S
↪  (K S)) K) (S ((S K) K) (K K)) (S ((S K) K) (K (S K)))) (+ 1) ((S K) K)) ((((S ((S (K ((S (K S)) K))) S)) (K K))
↪  (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)) (((S (K S)) K) ((S (K S)) K)))) (((S (K S)) K) (S ((S K) K) (K
↪  (S K))) (S ((S K) K) (K (S K))))))))
c1 = (S ((((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))) E (((S ((S (K ((S (K S)) K))) S)) (K K)) ((((S ((S (K
↪  ((S (K S)) K))) S)) (K K)) ((S K) K)) K) K) 0) ((((S (K S)) K) (((S (K S)) K) S) ((S (K S)) K)) (((S (K ((S (K ((S
↪  (S ((S ((S K) K)) K))) (S ((S K) K))))) S))) K) (((((S (K S)) K) ((S (K S)) K)) ((((S (K S)) K) ((S (K S)) K)) S))
↪  ((((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))) E (((S ((S (K ((S (K S)) K))) S)) (K K)) ((((S ((S (K ((S (K
↪  S)) K))) S)) (K K)) ((S K) K)) K) K) 0) (((((S (K S)) K) ((S (K S)) K)) ((((S (K S)) K) S) (((S (K S)) K) S)))
↪  (((((S (K S)) K) (((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))))) (((S (K S)) K) (((S (K S))
↪  K) (((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K))))) ((((S (K S)) K) (((S (K S)) K) ((S ((S (K
↪  ((S (K S)) K))) S)) (K K)))) (((S (K S)) K) (((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)))))
↪  ((((S (K S)) K) (((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)))) ((S (K S)) K))))) E (((S (K S)) K) (S ((S K)
↪  K) (K K)) (S ((S K) K) (K (S K)))) (+ 1) ((S K) K)) ((((S ((S (K ((S (K S)) K))) S)) (K K)) (((S (K S)) K) ((S ((S
↪  (K ((S (K S)) K))) S)) (K K)) (((S (K S)) K) ((S (K S)) K)))) (((S (K S)) K) (S ((S K) K) (K (S K))) (S ((S K) K)
↪  (K (S K)))))))) (((S (K S)) K) (S ((S K) K) (K K)) (S ((S K) K) (K (S K)))) (((S (K S)) K) (S ((S K) K) (K (S K)))
↪  (S ((S K) K) (K (S K))))))
```

Figure 4-2: A language which could be learned from Figure 4-1 using an approach in which each concept is described in terms of a fixed set of primitives. Here, the primitives include S and K from combinatory logic, E for establishing equality, and various numerical primitives.

This approach would eventually require less memory than memorizing input/output pairs for each concept, but it would take many more examples than those shown above. It also makes an oddly limited use of composition. Because each concept is learned independently and without updating the LOT, there is no curriculum-based learning. Concepts learned in the past can be referred to and used, but only when used in isolation. They cannot, however, be composed to make future learning easier. This approach is then a sort of sub-Fodorian approach to the LOT in that only the innate primitives can be composed. Because this set of compositional primitives is fixed, learning new concepts quickly becomes hard: things which should be easy become difficult, and things which should be difficult become intractable. This effect is exaggerated when trying to uniquely encode a large number of concepts using a small

```
# Fixed primitives
(S a b c) = ((a c) (b c))
(K a b) = a
(E x x a b) = a
(E x y a b) = b



# Target concepts
Empty = (Pair True True)
Cons = ((B D D) Pair False Pair)
c2 = (B Fst Snd)
c3 = (B Snd Snd)
c4 = (Y ((D (D S)) (C* E Empty 0) ((D (F F)) ((C*** (B C***
↪  (C** C*** (C** B)))) E c2 (+ 1) I) ((C (B C D)) c3))))
c1 = (S (C* E Empty 0) ((B F B) c4 c2 c3))
```

```
# Learned concepts
I = ((S K) K)
A = (S I)
Y = ((S (K ((S (K ((S (S (A K))) A))) S))) K)
B = ((S (K S)) K)
C = ((S ((S (K ((S (K S)) K))) S)) (K K))
C* = (B C)
C** = (B C*)
C*** = (B C**)
D = (B B)
F = (B S)
T = (C I)
True = K
False = (K I)
Pair = (C* T)
Fst = (A (K K))
Snd = (A (K (S K)))
```

Figure 4-3: A language which could be learned from Figure 4-1 using an intermediate approach in which new concepts can be learned but must resolve to a fixed set of primitives.

set of primitives. Oddly enough, this is perhaps the most common approach for program-induction-based models of learning in the LOT (e.g. Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2012, 2016; Piantadosi, 2016; Kemp & Tenenbaum, 2008), largely because most models only ever learn a single concept at a time[5].

One obvious way to improve on this approach is to allow the iterative learning of intermediate concepts (Dechter et al., 2013). The scenario is otherwise the same as before, but once a concept has been learned, it is now considered a compositional part of the LOT. Moreover, concepts can be added which are not specifically designated in the input data but instead capture useful latent structure. Every concept ultimately resolves to the innate set of primitives, but learned concepts provide a shorthand for referring to certain combinations of those primitives and can themselves be combined. This is a decidedly more compositional approach, where harder concepts can be defined in terms of easier and previously learned concepts (Figure 4-3). It is also perhaps the closest to Fodor's discussion of the LOT (Fodor, 1975; Fodor, 1980). The fundamental basis on which the LOT rests never changes, each new concept represents a specific combination of primitives from this basis, and the ultimate set of possible concepts—the possible compositions of LOT primitives—is always the same.

[5]Some readers might see this as a largely technical point. Many models are only designed to learn a single concept, so the question of whether that concept is added to the lexicon or not might seem irrelevant. It is, however, decidedly nontrivial to create a model which can learn multiple concepts in a curriculum-based way and identify useful latent structure while doing so. Moreover, people, especially children, are constantly faced with multi-task learning scenarios. It thus seems appropriate to distinguish models which do not address these issues from those which have made a serious effort to explicitly model them.

Any apparent change in the cognition of the learner merely reflects the discovery of more and more complex combinations of primitives.

Allowing new concepts certainly provides a more compact description of c_1–c_4 than memorization or search in a fixed language. None of these three approaches, however, allows variable binding outside the set of primitives. Named variables are useful, because they allow learners to more directly express concepts which abstract over portions of their compositional structure. The result is that a great deal of each definition in Figure 4-3 is spent reinventing ad hoc parts of variable binding on a per-concept basis, shuttling and routing arguments to the proper places. This makes things which feel intuitively easy to a human extremely difficult to describe. It also ignores work showing that LOTs which permit variable binding through the use of quantifiers provide a better explanation of Boolean set inclusion concepts than LOTs lacking variable binding (Piantadosi et al., 2016). To encode something as simple as

```
(f x y g) = (g x y)
```

means learning something as complex as

```
f = ((((S (K S)) K) ((S ((S (K ((S (K S)) K))) S)) (K K)))
     (((S ((S (K ((S (K S)) K))) S)) (K K)) ((S K) K)))!
```

It is possible to simplify that process by adding variable binding to the LOT. This does not change the theoretical expressiveness of an already Turing-complete LOT, but it does make some kinds of concepts easier to describe, namely any concept which can be modified by arguments or parameters. In a sense, this approach is slightly super-Fodorian in that a named piece of the LOT may no longer represent a single composition of primitives. Variable binding makes it possible to describe concepts which instead specify a family of compositions, namely those that can be created by substituting values for each variable. Several models take this approach, using an LOT in which variable binding is freely available (Rule et al., 2015; Ellis et al., 2018; Ellis et al., 2020), and such a model can describe c_1–c_4 much more compactly[6] (Figure 4-4).

[6]The language in Figure 4-4 both allows variable binding and uses the variable `x` multiple times in the definition of `eq` to enforce equality. Restricting rules so the left-hand side uses each variable only once approximates an untyped lambda calculus. The problem from Figure 4-1 can no longer be solved as given, as there is no way to judge equality between terms from within the language (Jay & Vergara, 2017). A modified problem operating over a subset of terms (e.g. Church numerals) can be solved but is sufficiently different to confuse the example being made here.

```
# Fixed primitives
(S a b c) = ((a c) (b c))
(K a b) = a
(E x x a b) = a
(E x y a b) = b
```

```
# Learned concepts
(True x y) = x
(False x y) = y
(Pair x y f) = (f x y)
(Fst x) = (x True)
(Snd x) = (x False)
Empty = (Cons True True)
(Cons x y) = (Pair False (Pair x y))
(c2 x) = (Fst (Snd x))
(c3 x) = (Snd (Snd x))
(c4 x y) = E y Empty
             0
             (E x (c2 y)
                (+ 1 (c4 x (c3 y)))
                (c4 x (c3 y)))
(c1 x) = E x Empty 0 (c4 (c2 x) (c3 x))
```

Figure 4-4: A language which could be learned from Figure 4-1 using an intermediate approach in which new concepts are introduced as needed, but all concepts are treated like named functions and each argument must be a variable.

Variable binding helps to provide a significantly more direct explanation of the data from Figure 4-1. A great deal of what remains is dedicated to constructing and destructing lists in each place they are used. That is, when variables on the left-hand side of a conceptual relation can only be used to describe entire concepts, a great deal of work has to be done on the right-hand side to manipulate those concepts and retrieve the desired sub-parts. This could be fixed by allowing the left-hand side of rules to use variables not just for entire arguments, but for specific parts of arguments. In that case, the left-hand side of each rule acts like a complex pattern, some parts of which must remain constant, and other parts of which can vary and are assigned to variables. This provides fine-grained control over exactly what kinds of structures a concept abstracts over. It can in this way be interpreted as a powerful way to define compositions over a fixed set of primitives.

If, however, all symbols are defined using these complex patterns, then there is no bottom, no fixed set of primitives whose behavior is defined apart from the other symbols. Each pattern is a composition of symbols and variables which refers to another composition of symbols and variables. Portions of that composition might refer to other compositions, and so on. In that case, these expressions, rather than defining individual concepts, or parameterized families of concepts, describe complex relationships between multiple concepts, namely all the symbols involved in the pattern and the composition to which it refers. The semantics then emerge as the sum total of these relationships, rather than reducing to a small set of primitive behaviors.

```
[x y] = (Cons x y)
(c2 [x y]) = x
(c3 [x y]) = y
(c4 z Empty) = 0
(c4 z [z y]) = (+ 1 (c4 z y))
(c4 z [x y]) = (c4 z y)
(c1 Empty) = 0
(c1 [x y]) = (c4 x y)
```

Figure 4-5: A language which could be learned from Figure 4-1 when allowing reuse, variable binding, primitive addition and removal, and pattern-matching. Note the use of patterns for pulling lists into parts.

At this point, it makes sense to allow primitives to be added and removed at will, as symbols no longer directly ground out in compositions of primitive behavior. They simply describe relationships between symbols, and there is no reason to prevent symbols from being added or removed. The learned language can thus become an increasingly finely-tuned domain-specific conceptual system as symbols are added, removed, or redefined to better match observed data.

Expressing c_1–c_4 that way provides a solution perfectly tailored to the problem (Figure 4-5). Concepts like c_1 rely compositionally on previously learned concepts like c_4 and the patterns described by c_2 and c_3. More generally, the overall behavior of the LOT is fit to the data to be explained: all extraneous dynamics have been removed, and those which remain directly contribute to explaining the observations.

The progression above can be seen as lying on a spectrum describing various restrictions on how the LOT can be updated[7]. In the first, new symbols can only be added when they are defined strictly in terms of innate primitives. It thus maintains a strict divide between what is innate and what is learned. The second blurs this distinction in that new symbols can be defined in terms of both primitives and previously learned symbols. Each newly named composition is added to the LOT, but each name also refers to only a single, specific composition. The third system loosens this restriction by adding variable binding, which makes it simple to add symbols which take arguments. These new symbols refer not to a single composition, but to a parameterizable family of compositions. All of these approaches,

[7]There are even more extreme points on this spectrum. Some languages might assume that it was impossible to add new symbols at all, or even to compose them. These would be radically nativist approaches that seem decidedly untenable given our knowledge about human learning and development. Other languages might use the complex patterns of the fourth system but allow those patterns to nondeterministically reference multiple distinct compositions of other symbols.

however assume that the innate primitives cannot be removed or redefined, and new symbols are ultimately defined in terms of basic primitives. They are thus all closely related to Fodor's notion of learning in the LOT. Learning a new concept in all three boils down to finding a composition of basic primitives which explains the observed data. The fourth system takes a different approach. It allows for symbols to be added or removed at any time. Moreover, meaning does not necessarily ground out in the behavior of a fixed set of primitives. It is possible to express that kind of LOT, but more generally, meaning comes from the full set of complex relationships in which all symbols participate.

Learning in this fourth system is thus importantly different from the Fodorian view. Instead of composing a fixed set of basic dynamics in such a way as to explain the observed data, it changes the basic dynamics to contain precisely those needed to most simply explain the data. The entire language can be adapted to make the data as easy to explain as possible. Learning is no longer restricted to defining a symbol as compositions of existing symbols, though this can be done when it provides the simplest explanation. In other cases, however, learning is about finding the best language, the simplest set of objects and their relationships, for explaining the data. In essence, learning produces a DSL whose domain is the observed data.

As the LOT grows to explain more and more observations of the world, the total dynamics of the entire LOT shift less and less after each learning episode. An individual conceptual system, however, might change radically: adding symbols, removing symbols, or redefining relationships between those that remain. The DSL approach thus has important connections with the psychological problem of conceptual change (Barner & Baron, 2016). Conceptual change is the idea that intuitive and formal theories can change radically; later theories may ultimately become completely incommensurable with older theories of the same domain (Kuhn, 1962; Feyerabend, 1962). Despite philosophical arguments as to the impossibility of conceptual change (Fodor, 1980), conceptual changes and incommensurabilities are well documented in developmental psychology for domains like biology, the natural and rational numbers, and for concepts of weight and density (Carey, 2009, 1985; Carey & Spelke, 1994; Gopnik, 1983). They are also well-known in the history of formal science, including the development of modern theories of heat and heliocentrism.

Figure	$\lvert\Sigma\rvert$	$\lvert R\rvert$	$S(R)$	$\mathrm{DL}((\Sigma, R))$
4-1	8	20	346	374
4-2	10	10	1,681	1,701
4-3	26	26	256	308
4-4	15	15	172	202
4-5	8	8	68	84

Table 4.1: A comparison of the languages in Figures 4-2–5 based on their description lengths as TRSs. $|\Sigma|$ is the number of required symbols, $|R|$ is the number of rules, $\mathrm{S}(R)$ is the total number of subterms across all rules (i.e. the sum over the description length of each rule), and $\mathrm{DL}((\Sigma, R))$ gives the total description length of the entire TRS. The figures do not include the assumed background knowledge of numbers and arithmetic operators, which remains constant across all cases.

The DSL approach provides a formal framework for modeling this process. Newly added symbols are initially void of meaning and act instead as placeholders whose meaning is constrained as they are incorporated into more and more of the language's rules. Even as the meaning of these placeholders is contrained by their relationship to existing concepts, those existing concepts are simultaneously constrained by their new relations to the placeholder. As a symbol is used in more rules, its meaning becomes more tightly constrained. As it is used in fewer rules, it loses meaning, and may eventually become meaningless if its becomes entirely unused. The result is a formal description of how incommensurabilities might develop.

The discussion above demonstrates the computational value of treating learning as the development of a DSL. The impact for this particular example is summarized in Table 4.1, but the point holds more generally: tailoring syntax and semantics in this way can often lead to a hundred-fold or even thousand-fold reduction in the size of the program needed to encode a particular computation (Ohshima et al., 2012). Moreover, there are strong correspondences between the operation of intuitive theories in psychology and domain-specific languages in computer science (Table 4.2).

Despite its plausibility, existing models of concept learning are poorly suited to test this domain-specific LOT hypothesis. First, most models are tailored to a single domain, such as motor programs for hand-written characters (Lake et al., 2015) or Boolean concepts for sets of colored shapes (Piantadosi et al., 2016), and do not have facilities for extending themselves to explain other domains. They thus require manually designed conceptual primitives. This

Intuitive Theories	Domain-Specific Languages
explain & predict	explain & simulate domains (fluids, protein folding)
simplify reality	define interfaces to domain algorithms & structures
interact with data	read, create, transform, destroy, and save data
have external structure	interact with other libraries
have internal structure	are sets of compositional functions
are learned piecemeal	are implemented iteratively
may be implicit/explicit	selectively import & export
are internally holistic	cannot usually be copied piecemeal
are externally modular	explicitly list dependencies on other libraries

Table 4.2: Comparing intuitive theories and DSLs, inspired by (Murphy & Medin, 1985).

is problematic because it suggests the need for a large set of primitives to explain the wide variety of human learning. The primitives also become more complex and thus typically require more detailed psychological justification. This justification, while necessary in the long run, can slow modeling research, as many human core cognitive resources are still poorly understood relative to what is needed for computational modeling. Comparing multiple sets of potential primitives developed this way is also costly for modelers and severely limits the space of considered hypotheses (Piantadosi et al., 2016). Second, models which operate across multiple domains typically assume a universal and often minimal set of primitives such as those found in various combinatory logics or lambda calculi (Piantadosi, 2016; Dechter et al., 2013; Ellis et al., 2020). This option is attractive because it requires only a small number of simple concepts; in many existing learning algorithms, each additional primitive slows learning significantly. Veering too far into minimalism, however, can ignore primitives for which there is strong empirical evidence and introduce primitives for which there is no meaningful evidence. It can also slow learning by making each learned concept unnecessarily difficult to describe. For large sets of concepts, many of the concepts grow to be complex and inefficient to evaluate. There is no way to balance the number of true primitives against the complexity of the domain. Third, very few models are organized so that they can define intermediate concepts in terms of which other concepts can be defined. Those models which do permit intermediate concepts often sharply limit the number which can be learned (Rule et al., 2015), or force intermediate concepts to resolve to fixed primitives (Dechter et al., 2013; Ellis et al., 2020).

The root of these problems, again, is relying on a fixed set of conceptual primitives in terms of which all concepts must eventually be defined. If those primitives are not well suited to the domain, that means that domain-specific concepts become complex to define and inefficient to use. Existing models of grammar induction are affected by a related problem in which they can easily express only a limited set of computational grammars (e.g. graph structures as in Kemp and Tenenbaum, 2008, or string concatenations as in Rule et al., 2015). Being able to add or remove primitive concepts as needed to tailor-fit a domain-specific LOT requires a new approach. Rather than fixing the language (i.e. fixing a set of conceptual primitives), or adopting a meta-language for describing one class of domains (e.g. strings, graphs), the model should fix a meta-language from which any DSL could be built. Individual expressions in this meta-language would themselves describe a complete language—syntax and semantics—that could be evaluated as an explanation of data. This approach would allow models to develop the set of primitives and rules of interaction best suited to their observed data instead of building from fixed primitives and the limited interactions they allow. The next section describes how HL implements these ideas using a formalism called term rewriting.

4.1.3 Term rewriting

Rather than leading to an infinite regress, the DSL approach favors the adoption of an initial language which is itself good for implementing other languages. This requires stepping back from fixed sets of primitives and behaviors—as are common in formalisms like combinatory logic (Schönfinkel, 1924; Curry, 1930), lambda calculus (Church, 1932), Turing Machines (Turing, 1936), context free grammars (Sipser, 2012), Post canonical systems (Post, 1943), and others (Minsky, 1967)—to study the class of systems to which they belong and the ways in which changing the primitives and their interactions affects the behavior of the system. This is the approach taken in the study of Term Rewriting Systems (TRSs)[8]. Term rewriting suits the purposes of theoretical computer science well in that it permits

[8]This is actually the approach of rewriting more broadly. Rewriting is a branch of computer science studying how discrete objects evolve over time using Abstract Rewriting Systems. Abstract Rewriting Systems exist to describe rewriting not only for terms but also graphs, strings, concurrent system traces, and other structures (Bezem et al., 2003).

abstracting over individual languages to draw general conclusions about the behavior of entire classes of languages. It suits the purposes of computational cognitive modeling in that it provides a uniform mechanism for defining an unboundedly large set of potentially Turing-complete programming languages, each serving as a model LOT. After a brief tutorial on term rewriting based on discussions in Baader and Nipkow (1999) and Bezem et al. (2003), we describe how HL uses term rewriting as a model of conceptual representations.

TRSs formalize the intuition that symbolic forms of computation like programming languages boil down to trees of symbols and rules for how those trees evolve, evaluate, or compute. They form the explicit basis for many functional programming languages, and, because both combinatory logic and lambda calculus are TRSs, the implicit basis for many others. While it has previously appeared in inductive learning systems (Rao, 2004; Hofmann et al., 2009), term rewriting is less common than alternative formalisms like first-order logic, combinatory logic, or lambda calculus. Defining a TRS formally requires several preliminary definitions, given here and illustrated with SK combinatory logic and Peano arithmetic.

A *signature*, Σ, is a set where each element, $s \in \Sigma$, is a symbol with some arity, $n \geq 0$. The signature for SK combinatory logic uses one symbol with arity 2, $\cdot$, and two symbols with arity 0, S and K:

$$\Sigma_{CL} \equiv \{\cdot/2, S/0, K/0\}$$

The signature for Peano arithmetic uses one symbol with arity 2, $+$, one symbol with arity 1, S, and one symbol with arity 0, 0:

$$\Sigma_{PA} \equiv \{+/2, S/1, 0/0\}$$

A *term* is a tree structure composed of atoms. An *atom* is either a variable or an operator. A *variable* represents an entire arbitrary term and thus has no arity. An *operator* is a symbol from Σ. A *constant* is an operator of arity 0. If X is a set of variables, then the set of terms given Σ and X, $T(\Sigma, X)$, is the set of expressions that can be composed from X and Σ while respecting the arity of the operators[9]:

[9]Technically, the definition given here is for *first-order* terms. Higher-order term rewriting systems have

$$T(\Sigma, X) \equiv X \cup \{f(t_1, \ldots, t_n) \mid f \in \Sigma,\ \text{arity}(f) = n,\ t_1, \ldots, t_n \in T(\Sigma, X)\}$$

If $X \equiv \{x, y\}$ and $\cdot$ is written using infix notation:

$$\begin{aligned}
\{x,\ y,\ S,\ K\} &\subset T(\Sigma_{CL}, X) \\
\{x,\ y,\ 0\} &\subset T(\Sigma_{PA}, X) \\
(x \cdot y) \cdot z &\in T(\Sigma_{CL}, X) \\
+(S(S(0))\ +(S(0)\ x)) &\in T(\Sigma_{PA}, X) \\
\cdot &\notin T(\Sigma_{CL}, X) \quad (\cdot \text{ has arity 2}) \\
\{S,\ +\} &\not\subset T(\Sigma_{PA}, X) \quad (S \text{ has arity 1, } + \text{ has arity 2}) \\
+(K\ 0) &\notin T(\Sigma_{PA}, X) \quad (K \notin \Sigma_{PA}) \\
x(K) &\notin T(\Sigma_{CL}, X) \quad (\text{variables are complete terms}) \\
+(0) &\notin T(\Sigma_{PA}, X) \quad (+ \text{ has arity 2})
\end{aligned}$$

Assuming that X is some unboundedly large set of variables, the set of terms is thus entirely defined by Σ and the rules for composing elements of X and Σ. Knowing Σ thus defines the syntax of the TRS, the set of terms about which it is concerned. If the TRS is viewed as defining a programming language, Σ defines the set of programs in that language. On its own, however, Σ provides no information about how these programs behave, how programs are transformed step-by-step to perform computations. That is, Σ provides syntax, but it lacks a meaningful semantics.

The semantics of a TRS come from its rewriting rules, each of which describes a specific way of transforming an input program into an output program. An *identity* is a pair $(s, t) \in T(\Sigma, X) \times T(\Sigma, X)$, commonly written as $s \approx t$. A *rewrite rule* is an identity where $s \notin X$ (i.e. s is not a variable) and $\text{Var}(s) \supseteq \text{Var}(t)$ (i.e. no terms need be invented to complete the rewrite). A *term rewriting system*, $H \equiv (\Sigma, R)$, is an ordered pair containing a signature and a sequence of rewrite rules, $R \equiv \langle R_1,\ R_2, \ldots, R_n \rangle$. For SK combinatory logic:

been studied, and place appropriately different constraints on what constitutes a term (Bezem et al., 2003; Van Raamsdonk, 1999)

$$R_{CL} \equiv \langle((S \cdot x) \cdot y) \cdot z \approx (x \cdot z) \cdot (y \cdot z),$$
$$(K \cdot x) \cdot y \approx x\rangle$$

$$H_{CL} \equiv (\Sigma_{CL}, R_{CL})$$

And, for Peano arithmetic:

$$R_{PA} \equiv \langle + (0\ x) \approx x,$$
$$+ (S(x)\ y) \approx S(+(x\ y))\rangle$$

$$H_{PA} \equiv (\Sigma_{PA}, R_{PA})$$

Each set of rewrite rules, R, defines a rewrite relation, $\rightarrow_R$, describing how arbitrary terms in $T(\Sigma, X)$ may be rewritten. According to $\rightarrow_R$, a term, s, can be rewritten to another term, t, or $s \rightarrow_R t$, if there is a rule $l \approx r \in R$, such that s can be matched with l, written $l \mapsto s$, to produce a substitution, σ. Intuitively, matching is a way of assigning a value to each variable in l such that replacing the variable with its value transforms l into s; the substitution, σ, is merely a collection of these assignments. Formally, matching produces σ if and only if applying σ to l, $\sigma(l)$ equals s. Computing $\sigma(l)$ consists of repeatedly replacing each variable in l according to the mapping defined by σ until no more replacements are possible.

For example, given the term $s \equiv ((S \cdot K) \cdot S) \cdot K$ and the rule $R_{CL,1} \equiv (S \cdot x) \cdot y) \cdot z \approx (x \cdot z) \cdot (y \cdot z)$ such that $l \equiv (S \cdot x) \cdot y) \cdot z$:

$$\{((S \cdot x) \cdot y) \cdot z \ \mapsto\ ((S \cdot K) \cdot S) \cdot K\} = \sigma \equiv \{x \mapsto K, y \mapsto S, z \mapsto K\}$$

because

$$\begin{aligned}\sigma(((S \cdot x) \cdot y) \cdot z) &= \sigma((S \cdot K) \cdot y) \cdot z \\ &= \sigma((S \cdot K) \cdot S) \cdot z \\ &= ((S \cdot K) \cdot S) \cdot K\end{aligned}$$

The rewrite step is completed by applying σ to r, the right-hand side of the rule. For example, given that $r \equiv (x \cdot z) \cdot (y \cdot z)$:

$$\begin{aligned}\sigma((x \cdot z) \cdot (y \cdot z)) &= (\sigma(x) \cdot \sigma(z)) \cdot (\sigma(y) \cdot \sigma(z)) \\ &= (K \cdot K) \cdot (S \cdot K)\end{aligned}$$

So, we say that $((S \cdot K) \cdot S) \cdot K \rightarrow_{R_{CL}} (K \cdot K) \cdot (S \cdot K)$. The decision of which rules can apply to which subterms at what times—that is, the questions of whether rules apply in particular orders when there are potentially overlapping rewrites, whether multiple rules can apply simultaneously, or whether the same rule can apply in multiple places simultaneously—are of great theoretical importance. The study of these various *evaluation orders* has a significant bearing on what guarantees can be made about the behavior of the TRS. In this thesis, we focus primarily on the well-known *normal-order* evaluation strategy, which assumes that at most one rule can apply at a time, and only to the leftmost-outermost position in the term to which any rule can apply. We further restrict normal-order evaluation by assuming that if multiple rules could apply in the same position, the first in the sequence of R is applied.

The set of rules R thus defines the relation $\rightarrow_R$. This relation describes the step-by-step process by which terms are rewritten. Equivalently, it defines the interpreter according to which programs are executed. This interpreter, as the name suggests, provides the semantics for the programs of the TRS. It describes the means by which a given program is related to all other programs. These relationships act as a sort of procedural semantics (Woods, 1981; Johnson-Laird, 1977), a perspective on meaning closely related to several prominent theories of meaning by inferential role in psychology and philosophy of mind (Block, 1987, 1997; Carey, 2009). For example, in our Peano arithmetic system, the meaning of `S(S(0))` is constrained by the fact that `S(+(S(0) 0))`, after a few steps, rewrites to `S(S(0))`. It is also constrained by the fact `+(0 S(S(0)))` also rewrites to `S(S(0))`, as does `+(S(S(0)) 0)`

`+(S(0), S(0))`, and so on. This network of relationships provides meaning to each program.

HL models learning as a search through the space of TRSs. Like other LOT-based models, it searches for a program which serves as a generative model of observed data. In contrast with other LOT-based models, however, the data which HL considers are programs and their outputs, and the programs which it learns are TRSs specifying the syntax and semantics of an entire DSL. This allows it to add, remove, and redefine primitives as needed, supporting genuine conceptual change as it searches for an LOT which provides a close fit to the data. The only significant deviation made from the description given above is that HL provides the ability to mark some rules as background rules which cannot be altered. This is more for the sake of computational efficiency than formal necessity in that it helps to clearly delineate which aspects of the language are relevant for learning and which can be safely ignored. It otherwise considers any valid TRS a valid hypothesis.

4.2 Learning Mechanisms: Learning as iterative meta-programming

This section describes an organizing principle of human learning essential to effectively using a diverse set of learning mechanisms, a related technique used by hackers, and their implementation in HL. The organizing principle we focus on here is the way that much of human learning is constructive thinking constrained by goals and hypotheses, rather than blind local search. Hackers do something similar through the use of structured techniques for revising programs. These techniques can be thought of as program-changing programs. HL implements these ideas by reparameterizing search. Rather than searching directly for TRSs, it searches for meta-programs describing a chain of revisions to some seed TRS. This operates like a compact generative process for transforming the seed into some target TRS. HL searches through this space using Monte Carlo tree search in order to iteratively explore the space of revisions in a way that favors the revisions which are likely to be most useful.

4.2.1 Hypothesis-and-goal-driven search

Developmental and cognitive psychology are filled with accounts of diverse learning mechanisms. The microgenetic methods pioneered by Siegler and colleagues have been used to show how variation, selection, and adaptation of learning strategies are key features of children's cognition (Siegler & Jenkins, 1989; Siegler, 1996). Significant bodies of work point to the importance of mechanisms as varied as: deductive reasoning (Rips, 1994), Bayesian inductive inference (Ullman & Tenenbaum, 2020) and many other forms of experimentation and inductive reasoning (Schulz, 2012b; Gopnik, 2012), analogy (Gentner, 1983; Holyoak, 2012), bootstrapping (Carey, 2009), heuristic reasoning (Kahneman et al., 1982), mental simulation (Ullman et al., 2017), mental modeling (Johnson-Laird, 1989, 2012), explanation and abduction (Lombrozo, 2012), and counterfactual and causal reasoning (Chater & Oaksford, 2013; Gerstenberg & Tenenbaum, 2017). Learners are constantly switching between mechanisms based on the current situation and adapting them to new purposes. The existence of modular learning mechanisms and core cognition (Carey, 2009; Fodor, 1983) is a reminder that natural selection also acts as a key phylogenetic learning mechanism.

If the list above reads like the table of contents for a psychology textbook, that is because much of learning is constructive (Xu, 2019). This sort of constructive learning occurs in the context of a hypothesis-and-goal-driven search. It makes use of highly structured mechanisms that make specific hypotheses in service of specific goals. As a result, a lot of learning is thinking, and a lot of thinking is learning (Lombrozo, 2019).

Like the empirical literature, cognitive models and artificial intelligence systems compatible with the idea of learning in an LOT are similarly filled with a variety of learning mechanisms. Exhaustive enumeration remains perhaps the simplest technique (Gulwani et al., 2017). Others focus on structured exploitation of constraints, including: deductive reasoning and theorem proving (Newell et al., 1959; Newell & Simon, 1956; Manna & Waldinger, 1980; Green, 1981; Joshi et al., 2002), syntactic pattern matching (Sussman, 1973), inductive logic programming (Muggleton et al., 2015; Cropper & Muggleton, 2016; Cropper et al., 2019), heuristic reasoning (Lenat, 1983; Lenat, 1976), type-directed synthesis (Polikarpova et al., 2016; Osera & Zdancewic, 2015), SAT/SMT solving (Solar-Lezama, 2008; Nye et al.,

2019) and answer set programming (Cropper & Morel, 2020). Examples of Bayesian inductive inference form another central class of learning mechanisms (Ullman & Tenenbaum, 2020), especially hierarchical Bayesian inference (Goodman et al., 2011; Ullman et al., 2012; Lake et al., 2015), exact and approximate inference in probabilistic graphical models (Gopnik & Tenenbaum, 2007; Gopnik & Wellman, 2012), and learning the weights of stochastic grammars (Dechter et al., 2013). Still others are fundamentally stochastic, such as: neural program synthesis (Balog et al., 2017; Devlin et al., 2017), Markov Chain Monte Carlo (Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi, 2011; Ullman et al., 2012), Sequential Monte Carlo (Ellis et al., 2019), and genetic programming (Koza, 1989; Koza & Koza, 1992). Reinforcement learning is another, less common, formalization of learning in compositional languages (Simmons-Edler et al., 2018).

The richness of human learning and the richness of computational models of learning, however, are importantly different. Critically, while computational models have explored a wide range of learning mechanisms, any given model typically makes use of just one. Even when a single model makes use of multiple mechanisms, this is most often done by composing the mechanisms into a more complex, but still singular, super-mechanism. Moreover, these mechanisms are often simple (even dumb) local search mechanisms. They act at random or by brute force rather than through a structured hypothesis-and-goal-driven search. This pattern stands in sharp contrast to the diversity of mechanisms available to human learners at any given moment and the sophisticated ways in which they decide which mechanism to use in any given situation. While closing this gap is likely to require sustained interdisciplinary effort over the course of years, if not decades, the child as hacker hypothesis around which this thesis is organized provides a concrete roadmap by which it might be accomplished. It identifies the diversity of mechanisms as a central feature of learning and focuses on the actual practices of human hackers as testable hypotheses about specific mechanisms in learning which remain largely unexplored.

4.2.2 Hacking as iterative revision

This section describes the fundamental pattern hackers use to write code and the implications of that approach for a computational model of learning. A mature codebase is not written

at a single stroke but instead develops iteratively through a series of revisions. This practice of iterative revision is perhaps the fundamental pattern of hacking (Sussman, 1973; Ellis et al., 2019; Fowler, 2018). At any given time, a hacker is working to introduce just a single change to her code. Ideally, this change is fairly targeted and easy to implement, so that she spends very little time with a codebase that she cannot run and test. This is sometimes impossible—for example, rewriting a program in terms of a new DSL is a big job—but she can often decompose even these big tasks into a series of smaller revisions. She then chains these revisions together, accumulating them to effect dramatic changes in her code. Of course, she might get excited and slip out of this pattern from time to time, working on many changes simultaneously or forgetting to break a big change down into a series of small changes. When she is hacking at her best, however, she is iteratively revising her code through a series of small changes.

Some of the changes she makes affect the meaning of her code—i.e. running a certain program produces a different output than it did previously. Many others, however, merely *refactor* the code. They change its shape, or *factor*, without changing its meaning. Similarly, some changes immediately and obviously improve her code, while others do not. She may occasionally make several revisions in a row which make her code longer or slower without adding any new functionality. This is in fact often the case for various types of refactoring. Chaining several of these seemingly useless refactorings, however, may reveal patterns which future revisions are able to exploit, such that the overall effect of the entire sequence is to dramatically improve her code.

Iterative revision may play such a fundamental role for several reasons. Perhaps one of the most obvious is that humans have limited working memory. Once code grows sufficiently complex, it is difficult to predict how it will behave without actually running it, difficult to understand how each of a series of changes will impact the code without making each change in turn, and difficult to remember where you are in the process of making a revision if the revision itself is complex. Serializing her work allows a hacker to scale even complex modifications to fit her own cognitive limits.

Iteration also provides a steady signal that a hacker is heading in the right direction. If each change overlapped with several others, the code would never be in a functioning state,

and she would be able to gather little information on whether the changes made so far had actually improved the code. By separating them into serial units of work, she accumulates a sequence of signals by which she can judge her progress, decide whether to continue, undo past changes, or pause to consider her options in more detail.

Finally, and perhaps most importantly, the iterative nature of her work allows her to reuse stereotypical patterns of revision. She is constantly in close contact with her code, and she is likely writing code unlike anything she has written previously. Her changes to the code, however, rarely occur at the level of individual symbols. Instead, much like the patterns which skilled chess players use to perceive a chess board (Chase & Simon, 1973) and generate likely moves (Wan et al., 2011), she develops a network of strategies for perceiving latent structure in her code and related techniques which revise her code to make that latent structure explicit. Many of these strategies and techniques are so well defined that they are, in effect, program-changing programs. Most are heavily parameterized, so that they can be easily adapted to a variety of situations. Each represents a specific hypothesis about the meaning of the code which can be used by a hacker, in the context of her current goals, to explore specific ideas about the underlying structure of her code. Most are also sensitive to the semantics of the code, changing it in ways that either preserve (e.g. refactor) the meaning, or make latent meaning more explicit through abstraction and generalization. Rather than operating at the level of individual symbols, hacking now takes place primarily in this meta-language developed for describing the generative process of hacking.

The impact of this approach is two-fold. First, the series of revisions accumulated over time essentially describes a meta-program by which her code has been constructed. It may not be the only such meta-program, nor even the optimal one, but it describes her code via a compact generative process by which the entire codebase can be recovered. Given the structure in each revision, this generative meta-program is likely to be much shorter than the actual code itself. Second, this approach makes it easier to hack useful code. Because the meta-program required to write useful code is typically much simpler than the code itself, the meta-program is easier to find than the program itself. In effect, there are fewer decisions that must be made correctly, fewer bits that need to be set accurately. Moreover, because most revisions are heavily parameterized, the process of constructing even this relatively

short meta-program can itself be factored into two parts. First, a hacker determines which of the many kinds of revisions at her disposal might be most appropriate, then she searches the much smaller space of possible parameterizations to find those which are likely to be most useful.

The idea of code as data is central to this process. Rather than searching blindly, the hacker is able to observe her code, itself an artifact in the world, analyze its structure, and use that analysis to guide her decision making. This allows her to transform the gargantuan task of finding one of a very small set of possible solutions to her problem into a series of much simpler problems sharing identical structure. Each step poses the same problem: given my current code and current goal, what change can I make that will bring me closer to that goal? This framing is useful because it explicitly transforms hacking into a hypothesis-and-goal-driven search, with the revision techniques playing a similar role to the mechanisms used during learning. A hacker's techniques are unlikely to be identical to processes like analogy or experimentation, but they are analogous in that they encode ways to apply specific hypotheses to code, i.e. a hacker's knowledge representation, to pursue certain goals. This framing also decomposes hacking into discrete steps, such that a hacker is likely to take many such steps for each program she writes. This not only makes each problem to be solved smaller, but it gives her more practice and more frequent learning signals than writing each program monolithically from scratch.

The implications of this approach for a computational model of learning extend well beyond the scope of a single model. HL, however, takes three primary steps toward applying these lessons toward building a computational model of learning. First, HL is built around a series of smart search mechanisms modeling structured techniques hackers use to revise code. Each mechanism is associated with a strategy for recognizing when the mechanism can be applied in addition to techniques for actually revising code appropriately. Many of these mechanisms are parameterized, describing families of possible revisions. Most also decompose into a series of submechanisms, allowing HL to explicitly learn which parameterizations are most useful. All of them are program-revising programs, taking a TRS as input and returning a TRS as output. Second, HL learns by constructing chains of mechanisms into meta-programs, which it then executes to generate concrete TRSs. This process effectively

decouples the complexity of finding a program from the length of the program itself, making it easy to find certain kinds of long programs, namely those whose input/output data contains structure that HL knows how to exploit. Third, HL uses Monte Carlo tree search to decide which meta-programs to explore in which order. It learns over time which partial meta-programs are most likely to lead to good solutions, avoids repeating itself, and balances exploration of novel meta-programs versus exploitation of known meta-programs. The next three sections describe how each of these components are implemented in HL.

4.2.3 HL's learning mechanisms

HL is equipped with 11 kinds of mechanisms (Table 4.3). Some are primarily refactoring moves, while others almost always introduce new behavior. Developing these mechanisms took substantial work, but they are only a first step. A skilled hacker likely has hundreds of such techniques available to them (Abelson et al., 1996; Fowler, 2018). Nonetheless, this first step is a significant shift toward a more hacker-like model of learning. This section describes briefly each mehcanism in turn and provides an example of the kind of transformation it can produce.

MemorizeDatum HL is designed to learn from observations, specifically ordered pairs of programs, (i, o), such that i represents some input program, and o represents the output of evaluating i. o might represent the complete evaluation of i or merely a partial evaluation (i.e. it may be possible to treat o as an input i' which could itself be evaluated to produce some further output o', as in $i \to_R o \equiv i' \to_R o'$). The `MemorizeDatum` mechanism selects a single input/output pair and adds it directly to a hypothesis TRS, effectively hard-coding an exception into the language. The new rule instructs the TRS to immediately rewrite i to o whenever the input program contains i. `MemorizeDatum` takes a single parameter, an index specifying which datum to memorize of those seen so far. A datum cannot be memorized if an equivalent rule already exists in the TRS.

MemorizeAll This mechanism is similar to `MemorizeDatum` except that it memorizes all the observed data, rather than a single datum. Like `MemorizeDatum`, however, data are not memorized if an equivalent rule already exists in the TRS. The mechanism takes no parameters.

I.MemorizeDatum(2)

```
# where datum 1: C [1, 38, 19, 4] = [19]
# where datum 2: C [31, 41, 59, 62, 5] = [59]
# where datum 3: C [68, 47, 3, 6, 0, 9, 77] = [3]
C [x, y, 19, z] = [19]
```

```
C [31, 41, 59, 62, 5] = [59]
C [x, y, 19, z] = [19]
```

I.MemorizeAll()

```
# where datum 1: C [1, 38, 19, 4] = [19]
# where datum 2: C [31, 41, 59, 62, 5] = [59]
# where datum 3: C [68, 47, 3, 6, 0, 9, 77] = [3]
C [31, 41, 59, 62, 5] = [59]
C [x, y, 19, z] = [19]
```

```
C [1, 38, 19, 4] = [19]
C [68, 47, 3, 6, 0, 9, 77] = [3]
C [31, 41, 59, 62, 5] = [59]
C [x, y, 19, z] = [19]
```

I.DeleteRule(3)

```
C [1, 38, 19, 4] = [19]
C [68, 47, 3, 6, 0, 9, 77] = [3]
C [31, 41, 59, 62, 5] = [59]
C [x, y, 19, z] = [19]
```

```
C [1, 38, 19, 4] = [19]
C [68, 47, 3, 6, 0, 9, 77] = [3]
C [x, y, 19, z] = [19]
```

I.SampleRule(·, C, ·, Cons, ·, 7, x, x)

```
C [x, y, z] = [z]
```

```
C (Cons 7 x) = x
C [x, y, z] = [z]
```

I.RegenerateRule(2, [1, 0, 1], ·, ·, +, x, x)

```
C [x, y, z] = [z]
C [17, x, 38, 4] = [38]
```

```
C [x, y, z] = [z]
C [17, x, 38, 4] = [(+ x x)]
```

I.Variablize(2, 63)

```
C [w, x, y, z] = [y]
C [91, 12, 63, 42, 35] = [63]
```

```
C [w, x, y, z] = [y]
C [91, 12, x, 42, 35] = [x]
```

I.AntiUnify()

```
C [25] = Empty
C [0] = Empty
C [9, 25] = (Cons 9 (C [25]))
C [81, 9, 25] = (Cons 81 (C [9, 25]))
C [50, 0] = (Cons 50 (C [0]))
C [28, 50, 0] = (Cons 28 (C [50, 0]))
C [37, 28, 50, 0] = (Cons 37 (C [28, 50, 0]))
```

```
C [x] = Empty
C (Cons x y) = (Cons x (C y))
```

I.Compose([1], [0])

```
C [15, 6, 38] = [0, 15, 6, 38, 99]
C [12, 71] = [0, 12, 71, 99]
C [74, 3, 8, 16] = [0, 74, 3, 8, 16, 99]
```

```
C x = g (f x)
f [15, 6, 38] = [15, 6, 38, 99]
f [12, 71] = [12, 71, 99]
f [74, 3, 8, 16] = [74, 3, 8, 16, 99]
g [15, 6, 38, 99] = [0, 15, 6, 38, 99]
g [12, 71, 99] = [0, 12, 71, 99]
g [74, 3, 8, 16, 99] = [0, 74, 3, 8, 16, 99]
```

I.Recurse(C, [1], [1, 1])

```
C [81, 9, 25] = [81, 9]
C [37, 28, 50, 0] = [37, 28, 50]
```

```
C [25] = Empty
C [0] = Empty
C [9, 25] = (Cons 9 (C [25]))
C [81, 9, 25] = (Cons 81 (C [9, 25]))
C [50, 0] = (Cons 50 (C [0]))
C [28, 50, 0] = (Cons 28 (C [50, 0]))
C [37, 28, 50, 0] = (Cons 37 (C [28, 50, 0]))
```

I.Generalize()

```
C [x 1] = [1 (* 2 x) (* x x)]
C [x 2] = [4 (* 4 x) (* x x)]
C [x 3] = [9 (* 6 x) (* x x)]
```

```
C [x y] = [(f y) (* (g y) x) (* x x)]
f 1 = 1
f 2 = 4
f 3 = 9
g 1 = 2
g 2 = 4
g 3 = 6
```

I.Stop()

```
C [x] = [x]
C (Cons x y) = (C y)
```

```
C [x] = [x]
C (Cons x y) = (C y)
```

Table 4.3: An example application of each learning mechanism implemented in HL. Each mechanism is listed, along with its arguments, including **I**, the input TRS, listed immediately below. The output is listed to the right.

DeleteRule This mechanism deletes a single rule from the TRS. It takes a single parameter, an index specifying which rule to delete.

SampleRule The `SampleRule` mechanism samples a new rule according to a grammar-based prior constructed from the signature of the TRS generated from the earlier revisions in the meta-program. This new rule is then prepended to the TRS. It takes a variable number of parameters specifying the structure of the rule atom by atom.

RegenerateRule The `RegenerateRule` mechanism chooses a unique location in an existing rule and replaces the subterm in that location with a newly sampled subterm of the same type. Like `SampleRule` it samples the subterm according to a grammar-based prior constructed from the signature of the TRS generated from the earlier revisions in the meta-program. The mechanism requires that the TRS contains at least one rule; the modified rule remains in its original order. The mechanism takes a variable number of parameters: the index of the rule, a vector encoding the location of the subterm, and one for each atom in the newly generated subterm. `SampleRule` and `RegenerateRule` provide a similar search dynamic to sampler-based models of learning in the LOT (e.g. Goodman, Tenenbaum, Feldman, et al., 2008; Ullman et al., 2012; Piantadosi et al., 2012, 2016), thus allowing HL to default to that approach should more structured mechanisms prove unfruitful. Operating at random or enumerating options exhaustively are sometimes the best options, but arguably only if other options have been exhausted (Sussman, 1973).

Variablize This mechanism selects a single subterm in a single rule in a TRS and replaces all occurrences of that subterm within the rule with a fresh variable. It therefore introduces an abstraction encoding the hypothesis that repetition within a rule is not accidental but represents a meaningful reuse of the same structure. The mechanism takes two parameters: the first indicating which rule to revise, and the second identifying the subterm to be replaced.

AntiUnify `AntiUnify` is the first mechanism which operates on an entire TRS rather than a single rule. It recursively attempts to anti-unify[10] each pair of rules in a TRS. If an

[10]Unification is a fundamental concept in computer science with a long history (Robinson, 1965; Martelli & Montanari, 1982; Baader & Snyder, 2001). It is the practice of substituting values for variables in a pair of expressions to find a most-general specialization of the two expressions. If variables in an expression are treated as gaps to be filled, unification finds a way to fill those gaps in a pair of terms such that they become the same term. The related concept of anti-unification looks for a way to replace values with variables to find a least-general generalization. It looks for ways to introduce gaps such that two terms become the same.

anti-unification can be found, the anti-unification is retained in place of the original rules. If not, the original rules are both kept, and the next rule is analyzed. Despite its apparent complexity, `AntiUnify` takes no parameters.

Compose This mechanism provides a way of hypothesizing that the effect of some function is actually best explained as the composition of functions. The mechanism looks for a subterm, T occurring on the left-hand side of one more rules and of type `t1 -> t1`—that is, a function which takes some type as input and returns the same type as output. It creates a new rule specifying that occurrences of T should be replaced with a composition of two new symbols, f and g, also of type `t1 -> t1`. For each rule using T, it then chooses a specific decomposition of the right-hand side of that rule, assigning responsibility for some portion of the right-hand side to f and another portion to g. The mechanism takes a single parameter specifying the term to be replaced and the way in which the right-hand sides should be decomposed. By focusing on introducing two new symbols of the same type, `Compose` actually implements a limited form of function composition. More generally, the two symbols could have different types so long as the output of the first served as the input to the second, and the input of the first and output of the second matched the overall input and output types, respectively, of the symbol they replace. This more general approach would have dramatically increased the implementation complexity of the mechanism, however, so this extension has been delayed to future work. The current mechanism does, however, allow for a limited form of conceptual change by introducing new symbols whose initial meaning is relatively unconstrained and can be changed by subsequent revisions.

Recurse The `Recurse` mechanism hypothesizes that some subterm in a TRS is best explained recursively. It specifically hypotheses that the subterm is structurally recursive, operating over successively smaller pieces of a recursively defined structure like numbers, trees, or lists. A structurally recursive function, f, consists of a general case describing the behavior of f on some recursive structure in terms of a recursive call to f, and a set of base cases which terminate the recursion. `Recurse` identifies a subterm representing a possibly structurally recursive function, and selects the portion of the recursive structure that might be affected by each application of the general case. Crucially, it does not describe a fully-formed recursive case. It leaves that to future revisions. It only identifies which

portion of the input and output structure the recursive case is responsible for processing. The mechanism then replaces each relevant rule with a set of rules created by repeatedly applying the general case until it cannot be applied and discovers a base case. `Recurse` is thus a refactoring mechanism. It typically makes a TRS longer but in doing so, exposes repeated structure that can often be abstracted away by a mechanism like `AntiUnify` or `Variablize`. The mechanism takes parameters specifying the recursive subterm and which portions of the left-hand side and right-hand side should be processed recursively.

Generalize The `Generalize` mechanism hypothesizes that one or more rules which are identical in all but a few subterms actually represent specific instances of a more general rule. It replaces each discrepancy on the left-hand side with a fresh variable, and it replaces each discrepancy on the right-hand side with a call to a new function symbol taking the variables from the left-hand side as arguments. It removes the original rules and replaces them with a copy of the generalized rule, as well as specific examples of each newly introduced symbol. In this way, `Generalize` refactors a set of rules into one or more subproblems, directly encoded in the TRS, which can be revised by future mechanisms. For the sake of efficiency, the current implementation limits the number of left-hand side discrepancies to one and the number of right-side discrepancies to three. The mechanism takes no parameters.

Stop Because HL operates iteratively, it can continue extending a given meta-program indefinitely. It is necessary to include a `Stop` mechanism instructing HL to consider the meta-program complete. It takes no parameters and has no impact on the TRS generated from the meta-program.

4.2.4 Chaining mechanisms into meta-programs

A large part of the success of hacking rests in the ability to iteratively revise programs. HL accommodates this by chaining its learning mechanisms together into meta-programs. These meta-programs describe a generative process for revising a seed TRS. If that seed TRS is fixed to a set of assumed background knowledge modeling the LOT before learning, the meta-program thus describes a generative process for revising that LOT over time, adding, removing, and redefining the meaning of various primitives in light of observed data.

The global structure of these meta-programs is straightforward. Each applies a linear se-

quence of mechanisms, $M_1, M_2, \ldots$, to some seed TRS, H_0, and terminates with `Stop`, written in the style of method chaining from object-oriented programming as $H_0.M_1.M_2.\ldots.$`Stop()`.

The local structure, however, is more complex. Only a few mechanisms—`AntiUnify`, `MemorizeAll`, `Stop`, and `Generalize`—are parameter-free. Other mechanisms, most notably `SampleRule` and `RegenerateRule`, require significant parameterization before they completely specify the next mechanism to be taken. Work on the next mechanism cannot begin until the current mechanism has been fully parameterized. Moreover, the various mechanisms available to HL can only be applied in specific contexts. `DeleteRule`, for example, requires that the TRS contain at least one rule. `MemorizeDatum` and `MemorizeAll` require that there be at least one datum which does not already appear in the TRS. Others, like `Compose` or `Recurse` require that the TRS contain rules with a specific structure. Decisions about which mechanisms can be chosen when and how each is parameterized are thus context-sensitive but can easily be computed given the TRS generated by the meta-program thus far.

To see how these mechanisms interact to improve learning, consider the following example. The goal is to learn a description of the function `F x`, where `x` is a list of natural numbers. Assume that the seed TRS, H_0, has symbols `Cons` and `Empty` for constructing lists, the numbers `0`–`99`, the basic arithmetic operators (e.g. `+`, `-`, `*`, and `/`), and rules describing arithmetic, but is otherwise empty:

(4.1)

H_0 contains no rules specific to understanding `F` and so is presented as an empty list of rules. Also assume that H_0 has access to four input/output pairs:

```
F [2, 5, 1, 4, 8] = [4, 2, 1]
F [3, 1, 4, 0, 3, 2, 2] = [9, 3, 4, 3]
F [0, 4, 6, 1, 4, 7, 1, 5, 8] = [0, 0, 6, 4, 1]
F [6, 0, 2, 5, 4, 9, 7] = [36, 6, 2, 4]
```

(4.2)

At this point, the only mechanisms available are `Stop`, `SampleRule`, `MemorizeDatum`, and `MemorizeAll`. HL cannot regenerate, delete, or generalize rules that do not yet exist. In

many cases, it does the same thing a human might do—it takes a look at the available data by selecting `MemorizeAll`. This gives:

```
F [2, 5, 1, 4, 8] = [4, 2, 1]
F [3, 1, 4, 0, 3, 2, 2] = [9, 3, 4, 3]
F [0, 4, 6, 1, 4, 7, 1, 5, 8] = [0, 0, 6, 4, 1]
F [6, 0, 2, 5, 4, 9, 7] = [36, 6, 2, 4]
```
(4.3)

As expected, all four observations have now been memorized, directly rewriting these known inputs to their known outputs. The TRS now accurately explains the data, but from the perspective of generalization, this first step appears unhelpful. It plays a potentially critical role, however, in that it provides direct access to the internal structure of the input/output pairs for refactoring and revision. For a human, a quick glance over the examples suggests that the first element of the output is unlike the others. It does not always appear in the list, while the others all appear to come from the list in order. Maybe the best way to tackle the task would be to split it into pieces, figuring out which elements are selected, and then figuring out how the first element is constructed. HL can capture this approach with the `Compose` mechanism[11]:

```
F x = H (G x)
G [2, 5, 1, 4, 8] = [2, 1]
G [3, 1, 4, 0, 3, 2, 2] = [3, 4, 3]
G [0, 4, 6, 1, 4, 7, 1, 5, 8] = [0, 6, 4, 1]
G [6, 0, 2, 5, 4, 9, 7] = [6, 2, 4]
H [2, 1] = [4, 2, 1]
H [3, 4, 3] = [9, 3, 4, 3]
H [0, 6, 4, 1] = [0, 0, 6, 4, 1]
H [6, 2, 4] = [36, 6, 2, 4]
```
(4.4)

This is a much longer TRS, but by hypothesizing that `F` is actually best explained as the interaction of two distinct processes, it carves the dynamics of `F` at a crucial joint. It factors

[11] `Compose`, like many of HL's learning mechanisms is parameterized. This example focuses on how the mechanisms compose to work together, so assume the appropriate parameterization is chosen for each mechanism.

F into two parts, G and H, which can be explained separately. This sort of decomposition is a basic pattern in human problem solving. Here, it effectively creates two subproblems that exist within the scope of the larger problem of explaining F. G captures the part of F which decides which elements of the input to copy over. Since the number of elements seems to vary, this process is likely recursive. Applying HL's Recurse mechanism to G gives:

```
G [8] = Empty
G [2] = Empty
G [7] = Empty
G [1, 4, 8] = (Cons 1 (G [8]))
G [2, 5, 1, 4, 8] = (Cons 2 (G [1, 4, 8]))
G [3, 2, 2] = (Cons 3 (G [2]))
G [4, 0, 3, 2, 2] = (Cons 4 (G [3, 2, 2]))
G [3, 1, 4, 0, 3, 2, 2] = (Cons 3 (G [4, 0, 3, 2, 2]))
G [1, 5, 8] = (Cons 1 (G [8]))
G [4, 7, 1, 5, 8] = (Cons 4 (G [1, 5, 8]))
G [6, 1, 4, 7, 1, 5, 8] = (Cons 6 (G [4, 7, 1, 5, 8]))
G [0, 4, 6, 1, 4, 7, 1, 5, 8] = (Cons 0 (G [6, 1, 4, 7, 1, 5, 8]))
G [4, 9, 7] = (Cons 4 (G [7]))
G [2, 5, 4, 9, 7] = (Cons 2 (G [4, 9, 7]))
G [6, 0, 2, 5, 4, 9, 7] = (Cons 6 (G [2, 5, 4, 9, 7]))
F x = H (G x)
H [2, 1] = [4, 2, 1]
H [0, 6, 4, 1] = [0, 0, 6, 4, 1]
H [3, 4, 3] = [9, 3, 4, 3]
H [6, 2, 4] = [36, 6, 2, 4]
```

(4.5)

Again, the length of the TRS grows dramatically, but the behavior of G has been neatly unrolled. The notation might look odd, but the behavior is fairly similar to something a human might do. HL scans along the list, repeatedly breaking the list into similarly-sized pieces and correlating them to sequential pieces of the output until it gets to the end of list, where it writes down a separate rule for terminating the recursion. That is, it aligns the

input and output as it processes the recursion. This would be a poor final solution, but refactoring in this way shows that the base cases (lines 1–3) share most of their structure, as do the recursive cases (lines 4–15). Seeing this kind of repeated structure is a good signal that some kind of abstraction is probably necessary. For HL, the `AntiUnify` mechanism readily introduces appropriately general descriptions for both cases:

```
G [x] = Empty
G (Cons x (Cons y z)) = (Cons x (G z))
F x = H (G x)
H [2, 1] = [4, 2, 1]
H [0, 6, 4, 1] = [0, 0, 6, 4, 1]
H [3, 4, 3] = [9, 3, 4, 3]
H [6, 2, 4] = [36, 6, 2, 4]
```
(4.6)

`G` has now been neatly explained via two simple rules. `AntiUnify` is an exceptionally powerful mechanism, introducing multiple variables in a highly structured way that recognizes repeated structures across rules. It is in this way analogous to the sort of analogy proposed to crucial to children's learning (Carey, 2009; Gentner, 1983). It captures an important way of articulating abstraction that is nonetheless parameter free. Turning to `H`, it appears that most elements are directly copied. The `Variablize` mechanism recognizes this kind of repeated structure within a rule and uses a variable to abstract over that structure. One way to proceed, then, is for HL to use this mechanism to ignore all the copied elements except those adjacent to the first element. Abstracting over repeated structure within a rule is another limited form of analogical reasoning built around the hypothesis that the repetition is more important than the specific structure being repeated. This significantly simplifies the rules for `H`:

```
H (Cons 2 y) = (Cons 4 (Cons 2 y))
H (Cons 0 y) = (Cons 0 (Cons 0 y))
H (Cons 3 y) = (Cons 9 (Cons 3 y))
H (Cons 6 y) = (Cons 36 (Cons 6 y))
G [x] = Empty
G (Cons x (Cons y z)) = (Cons x (G z))
F x = H (G x)
```
(4.7)

At this point, the `H` rules are identical save for a single subterm on the left-hand side and two subterms on the right-hand sides. One way to continue is to try to explain the differences in the right-hand side in terms of the differences on the left-hand side. The `Generalize` mechanism proposes a structural relationship between these subterms:

```
H (Cons x y) = (Cons (J x) (Cons x y))
J 2 = 4
J 0 = 0
J 3 = 9
J 6 = 36
G [x] = Empty
G (Cons x (Cons y z)) = (Cons x (G z))
F x = H (G x)
```
(4.8)

A human might immediately recognize the functional relationships at play here. Right now, HL can recognize simple things like copying, but it has no learning mechanisms for dealing with numerical and arithmetical structure unless that structure is explicitly represented via something like Peano arithmetic. In this case, where it is presented with a base-10 encoding, its structure-based search mechanisms will fail to discover the squaring relationship encoded in `J`. Because the relationship is fairly simple, however, random sampling can close the gap. The `SampleRule` mechanism could easily propose that `J x = * x x`, giving:

```
J x = * x x
H (Cons x y) = (Cons (J x) (Cons x y))
G [x] = Empty                                                    (4.9)
G (Cons x y) = (Cons x (G y))
F x = H (G x)
```

Because this chain of revisions could continue indefinitely, HL has to explicitly decide that it's done and generate a complete meta-program by applying the `Stop` mechanism. The result is the same as TRS 4.9. This final language is significantly shorter and simpler than the data, yet is completely consistent with it. Moreover, it embodies a general hypothesis about the domain and consequently covers many lists which the original could not. It discovered this hypothesis primarily by making structured transformations rather than through blind sampling or exhaustive enumeration. Moreover, these transformations introduced three new primitives into the language. When the structured methods failed, HL was able to rely on random sampling to produce the necessary small changes to complete the language. Overall, the meta-program approach required just nine mechanisms to describe these data. Moreover, this process is similar to the process a human might take to solve this process. While the discussion likely took much longer to read than it would have to perform as a learner, the steps are similar. Both HL and people scan the data, identify structural similarities, abstract them into general patterns, and repeat until discovering a sufficiently compact description of the data.

4.2.5 Monte Carlo tree search

At a given point in a meta-program, the structure of which mechanisms can be chosen next and how each can be parameterized depends heavily on the TRS described by the mechanisms used up to that point. These options can be easily computed given that TRS, but they cannot be compactly described as a grammar. It is instead easier to explicitly organize the possible meta-programs as a tree, with the root representing the seed TRS, H_0, each branch representing the application of a possible mechanism, and each child representing the outcome of revising its parent by applying the mechanism specified by the connecting

branch. Paths from the root continue indefinitely until encountering a `Stop` branch. Each leaf thus represents the result of applying a complete meta-program to H_0—the meta-program specified by the path from the root to that leaf. Intuitively, such a tree represents the space of all possible ways for HL to revise H_0, with each leaf representing a possible outcome.

This tree, however, is unboundedly large. Given the finite time and computational resources available during learning, a learner can only explore a portion of this tree; ideally, only those parts which describe the most valuable meta-programs, those providing the best explanations of the observed data. If the learner knew these paths in advance, however, there would be no need to learn at all. The learner could simply report the known best options. Instead, the learner needs a way to iteratively construct a partial search tree which favors those parts of the full search tree which are likely to lead to valuable solutions while at the same time remaining cognizant of its own ignorance. As a terminological note, this partial tree now contains two kinds of leaves. Some follow a `Stop` branch and thus represent the application of a complete meta-program. Call these leaves *terminal* nodes. Others represent the application of partial meta-programs that must be extended to become complete. Call these leaves *unvisited* nodes, as they have not yet been selected and expanded during search.

Monte Carlo tree search (MCTS; see Browne et al., 2012, for a detailed review) provides a solution to this problem. MCTS is an algorithm for iteratively constructing a partial search tree like the one described above. It has been actively explored in recent years, and there are many variants. The basic algorithm used in HL is as follows (Figure 4-6):

1. **Selection**: Starting from the root node, repeatedly select children according to some selection algorithm until reaching an unvisited node, P.
2. **Expansion**: Select a branch, M, leaving P, and create a child node, C, by applying the mechanism at M to the TRS at P.
3. **Simulation**: If M is `Stop`, score the resulting hypothesis. If not, simulate a possible completion by randomly selecting possible moves until selecting `Stop`. Score the resulting hypothesis.
4. **Backpropagation**: Use the score, S, gained during Simulation to update statistics along the path leading to C. These statistics are used during Selection. If C is a

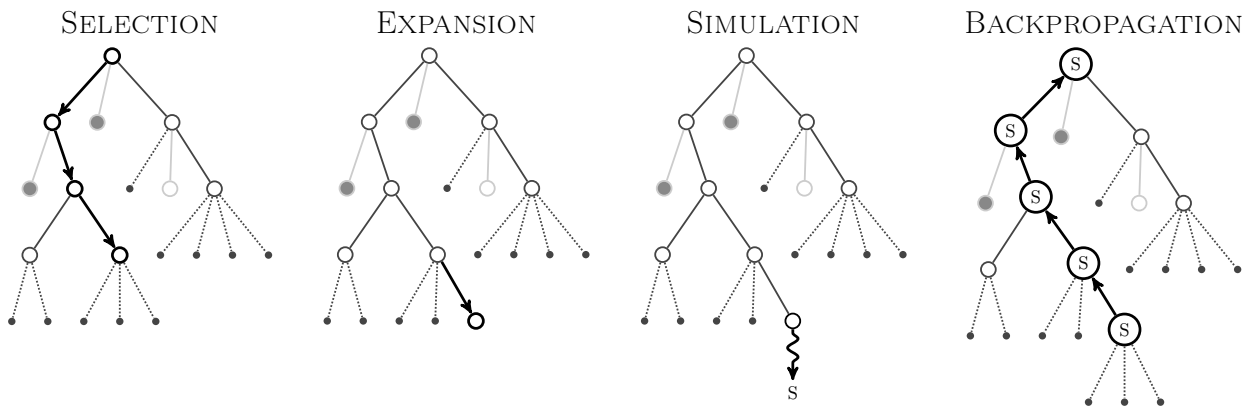


Figure 4-6: A visual overview of Monte Carlo Tree Search: each of the four stages are illustrated. Visited nodes are white, terminal nodes are gray, and unvisited nodes are small and dark gray. S represents the score acquired during simulation. Nodes which are pruned, either due to a failed mechanism, or reaching a terminal node, are outlined in a lighter gray. Image based on Figure 1 of James et al. (2017).

terminal node, or if there are no possible moves leaving C, mark C as fully explored, as well as any ancestor nodes which are similarly fully explored. This information is used to ensure that Selection always reaches an unvisited node. Otherwise, add unvisited nodes for all new branches.

Completing these four steps constitutes a single iteration. Each iteration adds a single leaf node to the search tree and adds a single scored meta-program to the set of explored hypotheses, either by simulation or by reaching a terminal node. This iterative nature means MCTS can be stopped at anytime, producing as output the set of all explored hypotheses. These hypotheses can then be queried in any number of ways, e.g. to identify the best program seen so far.

Intuitively, the dynamics of MCTS can be interpreted as iteratively expanding a set of possible futures. Each path in the search tree represents a particular line of thinking about how to proceed from the current state. Each iteration adds a single step to a single line of thinking based on how promising it seems to the learner. The learner might extend the same line of reasoning for several steps, or it might jump from one chain to the next, uncertain about where the most promising options are likely to be. In HL, each extension is deterministic and structured, but the choice of which extension to make is stochastic. The overall effect is that HL considers many ways in which it might revise a given TRS, noisily

jumping between them in search of options that provides a good explanation of the data.

MCTS was originally developed as part of artificial intelligence agents for two-person board games like Go (Silver et al., 2016). Learning mental representations, however, is a significantly different task, and we thus alter the basic MCTS algorithm in several key ways. First, two-player MCTS alternates between the moves of the agent and the moves of that agent's opponent. HL, however, models the learning of a single individual, with no opponent, and so expands something more akin to a single-player tree. Second, two-player MCTS typically allows for the SELECTION of the same leaf node multiple times. HL, however, is seeking to explore as many TRSs, as many LOTs, as possible. When search stops, it can select the best TRS seen so far. This is based entirely on the score assigned during SIMULATION, so no new information can be gained by visiting a node twice. Third, two-player MCTS is fundamentally more uncertain, because the behavior of the opponent can be explored but not fully predicted. It is thus important to seek areas in the search tree that are valuable *on average*, areas which maximize the probability that the agent will win, no matter what the opponent does. HL cares more about finding areas in the search tree containing the most valuable *individuals*. Rather than considering the mean value of a subtree, it considers the maximum values in that subtree.

The fact that HL is seeking good individuals rather than good subtrees is directly reflected in its selection algorithm. The goal of the selection algorithm is, for a given parent node P with child nodes $c_1 \ldots c_n$, to select some child node c_j maximizing some objective function. It is thus a form of the multi-armed bandit problem from reinforcement learning (Sutton & Barto, 2018); our focus on seeking maximally good individuals is a variant known as the max bandit or extreme bandit problem (Cicirello & Smith, 2005; Carpentier & Valko, 2014). The SELECTION step, described above, repeatedly uses this selection algorithm until it reaches an unvisited node. The selection algorithm thus plays a key role in biasing which portions of the search tree get explored. For its selection algorithm, HL uses a variant of Thompson sampling (Thompson, 1933) parameterized by a schedule $S : \mathbb{N} \to \mathbb{R}^+$ and a count $N : \mathbb{N}$. For the experiments reported in this thesis, we use $S(t) = 5/\ln(1+t)$ and $N = 10$. Assuming that each node n tracks the number of times it has been visited $\mathrm{v}(n)$ (i.e. the number of nodes in the subtree rooted at n), and an array, $\mathrm{t}(n)$, of the N highest scores seen during

the SIMULATION step for any node in the subtree, the algorithm is as follows:

1. If `Stop` is available and its child unvisited, visit it.
2. If there are other unvisited children, select one at random.
3. If all children have been visited, return

$$\underset{c_{1\leq i\leq n}}{\text{argmax}}(\text{MaxThompson}(S, N, c_i)).$$

For a subtree rooted at some node n, MaxThompson(S, n) first adjusts each score $\text{t}(n)_{1\leq i\leq N}$ as $e^{\text{t}(n)_i/S(\text{v}(n))}$. It then randomly samples an adjusted score in proportion to its value.

This selection algorithm has several useful features. First, it uses the `Stop` mechanism whenever possible and is thus explicitly biased toward short meta-programs. Second, it balances exploration of relatively underexplored paths against exploitation of known good paths. Third, the exploration bias is strongest when a subtree has never been visited (i.e. available moves which have not yet been taken are always taken). Fourth, this bias weakens monotonically with each subsequent visit, assuming that $S(x)$ monotonically decreases as x increases. This means that nodes which have been visited many times will only continue to be visited if their subtrees contain high-scoring meta-programs relative to the alternatives. Fifth, because the algorithm tracks only the N top scores in each subtree, it is sensitive only to the best individuals in a subtree, rather than to the mean individual. When N is larger than one, the algorithm does, however, have a limited ability to favor nodes with several high-scoring individuals. For example, when each c_i has been visited a moderate number of times, the bias toward exploitation will tend to favor either nodes whose subtrees contain a single exceptionally good score or nodes whose subtrees contain several moderately good scores. In sum, this selection algorithm provides a computationally efficient means of favoring nodes which remain relatively underexplored or which have been shown to contain one or more exceptional TRSs, increasingly favoring the latter as search progresses.

Another notable feature of HL's MCTS algorithm is the way it explicitly represents mechanism parameterizations in the structure of the search tree using using *move groups* (Childs et al., 2008). If a mechanism has parameters, those parameters represents various

ways of modifying the behavior of the mechanism. Each distinct parameterization reflects a different set of modifications. Some mechanisms have a fairly simple parameterization, requiring only a single choice. For example, the `DeleteRule` mechanism only requires selecting which rule will be deleted. In this cases, the overall `DeleteRule` mechanism is decomposed into two steps. First, HL selects `DeleteRule`, as opposed to other possible mechanisms like `SampleRule`, `RegenerateRule`, `Stop`, etc. After selecting `DeleteRule`, it then faces a decision between `DeleteRule(1)`, `DeleteRule(2)`, and so on. After choosing any one of these, it returns to deciding among mechanisms: `SampleRule`, `RegenerateRule`, `Stop`, and so on. As mechanism parameterizations grow more complex and multiple decisions must be made in sequence, the complexity of the subtree representing the various decisions necessary to complete the parameterization also grows. For a mechanism like `SampleRule`, which imposes no bound on the size of the sampled rule, this subtree can grow unboundedly large. Subsequently, only the most valuable portions will ever be explored.

The overall effect is to decompose parameterized mechanisms and explicitly represent the structure of the decisions necessary to construct a specific instantiation of the mechanism. By explicitly representing the structure, HL can learn not only which mechanisms are useful, and which parameterizations of that mechanism, but it can also learn which parts of that parameterization are useful. It can learn, for example, that it is helpful to regenerate the first rule but not the second, and that within the first rule, it is most helpful to regenerate a specific subterm on the left-hand side.

Finally, HL's MCTS implementation supports online learning. That is, it is capable of updating its behavior as new data become available. To do so, it makes use of a second form of pruning. When a new input/output pair becomes available, HL identifies the highest scoring paths in its current search tree (the exact number is configurable, but this thesis uses 100) which are still valid given the new data. A path can be invalidated if, for example, it contains `MemorizeAll`, followed by a mechanism whose preconditions are no longer met given the new data. It records these paths and then reconstructs the search tree to contain just these paths, adding any new mechanisms made possible by the new datum, deleting mechanisms whose preconditions are no longer met, running new simulations, and updating the scores for each meta-program. For moderate numbers of retained nodes, this process is

fast yet still retains the most useful hypotheses.

4.3 Learning Objectives: simple, accurate, discoverable, and well-formed

Chapter 2 argues that hackers and learners actively maintain and navigate a network of goals rather than pursuing a single objective function. This section describes one particular reason for which learners maintain multiple objectives. It specifically examines ways in which people are often sensitive to good structure in suboptimal hypotheses, entertaining ideas they know are deficient, perhaps flatly wrong, if they are beneficial in useful ways. It then relates this to the hacker maxim of avoiding premature optimization and details the implementation of these ideas in HL. HL scores hypotheses differently when building the search tree than when deciding which, of all the hypotheses it's found, to use for making predictions. It specifically relaxes its objective during search to make it more likely to explore seemingly suboptimal revisions that have the right overall structure. Moreover, the model favors TRSs which provide a good fit to observed data and produce well-formed guesses for novel inputs while being easy to find and relatively simple.

4.3.1 Abstract error maps

Entertaining hypotheses which are known to be wrong is a common feature of human thinking. It is in fact often the case that a learner understands not just that the hypothesis is wrong but also which parts of it are wrong and why. Laura Schulz (2012a) calls these detailed understandings of incorrect hypotheses *abstract error maps* and argues that they are an essential guide during learning. To illustrate, she gives the following example of an interaction she had with her daughter, Adele:

> Adele (age 4): "Mommy, I know why they make you turn off your phone when the plane is taking off."
>
> Laura: "Oh really? Why?"
>
> Adele: "Because when the plane takes off it's too noisy to talk on the phone."

To anyone who has flown as an adult, this explanation is obviously incorrect in nearly all

its particulars. It is, however, also importantly right in several key respects: 1) it involves phones and planes; 2) it posits a causal link between them; and 3) the causal link could result in the airline asking passengers to power off their phones during takeoff. In these respects it is a good explanation. It is true that the causal link is incorrect and even reverses the direction of causality, but these details can be revised. The essential structure of the hypothesis is right. Adele did not propose, for example, that we turn phones off on planes because camels dislike cheese or because two and three makes five. Neither links phones and planes, and the latter fails even to be causal. These are simply the wrong kinds of hypotheses to consider and are hardly worth revising.

This knowledge of which parts of the hypothesis are useful and which need revision is precisely what is included in an abstract error map[12]. Abstract error maps thus help to constrain the immediate goals and hypotheses we consider, making hypothesis-and-goal-driven search a viable approach to learning. By making it clear where an explanation stands up and where it needs further work, abstract error maps allow a learner to efficiently allocate their effort, expending energy primarily on those aspects which are most in need of improvement. Repeatedly updating such an error map allows a learner to iteratively improve a hypothesis until arriving at a satisfactory explanation.

4.3.2 Avoiding premature optimization

Hackers operate over a complex space of complex objectives. First, their goals are complex. Rather than being a function of just one or two dimensions—such as the tradeoff between accuracy and description length that is common in Bayesian models of cognition (Ullman et al., 2012; Ullman & Tenenbaum, 2020)—they routinely involve a much larger number of dimensions. These include: classic normative constraints like description length and accuracy (Chater & Vitányi, 2003; Baum, 2004); engineering considerations like efficient use of energy, memory, and computation, as considered in resource rational analysis (Lieder & Griffiths, 2020; Griffiths et al., 2015; Lewis et al., 2014); aesthetic considerations such as

[12]As Schulz notes, the map may include signals which are not about error per se, i.e. the gap between data and explanation. It is instead intended to capture the general notion of our awareness of any place where our hypotheses fail to meet our own subjective standards of explanatory adequacy.

elegance, clarity, and cleverness (Abelson et al., 1996); basic measures of utility as might be considered as part of a naive utility calculus Jara-Ettinger et al., 2016; novelty (Lehman & Stanley, 2011b, 2011a); fun; and a host of software engineering principles like modularity and reusability (Martin, 2009; Thomas & Hunt, 2019). Second, hackers make use of multiple objective functions. Rather than maintaining a fixed objective function, always seeking to maximize the same combination of values, hackers make use of an entire space of possible objectives, shifting between them based on context and past experience. Third, hackers want to produce better code. They are interested in writing the best code possible. This may be the most basic reason for hacking. They are therefore less interested in exploring the space of all possible programs than they are in quickly finding the best programs in this space.

One important technique which hackers use to manage their movement through the space of objectives is avoiding premature optimization (Knuth, 1973). The technique is typically invoked to explain why a hacker might choose to deliberately ignore certain values, such as making a program fast or short in order to more heavily value other things, like making a program accurate or sufficiently general. She knows that her code is suboptimal in certain important ways, but explicitly avoid fixing some aspects to focus on others. The hope is that, in time, she will be able to broaden her values, incorporating additional dimensions like speed or description length and thereby produce even better code. Trying to manage a large number of values early in a program's development, however, frequently leads to code which scores poorly on all dimensions. This is by no means the same idea as abstract error maps, but it is deeply related. In both cases, the agent maintains an awareness of a hypothesis' strengths and weaknesses and is willing to continue working with it despite known weaknesses in hopes that future work will be able to resolve them.

There is a great deal more to say and to learn about the internally-motivated nature of goals, the way that hackers actively manage which dimensions of value they are attempting to improve in their code at any given time, and the way they select goals in service of these values. This is the work of another thesis, if not several others. The implementation of HL focuses on applying the considerations outlined above in the following ways. First, it defines multiple distributions over programs, including a search-based prior over meta-programs assessing ease of discovery, a grammar-based prior assessing TRS description

length, a trace-based likelihood over TRSs assessing fit to data and including a weak bias toward computational efficiency, and a trace-based likelihood over TRSs assessing a TRS's ability to produce well-formed outputs for novel inputs. Second, it combines these various distributions into a posterior assessment of a meta-program's value using an extensible framework based on mixture distributions and products of experts. Third, HL uses two distinct but related objective functions depending on its current task and phase of learning. One guides search itself (i.e. the iterative expansion of the search tree), and the other decides which meta-program, of all those observed in the search tree, to use as the basis for making novel predictions. This latter objective basically acts like a Bayesian posterior over meta-programs, while the former relaxes several terms to help HL entertain suboptimal hypotheses that might be improved given additional search.

In classic two-player MCTS, the objective function is typically very simple, scoring simply whether a given pattern of play represents a win, loss, or draw for each player. It was not therefore necessary to take a Bayesian approach to objectives in HL. The decision to do so, however, brings a host of benefits, including: the ability to interpret scores as subjective degrees of belief in the goodness of individual hypotheses, principled mechanisms for incorporating data-independent and data-dependent contributions as data accumulate, a deep literature of formal tools, and a long history of successful application in cognitive science. See Ullman and Tenenbaum (2020) for more detailed discussion of these issues.

4.3.3 A prior assessing discoverability

HL uses MCTS to construct a search tree. The path to a given non-leaf node represents a partial meta-program, and the branches leaving it represent each possible way to continue that meta-program. Taking a particular branch thus represents a choice, namely to continue the meta-program in the fashion described by this branch as opposed to any of the others. Each path then represents a series of choices, one for each node in the path. Some choices might discriminate among dozens or even hundreds of options, while others might be trivial choices with only a single option.

Assuming that each branch is chosen at random, the prior probability of a given meta-program is simply the inverse product of the number of options leaving each non-terminal

node. Because HL selects each branch leaving a node at least once before selecting any a second time, this prior also represents how easily HL can discover a meta-program. Meta-programs requiring few choices, each of which presents just a few options, are easy to discover. Meta-programs which are long or which contains choices among hundreds of options, by contrast, are harder to discover.

We can formalize these intuitions by defining the prior probability, $p(M \mid T)$, of a meta-program, $M \equiv H_0.M_1.M_2...M_{m-1}.\texttt{Stop()}$, given a search tree, T, with root TRS, H_0, as follows. Recall that parameterized mechanisms are represented by a series of branches in the tree. Let $\text{path}_T(M)$ be the series of branches in T which represents M, and $\text{siblings}_T(b)$ returns a set of containing all the branches sharing the same parent as b in T. Then:

$$P_{\text{META}}(M \mid T) = \prod_{b \in \text{path}_T(M)} \frac{1}{|\text{siblings}_T(b)|} \tag{4.1}$$

4.3.4 A prior assessing simplicity

Each meta-program M in a search tree T can be applied to the seed TRS, H_0, represented by the root to produce some output TRS H_M. The search-tree prior in the previous section, $P(M \mid T)$, describes the complexity of M and thus provides an upper bound on how difficult it is to find H_M. It does not, however, tell us about the complexity of H_M itself. It focuses on the complexity of the generative process rather than the complexity of the final result. For example, say HL had made two observations. In that case, $M_1 \equiv \texttt{H}_0\texttt{.MemorizeAll().Stop()}$ and $M_2 \equiv \texttt{H}_0\texttt{.MemorizeDatum(1).MemorizeDatum(2).Stop()}$ would compile to the same TRS (i.e. $H_{M_1} = H_{M_2}$), but $P(M_1 \mid T) > P(M_2 \mid T)$, because M_1 has a simpler generative process than M_2. Moreover, after three additional observations, the complexity of M_1 as a generative process would remain the same, but H_{M_1} would contain three additional rules, one for each new observation.

For a given TRS, $H = (\Sigma, R)$, we can, however, define a grammar-based prior, $P(R \mid \Sigma, p, w)$, which is sensitive to the complexity of the rules of H itself. Intuitively, this prior corresponds to repeatedly flipping a coin to decide how many rules to include in R, and then using Σ to sample each rule in turn. Like other grammar-based priors (e.g. Goodman, Tenenbaum,

Feldman, et al., 2008), it penalizes complexity and favors small sets of simple rules.

Formally, we can define $P(R \mid \Sigma, p, w)$ as follows. Let p be the stopping probability and $w = (w_c, w_f, w_v, w_i)$ be a tuple containing the weights associated with constant operators, function (non-constant) operators, reusing an existing variable, and inventing a new variable, respectively:

$$P_{\text{TRS}}(R \mid \Sigma, p, w) = p(1-p)^{|R|} \prod_{i=1}^{|R|} P_{\text{RULE}}(R_i \mid w, \Sigma) \tag{4.2}$$

Let $w' \equiv (w_c,\, w_f,\, w_v,\, 0)$. Then, for a given rewrite rule, $l \approx r$, where τ is the type associated with $l \approx r$ and Γ is the typing environment generated by computing τ:

$$P_{\text{RULE}}(l \approx r \mid w, \Sigma) = P_{\text{TERM}}(l \mid w, \tau, \Gamma, \Sigma)\, P_{\text{TERM}}(r \mid w', \tau, \Gamma, \Sigma) \tag{4.3}$$

Then, for a given term m, assuming that $[m]$ returns a preorder traversal of m and $\text{var}(x)$ returns the set of variables observed in a sequence of atoms:

$$P_{\text{TERM}}(m \mid w, \tau, \Gamma, \Sigma) = \prod_{i=1}^{|m|} P_{\text{ATOM}}([m]_i | w, \tau, \Gamma, \text{var}([m]_{1\ldots i}), \Sigma) \tag{4.4}$$

Finally, for a given atom, a, and sequence of previously observed variables, v:

$$P_{\text{ATOM}}(a \mid w, \tau, \Gamma, v, \Sigma) = \frac{\text{weight}(a, w, \tau, \Gamma, v, \Sigma)}{\sum_{x \in \Sigma \cup v} \text{weight}(x, w, \tau, \Gamma, v, \Sigma)} \tag{4.5}$$

where, assuming $\doteq$ represents type unification and $\Gamma(a)$ represents the type of a given Γ:

$$\text{weight}(a, w, \tau, \Gamma, v, \Sigma) = \begin{cases} 0 & \text{if } \Gamma(a) \not\doteq \tau \text{ or } a \text{ is an operator and } a \notin \Sigma \\ w_c & \text{if } a \in \Sigma \text{ and } \text{arity}(a) = 0 \\ w_f & \text{if } a \in \Sigma \text{ and } \text{arity}(a) > 0 \\ w_v & \text{if } a \in v \\ w_i & \text{if } a \text{ is a variable and } a \notin v \end{cases} \tag{4.6}$$

4.3.5 A likelihood assessing accuracy

Each TRS $H \equiv (\Sigma, R)$ defines a rewrite relation $\rightarrow_R$. For a given input term t_i, $t_i \rightarrow_s t_o$ indicates that t_i can be rewritten to t_o in a single rewrite step. Depending on the rewriting strategy and whether s is nondeterministic, there may be several possible single step rewrites for any given input. Moreover, it may be possible for the output of any of these rewrites to serve as input to subsequent rewrites. These rewrite steps can be organized as a tree called an evaluation trace. The root node represents the initial input, t_i, each outgoing branch represents rewriting a particular subterm of t_i according to a particular branch of a particular rewrite rule in R, and the corresponding child node represents the outcome of this rewriting step.

Given a particular input/output pair (i^*, o^*), HL assesses the likelihood of using H to rewrite i^* to o^* using the following generative model. It first flips a coin with weight p. If the coin flip succeeds, it stops and returns i^* as the output. Otherwise, it considers all possible rewrites of i^*, chooses one at random, and makes the output of that rewrite its new input. It repeats this process until producing an output. This generative process defines a distribution over input/output pairs, written $P_{\text{TRACE}}(n \mid i, H)$, associating a probability with every node, n, in the evaluation trace rooted at i^*. Outputs which are reached quickly and which compete with few alternative outputs have a high probability, while those which take many rewrite steps or for which certain steps could have resulted in many distinct rewrites have a correspondingly lower probability. Depending of the setting of p, this distribution also favors earlier outputs over later outputs, a weak efficiency bias.

It would be simple to convert this generative process to a likelihood by summing up the probability of each node n for which the associated output $o_n = o^*$. This likelihood, however, assigns no partial credit. If o_n is not exactly equal to o^*, n contributes nothing to the likelihood. HL resolves this problem by amending the generative model above. After producing an output, it then assumes that the output is probabilistically corrupted. In the case of the list function domain, it assumes elements are inserted, deleted, or randomly replaced, each with a certain probability according to the algorithm described in Kashyap and Oommen (1984). For a given observed output, o, target output, o^*, and parameterization

θ, we write this distribution as $P_{\text{STRING}}(o \mid o_n, \theta)$. In combination, with p_{TRACE}, the resulting distribution then allows each node in the trace to contribute to the likelihood based on the combined cost of reaching the node n and corrupting o_n into o^*.

HL defines the likelihood of a set of input/output data, D, $P(D \mid H, \theta)$ as follows, where $\mathbf{1_o}(o_n)$ is an indicator function testing whether $o_n = o$, and $\text{Trace}_H(i)$ is an evaluation trace rooted at i:

$$P_{\text{ACC}}(D \mid H, \theta) = \prod_{(i,o) \in D} \sum_{n \in \text{Trace}_H(i)} \mathbf{1_o}(o_n)\, P_{\text{TRACE}}(n \mid i, H)\, P_{\text{STRING}}(o_n, o, \theta) \tag{4.7}$$

Because the trace may be unboundedly large, HL artificially limits the depth and maximum number of nodes in the trace. Both parameters are configurable.

4.3.6 A likelihood assessing well-formedness

In the online learning experiments described in Chapters 5 and 6, learners are asked to make a series of predictions about list function problems. For each problem, they are given an initial input and asked to predict its output. After making their prediction, they are given the correct output, shown a novel input, and asked to make another prediction. At all times, the correct input/output pairs for previous trials remain on screen as well as a single novel input. The learner may or may not have a strong hypothesis about how the list function concept will apply to this particular novel input, but they do know at least one thing. They know that it will apply and produce a literal output list. It will not produce a number, nor a partially evaluated program, nor anything else except a literal list. Given the sometimes rich structure of the input lists, not all possible inputs are guaranteed to produce literal outputs. The inputs provided during the experiment, however, are always guaranteed to produce meaningful outputs. This knowledge allows the learner to rule out any hypothesis which fails to produce a literal output list for the novel input. It therefore provides a reliable test about a hypothesis' ability to produce well-formed outputs and acts as a form of query-guided search (Chu et al., 2019). The signal is admittedly weak—it cannot tell the learner whether a given hypothesis will produce correct outputs—but it is nonetheless helpful in

constraining search.

Intuitively, HL can use this information to construct a very simple likelihood function favoring hypotheses that satisfy this well-formedness constraint. For each TRS, H, it evaluates the novel input, i, to construct a trace. Each node, n, contributes its probability under the trace weighted according to whether it is a literal list. For a set of inputs I, HL formalizes this likelihood as follows:

$$P_{\text{WF}}(I \mid H, \alpha) = \prod_{i \in I} \sum_{n \in \text{Trace}_H(i)} \text{LIT}_\alpha(o_n)\, P_{\text{TRACE}}(n \mid I, H) \tag{4.8}$$

where

$$\text{LIT}_\alpha(o) = \begin{cases} 1-\alpha & \text{if o is a literal list} \\ \alpha & \text{otherwise} \end{cases} \tag{4.9}$$

4.3.7 Two objectives for HL

In the list function experiments reported in Chapters 5 and 6, HL makes use of two different objective functions. The first is used during search: it is computed during the SIMULATION step and used during the SELECTION step. The second is used after search when deciding which, of all known meta-programs, to select, compile into a TRS, and use as the new LOT. This LOT is then used to make predictions on novel inputs. This is a first step toward a model which adopts objective functions that allow it to consider and revise suboptimal hypotheses in a style similar to the way people deploy abstract error maps during learning or the way in which hackers avoid premature optimization.

HL uses four distinct probability distributions to assess the value of a meta-program—a prior distribution over meta-programs favoring discoverability, a prior distribution over TRS rules favoring simplicity and generalization, a likelihood over input/pairs favoring accuracy and efficiency, and a likelihood over lone inputs favoring well-formed guesses. Table 2.1 suggests that future hacker-like models of learning may want to include many additional terms. Bayesian frameworks for learning in the LOT, however, most commonly feature posteriors combining just two distributions: a prior favoring expressions simplicity and a

likelihood favoring accuracy.

HL resolves this tension by combining distributions using tools from probabilistic logic in the following way (Pearl, 1988).

During prediction, HL favors meta-programs that produce a short TRS that accurately explains previously observed input/output pairs and makes well-formed guesses about novel inputs. All else being equal, it also favors short meta-programs. It is valuable to select a short TRS because accurate predictions require hypotheses that generalize well. A hypothesis is more likely to generalize if it is short, because it has less capacity for memorizing specific exceptions. It is also valuable to have a hypothesis which fits the previously observed data as well as possible, and it is necessary that the hypothesis produce a well-formed guess for novel inputs. Hypotheses which score poorly on either of these two criteria hardly count as solution at all. All else being equal, efficiency considerations also suggest favoring hypotheses which are easy to find. Because search has ended, however, it is senseless to favor efficiency at the cost of probable generalization. So, HL uses the following objective during prediction for data, D, and meta-program, M:

$$\begin{aligned} &P_{\text{PRED}}(M|D,T,p,w,\theta) = \\ &\quad P_{\text{TRS}}(M\,|\,p,w)\ \cdot\ P_{\text{ACC}}(D\,|\,M,\theta)\ \cdot\ P_{\text{WF}}(D\,|\,M,0))\ \cdot\ P_{\text{META}}(M\,|\,T) \end{aligned} \tag{4.10}$$

The product of P_{WF} and P_{ACC} form an unnormalized likelihood, where P_{WF} is parameterized to act as a hard bias toward well-formedness (i.e. $\alpha = 0$). The product of P_{TRS} and P_{META} acts as a prior. Both products are a sort of noisy-and (Pearl, 1988). The likelihood favors hypotheses which are accurate *and* produce well-formed guesses on novel inputs. The prior favors hypotheses with simple meta-programs *and* simple TRSs. In cases where multiple TRSs are equally simple, this prior favors the TRS generated by the simplest meta-program[13]. Their product is then proportional to a Bayesian posterior favoring simple meta-programs that generate TRSs which are also simple, accurate, and generate well-formed guesses. The

[13]This all-else-being-equal property only holds when all else is truly equal. Another approach would be needed to select, say, the simplest meta-program which generates the simplest TRS. This might be possible, for example, with an appropriately tuned softmax function, though exploring this and other formalizations remains future work

overall effect is to select the simplest meta-program leading to the best TRS.

HL relaxes both the prior and the likelihood during search. It relaxes the prior by favoring hypotheses which either have a short meta-program *or* compile to a short TRS. This is accomplished using a noisy-or operator. A short TRS is likely to generalize rather than memorize, while a short meta-program is fairly easy to find and compile into a TRS. Ideally, hypotheses score well on both dimensions. During search, however, we may not want to rule out hypotheses which score poorly on one of these dimensions. It is possible that a long TRS with a short meta-program can be shortened by applying a small number of additional learning mechanisms. Also, a short TRS with a long meta-program has already been discovered. The effort required to find it should not count against it, because the effort has already been expended. It relaxes the likelihood by softening the hard bias toward well-formedness (i.e. $0 < \alpha \leq 1$). This is useful because the search tree is built iteratively and takes the `Stop` mechanism whenever possible. It thus accumulates information about the opening mechanisms of meta-programs much more rapidly than it does for later occurring mechanisms. It is also often helpful in the first few steps to memorize and refactor the data. During this time, hypotheses typically do not generalize. Being relatively insensitive to generalization during search allows HL to persist through this phase and discover refactorings that do generalize well. It is otherwise led astray by hypotheses are short, easy to find and which generalize trivially but are highly inaccurate. HL thus uses the following objective during search, where $\oplus$ is the noisy-or operator:

$$\begin{aligned} &P_{\text{SEARCH}}(M|D,T,p,w,\alpha,\theta) = \\ &\quad (P_{\text{TRS}}(M\,|\,p,w) \oplus P_{\text{META}}(M\,|\,T)) \cdot P_{\text{ACC}}(D\,|\,M,\theta) \cdot P_{\text{WF}}(D\,|\,M,\alpha) \end{aligned} \tag{4.11}$$

To better illustrate how these two objectives work in tandem, consider the progression from the TRS 4.1 to TRS 4.9. TRS 4.1 is an empty TRS containing no rules and constructed by a meta-program which has not yet applied any learning mechanisms. P_{TRS} and P_{META} are thus high. It thus cannot rewrite a query like `F [1, 2, 3]` and so returns the same term as output, making it both inaccurate and unable to produce well-formed guesses. From a prediction standpoint, it thus scores well in terms of having a short meta-program and short

TRS but scores poorly on accuracy. Moreover, its inability to produce well-formed guesses gives it an overall score of 0. During search, P_{TRS} and P_{META} remain high, so their noisy-or is also high, and P_{ACC} remains low. P_{WF}, however, being only a soft bias during search, is no longer 0. Thus, the overall score suggests that the hypothesis could be extended by future revisions. This is good, seeing as its the only available hypothesis at the start of search.

After memorizing the data in TRS 4.3, progress appears mixed. P_{META} remains high, though P_{TRS} is lower because the TRS now has four complex rules. More importantly, these rules perfectly predict the data seen so far, so P_{ACC} now has a high score. Memorization, however, will not produce well-formed guesses, so P_{WF} remains 0. From a prediction standpoint, the TRS thus continues to have an overall score of 0. From a search perspective, however, TRS 4.3 is much better than TRS 4.1. The noisy-or of P_{TRS} and P_{META} remains relatively unchanged, as does P_{WF}, but P_{ACC} is now also high. The overall search score then suggests that TRS 4.3 is a good candidate for extending in future revisions.

A couple revisions later, TRS 4.5 appears decidedly worse under both objectives. P_{ACC} remains high, but two additional parameterized learning mechanisms lower P_{META}. A much more complex set of rules means P_{TRS} is also very low. Moreover, none of the learning mechanisms have introduced any form of abstraction so P_{WF} remains 0. The TRS thus continues to have an overall prediction score of 0. The search score is also worse for the same reasons. Both P_{TRS} and P_{META} are lower, so their noisy disjunction is also lower. The overall search score is non-zero and so does not disqualify TRS 4.5 from future revision, but it looks less appealing than either TRS 4.3 or TRS 4.4.

Everything changes, however, by the time search reaches TRS 4.9. For the prediction score, P_{META} is significantly worse, but P_{ACC} remains high, P_{TRS} improves dramatically, and the hard bias of P_{WF} is now non-zero. The meta-program for TRS 4.9 is one of the shortest meta-programs with this combination of factors, so it is also among those with the highest overall prediction scores. The search score is also much better. Because P_{TRS} is much higher, the noisy-or with P_{META} is also better. Both P_{ACC} and P_{WF} are also high. Not only is TRS 4.9 a good final solution, but it is a great place to consider extending if time remains for additional search. Entertaining TRS 4.5, despite its flaws, was necessary for reaching this significantly better solution.

The two objectives thus work together to guide different kinds of decision making throughout learning. The prediction objective fails to make allowances for things later revisions might fix, which is necessary during search. The search objective makes these allowances, but is too relaxed about features critical to making good predictions. Together, however, they allow HL to delay certain kinds of judgment while actively exploring the hypothesis space, bringing them online only when it becomes time to decide which hypothesis, of all it has considered, is most likely to generate useful predictions.

4.4 Conclusion

HL defines a model of inductive learning inspired by the child as hacker. It is aimed at structurally rich domains like the list functions described in Chapter 3 and takes steps toward incorporating hacker-like insights in its approach toward mental representations, objective functions, and learning mechanisms. This enables HL to model learning as the iterative development of an entire LOT. It is also sensitive to multi-part objective functions that vary based on the task at hand. By using a variety of structure-sensitive learning mechanisms, HL dissociates the complexity of a program from the complexity of the generative process that creates that program, and is able to rapidly discover certain kinds of complex LOTs that, despite their length, are among the simplest explanations providing an accurate and general account of the data. Chapter 5 reports the results of a large-scale human concept learning experiment in the domain of list functions, and Chapter 6 describes how well HL explains human performance in this experiment relative to several alternative models of learning as programming.

Chapter 5

Human learning of list functions: Structural sources of difficulty

5.1 Introduction

Human learning is pervasively rich and complex. It is sometimes swift and powerful (Tenenbaum et al., 2011) and other times slow and labored (Carey, 2009; Barner, 2017). Variation among learning mechanisms is the rule rather than the exception (Siegler, 1996), and these mechanisms constantly adapt to new data (Siegler & Jenkins, 1989). The domains about which people learn span the literal cosmos, including: subatomic particles, galactic superclusters, the origins of the universe, and the end of time. It also encompasses everything in between, especially everyday, commonsense abilities such as navigating an environment, communicating and collaborating with others, and performing daily routines. The motivations and goals of learners are equally diverse, as are their outcomes. Some remain novices, while others excel to world-class performance and beyond, advancing the collective repertoire of human accomplishment (Ericsson, 2006). Moreover, much of this learning occurs across multiple domains simultaneously and informally, as part of the daily fabric of human life.

Studying this rich texture of cognitive development has led to a number of key hypotheses about learning. Several of these hypotheses work together to provide computational cognitive science with—for lack of a better term—a standard approach to learning difficulty[1].

[1]The name "standard approach" is not intended to imply that these hypotheses sit on equal footing with,

In keeping with the rest of this thesis, we frame the discussion of this standard approach in terms of learning over a system of compositional parts which behave like a mental language or language of thought (LOT; Fodor, 1975). We specifically focus on the learning as programming paradigm introduced in Chapter 1, which treats the LOT as something like a probabilistic programming languages and learning as something like programming

The first key idea in this standard approach is that the amount of information required to uniquely specify a concept plays a key role in explaining its learnability. Specifically, learners prefer the simplest concepts that will explain the data, namely those requiring the least amount of information to identify. There are strong normative arguments in favor of simplicity as a guiding principle for learning (Chater & Vitányi, 2003; Baum, 2004), not the least of which is that simpler explanations have less capacity for memorization and overfitting and are thus more likely to generalize. The amount of information required to specify a concept is deeply connected with the size of the hypothesis space and the methods which a learners uses to explore hypotheses. This idea has been formalized and studied under multiple names, including Kolmogorov complexity (Kolmogorov, 1963), Solomonoff induction (Solomonoff, 1964a, 1964b), the minimum description length principle (Grünwald, 2007), information theory (MacKay, 2003), algorithmic information theory (Grünwald & Vitányi, 2008), and probably approximately correct (PAC) learning (Valiant, 1984; Kearns & Vazirani, 1994). In a Bayesian setting, the drive for simplicity can be realized through a prior distribution over hypotheses which penalizes a hypothesis in proportion to its complexity (Jefferys & Berger, 1992). Bayesian models of learning in the LOT (Ullman & Tenenbaum, 2020) then balance this prior against a likelihood favoring a good fit to observed data. The overall effect is to favor hypotheses which are both simple and accurate. All these methods formalize Occam's Razor, suggesting that, all else being equal, learners will eventually settle on the simplest explanation of the data because, all else being equal, simpler explanations are better.

A growing body of computational and empirical work, however, supports an even stronger

for example, the much more established standard model in physics. It is instead to communicate that these hypotheses form a key part of the substrate on which modern computational cognitive science is built. They are part of the basic worldview which many model builders bring to understanding learning, specifically to understanding what makes learning easy or hard.

claim: knowledge that can be encoded with a short expression in the LOT is learned more easily than knowledge requiring a long expression. This forms the second key idea of the standard approach. Feldman (2000; 2003), for example, has shown this convincingly for Boolean concepts (cf. Lafond et al., 2007), and the idea has formed the basis for many empirically successful computational models of concept learning, including subsequent models of learning for Boolean concepts (Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2016), counting procedures (Piantadosi, 2011), graph structures (Kemp & Tenenbaum, 2008), and scientific theory learning (Kemp et al., 2010; Ullman et al., 2012). Given the complexity of their LOTs and the infeasibility of considering all possible concepts, such models rely on algorithms which closely link learning with description length, such as exhaustive search based on enumeration or stochastic search based on random sampling. Their findings suggest that learners not only prefer simple solutions, but that the simplicity of a solution strongly determines how easily a concept can be learned at all.

This standard approach, however, cannot be easily reconciled with findings that paint a richer picture of learning. It argues for a strong connection between learning and description length in the LOT. The more compactly a concept can be expressed in the LOT, the more easily it can be learned. As discussed in previous chapters, however, learners develop entirely new conceptual systems in which totally new symbols are defined not in terms of some fixed set of preexisting primitives, but in terms of their overall conceptual role. They are keenly aware of the strengths and weaknesses of their hypotheses and apply constructive thinking in highly structured goal-and-hypothesis-driven searches. They are sensitive to many values and goals and appear to move fluidly between them.

The child as hacker hypothesis developed in this thesis emphasizes the diversity of the values and mechanisms which humans bring to bear during learning. It specifically suggests that, like hackers (Fowler, 2018; Abelson et al., 1996), learners may make use of a wide variety of context-specific values and learning mechanisms in the development of domain-specific languages. For hackers, these values and mechanisms are often highly targeted, applicable only to particular kinds of problems. The mechanisms frequently manipulate programs in highly structured ways, acting essentially as program-changing programs. Hackers maintain a catalog of these mechanisms and understand the conditions in which they are likely to be

helpful. They also understand the relationships between these mechanisms and the many dimensions of value they might affect. If human learning mechanisms operate in hacker-like ways and are sensitive to structure in observed data, learning itself might be strongly affected by even small amounts of data. As a result, human learners might deviate significantly from models which explain learning primarily as a function of description length in the LOT. People may instead be able to learn much more quickly than these models would predict, making use of structure-sensitive methods to narrow the space of hypotheses and rapidly discover long programs that are, nonetheless, the simplest explanations of the data.

In this chapter, we present a large-scale human concept learning experiment that tests these hypotheses in human learners. We extend preliminary work in (Rule et al., 2018) to develop a benchmark set of 250 list transformation concepts and assess their learnability through a large scale online behavioral experiment. We encode each concept as a program in a rich, domain-appropriate model LOT and analyze learning performance in terms of a variety of syntactic and semantic features. These features include alphabet size and three measures of description length, all of which are strongly connected to the simplicity of each program. We also include factors encoding the use of conditional or recursive reasoning, the use of counting knowledge, argument structure, and visibility—a concept we introduce as a measure of how transparently each symbol in a program is encoded in its associated input/output data. We find that while description length has some predictive value, the contribution of each symbol is heavily modulated by its visibility. Moreover, language size has no significant effect in our data, while more semantic features significantly impact overall performance. These results show that humans are strongly sensitive to structural cues of semantic content in observed data and are able to use them to improve learning. We argue that humans may use learning mechanisms which—unlike enumeration or random sampling—decouple the complexity of learning a concept from the complexity of that concept's representation in the LOT.

5.2 Method

Participants were told that they were going to play a guessing game with the computer. The game was divided into rounds, and in each round, the computer would devise a rule

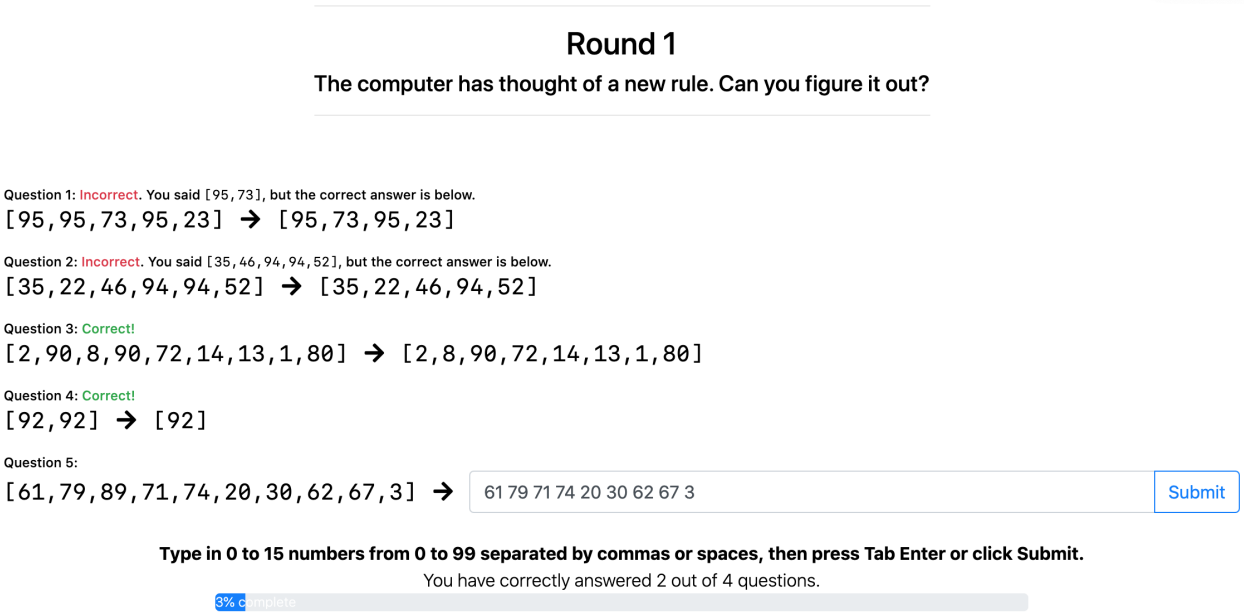


Figure 5-1: A sample display from the list transformation paradigm used in the behavioral experiment. Previous trials remain onscreen, with a record of the input, output, participant response, and whether the response was correct. The concept is to remove the first instance of the largest element in the list.

for transforming a list of numbers given as input into some output list of numbers. To help participants learn about the rule, the computer would ask a series of questions. In each, the computer would show a novel input list and ask the participant to predict the output associated with the input. Participants were told that their job was to guess the rule and use it to correctly respond to as many of the computer's questions as possible.

After each prediction, the computer revealed the correct output. The input/output pair remained on screen for the rest of the experiment. We thus created an online learning paradigm in the style of Piantadosi et al. (2016) and Rule et al. (2018), in which learners had access to correct information associated with all past trials and could review it at any time during learning (Figure 5-1). This paradigm also provided us with a trial-by-trial record of learners' generalizations.

5.2.1 Participants

498 people participated in the experiment, hosted on Amazon Mechanical Turk using PsiTurk. While we attempted to define highly learnable concepts, this is still a difficult exper-

Type	Description
`t1`, `t2`,...	Universally quantified type variables
`Int`	Integer values
`Bool`	Boolean values
`[X]`	List of type `X` (e.g. `[Int]` for a list of integers)
`X → Y`	Arrow type representing a function from `X` to `Y`. All functions are curried (i.e. arrows are chained to represent functions of multiple arguments). E.g. `Int → [Int] → Bool` takes an `Int` and a list of `Int`s and returns a `Bool`.

Table 5.1: The Hindley-Milner typesystem used in the concept language.

iment. After reviewing pilot data, we excluded participants who completed the experiment in less than 20min, with fewer than 10 correct responses, or with more than 20 identical responses. We therefore excluded 106 participants, analyzing data from the remaining 392. Participant age for this group ranged from 18.6yrs to 69.4yrs (median: 39.2yrs), with 253 males and 132 females. We did not actively assess language skills but requested that participants speak English fluently.

5.2.2 Materials

Both humans and models were tested on a set of 250 concepts (Appendix A). Each concept was drawn from a rich domain-specific language based on a typed lambda calculus. Lambda calculus is a Turing-complete (i.e. computationally universal) formalism that models computation as function abstract and application (Barendregt et al., 1984). It plays a fundamental role in computer science and frequently appears in LOT-based computational models of learning (e.g. Piantadosi, 2011; Liang et al., 2010; Zettlemoyer & Collins, 2005). See page 68 and following for more discussion of the lambda calculus and an example computation. We equip our language with a Hindley-Milner typesystem (Hindley, 1969; Milner, 1978; Pierce, 2002) which provides syntactic guarantees on the semantic correctness of programs. The type system makes it impossible to construct programs which are semantically non-sensical (i.e. *take the second element of the number 3*) while still allowing the full range of semantically meaningful programs. Table 5.1 describes the type system, Table 5.2 describes the primitives, and Table 5.3 gives several example programs for concepts used in the experiment.

Usage	Type	Description
(λ x body)	t1 → t2 → (t1 → t2)	lambda abstraction; binds x for use in body
0, 1, 2,...,99	Int	natural numbers
true, false	Bool	Boolean values
[]	[t1]	empty list
(+ x y)	Int → Int → Int	add x and y
(- x y)	Int → Int → Int	subtract y from x
(* x y)	Int → Int → Int	multiply x and y
(/ x y)	Int → Int → Int	quotient of x divided by y
(% x y)	Int → Int → Int	remainder of x by y
(< x y)	Int → Int → Bool	true if x is greater than y
(> x y)	Int → Int → Bool	true if x is less than y
(is_even x)	Int → Bool	true if x is even
(is_odd x)	Int → Bool	true if x is odd
(and x y)	Bool → Bool → Bool	Boolean conjunction of x and y
(or x y)	Bool → Bool → Bool	Boolean disjunction of x and y
(not x)	Bool → Bool	Boolean negation of x
(if p a b)	Bool → t1 → t1 → t1	a if p is true, else b
(== x y)	t1 → t1 → Bool	true if x and y are structurally identical
(singleton x)	t1 → [t1]	list with a single element, x
(repeat x n)	t1 → Int → [t1]	list repeating x n times
(range i j n)	Int → Int → Int → [Int]	list of numbers from i to j, inclusive, counting by n
(cons x xs)	t1 → [t1] → [t1]	prepend x to xs
(append xs x)	[t1] → t1 → [t1]	append x to xs
(insert x i xs)	t1 → Int → [t1] → [t1]	insert x at index i in xs
(concat xs ys)	[t1] → [t1] → [t1]	concatenate xs and ys
(splice ys i xs)	[t1] → Int → [t1] → [t1]	insert ys into xs, beginning at index i
(first xs)	[t1] → t1	first element of xs
(second xs)	[t1] → t1	second element of xs
(third xs)	[t1] → t1	third element of xs
(last xs)	[t1] → t1	last element of xs
(nth i xs)	Int → [t1] → t1	element i of xs
(replace i x xs)	Int → t1 → [t1] → [t1]	replace element at index i in xs with x
(swap i j xs)	Int → Int → [t1] → [t1]	swap elements at indices i and j in xs
(cut_idx i xs)	Int → [t1] → [t1]	remove element at index i from xs
(cut_val x xs)	t1 → [t1] → [t1]	remove first occurrence of x from xs
(cut_vals x xs)	t1 → [t1] → [t1]	remove all occurrences of x from xs
(drop n xs)	Int → [t1] → [t1]	remove first n elements from xs
(droplast n xs)	Int → [t1] → [t1]	remove last n elements from xs
(cut_slice i j xs)	Int → Int → [t1] → [t1]	remove elements at indices i to j, inclusive from xs
(take n xs)	Int → [t1] → [t1]	first n elements of xs
(takelast n xs)	Int → [t1] → [t1]	last n elements of xs
(slice i j xs)	Int → Int → [t1] → [t1]	sublist of xs from indices i to j, inclusive
(fold f acc xs)	(t2 → t1 → t2) → t2 → [t1] → t2	iteratively accumulate elements of xs into acc via f
(foldi f acc xs)	(Int → t2 → t1 → t2) → t2 → [t1] → t2	like fold, but p is also given the element's index
(filter p xs)	(t1 → Bool) → [t1] → [t1]	keep only elements of xs for which p is true
(filteri p xs)	(Int → t1 → Bool) → [t1] → [t1]	like filter, but p is also given the element's index
(count x xs)	(t1 → Bool) → [t1] → Int	count occurrences of x in xs
(find p xs)	(t1 → Bool) → [t1] → [Int]	returns indices of xs for which p is true
(map f xs)	(t1 → t2) → [t1] → [t2]	apply f to each element of xs
(mapi f x)	(Int → t1 → t2) → [t1] → [t2]	like map, but f has access to each element's index
(group f xs)	(t1 → t2) → [t1] → [[t1]]	group elements, x, of xs based on the key, (f x)
(is_in xs x)	[t1] → t1 → Bool	true if x is in xs
(length xs)	[t1] → Int	length of xs
(max xs)	[t1] → Int	largest element in xs
(min xs)	[t1] → Int	smallest element in xs
(product xs)	[Int] → Int	product of elements in xs
(sum xs)	[Int] → Int	sum of elements of xs
(unique xs)	[t1] → [t1]	unique elements of xs
(sort f xs)	(t1 → Int) → [t1] → [t1]	sort elements, x, of xs by the output of (f x)
(reverse xs)	[t1] → [t1]	xs in reverse order
(flatten xs)	[[t1]] → [t1]	concatenates the list of lists, xs, into a list
(zip xs ys)	[t1] → [t1] → [[t1]]	join xs and ys into a list of two-element lists

Table 5.2: The primitives from which the concepts were formed

Program	Description
`(λ x (singleton 5))`	the list [5]
`(λ x x)`	identity function
`(λ x (take 2 x))`	first two elements of `xs`
`(λ x (singleton (index (first x) x)))`	element N of `x`, N = element 1
`(λ x (concat (repeat (third x) 3) (drop 3 x)))`	element 3 replaces elements 1 & 2
`(λ x (map (λ y (+ y 1)) x))`	increment every element by 1
`(λ x (filter (λ y (== 1 (count (λ z (== y z)) x))) x))`	remove repeating elements

Table 5.3: Example concepts in the model LOT.

84 of the concepts exclusively used the numbers 0–9, while the rest also included 10–99. All concepts were manually generated with the intention of capturing broad variation both in the kinds of algorithmic reasoning required and in their difficulty for human learners.

To generate input/output pairs for each concept, we randomly generated one million sets of 11 input/output pairs and selected the best according to a per-concept custom scoring function. Both input and output lists were restricted to contain 0 to 15 elements. The per-concept scoring function always favored variance in the lengths of the inputs and outputs, variance in the elements of the lists, a high number of unique outputs, and a low number of examples in which the input and output were identical. Each was then also customized to favor features relevant to the given concept. For example, a concept indexing the third element might favor inputs with three or more elements, while a concept using the first element as an index might favor lists in which the first element was less than or equal to the length of the list. After selecting a set of examples, we then generated five thousand random orderings and selected the one with the highest score based on: applying the per-concept scoring function to the first five pairs, applying the per-concept scoring function to the last six pairs, whether the input differed from the output in the first example, and the distance between 5 and the length of the first input.

5.2.3 Procedure

Participants began by reviewing a set of instructions detailing the experimental design and completing a short quiz verifying their understanding of the task. They then completed 110 trials of the list-routines task—10 blocks of 11 trials each. In each trial, participants were

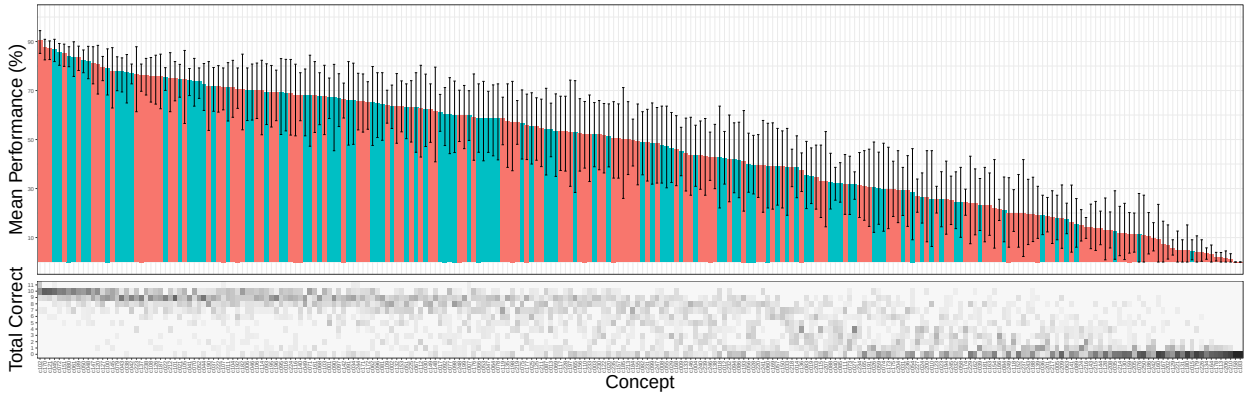


Figure 5-2: (Top): Mean accuracy (y-axis) on each concept (x-axis) in descending order of mean accuracy. Error bars are bootstrapped 95% CIs. Concepts examined in Chapter 5's model comparison are marked in blue; (Bottom): For each concept (x-axis), the percentage of participants with each possible number of correct responses (y-axis). Color varies from white (0%) to black (100%).

shown an input list and asked to predict the output according to the rule they thought the computer was using to transform inputs into outputs. Each block tested a different concept. Because each participant completed 10 blocks, we collected data from about 16 participants for each concept. The experiment concluded with a brief demographical survey.

The ten concepts were sampled uniformly from the total pool of 250 concepts being tested, and their order was also randomized. The paradigm encouraged online learning: for trial $N + 1$ of a block, participants saw the correct inputs and outputs from the previous N trials and were able to use them, and their past thinking about the function, to inform their prediction. At the conclusion of each block, participants were asked to describe in words the rule they thought the computer had been using.

Participants were paid a flat fee of \$7.50 for participating plus an additional \$0.01 for each correct response, with a median compensation of \$8.00 for a median 72min of work. Participants found the task difficult but engaging with a mean self-reported difficulty rating of 4.9 and a mean self-reported engagement rating of 5.9, both on a 7-point Likert scale.

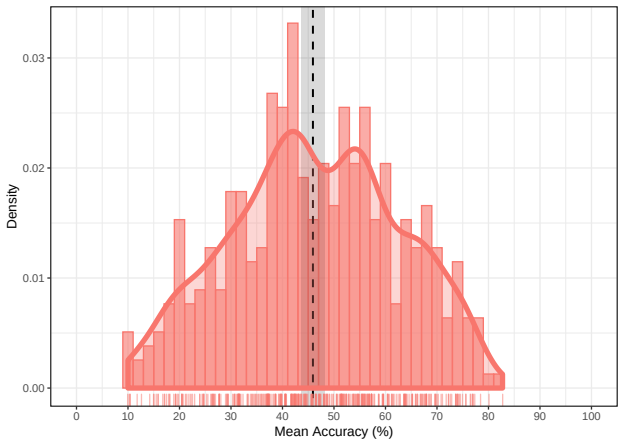
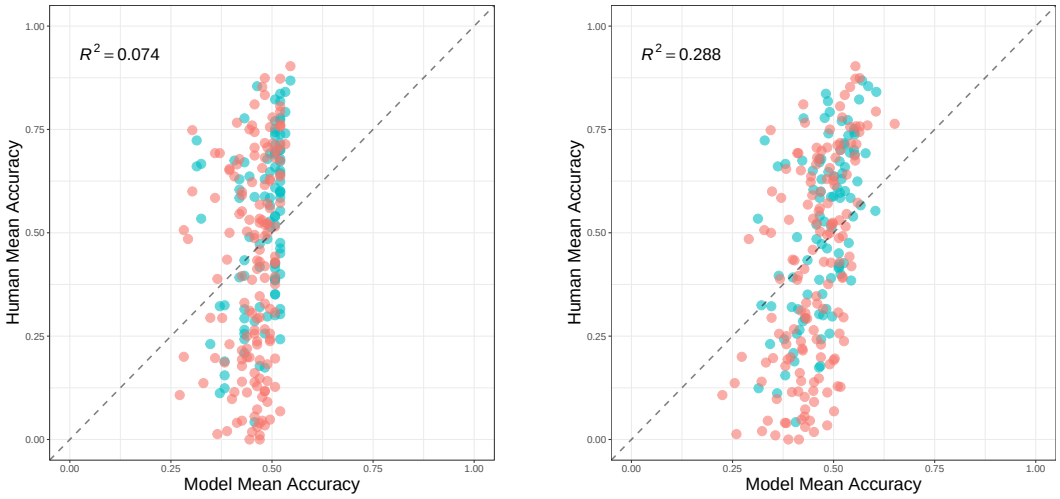


Figure 5-3: Histogram (bars) and Gaussian kernel density estimate (curve) of mean concept-level performance (x-axis) by participant, including median (black dashed line) with bootstrapped 95% CI (gray region), and individual participants (rug plot).

5.3 Results

The top portion of Figure 5-2 plots the mean accuracy across subjects for each concept. It depicts, on average, how accurately all the people who completed a given concept responded to all trials for that concept. The figure shows wide variation in performance across concepts. Nearly all participants learn the identity function after completing just a single trial. This performance is at the effective ceiling. Before completing the first trial, participants have no data on which to base their predictions; it is incredibly unlikely that a learner would select the correct function from an unboundedly large space in the complete absence of a learning signal. Other concepts—like *replace each element with* `1` *if the element is equal to its index and* `0` *otherwise*, or *list the indices of the elements of the tail of the list equal to the head of the list*—were so difficult that no participant ever responded correctly to a single trial. The remaining concepts smoothly vary between ceiling and floor. There are no notable plateaus or discrete levels of difficulty as might be expected if description length were the dominant predictor of performance.

In addition to showing a wide variance at the concept level, the top of Figure 5-2 also shows pronounced patterns of variation at the participant level. Once mean performance is sufficiently far from floor or ceiling, the confidence interval on mean accuracy for each concept tends to be wide. The bottom portion of the figure demonstrates why, plotting the

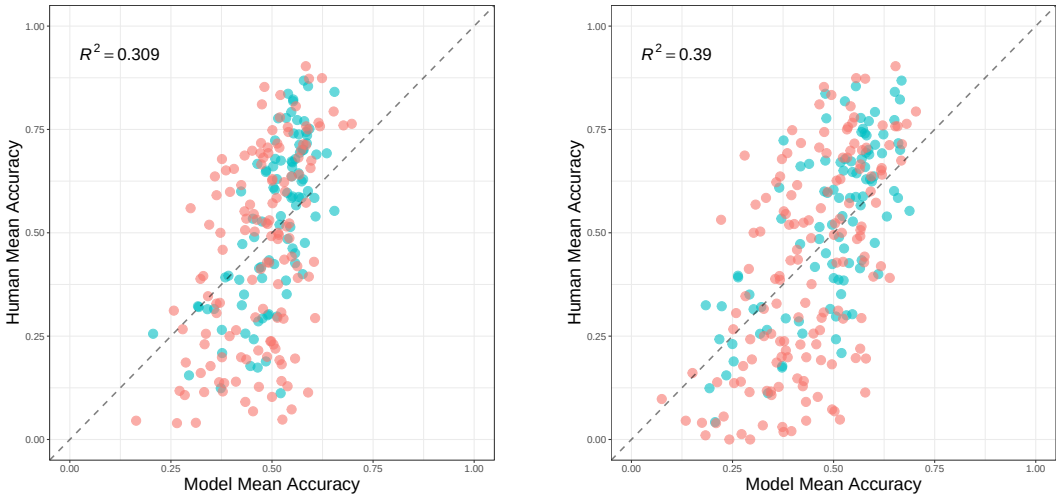


(a) (b)

Figure 5-4: Logistic model predictions (x-axis) of mean human accuracy (y-axis) by concept (circles) based on description length in a rich model LOT. (a): predictions based on description length alone; and (b): predictions based on length with a mixed effect of participant including intercepts and trial slopes. Concepts examined in Chapter 5's model comparison are marked in blue. R^2 is variance in human performance explained by model predictions.

percentage of participants who correctly responded to N trials, $0 \leq N \leq 11$. The figure shows that while extremely easy and extremely difficult concepts tend to cluster around just a few values of N, moderately difficult concepts show a fairly diffuse distribution of values. While it is possible that these variations are naturally expected, Figure 5-3 suggests that they reflect genuine differences in participant performance. This figure plots a histogram over mean performance by participant, averaging across the 10 concepts each participant completed. As for individual concepts, there is a wide variation in the performance of individual participants, suggesting that some were generally more accurate or faster to learn than others.

Based on past successes in explaining human learning as a function of description length, however, we fit a logistic model predicting accuracy on each trial based solely on a scaled and centered measure of description length for the program encoding each concept in the model LOT. Each program can be represented as a tree whose leaves are the visible symbols of the program. These include function names like `first` and `last`, as well as variable names like `x` or `y`. The internal nodes of the tree represent either function application, which has two children, and lambda abstractions, which have a single child. We used the total number of



(a) (b)

Figure 5-5: Logistic model predictions (x-axis) of mean human accuracy (y-axis) by concept (circles) based on English description length. (a): predictions based on participant provided English descriptions with a mixed effect of participant; and (b): predictions based on gold standard English descriptions with a mixed effect of participant. Concepts examined in Chapter 5's model comparison are marked in blue. R^2 is variance in human performance explained by model predictions.

nodes in each program's tree as its description length.

Figure 5-4a shows the match between the model predictions and human performance, averaged on a per-concept basis. This model explains very little variance ($R^2 = 0.074$). The particular model LOT used to describe these data, then, is a poor predictor of human learning performance on these concepts. This initial model did not account for variation in the performance of individual learners. We therefore extended it to include a random effect of participant sensitive to both overall accuracy and the speed with which each participant learned trial-to-trial within an individual concept. Figure 5-4b shows that this provides a significantly better fit than length alone ($R^2 = 0.288$, difference $= 0.215$, 95% CI $= [0.180, 0.233]$, $p < 0.001$). Even so, the bulk of the variance remains unexplained. Even correcting for differences in performance across individual participants, many concepts are much easier or much harder than can be explained given their length in our model LOT.

This result is surprising given the success of past work in correlating description length and learning difficulty. Moreover, our model LOT has not been empirically verified as an accurate description of the human LOT relevant to this task. To confirm our results, we

therefore scored participant natural language responses on a three-point scale: incorrect, partial, and correct. Incorrect indicated that the participant did not identify any portion of the correct concept, correct indicated that they fully identified the correct concept, and partial indicated that they correctly identified some aspects of the concept but did not describe it in full. Participants provided correct responses for 231 of the 250 concepts. After preprocessing correct responses to remove unnecessary punctuation and separate grouped mathematical symbols (e.g. "n+1" was rewritten as "n + 1"), we computed the mean number of words per response by concept. This mean correct English description length serves as a secondary measure of description length. The claim is not that the LOT uses natural language, but instead that important concepts in the LOT are likely to be named in a lexically rich language like English. Concepts which are simple to describe in English are then likely to correspond to concepts which have a compact description in the LOT, and similarly for concepts which have long English descriptions.

Because this measure was only moderately correlated with model LOT description length ($\rho = 0.448$), it is possible that English description lengths might provide a significantly different and perhaps more predictive measure of learning difficulty. We therefore constructed a mixed effect logistic model predicting human performance using a fixed effect of mean correct English description length and a random effect of participant as described previously. Figure 5-5a shows the results of this analysis. As with description length in the model LOT, verbal description length does not appear to be a strong predictor of human performance for these tasks ($R^2 = 0.309$). We repeated this analysis with a set of gold standard English glosses (Figure 5-5b; see Appendix A for glosses), and while the results are significantly better ($R^2 = 0.390$, difference= 0.081, 95% CI=[0.071, 0.089], p< 0.001), they were constructed by individuals deeply familiar with the domain and with technical vocabulary used in computer science for describing list processing programs. Furthermore, the length of these gold standard descriptions is strongly correlated with the length of participant-provided descriptions ($\rho = 0.670$) and, like the participant-provided descriptions, only moderately correlated with concept length in the model LOT ($\rho = 0.486$), so we use participant-provided descriptions for the remainder of our analysis.

Table 5.4 reports several cases where the regression model based on length in the LOT

ID	μ_h	μ_l	Δ	Concept
c134	0.034	0.484	-0.450	(λ x (cut_slice (first x) (second x) x))
c113	0.018	0.451	-0.433	(λ x (filter (λ y (> (first x) (% y 10))) x))
c202	0.068	0.501	-0.433	(λ x (find is_even x))
c183	0.000	0.414	-0.414	(λ x (find (== (first x)) (drop 1 x)))
c164	0.030	0.430	-0.399	(λ x (map (λ y (+ (/ y 4) 5)) x))
c131	0.045	0.442	-0.396	(λ x (filter (λ y (is_even (/ y 10))) x))
c208	0.127	0.517	-0.390	(λ x (takelast (last x) x))
c158	0.000	0.388	-0.388	(λ x (mapi (λ y (λ z (if (== y z) 1 0))) x))
c128	0.129	0.510	-0.381	(λ x (sort (λ y y) (cut_idx 3 (drop 2 x))))
c129	0.056	0.428	-0.373	(λ x (slice (first x) (second x) (drop 2 x)))
c151	0.853	0.541	0.312	(λ x (flatten (map (λ y (repeat y y)) x)))
c121	0.873	0.554	0.319	(λ x [(last x)]) (using 0–99)
c048	0.818	0.486	0.332	(λ x (take 1 x))
c223	0.766	0.429	0.337	(λ x (map (λ y (+ (* (% y 10) 10) (/ y 10))) x))
c102	0.903	0.554	0.349	(λ x x)
c043	0.777	0.426	0.351	(λ x [8, 2, 7, 0, 3])
c061	0.836	0.481	0.355	(λ x [(last x)]) (using 0–9)
c147	0.811	0.425	0.385	(λ x (flatten (mapi (λ y (λ z [z, y])) x)))
c044	0.723	0.330	0.394	(λ x [1, 9, 4, 3, 2, 5, 8, 0, 4, 9])
c101	0.748	0.344	0.404	(λ x [11, 19, 24, 33, 42, 5, 82, 0, 64, 9])

Table 5.4: 20 concepts whose length-based predictions strongly strongly deviate from human performance. μ_h and μ_l report mean human and length-based accuracy, respectively; $\Delta = \mu_h - \mu_l$. The table includes the 10 largest and the 10 smallest values of Δ. Literal lists were rewritten for legibility.

strongly deviated from human performance, predicting that concepts would be either easier or harder for people than they actually were. Six of the ten concepts listed as being more difficult for people than predicted by the regression involved slicing or filtering the list in some way. It is tempting to say that such functions are either not part of people's LOT or are much more costly than a simple measure of description length would suggest. Models which learn a weighted grammar (e.g. Ellis et al., 2020) might be more appropriate. At the same time, however, there are also examples of filtering and slicing which are much easier for people than predicted by length alone, such as removing every third element ($\Delta = 0.248$), or extracting a sublist containing the second, third, and fourth elements ($\Delta = 0.186$). The same is also true for other concepts in the table which the regression underpredicted—`map`, `takelast`, and `sort` are all used in concepts which are significantly more difficult and significantly easier than predicted by description length in our model LOT.

The concepts in Table 5.4 which are significantly easier than predicted by description length share a key feature in common. They often describe very simple, structured transformations that can be easily inferred directly from input/output pairs. The functions which return constant lists, for example, have a long description length, but nearly the entire struc-

ture of the concept is directly encoded in the output. The only inference required is that the output given in the first few trials should be given for any input. Similarly, *return the input*, *remove all but the first element*, and *remove all but the last element* all rely on very simple structural transformations. So, too, do *repeat each element, n, n times*, and *swap digit orders in each element*, and *insert each element alongside its index in the input*, though these are slightly more complex because the transformation is applied elementwise. The individual transformations of swapping digits or inserting an index, however, are each readily visible in the input/output relation, as is the fact that the transformation is applied to each element.

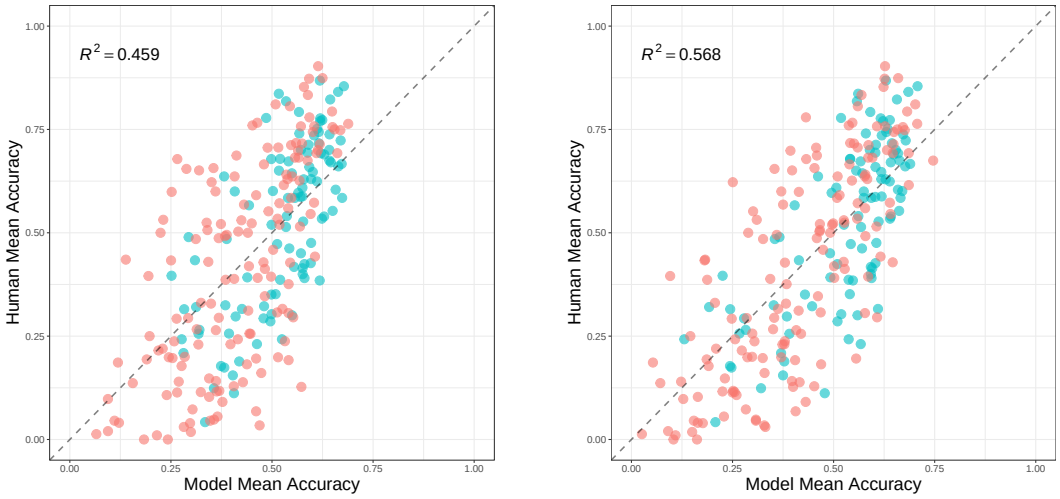
In light of these observations, we analyzed the programs for every concept in our model LOT, assigning each symbol in each program a visibility: visible, semi-visible, or hidden. Visible symbols can reasonably be inferred directly from 1–3 examples. For example, in `(λ x (slice 2 4 x))`, which extracts a sublist containing the second through fourth elements, `slice` and `x` are considered visible because outputs contain a non-trivial, continuous sublist of the input which excludes both endpoints. That the output always includes a portion of the input immediately suggests that the function uses its argument `x`. `slice` is not the only symbol whose use signals the extraction of a sublist. `take`, `takelast`, and `filter` can also return sublists, though `filter` is unlikely to be continuous, and `take` and `takelast` always include an endpoint. After looking at just a couple of examples, `slice` can be reasonably inferred. Similarly, semi-visible symbols can be reasonably inferred directly from 3–5 examples or indirectly from 1–3 examples. For example, the symbols `2` and `4` in the program above can be inferred directly by reviewing several examples to notice that the first element in an output is always element 2 of the input, and the last element of the output is always element 4 of the input. They can be inferred indirectly by first inferring the use of `slice` and then searching for suitable values for each argument. All other symbols are scored as hidden, meaning not that they are necessarily difficult to learn, but that they likely have to be guessed. For example in `(λ x (singleton (- (length x) (length (unique x)))))`, which computes the number of repeated items in a list, the use of `singleton` and `x` are semi-visible, because a few examples will show that the output is always a single element and varies based on the input. The other elements, however, which compute the difference between the length and the number of unique elements, cannot be obviously inferred directly

from the data. They are thus considered hidden. This notion of visibility is an initial attempt to capture how much information is required to infer a symbol's existence. Visible symbols require relatively less information than semi-visible symbols, which require relatively less information than hidden symbols.

Because each symbol in a program's description length is assigned a score, visibility effectively subdivides description length into three categories. If description length is the primary driver of learning difficulty, then breaking it into multiple categories should have no impact on the ability to predict human performance. If however, people are sensitive to the kinds of inferences which visibility attempts to capture, then predictive models based on visibility may be significantly more accurate. The child as hacker suggests the latter hypothesis. We therefore constructed a third mixed effect logistic model including counts of visible, semi-visible, and hidden symbols in each program as well as the random participant effect used previously. All counts were centered and scaled. Figure 5-6a shows that assigning a visibility to each node significantly improves the ability to predict human performance compared to the description length models based on the model LOT ($R^2 = 0.459$, difference= 0.171, 95% CI=[0.159, 0.179], p< 0.001) and the gold standard English descriptions (difference= 0.069, 95% CI=[0.059, 0.079], p< 0.001). This result suggests that the individual symbols which compose a concept's description do not impose uniform costs on learning. The impact instead appears to be significantly modulated by how easily that symbol can be inferred from input/output data.

Even so, the visibility measure used here is a crude measure of the inferential cost of an individual symbol. It fails to take into account possible differences between various kinds of inferences. For example, inferring that a list function's behavior depends on the value of a particular element in the list might be more difficult than recognizing that it relies on some form of highly familiar behavior like counting. We thus further analyzed each concept for several additional factors:

Internal Argument True if a concept relied on using an element or property of the list as an argument to another function, as in `(λ x (take (first x) (drop 1 x)))`, which takes n items from the tail of the list, where n is the head.



(a) (b)

Figure 5-6: Logistic model predictions (x-axis) of mean human accuracy (y-axis) by concept (circles) using semantic features. (a): predictions based on visibility and mixed effect of participant; and (b): predictions incorporating a variety of syntactic and semantic features and mixed effect of participant. Concepts examined in Chapter 5's model comparison are marked in blue. R^2 is variance in human performance explained by model predictions.

Counting True if a concept relied on counting knowledge, such as computing the length of the list or counting the number of times a given element appeared.

Conditional True if the concept required conditional reasoning, such as filtering by a predicate or handling exceptional cases.

Recursive True if the concept required recursive reasoning, as with `map`, `filter`, or `fold`.

$\geq$ 10 True if the concept used numbers greater than nine.

Variables A count of the number of variable references in the program.

The presence of many of these features, particularly Counting, Conditional, and Recursive, often affect programs such that they can, like visible nodes, be directly inferred from a few input/output examples. Others, particularly Internal Argument and Variables, require multiple inference steps or guessing to discover. Having analyzed each concept, we constructed a final mixed effect logistic model including fixed effects for each of these features. We also included counts of visible, semi-visible, and hidden symbols for each program and a mixed effect for participants. All counts were centered and scaled. Figure 5-6b shows the results of this analysis. The addition of these semantic features again improves the ability to

predict human performance compared to all previous models, including the visibility model ($R^2 = 0.568$, difference= .109, 95% CI=[0.093, 0.124], p< 0.001). This result suggests that the ease of inferring conceptual dynamics from structural cues in input/output data differs significantly not only with the overall complexity of inference but also with the kinds of dynamics being inferred.

If description length served as the primary driver of learning difficulty, we would expect to see similar coefficients for Visible, Semi-Visible, and Hidden. We would also expect semantic features like Internal Argument, Conditional, Counting, Variables, and Recursive, to be close to 0. Finally, ≥10 would be expected to have a significant negative influence on performance, as it would indicate a significantly larger set of possible constants, and thus possible programs, to consider during learning. That is, the difficulty of learning would be determined primarily by the length of the program and the number of possible programs. The meaning of the nodes and their impact on the behavior of the program would have relatively little influence on learning.

By contrast, the child as hacker suggests that people rely on a diversity of structure-sensitive learning mechanisms. At the same time, it recognizes the impact of Occam's razor: people likely prefer simpler hypotheses. As a result, description length should play a role in predicting learning difficulty, but the impact of a given symbol in a program would be heavily influenced by its visibility. Moreover, semantic features are likely to have a significant influence on learning difficulty. Features which are directly reflected in the structure of the output lists, such as counting and certain kinds of recursive patterns, should make concepts easier to learn, while hidden structures like conditionals or internal arguments should make concepts harder to learn. By contrast, alphabet size might have a relatively insignificant role, as learners would, whenever possible, use semantic inferences to dramatically reshape and reduce the size of the hypothesis space.

To investigate these dynamics, Figure 5-7 plots the coefficients for each component of the final regression. The results are consistent with the predictions of the child as hacker. Hidden, Semi-Visible, and Visible all are significant factors with notably different impacts on learning performance. The level of impact is also consistent with the predictions of the child as hacker: many hidden nodes make a program much more difficult to infer, while

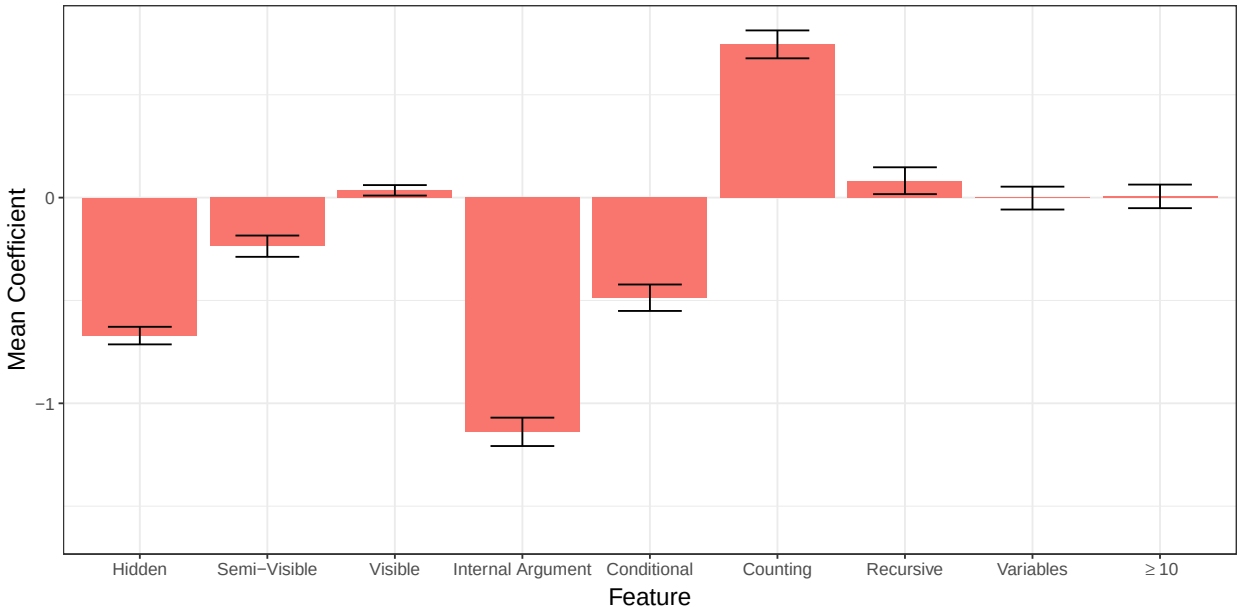


Figure 5-7: Regression coefficients from the feature-based model with 95% CIs, sorted by absolute value.

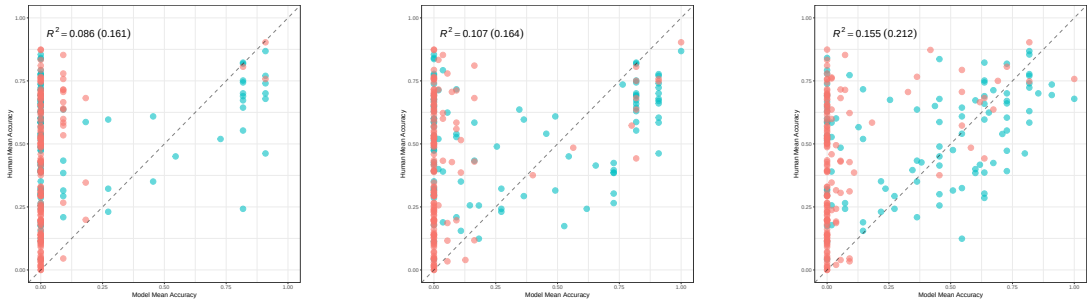
many visible nodes actually make the program less difficult to infer. This suggests not only that visible symbols have essentially no learning cost, but they might even make it easier to infer other symbols in the overall program. Additionally, features which are difficult to infer directly from the structure of input/output pairs, such as Internal Argument and Conditional, negatively impact human learning. Those which relate to structure that can be directly inferred—Counting and Recursive—positively impact performance. The set of numbers required to explain each concept, and the number of variables tokens in the concept, meanwhile, have no significant impact.

The analysis so far has focused on predicting learning accuracy from summary statistics of a concept's structure. The results suggest that people are sensitive to various kinds of structure in observed data and are able to use that structure to more or less easily infer certain aspects of a concept's compositional structure. The analysis has not, however, made any attempt to explain human performance in terms of a causal model of learning, proposing specific kinds of inference and how they might interact to explain these results. This is largely the focus of Chapter 6, but Figure 5-8 compares three computational models of learning in the LOT in terms of their ability to explain human performance on all 250 concepts: Enumer-

ation, Fleet, and HL[2]. Enumeration and Fleet represent two learning algorithms—exhaustive enumeration and stochastic search, respectively—well-known to match description length to learning difficulty. HL, by contrast, contains a variety of structure-sensitive learning mechanisms. Please see Section 6.3 for more details. Each model used the minimal model LOT shown in Table 6.1, learned for 10min per trial per concept (110min total per concept), and searched in an online fashion (i.e. performance in each new trial built on learning from previous trials). The minimal model LOT used here is almost certainly a poor match for the relevant portions of a human learners' conceptual repertoire. It is, however, technically useful in that exhaustive enumeration and stochastic search are sensitive not only to description length but also alphabet size. The larger the language, the more difficult search becomes for these models. Choosing a language requires balancing description length and alphabet size. Their performance here should then be interpreted as a lower bound on their overall ability to explain these data.

Perhaps the most striking feature is that there are many list functions for which the models never make a single correct prediction. This result is expected. List functions are a challenging domain, and many of the concepts are not feasibly within reach of the minimal DSL used to generate these results. It therefore makes sense to focus on those places where the models did make accurate predictions. For these concepts, HL has the best performance: it has non-zero performance for 125 concepts, Fleet for 99, and Enumeration for 50. Combined, the three models have non-zero performance on 141 of the 250 concepts: 35 are unique to HL, 13 to Fleet, and 5 to Enumeration. Humans, by contrast, have non-zero performance for 248 of the 250 concepts. HL also explains significantly more variance than the other models (R^2: HL = 0.212, Fleet = 0.164, Enumeration = 0.161). Moreover, HL is within 5% of having the most accurate prediction of the three models for 107 of the 141 concepts, while Fleet and Enumeration are within this bound for just 57 and 46 concepts,

[2]The implementation of HL used in the analyses that follow is a slightly earlier version than that reported in Chapter 4, and the two objective functions used during prediction and search, while capturing similar intuitions as in the reported model, took slightly different forms. Specifically, the prediction objective replaced P_{META} with an equally-weighted mixture between P_{META} and P_{TRS}, i.e. $(P_{\text{META}} + P_{\text{TRS}})/2$. The prediction objective used a similar mixture in place of the noisy-or used in the reported model. For the 100 concepts analyzed in the model comparison of Chapter 6, however, performance on the two models was highly correlated ($\rho = 0.987$), so the performance reported here on all 250 concepts using the older model is likely very similar to what would be seen with the reported model.



(a) (b) (c)

Figure 5-8: Program-induction-based model predictions (x-axis) of mean human accuracy (y-axis) by concept (circles). (a): Enumeration; (b): Fleet; and (c): HL. Concepts examined in more detail in Chapter 5's model comparison are marked in blue. R^2 values show the coefficient of determination for all concepts and, in parentheses, for concepts on which the models had non-zero performance.

respectively. While these results are by no means conclusive, they do show that a learning algorithm designed around a diversity of structure-sensitive learning mechanisms is better able to account of human performance on a broad range of inductive learning tasks.

5.4 Discussion

This chapter presented a large scale behavioral experiment in human concept learning testing online inductive learning for 250 list functions. This experiment was the first large scale test of list functions in human learners and showed high variance in performance across participants and across concepts. We modeled this variance in terms of a logistic model incorporating a wide variety of syntactic and semantic features including visibility, a measure we introduced to capture the local learnability of symbols in an LOT expression. We showed that this approach is significantly more predictive than models based on description length alone, whether length is measured in a rich, domain-appropriate model LOT or in English text-based descriptions. We further showed that the impact of the individual symbols that make up description length is heavily modulated by their visibility and that a number of other semantic features strongly impact learning performance. A significant change in the size of the hypothesis space, however, had no measurable effect. Finally, we found that a model which uses structure-sensitive learning mechanisms is better able to account for these findings

than models which strongly connect description length and learning difficulty. Overall, these results suggest that the traditional concerns of description length and hypothesis space size perhaps matter less than suggested by the standard approach, while structural and semantic cues perhaps matter significantly more.

Piantadosi et al. (2016) showed that the specific LOT used to measure description length significantly impacts a LOT-based model's ability to predict human learning. We partially mitigate that concern by also including a natural-language based measure of description length. Given the long line of mathematical, computational, and empirical results in favor of a simplicity bias, it is still possible that the results reported here are the result of using a poorly fit LOT. While a search over possible LOTs might reveal several with better fits than that used here, we think the general conclusion of this work still stands. Our model LOT contained primitives supporting a broad spectrum of algorithmic abilities, including those analyzed: higher-order functions supporting iteration and recursion, conditional reasoning, several functions for dealing with count knowledge, and so on. Moreover, it seems unlikely that any reasonable LOT would include `f x = [8, 2, 7, 0, 3]`, much less `f x = [11, 19, 24, 33, 42, 5, 82, 0, 64, 9]`, yet these are both in the top 10–15% of concepts in terms of mean accuracy, not much more difficult than the identity function or extracting individual elements.

At the same time, simplicity, particularly as embodied in the principle of Occam's Razor, seems to be an essential feature of theory building and of learning more generally. As has been shown many times, learners prefer hypotheses which are themselves simple, at least in part because simpler hypotheses are forced to make stronger predictions and are thus more likely to generalize rather than memorize. Our verbal response data suggest that when participants learned a concept, they were able to give a simple verbal description of the concept. They did not "needlessly multiply entities". Rather than interpreting these results as saying that simplicity is somehow unimportant for learning, we instead see them as reinforcing simplicity as an absolutely essential element of learning.

Our results show that learning performance is more strongly predicted by specific kinds of structure in observed data than by simple measures of description length. That is, learners appear to value simple hypotheses—even if the simplest accurate hypothesis happens

to be quite complex—but their learning mechanisms appear to be sensitive to more than description length alone. These results suggest that people are able to recognize specific patterns in data and enable exceptionally strong data-dependent inductive biases that dramatically narrow the hypothesis space. By recognizing the right patterns, the search process itself thus becomes much simpler, reaching some kinds of complex hypotheses much more quickly than length-based search methods like enumeration or random sampling. We therefore see these results as suggesting that learners not only have a strong preference for simple hypotheses, but also a strong preference for simple searches. Uniform search over a large hypothesis space may be appropriate in some cases, but strong structure-sensitive inductive biases appear preferable when available.

The set of numbers required to explain each concept has no significant impact on learning in this experiment. This result is surprising from the perspective of past work relating simplicity to learning difficulty, because the minimum description length of a concept would be strongly influenced by the number of possible primitives and sub-concepts. In any formal model, the number of primitives combinatorially impacts the number of possible programs. Adding 90 additional primitives of the same type would typically strongly and negatively affect the performance of grammar-based learning algorithms like enumeration or random sampling. In this case, each time a program could possibly include a number, the set of numbers to consider would be ten times larger. A learner would therefore need significantly more information to decide which number to use. This point stands even if numbers are constructed compositionally rather than treated as primitives. A learner would require significantly more information to determine which of 100 possible option to use than they would when deciding between just 10 options. In the presence of learning mechanisms which frequently reshape the hypothesis space, however, learners may spend very little time working with the full space of all possible programs. They may, at any given time, consider only a few symbols, making the effective language size much smaller.

In sum, these data provide three results which suggest structural sources of learning that go significantly beyond those of hypothesis space size and description length favored by the standard approach. First, visibility strongly modulates and may even negate a symbol's impact on learning difficulty. Second, specific semantic features like the use of recursion,

counting knowledge, or conditional reasoning are highly predictive of learning performance. Third, a significant change to the size of the program space—i.e. order of magnitude differences in the set size of numbers needed to explain a concept's behavior—has no effect on learning. This suggests the need for a richer understanding of simplicity that goes beyond description length in a proposed LOT to incorporate the rich interaction of data, changing objectives, current hypotheses, and diverse learning mechanisms. Simplicity is not measured purely in terms of the final concept but also in terms of the entire generative process giving rise to that concept.

Even so, the significance of these results is strongly limited. First, the analysis here focuses on predicting accuracies based on features of gold standard encodings of target concepts rather than the hypotheses produced by a concrete learning algorithm. It thus fails to account for the existence of concepts which are similar to the targets and provide high accuracy while being significantly shorter or otherwise easier to learn. It also fails to account for cases where learners were nearly correct and perhaps miskeyed a small number of digits during the experiment. Many of these features were computed manually, so the analysis focused on a much smaller number of features than participants might have used.

Second, our account of visibility focused on the approximate difficulty of inferring the existence of each symbol in a concept definition. Because the inferences frequently rely on the interaction of inductive and deductive reasoning, our initial assessment was subjective and defined just three levels of visibility: visible, semi-visible, and hidden. This work would benefit from a more graded and rigorously computational account of visibility. Such an account could focus on defining visibility in terms of an execution trace, the step-by-step record of a program's behavior. A first step in this direction would be to measure visibility in terms of how many times a symbol was rewritten during the program's execution. Symbols which were never rewritten would be highly visible, while those rewritten several times would be less visible. An even better assessment, however, would be to provide a model of learning whose dynamics match the inferential dynamics of human learners. Such a model could then be analyzed to predict the visibility of given symbols in an LOT expression as well as the overall complexity of learning.

Perhaps the key limitation of this work, then, is that it did not center on comparing

human learners with implemented theories of learning whose performance is objective and whose dynamics can be analyzed in detail. It briefly applied three models of learning to explore ways in which structure-sensitive learning mechanisms might differ from description-length-based accounts of learning. It did not, however, deeply investigate the dynamics of learning in an implemented computational model. Nonetheless, the results suggest that such a model would likely benefit from an algorithm which can dissociate the complexity of learning from the complexity of the learned hypothesis and which is able to combine inductive and deductive inferences to quickly infer certain kinds of conceptual structure. The HL model discussed in Chapter 4 is designed in precisely this way. Our results show that it decidedly does not yet match the scope and flexibility of human learners[3], but its diversity of learning mechanisms appear to allow it to quickly acquire certain kinds of concepts. Some of these mechanisms are sensitive to patterns demonstrated to be important here, such as recursive and conditional reasoning. Moreover, its ability to memorize data, analyze its detailed structure, and chain together moves that combine inductive and deductive reasoning captures the basic intent of our visibility analysis. It goes beyond that analysis, however, by providing a generative account of how each symbol is inferred.

Chapter 6 more deeply investigates these issues by comparing HL against several alternative models of program-induction-based learning in the LOT in terms of their ability to predict human learning performance. Given the scope of HL's initial learning mechanisms and the dramatic impact which language size has on algorithms which rely on exhaustive or stochastic search, the comparison focuses on 100 of the 250 concepts reported here. These concepts were specifically designed to be learnable from a minimal DSL while exhibiting a variety of learning difficulties and algorithmic patterns. The comparison looks at both how accurately these models are able to predict human performance as well as how sensitive their algorithms are to the semantic features analyzed here.

[3]For example, it cannot yet capitalize on the subtle differences in behavior between `filter`, `take`, and `slice` used above to introduce the concept of visibility.

Chapter 6

Human-like learning of list functions

6.1 Introduction

Developing a computationally precise account of human learning is one of the central goals of cognitive science. It has fascinated those interested in the mind since before cognitive science could be rightly identified as its own discipline. Boolean logic (Boole, 1854), the Turing machine (Turing, 1936), and artificial neural networks (McCulloch & Pitts, 1943) were all inspired, if not explicitly motivated, by the desire to provide a computational explanation of learning and cognition. The earliest works of modern cognitive science share a similar goal, using computational language to explain mental behavior (Newell et al., 1958; Newell & Simon, 1956; Newell et al., 1959; Chomsky, 1959; Miller, 1956). These early efforts have since evolved into several disciplines focused on providing computational models of intelligent behavior including: cognitive science itself, artificial intelligence (Russell & Norvig, 2002), machine learning (Murphy, 2012), and even the newly emerging science of machine behavior (Rahwan et al., 2019).

Even so, the models produced after decades of research by tens of thousands of scientists fall far short of the accomplishments of children born three or four years ago. These children can: identify objects and their affordances; communicate verbally through talking, singing, whispering, shouting, and joking; introspect on their own desires, preferences, and beliefs; follow instructions; coordinate in social situations to help, play, or determine ownership; perform routine tasks like making beds and dressing themselves; navigate any environment

they find themselves in by walking, running, crawling, climbing, jumping, rolling, and any number of other means they happen to invent; count and perform basic arithmetic; provide causal explanations for what they observe in the world and in the behavior of other people; invent stories about princesses and dinosaurs and the dog down the street; compose complex physical artifacts using wooden blocks, sand at the beach, or whatever is at hand; and the list goes on. Models of any one of these things fall short of human abilities, yet children do all these things flexibly and in combination with one another.

Despite the flexibility and expansiveness of human cognition, or perhaps because of it, those interested in the organization of the mind have long noted its seeming ability to compose smaller units of meaning into larger ones in rule-like ways. These observations led Boole, for example, to hypothesize that thought occurred in a mental language based on Boolean logic. The idea of a mental language, or language of thought (LOT; Fodor, 1975), has been used to explain the systematic, productive, and compositional nature of thought. The core claim is that the mind is organized by means of a mental language such that individual concepts are expressions in that language, and learning consists of discovering expressions which make sense of observed data. Modern formulations have converged on the idea that this LOT behaves like a probabilistic programming language, which additionally helps to explain gradedness, variation, and computational expressiveness in human thought (Goodman et al., 2015) while making strides to resolve long-standing debates in the field (Piantadosi & Jacobs, 2016).

If knowledge is expressed as programs, learning is expressed as programming. More specifically, learning becomes program induction (Flener & Schmid, 2008)—the construction of programs to explain observed data. Many early learning as programming models grew out of a tradition focused on providing a computational-level[1] explanation of learning as

[1]David Marr (1982) discusses three levels at which cognitive phenomena can be explained. The computational level focuses on function. It considers the problem to be solved, the space of hypotheses, and the objective to be satisfied. The algorithmic level focuses on specific algorithms and representations by which the problems, hypotheses, and objectives of the computational level can be realized. The implementational level focuses on how these algorithms and representations are implemented in a physical system like the brain, a silicon computer, or your sock drawer. Work since Marr has generalized the three levels suggesting for instance, a level between the computational and algorithmic (e.g. Lieder & Griffiths, 2020; Griffiths et al., 2015). To avoid confusion in the discussion of models which use computation to explain cognitive phenomena and models which operate at Marr's computational level, I use *computational* to describe the former and *computational-level* to describe the latter.

Bayesian inference (see Ullman & Tenenbaum, 2020, for a review). The core of the idea is that people maintain a distribution over possible solutions for any given problem. This distribution initially reflects prior knowledge but is updated using Bayes' Law to incorporate new observations in a principled way. Early models focused on fairly small hypothesis spaces in which each hypothesis was relatively unstructured and could be explicitly considered and updated as each new datum was observed (Tenenbaum, 1999, 2000; Kemp & Tenenbaum, 2009; Kemp et al., 2007; Goodman et al., 2011).

The formalisms used by many learning as programming models, however, defined unboundedly large hypothesis spaces, making it infeasible to exhaustively consider all possible hypotheses. Models were instead forced to search, considering only a small part of the hypothesis space. Still motivated by providing a computational account of learning, these models focused on search algorithms with clear computational-level interpretations and guarantees, models that moved through the posterior distribution over hypotheses in predictable ways. Markov chain Monte Carlo (MCMC) and other probabilistic inference algorithms thus became common search algorithms in computational models of learning (Kemp & Tenenbaum, 2008; Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2012, 2016; Ullman et al., 2012; Griffiths et al., 2015). Artificial intelligence models of learning as programming, and algorithms developed in the field of program synthesis more generally (Gulwani et al., 2017), have tended to take a similar approach. These algorithms tend to focus on the highly-engineered application of a single technique, including MCMC (Schkufza et al., 2013; Alur et al., 2013), as well as enumeration (Feser et al., 2015; Akiba et al., 2013; Gulwani et al., 2011; Katayama, 2013), SAT/SMT-solving (Solar-Lezama, 2008; Alur et al., 2013), type-directed synthesis (Osera & Zdancewic, 2015; Polikarpova et al., 2016) meta-interpretative learning (Muggleton et al., 2015; Cropper et al., 2019), genetic programming (Koza, 1989; Koza & Koza, 1992; Poli et al., 2008; Langdon & Poli, 2013), and neural program synthesis (Bošnjak et al., 2017; Feser et al., 2017; Gaunt et al., 2016; Parisotto et al., 2016; Devlin et al., 2017).

Humans, however, are not computational-level abstractions. Our history as a species and as individuals has given us specific mental representations, values, and learning mechanisms that deeply impact the kind of learning that is easy (Fodor, 1983; Carey, 2009) or even

possible (Chomsky, Keyser, et al., 1988; McGinn, 1993). Rather than operating primarily by means of a single learning mechanism to advance a single objective function, people make use of a wide variety of learning mechanisms, selecting between them as appropriate for the task at hand (Siegler, 1996; Siegler & Shipley, 1995). They are constrained by the need to solve many kinds of problems with sharp limits on their time and energy (Lieder & Griffiths, 2020; Lehman & Stanley, 2011a). They flexibly devise interventions on the world to tease apart confounding evidence, interpret evidence according to context, and form sophisticated causal theories (Schulz, 2012b; Gopnik, 2012). They make use of various forms of analogy, thought experiments, and mental simulation to reason constructively (Lombrozo, 2019; Xu, 2019). They posit changes to intuitive and formal theories by means of placeholder concepts that bring about radical conceptual change and bootstrap the development of entirely new domains of thought (Barner & Baron, 2016; Carey, 1985, 2009; Gopnik, 1983).

While the idea of learning as programming has produced successful computational-level models in a number of domains, these efforts frequently require extensive hand-engineering (e.g. Lake et al., 2015; Piantadosi et al., 2016). A new approach that introduces strong but general algorithmic-level constraints is likely needed to drive the next generation of computational models of learning. The child as hacker hypothesis refocuses the idea of learning as programming around hacking, a particular type of programming practiced by people. Hacking pursues internally motivated goals through an open-ended set of values and techniques. It claims that a diverse toolkit of learning mechanisms is a central feature of learning essential to explaining the richness of human development. It simultaneously frames known learning mechanisms as techniques for revising code-like mental representations while identifying the practices of actual hackers as a rich source of concrete hypotheses for mechanisms that humans might use during learning.

This chapter compares HL, a computational model of learning as hacking (Chapter 4), against several paradigmatic algorithms common in computational-level models of learning as programming[2]. The models represent a broad spectrum of single-technique approaches to learning including: exhaustive search, stochastic search, deductive proof, and neural program

[2]I am exceptionally grateful to Andrew Cropper, Kevin Ellis, Max Nye, and Steve Piantadosi for their effort and expertise in contributing data for the comparison models.

synthesis. We compare the models in terms of their ability to predict human performance on a large set of concept learning tasks focused on list functions. The human behavioral data analyzed in this chapter come from the experiment described in Chapter 5. See the description and analysis there for more details on the experimental method and the patterns of results seen in humans.

We focus our comparison on 100 of the 250 concepts tested with humans. These concepts are amenable to description in a relatively minimal domain-specific language (DSL) that serves as our model LOT. At the same time, the concepts vary widely in terms of how easily humans acquire them and in the algorithmic abilities required to express them, which include conditional, recursive, and pattern-based reasoning. Moreover, we test concepts using two different language sizes by varying the numerical constants available during learning: some concepts rely on just the numbers 0–9, while others also require the numbers 10–99.

We restrict our analysis to concepts learnable from a minimal DSL for three reasons. First, several of the models are extremely sensitive to the size of the language in which they search. Providing a rich DSL would make it extremely time consuming to learn any program more than a few symbols long. In order to compare a large set of concepts in a timely manner while performing meaningfully complex searches, the DSL needed to be relatively small.

Second, one of our conclusions in Chapter 5 is that humans appear to make use of learning mechanisms which exploit particular features of observed data to make structured transformations either to individual hypotheses or to the entire hypothesis space. This point focuses primarily on the algorithmic-level dynamics of learning. By restricting the DSL, we are able to more clearly assess each model's ability to capture these dynamics. We show that HL is able to learn more of these 100 programs than the other models in our comparison, and that it does so while predicting human performance at least as well as any other model. We also show that HL far exceeds the other models considered here in terms of the differences between its mean accuracy and that of human learners. These results strengthen the conclusions of the empirical analysis in Chapter 5 by showing how a model making use of mechanisms that dissociate length from learning difficulty compares favorably with models in which that association is a defining feature. Our results also provide empirical support for the usefulness of the child as hacker hypothesis in generating testable hypotheses

about human learning.

Third, HL adopts a view of learning which evolves the basic structure of a model LOT over time. Rather than defining new concepts in terms of innate primitives, it adapts the set of primitives and their meaning to fit the data. This is perhaps more similar to psychological accounts of conceptual change (Barner & Baron, 2016; Gopnik, 1983; Carey, 1985, 2009) than to the traditional Fodorian view in which all concepts are merely compositions of an unchanging set of primitives (Fodor, 1975; Fodor, 1980). By restricting the model LOT such that it provides little more than the basic symbols required to express the data themselves, we model learning in an environment that is thus much more similar to that in which children find themselves. The models know the structure of the data as given, but they cannot draw on richly developed theories describing how to manipulate these data. The dynamics of each concept must basically be learned from scratch. The overall complexity of learning is then somewhat comparable to the kinds of learning underlying key conceptual achievements throughout childhood, such as the cardinal principle (Piantadosi et al., 2012) or basic algorithms for addition (Shrager & Siegler, 1998).

6.2 Concepts

The 100 concepts, concepts c001 to c100 in Appendix A, can be organized into five groups of 20 concepts each.

The first group contained four subgroups of five concepts each. Each subgroup focused on a simple manipulation task: indexing into a list, extracting a prefix, extracting a sublist, or replacing the value of an element. The five problems in each subgroup were all variants of the basic problem associated with the subgroup. These problem varied systematically to test for differences in: recursive and non-recursive reasoning, the ability to handle exceptions, and biases toward elements occurring early in a list. The first problem was a non-recursive variant occurring early in the list (e.g. *return a singleton list containing the third element*). The second problem extended the first by introducing a case for dealing with exceptional examples (e.g. *return a singleton list containing the third element if there is one, else return the empty list*). The third and fourth repeated the first and second, respectively, but using indices

occurring much later in the list (e.g. *return a singleton list containing the seventh element* and *return a singleton list containing the seventh element if there is one, else return the empty list*). The fifth problem in each subgroup required some form of recursive reasoning, replacing any arbitrary constants with values computed directly from the list (e.g. *return the n^{th} element of the tail of the list, where n is the first element*).

The second group was designed to test various kinds of conditional reasoning and contained five subgroups of four concepts each. The first two concepts were trivially different variants of a concept which did not require recursive reasoning (e.g. *insert 5 as the second element* and *insert 8 as the second element*). The third concept introduced a condition based on structural properties of the list such as equality between elements or the length of the list (e.g. *insert a new second element—8 if the list length is greater than 5, else 5*). The fourth concept introduced a condition based on specific numerical values in the list (e.g. *insert a second element—8 if the first element is greater than five, else 5*).

The third group contained 20 assorted tasks that could be solved without recursive reasoning, and the fourth group contained 20 problems requiring recursive reasoning. The fifth group contained 20 problems representatively sampled from the first four groups, modified such that any constants in their definitions, as well as the associated input/output pairs, were drawn from the range 0 to 99 rather than 0 to 9.

6.3 Models

The primary purpose of this work is to compare computational models of learning as programming against human learners. We are particularly interested in explaining how humans are able to draw semantic inferences from observed data that help them learn a concept more quickly than might be expected given its complexity in the LOT. The specific hypothesis under investigation is whether a computational model of learning explicitly incorporating hacker-like elements provides a better explanation of human learning than models inspired by learning as programming more broadly. To test this hypothesis, we compared the performance of five models of concept learning. The HL model described in Chapter 4 serves as the hacker-like model. Each of the four other models represents a broad class of learning

Usage	Type	Description
(λ x body)	t1 → t2 → (t1 → t2)	lambda abstraction; binds x for use in body
0, 1, 2,...,99	Int	natural numbers
nan	Int	an out-of-bounds number (i.e. < 0 or > 99)
true, false	Bool	Boolean values
[]	[t1]	empty list
(cons x xs)	t1 → [t1] → [t1]	prepend x to xs
(+ x y)	Int → Int → Int	add x and y
(- x y)	Int → Int → Int	subtract y from x
(> x y)	Int → Int → Bool	true if x is less than y
(if p a b)	Bool → t1 → t1 → t1	a if p is true, else b
(== x y)	t1 → t1 → Bool	true if x and y are structurally identical
(is_empty xs)	[t1] → Bool	true if xs is empty
(head xs)	[t1] → t1	first element of xs
(tail xs)	[t1] → [t1]	drop the first element of xs
(fix x f)	t1 → ((t1 → t2) → t1 → t2) → t2	recursively apply f to x

Table 6.1: The primitives initially provided to each model.

algorithm common in the learning as programming literature: exhaustive search (Enumeration), stochastic sampling (Fleet), deductive proof (Metagol), and neural program synthesis (RobustFill). We equipped each model with a minimal yet theoretically expressive DSL—a typed lambda calculus like the model LOT in Chapter 5. The primary difference between the two DSLs is that we deliberately restricted the set of primitives to just those listed in Table 6.1.

For each concept, every model completed 5 runs of all 11 trials, searching for 10min per trial in an online setting. Several of the algorithms rely heavily on stochastic behavior; running each algorithm multiple times per concept provides a better measure of variance in the algorithm due to randomization. Each trial ran for a total of 10min. That is, each model was allotted 10min search on trial one, another 10min on trial two, and so on, for a total of 110min of search per run of each concept. For each trial $1 \leq i \leq 11$, the correct input/output pairs for the previous $i-1$ trials were made available as training data, as well as the input for trial i. The correct output of trial i was held out as test data. As a result, the training data set was empty during the first trial, as it was for human participants. Each model except Metagol was setup so that search during trial $i+1$ started where trial i finished, reusing some portion of the computation from trials $1 \ldots i$ to hotstart trial $i+1$. Metagol's design makes online learning difficult, so it treated each trial independently. At the end of trial i's search period, each model selected a best hypothesis and used it to predict an output for the current input. We describe each model in more detail below.

6.3.1 Enumeration

Exhaustive enumeration is an extremely general program induction algorithm. Given some ordered generative process, it generates each program in order and evaluates it. If a solution exists and the generative process includes that solution in its hypothesis space, enumeration provides a simple brute-force method guaranteed to eventually discover the correct hypothesis (Solomonoff, 1964a).

We used the high-performance enumeration algorithm described in Ellis et al. (2020). This model performs type-directed top-down grammar-based enumeration in approximately decreasing order of prior probability. That is, it treats the type system as a grammar over programs and, starting from a requested type, iteratively lists all programs matching the given type, starting with the shortest. The enumeration proceeds in depth-first fashion, with an outer loop of iterative deepening: it first enumerates programs whose description length lies is 0–Δ, then all programs whose description length is Δ–2Δ, then 2Δ–3Δ, and so on until the end of the trial. Δ was set to 1.5 nats; each task used a single CPU with no offline training or parameter learning.

To accommodate online learning, Enumeration used a simple win-stay, lose-shift strategy (Nowak & Sigmund, 1993). When asked to make a prediction, it used the first program discovered which correctly explained all previously observed input/output pairs. If its predicted output was also correct, it continued to use that program to make predictions on subsequent trials. If the predicted output was incorrect, it would select the first program to correctly explain all previously observed input/output pairs *plus* the newly observed pair revealed after making the prediction.

6.3.2 Fleet

Enumeration is a strictly deterministic algorithm; it lists all programs in order of length until it finds a suitable hypothesis. Many other algorithms, however, are fundamentally stochastic. These algorithms are broadly termed *stochastic search*. Genetic programming (Koza & Koza, 1992; Langdon & Poli, 2013), for example, takes inspiration from random processes at work in biological reproduction and natural selection. It randomly mutates and

crosses programs to produce a population of offspring. A subset of these are selected based on fitness, though this process is also frequently stochastic. Another common approach to stochastic search is to treat learning as sampling from a distribution over programs (Alur et al., 2013; Schkufza et al., 2013). A common approach in cognitive models using stochastic search in a model LOT is to frame synthesis as sampling from a Bayesian posterior combining a grammar-based prior with an accuracy-based likelihood (Ullman et al., 2012). Based on the specific structure of the distribution, a variety of algorithms can be used to sample from this posterior, including Markov chain Monte Carlo (MCMC; Goodman, Tenenbaum, Feldman, et al., 2008; Piantadosi et al., 2012, 2016) and Sequential Monte Carlo (SMC; Ellis et al., 2019).

This comparison represents stochastic search using Fleet (Piantadosi, 2020). As used here, Fleet runs a high-performance version of the rational rules (Goodman, Tenenbaum, Feldman, et al., 2008) version of Metropolis-Hastings over programs in the DSL, similar to the approach taken in Piantadosi et al. (2012, 2016). This MCMC technique proposes changes to entire subtrees of expressions by selecting a node from the tree uniformly at random and regenerating it from the grammar. It also used a parallel tempering scheme (Vousden et al., 2015). Intuitively, parallel tempering simultaneously conducts multiple searches in parallel MCMC chains, each at a different temperature. A chain with a higher temperature is more likely to consider low-probability hypotheses and thus to move more effectively around the entire space of programs with a fairly low degree of precision. Chains with lower temperatures, by contrast, provide more precise search but are more likely to be caught in local minima. Parallel tempering combines the best of both kinds of search by periodically and stochastically swapping hypotheses between chains. This allows the overall scheme to search broadly for promising areas of investigation and progressively narrow in on the most valuable hypotheses.

The particular scheme used in this comparison simultaneously explored five chains at different temperatures, adaptively spaced to have efficient proposal acceptance rates. The maximum temperature was set to the trial number plus one, and the minimum temperature was fixed to 1.0, meaning the lowest temperature chain theoretically sampled from the target posterior. Swaps between chains were proposed every second and temperatures were adapted

every 30s. The Fleet grammar did not include lambda abstraction due to limitations of the current implementation. Fleet is explicitly Bayesian. In these simulations, it used a grammar-based prior and a likelihood based on string edit distance (treating lists as strings of characters) which deleted each character from the end of a list with probability 0.0001, and then appended uniformly random characters with the same probability. This likelihood allows the model to assign nonzero probability to all data, but favors data for which the model generates prefixes. To support online learning, each new trial was started on the hypothesis with the best posterior hypothesis found previously.

6.3.3 Metagol

Another major category of inductive learning algorithms are based on various kinds of deductive proof. The basic idea is to encode all the semantic and syntactic constraints of both the task and the programming language into a logical formula. A solution to this formula—i.e. an instantiation of any free variables—is then guaranteed to describe a syntactically correct program which solves the task. Boolean SATisfiability (SAT) and Satisfiability Modulo Theories (SMT) techniques encode the entire problem as a propositional Boolean formula (Jha et al., 2010; Polozov & Gulwani, 2015; Torlak & Bodik, 2013). This approach has also been used successfully in systems which fill in high-level program *sketches* (Solar-Lezama, 2008) or which additionally encode distributional information to allow constraint-solvers to act as probabilistic samplers (Ellis et al., 2016). Inductive Logic Programming (ILP) systems take a different approach (Muggleton & De Raedt, 1994), encoding constraints as first-order Horn clauses (i.e. $X \rightarrow Y$, where Y is a single literal, and X is a conjunction of literals). The goal is to discover a first-order theory, also encoded using Horn clauses, which implies the data.

Metagol (Muggleton et al., 2015; Cropper et al., 2019; Cropper & Muggleton, 2016) is an ILP system based on a Prolog meta-interpreter which induces Prolog programs, which are expressed as a series of Horn clauses, written as `Head :- Body.`. `Head` is true if each literal in `Body` is true. Empty bodies are also true. The key idea of Metagol is to use metarules, or program templates, to restrict the form programs can take, and thus the hypothesis space. A metarule is a higher-order clause. For instance, the *chain* metarule

Name	Logical Form	Description
Identity	$P(A, B) \leftarrow Q(A, B)$	P is Q
Split	$P(A, B) \leftarrow Q(A) \wedge R(B)$	Make independent assertions, Q and R, about P's arguments
Pre-Condition	$P(A, B) \leftarrow R(A) \wedge Q(A, B)$	P is Q with a pre-condition R
Post-Condition	$P(A, B) \leftarrow Q(A, B) \wedge R(B)$	P is Q with a post-condition R
Chain	$P(A, B) \leftarrow Q(A, C) \wedge R(C, B)$	Compose Q and R to explain P
Memorize	$P(A) \leftarrow \top$	Assert that P always holds for A
Infer-A	$P(B, C) \leftarrow Q(A, B, C)$	P is Q, assuming some latent argument A
Infer-B	$P(A, C) \leftarrow Q(A, B, C)$	P is Q, assuming some latent argument B

Table 6.2: The metarules used by the Metagol model.

is `P(A,B) :- Q(A,C), R(C,B).`, where `P`, `Q`, and `R` denote higher-order variables and `A`, `B`, and `C` denote first-order variables. The goal of Metagol is to find substitutions for the higher-order variables. For instance, the *chain* metarule allows Metagol to induce programs such as `f(A,B) :- tail(A,C), head(C,B)`, which can be loosely translated to the DSL as `(λ x (head (tail x)))`. Metagol can induce longer clauses though predicate invention, similar to the introduction of lambda abstractions. Metagol works by partially constructing and evaluating programs, pruning the search space when a partial program fails to cover the positive examples. Metagol is also capable of using negative examples to constrain search, backtracking when a program covers a negative example. Our simulations, however, only drew on positive examples.

Because Metagol learns relations, the evaluation is slightly unusual. For instance, suppose it learned this two clause program:

```
f(A,B):-head(A,B).
f(A,B):-tail(A,C),head(C,B).
```

This `f` relation is nondeterministic and holds both over lists and their head elements and lists and their second elements. To evaluate Metagol on each induced relation, we called the Prolog program with the first argument and asked for answer substitutions for the second argument, taking the first provided substitution as the output. Metarules are key to Metagol, but deciding which metarules to use for a given task is an unsolved problem (Cropper & Muggleton, 2015). To compute the benchmark, we gave Metagol a small set of 8 metarules shown in Table 6.2.

6.3.4 RobustFill

Deep learning has become an increasingly important machine learning technique (LeCun et al., 2015). Its basic approach is to encode a problem as a series of numbers which are given as input to a multi-layered neural network. The inputs are propagated through the network layer-by-layer to produce a series of numbers as output. These outputs are then decoded. Learning occurs by feeding in training inputs, observing the output, and providing an error-correction signal which is backpropagated through the network, adjusting the strength of the connections between neurons as needed. These ideas have been applied to a broad spectrum of learning problems, including program learning. Some approaches have used neural networks to replicate the input/output relation encoded by a programs without learning an interpretable representation of the program itself (Graves et al., 2014; Reed & de Freitas, 2015; Joulin & Mikolov, 2015). We are more interested here in neural program synthesis, using a neural network to transform observed data into an explicit program representation (Balog et al., 2017; Bošnjak et al., 2017; Gaunt et al., 2016; Chen et al., 2019).

We focus specifically on RobustFill (Devlin et al., 2017), a neural sequence-to-sequence encoder-decoder model with attention. It is specifically designed for program induction domains specifying tasks via input/output pairs. Intuitively, RobustFill separately encodes each observed input and output. It then decodes each observation into a distinct distribution over programs by producing a distribution over the first symbol in the program, then the second, and so on. These per-observation distributions are then combined and interpreted to produce a single distribution over the next symbol in the program. RobustFill can be interpreted as a form of stochastic search but is so different in its implementation and dynamics from genetic programming or probabilistic sampling that we investigate it separately. The use of an attention network allows RobustFill to learn which parts of the input representations are most important for constructing the output representations, and similarly, which parts of the output to use to inform the construction of the program.

The model thus consists of three components, each of which is parameterized by a Long Short-Term Memory (LSTM; Hochreiter & Schmidhuber, 1997) recurrent neural network: an input encoder, output encoder, and a program decoder. Our implementation is nearly

identical to the Attn-A RobustFill model from Devlin et al. (2017). For each input-output example: the input encoder encodes the input example; the output encoder encodes the output example while attending to the hidden states of the input encoder; and the decoder attends to the hidden states of the output encoder and produces a hidden state for each decoding timestep. This process is done separately for each input/output pair, and the decoder hidden states for each IO pair are max-pooled to produce a final output vector which is used to produce a distribution over program tokens. The model differs from the Attn-A RobustFill model by adding a learned grammar mask. As in Bunel et al. (2018), we also learned a separate LSTM language model over the program syntax. The output probabilities of this LSTM are used to mask the output probabilities of the Robustfill model, encouraging the model to put less probability mass on grammatically invalid sequences.

The model uses standard supervised, teacher-forcing techniques for training for sequence to sequence models, minimizing cross-entropy loss on the training data. We used a hidden size of 512 and an embedding size of 128. We trained the network for 3 days. This meant approximately 105,000 iterations with a batchsize of 16 programs ($\sim$1.6 million random programs seen during training). Training programs could have a maximum depth of 6, and each was associated with 1 to 10 input/output pairs, with the number of examples being sampled uniformly at random for each program.

6.3.5 HL

Details of the HL model architecture are given in Chapter 4. Briefly, HL is a model of learning as hacking: it adopts representations, objectives, and learning mechanisms motivated by the child as hacker. It searches over the space of Term Rewriting Systems (TRSs), each of which serves as a program specifying the syntax and semantics of an entire programming language. This allows HL to add and remove primitives and adapt the meaning of existing primitives to fit observed data. It models learning as the iterative application of a toolkit of structured revisions, searching for meta-programs describing how a TRS is constructed rather than searching directly for the TRS itself. HL uses Monte Carlo Tree Search (MCTS) to construct a longterm memory which helps it avoid repetition and balance exploration of relatively unknown meta-programs against exploitation of known meta-programs. It uses multiple

context-dependent objective functions sensitive to the complexity of the meta-program, the complexity of the TRS, the accuracy of the TRS in explaining observed data, and the ability of the TRS to generalize sensibly on novel inputs. Finally, HL supports online learning by pruning its tree as new information becomes available.

For these experiments, we restrict HL to learn only deterministic TRS. We restrict the MCTS tree to a maximum depth of 50 steps or the complete application of 7 learning mechanisms, whichever comes first. In between trials, we retain paths to the 100 top-scoring solutions discovered in the previous trial. The grammar-based prior over TRS rules is set so that the probability of adding each additional rule is $\frac{1}{2}$ and the probability of selecting a given variable or constant is 50% higher than selecting function application, the only operator with an arity greater than 0. The generalization likelihood used $\alpha = 0.001$. The accuracy likelihood discounted previous trials with a discount factor of $\lambda = 0.9$ and constructed traces of with a maximum of 25 evaluation steps and a maximum term size of 200. It used a normal-order evaluation strategy—i.e. it always rewrote using the rule which applied to the left-most, outer-most part of the term—and only assigned probability mass to the final output of the trace.

6.4 Results

Every model completed each of the 100 problems, making predictions for every trial seen by human participants. Figure 6-1 compares each model to human performance. One striking feature of the figure is how many concepts there are for each system in which the model never makes a correct prediction. While there are only 19 such concepts for HL, there are 36 for Fleet, 64 for Metagol, 70 for Enumeration, and 72 for RobustFill. That is, three of the comparison models fail to produce a single correct answer for approximately two-thirds of the concepts. By contrast, there are 0 such concepts for humans. Moreover, for the concepts on which they have positive performance, many of the models do a poor job of predicting human performance. In many cases, they are accurate for concepts which humans find difficult and inaccurate for concepts which people reliably produce correct responses.

Another interesting feature of these data is that the performance of the models is roughly

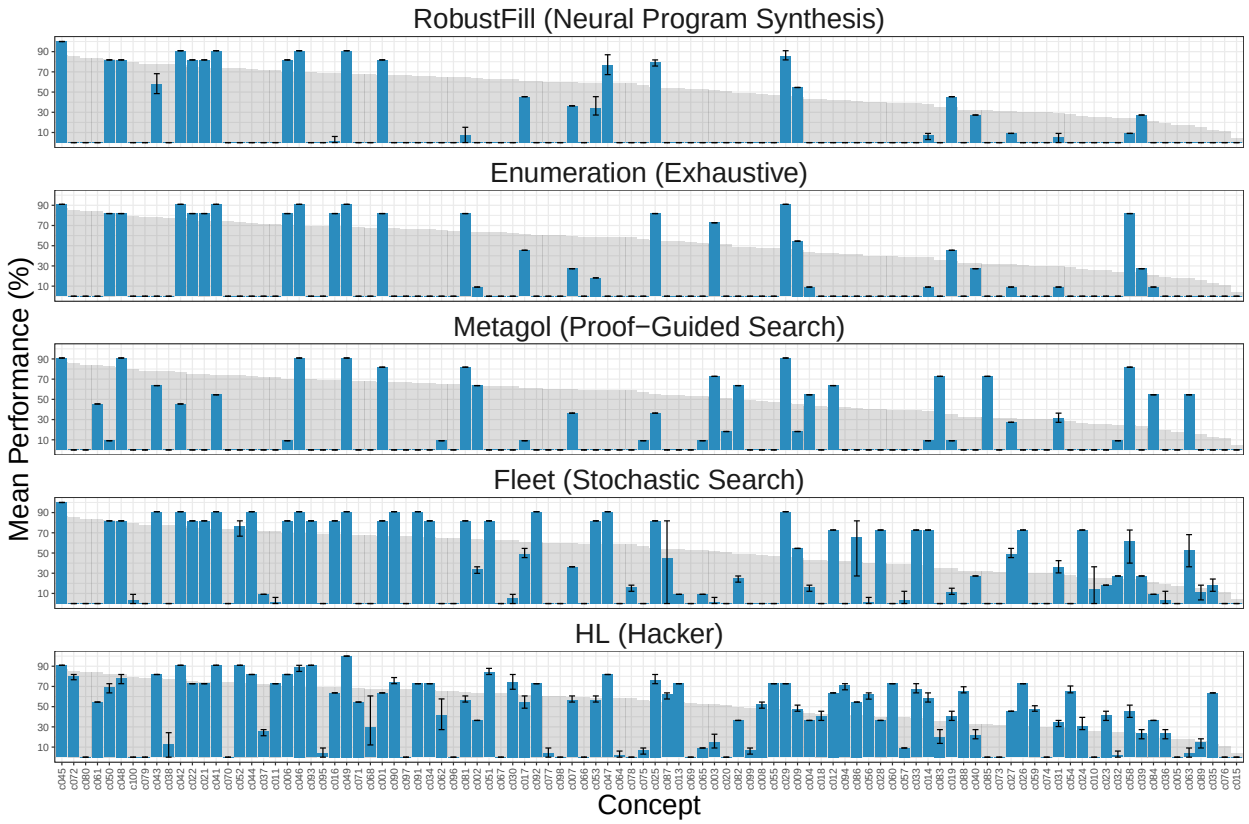


Figure 6-1: Mean accuracy (y-axis) on each concept (x-axis) by model (subplots). Concepts are ordered by mean human accuracy. Error bars are bootstrapped 95% CIs and gray region is human mean accuracy.

nested. With only two exceptions, Enumeration has non-zero performance on the same 28 functions as RobustFill, as well as 4 additional functions. These exceptions are *the list [8, 2, 7, 0, 3]* and *concatenate [9, 6, 3, 8, 5] with the input*, two functions using constants that might be easier to pick out using a neural network than to enumerate. Similarly, Fleet has non-zero performance on all of the 30 functions which Enumeration learns, as well 34 more for a total of 64. These include the two which RobustFill catches that Enumeration missed. HL's non-zero performance includes all but three of the functions from Fleet, including all the functions hit by RobustFill and Enumeration, as well as an additional 20. For two of these three, *reverse the list* and *select the maximum element*, Fleet gets a few trials right by memorizing special cases which reoccur by chance rather than reflecting a general algorithm. Only for *remove the first* $N+1$ *elements,* N = *element 1*, does it learns a working algorithm that HL never captures. Metagol does not fit quite as neatly into the nesting, but it only

has non-zero performance for two concepts on which HL is not also accurate at least some of the time. On one of these, *replace the first element with the last*, it gets a few trials right by randomly guessing a number with which to replace the first element. On the other, *remove all but element N+1, N = element 1*, it stochastically selects a number of elements to remove rather than learning a recursive rule. In only one instance, then, does a comparison algorithm find a general solution where HL finds no solution, and for only three other problems does a comparison algorithm discover a partial solution where HL does not also discover at least a partial solution.

By contrast, HL uniquely discovers partial solutions to 16 problems. That is, for these 16 problems, HL is the only model to produce one or more correct responses. For 11 of these, its accuracy is greater than 25%. Table 6.3 lists these 11 concepts and a learned meta-program along with the mean performances of HL and human participants. HL does not always closely predict human performance, but in so far as it is the only model which makes any correct predictions on any of these 11 concepts, it is also the most accurate. The meta-programs show that the key to HL's performance here is its ability to rely on structured revisions like `MemorizeAll`, `Generalize`, `Recurse`, `AntiUnify`, and so on. For none of these 11 is random sampling part of the learned meta-program. By contrast, all of them rely on the ability to closely analyze the data using `MemorizeAll` and then perform abstractions via `AntiUnify` and `Variablize` to expose the data's latent structure. Several also rely on refactoring moves like `Recurse` and `Generalize`; these moves typically make a solution worse, but they setup future revisions which can dramatically improve the overall value of a hypothesis. In short, the ability to systematically explore hypothesis-driven changes to candidate meta-programs, including the continued exploration of seemingly suboptimal solutions, is essential to explaining cases where HL is unique among the models examined here in explaining why certain concepts might be learnable for humans.

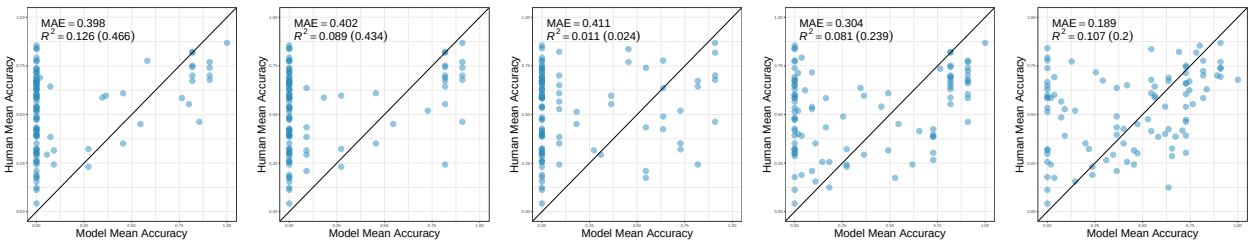
Figure 6-2 examines the relationship between human and model predictions more closely. Each subfigure plots the relationship between a particular model's mean accuracy and human accuracy on each concept. In terms of overall variance examples, no model provides a particularly compelling account of human performance. This is due at least in part to the number of concepts for which the models provide no correct predictions at all. When

Description & Meta-Program	Accuracy (%) HL	Humans
repeat every element 2 times in order of appearance `TRS.MemorizeAll().Recurse(...).AntiUnify().Stop()`	0.80	0.86
swap elements 1 and 3 and elements 2 and 4 (0–9) `TRS.MemorizeAll().AntiUnify().Variablize(...).Variablize(...).Variablize(...).Stop()`	0.73	0.47
elements 3, 2, 1, the number 4, then elements 5 and 7, in that order `TRS.MemorizeAll().AntiUnify().Stop()`	0.73	0.39
swap elements 1 and 3 and elements 2 and 4 (0–99) `TRS.MemorizeAll().AntiUnify().Variablize(...).Variablize(...).Variablize(...).Stop()`	0.71	0.42
replace elements 1 and 2 with element 3 `TRS.MemorizeAll().AntiUnify().Variablize(...).Variablize(...).Stop()`	0.66	0.29
swap elements 1 and 4 if element 2 = element 3, else swap elements 2 and 3 `TRS.MemorizeAll().AntiUnify().Variablize(...).Variablize(...).Variablize(...).Stop()`	0.66	0.33
add 2 to every element `TRS.MemorizeAll().Recurse(...).Generalize(...).AntiUnify().Stop()`	0.55	0.68
the first 6 elements `TRS.MemorizeAll().AntiUnify().Stop()`	0.53	0.48
swap elements 4 and 8 `TRS.MemorizeAll().AntiUnify().Variablize(...).Variablize(...).Stop()`	0.47	0.30
replace element 6 with a 3 `TRS.MemorizeAll().AntiUnify().Variablize(...).Stop()`	0.40	0.43
concatenate input and [7, 3, 8, 4, 3] `TRS.MemorizeAll().Recurse(...).AntiUnify().Variablize(...).Stop()`	0.29	0.68

Table 6.3: Concepts for which HL performs above 25%, while alternative models fail to give a single correct response, along with a representative meta-program learned by HL, HL's mean accuracy, and human mean accuracy.

examining just those concepts for which the models have non-zero accuracy, the fits are slightly better but again, the models leave most variance unexplained. The coefficient of determination, however, tells an incomplete story. For example, RobustFill and Enumeration have the highest R^2 values for concepts with non-zero performance, but this is largely because most of the concepts for which they make any accurate predictions were both easy for humans and for the models. There are very few cases where either model accurately predicts low or moderate levels of performance. Moreover, HL seems to more closely predict human accuracy for far more concepts than the other models. Because there are a few significant inaccuracies, the total variation explained remains low. The median absolute error (MAE), however, is much lower for HL than for the other models.

To examine this last point more closely, we computed the difference between humans and models for each concept. Figure 6-3 plots these values, focusing particularly on the models which are closest to human performance for each concept. For 82 of the 100 concepts, HL is either the closest to predicting human performance or within 5%, and there are only 2



(a) (b) (c) (d) (e)

Figure 6-2: A comparison of program induction models plotting mean model accuracy against mean human accuracy. MAE lists the median absolute error across all concepts, and R^2 lists the overall coefficient of determination for all concepts and, in parentheses, for concepts on which the model made at least one correct prediction. (a) RobustFill; (b) Enumeration; (c) Metagol; (d) Fleet; and (e) HL.

for which it is more than 20% away from being the closest. By contrast, Fleet is within 5% for only 43 concepts, Metagol for 36, Enumeration for 36, and RobustFill for 35. Moreover, HL's absolute difference from human performance is less than 25% for 61 of the 100 concepts, while these numbers are significantly lower for other models (Table 6.4).

Figure 6-4 reinforces this point by plotting the distribution over absolute error between humans and models for each model. RobustFill, Enumeration, and Metgaol have an approximately equal MAE, and their absolute errors appear to be fairly uniformly spread between 0 and 0.8, near the ceiling for human performance in this experiment. That is, they are roughly as likely to capture human performance perfectly as they are to learning nothing for a concept that humans learn after one or two trials. Fleet is notably better, but a quarter of the concepts have an absolute error greater than 50%. HL is significantly better in this regard than any comparison model. The cases in which it significantly deviates from humans, however, are telling. All but one concept for which HL has an absolute error greater than 50% require recursive reasoning. While HL has the ability to represent certain kinds of recursion, the way it does so (or fails to do so) does not match the ways in which humans solve these same problems. This problem appears widespread; recursive concepts appear disproportionately difficult for all five models. By contrast, 26 of the 30 concepts for which HL is within 10% of human performance rely on pattern matching and case-based reasoning. Only four require recursive reasoning, and of these, human performance was near zero for one and near ceiling for another. Only two represented moderately difficult recursive prob-

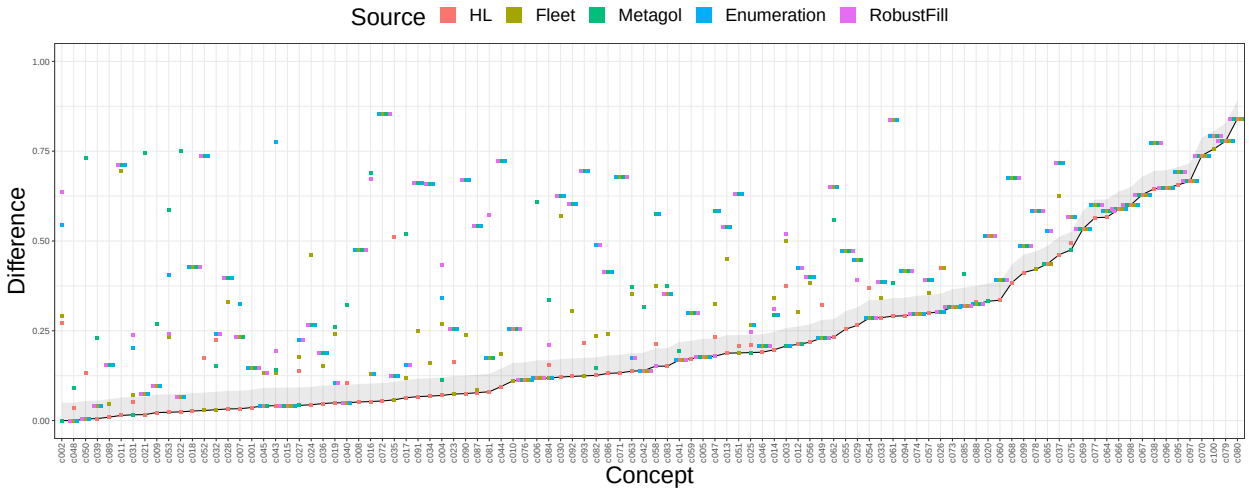


Figure 6-3: Difference of mean model accuracy from mean human accuracy (y-axis) for each concept (x-axis) by model (colored dots). Line plots minimum difference. Points in shaded region are within 5% of minimum difference.

lems. The more general point is then that HL appears to excel at capturing pattern-based and conditional reasoning in people, while its dynamics for recursive reasoning are not yet a good match for humans.

The main hypothesis of this chapter is that its diversity of structure-sensitive learning mechanisms enable it to better account for learning than the comparison models. Part of accounting for human learning is being sensitive to the same kinds of structure in the same ways. The preceding analysis of HL's absolute errors touches on this subject. To investigate it further, we fit a logistic model like that in Figure 5-6b for each learning model. We specifically predicted the model's trial-by-trial accuracy using the semantic features associated with each concept. That is, for each kind of learner, human, HL, Fleet, and so on, we used the concept-specific features described in Chapter 5 to explain the performance of that learner. We then compared the coefficients of these models against the coefficients determined from human learners in Figure 5-7. Figure 6-6 shows the results of this analysis. For each feature, it plots the coefficients for humans, followed by the coefficients for each model.

Several results stand out. First, HL is the only model to capture both the quantitative scale and qualitative pattern humans display for visibility, in which an increase in hidden symbols makes a concept harder than an increase in semi-visible symbols which in turn makes a concept harder than an increase in visible symbols, which actually may have an

	Non-Zero	Best $\pm 5\%$	$\lvert\text{Error}\rvert < 25\%$
RobustFill	28	35	33
Enumeration	30	36	30
Metagol	36	36	25
Fleet	64	43	43
HL	**81**	**82**	**61**

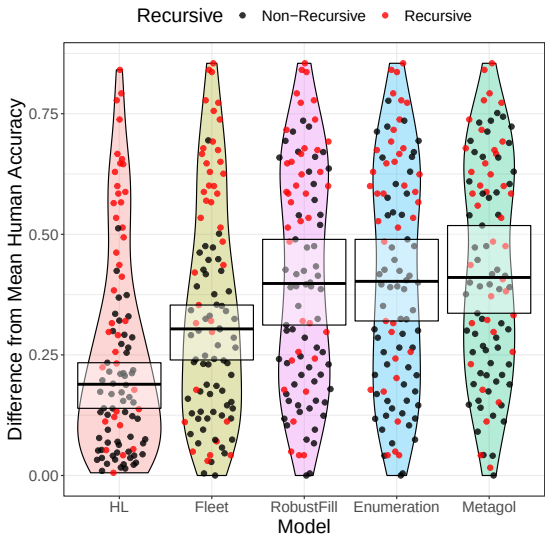


Table 6.4 & Figure 6-5: Table: A summary of model performance relative to human learners, including the number of concepts for which each model gives non-zero performance, the number for which it is within 5% of providing the closest prediction of human accuracy, and the number of concepts for which each model's absolute error in predicting human performance is greater than 25%. Figure: Differences between mean human accuracy and mean model accuracy (dots) summarized with a Gaussian kernel density estimate (colored regions). Red dots indicate recursive concepts. The crossbar plots the median with a 95% bootstrapped CI.

overall positive effect on learning.

Second, HL accurately captures the approximate contribution of several semantic features, including counting, the use of conditionals, and the relative insignificance of the number of variables. All models in fact found that concepts requiring counting knowledge were easier to learn than other concepts. This is perhaps surprising in that none of the primitives specifically supported any sort of counting-based reasoning.

Third, there are several cases where HL fails to accurately capture the contribution of other effects, as with the use of numbers greater than ten, the role of internal arguments, and recursion. Fleet and Metagol capture the insignificance of the changes in alphabet size marked by ≥ 10. HL associates a moderate penalty with larger alphabets, perhaps because numerical information is not currently represented in such a way that its internal structure is exposed to HL's learning mechanisms. This would make certain kinds of mathematical operations more difficult to discover. As expected, however, enumeration and Robustfill are strongly and negatively impacted by the larger alphabet. Surprisingly, all models found

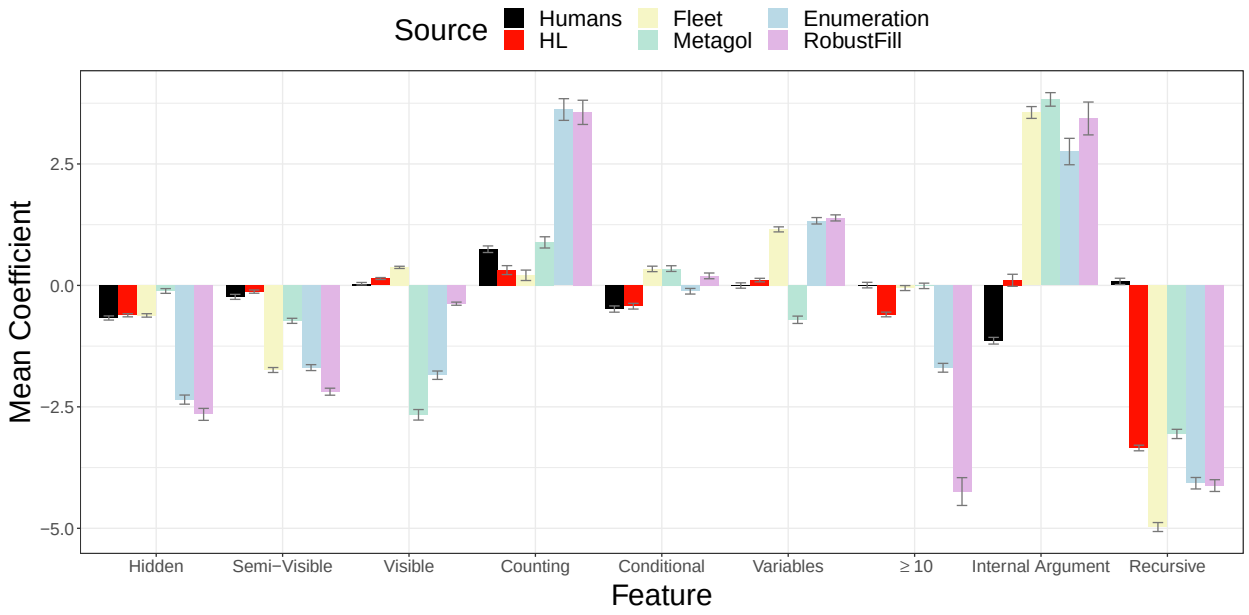


Figure 6-6: Regression coefficients for a series of feature-based logistic models, one for humans as well as for each learning model. Error bars are 95% CIs.

that the use of internal arguments made concepts easier to learn rather than more difficult, though HL is closest to matching humans. Finally, and as noted earlier, all models found recursion far more difficult than humans. The model LOT includes `fix`, a basic operator for constructing recursive functions, but it does not provide any specific utilities for common recursive patterns like mapping, filtering, or folding. Moreover, none of the models, including HL, are designed to identify signatures of, say, folding as opposed to mapping, or filtering as opposed to folding. These may be necessary to mimic human reasoning for recursive tasks.

In short, this analysis shows that HL captures a broader set of the structural influences on learning detected in Chapter 5 than the comparison models. This analysis also serves as a roadmap for future work in that it clearly identifies aspects of learning, such as recursive reasoning, where each model fails to capture human performance.

6.5 Discussion

This work provides an initial demonstration of the empirical value of the child as hacker hypothesis. It compared HL, a hacker-like model of inductive learning, against several alter-

native models of learning as programming. It has shown that HL more accurately predicts human learning performance than the comparison models in a richly structured domain with complex, rule-like concepts. It also identifies the ability to recognize structured relationships between input and outputs, particularly pattern-based and case-based reasoning, as playing a key role in HL's ability to accurately predict human learning performance for list functions. HL is based on fundamental insights about the representations, values, and techniques of hacking that work in concert to provide a novel algorithmic-level model of learning. The relatively poor fit of all models—not one fits nearly as tightly as a simple feature-based regression (Figure 5-8)—indicates that a great deal of work remains to be done to fully apply these ideas. All told, however, these results set the stage for a new generation of computational models based on applying insights from hacking to better understand the algorithmic-level dynamics of learning and the representational substrate supporting them.

Perhaps the most important lesson to draw from these results is that learning algorithms based on iteratively updating hypotheses by means of structured, semantically-sensitive revisions are better able to explain the dynamics of human inductive learning than competing models based on local search methods such as exhaustive or stochastic search. HL was explicitly motivated by these concerns and learns more concepts more accurately than the competing models examined here. It is also better able to capture the structural sources of difficulty observed in human performance. This suggests that the standard approach of explaining learning difficulty as a function of description length in a model LOT must be enriched to include more of the factors which cognitive psychologists have argued are central to learning. Computational accounts of learning would benefit by incorporating features like hypothesis-and-goal-driven search through patterns of constructive thinking (Carey, 2009; Lombrozo, 2019; Chu et al., 2019), a detailed sensitivity to the context-specific merits of a hypothesis (Schulz, 2012a), and the possibility of genuine conceptual change (Gopnik, 1983; Carey, 1985; Barner & Baron, 2016) by means of conceptual systems defined via conceptual role. Hacking and, more generally, the field of computer science have discovered techniques analogous to many of these aspects of learning and thus serves as a rich source of hypotheses for more realistic models of learning.

The approach we take in HL specifically relies on several essential ideas about hacking.

First, we recast program learning as language learning by learning programs which encode the syntax and semantics of a domain-specific language (Fowler, 2010). The goal of learning then becomes finding a language well-suited to the observed data—adding, removing, and redefining primitives as needed. The decision to implement this approach using TRSs (Bezem et al., 2003; Baader & Nipkow, 1999) focuses learning on identifying rules for evaluating input programs into output programs. The format of these rules permits sophisticated forms of symbolic pattern matching that make it possible to express the sort of structured relations between input /output pairs that are so crucial to HL's performance. Performing similar sorts of pattern matching in more common formalisms like lambda calculus, combinatory logic, or a fixed context-free grammar requires that either: the learner accomplishes the unlikely task of developing a pattern matching program alongside the target program; or the modelers implement pattern matching as part of the language, essentially extending it to behave more like term rewriting. Similarly, TRSs are, by definition, composed of a series of rules that naturally encode case-based reasoning. These rules can be learned independently, each identifying a specific kind of structure in the data. Other formalisms must learn to frame programs as a set of cases and typically must develop these cases concurrently to form a syntactically valid program. In short, TRSs provide basic representational tools for pattern matching, conditional reasoning, and framing learning as language evolution that appear to be cognitively important and which are lacking in the basic machinery of other common formalisms.

Second, HL adopts complex context-sensitive objective functions which allow it to trade off the contributions of multiple dimensions of value. The use of complex objectives helps the model to decouple the process of generating a program from the structure of the program itself while still being aware of both. Each can be represented by a separate term in the objective function and traded off against one another. Making these objective functions context-sensitive allows HL to care more about the description length of the program itself when reasoning about how likely a hypothesis is to generalize, and less when deciding which partial hypotheses to extend.

Third, HL adopts a diverse range of learning mechanisms which are sensitive to various kinds of structure in the current hypothesis. These are perhaps the most important feature

for allowing HL to dissociate the complexity of discovering a program from the complexity of the program itself. The iterative application of these mechanisms can quickly transform a complex set of rules into a fairly compact and highly general description of the same underlying dynamics. This dissociation, in turn, allows HL to quickly learn certain kinds of complex programs, namely those whose structure matches the structures which the various mechanisms can detect and exploit. `MemorizeAll` is particularly important in this regard, because it makes the structure of the observed input/output pairs available to HL for scrutiny. It allows HL to treat them as a sort of degenerate language which can be revised and generalized into a mature domain theory. This sort of sensitivity to structure in the data goes far beyond simply tallying the number of correctly predicted observations—a common practice in AI/ML—and seems to be a key piece of human learning often missing in computational models.

This initial comparison does not fully disentangle the contributions of these various elements. This is partially because these ideas build on one another in complex ways. The pattern-based, case-based approach of term rewriting not only makes it possible to tailor the LOT to data, but it makes many of the revision-based learning mechanisms technically feasible. Similarly, the use of an iterative, revision-based approach to learning relies on objective functions that are sensitive to the possibility for future revisions to improve existing hypotheses. More work remains to carefully explore the relative contributions of each element Even so, these initial results show that their total effect of HL's architecture is to provide a model which is more sensitive to features which impact human learning in our data. It also demonstrates the potential for hacker-like models to explain human learning more accurately than models inspired by learning as programming more generally.

That said, this work is only a first step. For example, the primitives with which the models were equipped are almost certainly insufficient to accurately model the relevant domain knowledge people bring to the list transformation domain. We used a small language here to emphasize the importance of a diverse and hacker-like set of learning mechanisms, but this decision likely harmed the fit of all models. Extending the set of primitives and providing HL with facilities for taking greater advantage of the input/output patterns associated with these primitives is a key area for future research. Perhaps the most urgent need is for

richer patterns of recursive reasoning. That said, the decision of which primitives to include, without an extensive and costly analysis (e.g. Piantadosi et al., 2016), is ultimately subjective and can be tuned to favor nearly any search algorithm. Enumeration, for example, could easily learn the 100 concepts tested given here if each were provided as a primitive, and various dummy primitives could be added such that it could perfectly model our human data.

The work we have done here demonstrates potential limits of simple computational-level algorithms as domains grow more complex and the precise dynamics of learning become more important for explaining humans patterns of generalization. At the same time, it proposes a model which only begins to close the gap between previous computational models and human learners. A great deal of work thus remains to produce any model which can compellingly capture the dynamics of learning even for seemingly simple domain like the list functions studied here.

Chapter 7

Conclusion

This thesis presented the child as hacker as a hypothesis about distinctively human-like learning. The hypothesis revolves around hacking—iteratively improving code through the internally-motivated and actively-managed pursuit of goals via diverse and open-ended sets of values and techniques. It claims that the representations, objectives, and mechanisms of hacking form a rich source of concrete hypotheses about human learning.

We then applied these ideas to construct, HL, a hacker-like model of inductive learning which builds on, but is importantly different from, existing program-induction-based learning models and is designed to reflect core principles of hacking. HL frames learning as the iterative development of an entire LOT. It uses objective functions favoring accuracy, well-formedness, discoverability, and simplicity and varies them in context-sensitive ways to entertain and revise hypotheses which are suboptimal but nonetheless contain useful structure. HL also uses a diverse set of structured learning mechanisms to iteratively explore a space of meta-programs describing compact generative processes for fitting an LOT to data.

We also introduced list functions as a domain for psychological investigation. Despite sharing many positive features with classic concept learning domains and a long history in artificial intelligence, this domain is relatively unknown in psychology. Even so, the combination of psychological familiarity, rich structure, algorithmic sophistication, and formal tractability make them a prime candidate for studying inductive learning.

A significant body of normative arguments and empirical results show that learners prefer simple hypotheses and find them easier to learn. The child as hacker, meanwhile, emphasizes

the diversity of values and mechanisms available to learners, many of which are tailored to exploit structure in data in ways that might dissociate the complexity of learning a concept from the complexity of the concept itself as expressed in some mental lexicon.

We therefore conducted a large scale investigation of human and machine concept learning. We facilitated an online concept learning experiment over 250 list functions designed to capture a wide range of learning difficulties and algorithmic operations. Our results show that in such a richly structured domain, simplicity—as measured by a concept's description length—predicts human learning. Predictions are much stronger, however, when the semantic features of individual concepts are taken into account. As part of this work, we develop a notion of visibility based on how transparently a symbol in a model LOT expression affects its input/output relation and show that the visibility of a symbol strongly modulates its overall contribution to learning difficulty. These results are unexpected if learning is primarily governed by a concept's description length in some mental language. Finally, we show that HL's structure-sensitive architecture more accurately predicts human performance in the list function domain than competing benchmark models of learning as programming. Its improved performance critically relies on its relative sensitivity to the semantic features which are so important for human learners.

We draw three conclusions from these results. The first conclusion is that the idea of hacking as the way people make code better can productively inform our hypotheses about learning. It has done so in at least four ways throughout this thesis. First, it suggested ways to explain developmental phenomena as a natural result of revising programs using a diverse set values and activities. While these ideas have not been explored in detail for the domains mentioned—small number addition, acquiring a count routine, learning there is no largest number, mastering a kinship system, or learning a formal theory like Mendelian inheritance—these examples make the broader point that hacking is a deep metaphor that provides a wealth of concrete hypotheses to explore. Second, the emphasis on the algorithmic character of thought led us to list functions, a new domain for studying human concept learning. Third, it suggested specific techniques to serve as a bridge between findings in cognitive psychology and computational tools for modeling them. These included the development of domain-specific languages, the iterative revision-based pattern underlying hacking, and the

maxim of avoiding premature optimization. These then drove the development of a model based on formal tools—including term rewriting, meta-programming, and Monte Carlo tree search—that helped to provide a more explanatory model of learning.

The second conclusion is that the idea of hacking has also helped us begin to capture cognitvely rich aspects of learning as thinking. Computational models of learning have grown significantly richer over the last decade or so in terms of the representations they use. Instead of the small and easily-enumerable hypothesis spaces of 20 years ago (e.g. Tenenbaum, 2000), Turing-complete formalisms are now increasingly common. Even so, models remain fairly similar in terms of their objective functions and learning mechanisms. Moreover, even where expressive formalisms are in use, they are often theoretically expressive without being practically useful. The set of background knowledge is typically small, and the formalisms rarely include innovations from programming language theory in pattern-matching, case-based reasoning, or sophisticated type systems. Yet, expressive representations and a diverse set of values of learning mechanisms are well attested both in learners and in the ways people write programs. The work in this thesis has begun to bring together several rich aspects of learning into a single model: analogy, goal-and-hypothesis-driven search, a willingness to tolerate errors that can be fixed later, and the need to base mental representations on conceptual systems defined according to inferential role. Our evaluation of this model suggests that while humans favor simple hypotheses and find them easier to learn, they may have also developed learning mechanisms that are sensitive to certain kinds of structure in observed data. These mechanisms perhaps allow them to dissociate the complexity of a program from the effort required to find it, quickly narrowing search to a small set of programs which, despite their complexity, are still likely the simplest explanations of the data. This suggests the need for a richer understanding of simplicity that goes beyond description length in a proposed language of thought to incorporate the rich interaction of data, changing objectives, and diverse learning mechanisms. Simplicity is not measured purely in terms of the final concept but also in terms of the generative process giving rise to that concept.

The third conclusion is that the child as hacker not only provides a roadmap for psychological work, but it also points the way toward more powerful algorithms for machine learning. This is not the main focus of the thesis, but the fundamental tools underlying the

models to which we compared HL were originally designed not to explain human learning but as powerful, general-purpose, machine learning systems. In a domain with a long history in artificial intelligence, list functions, our results showed that taking cues from hacking can provide a better learning algorithm.

In sum, this thesis has set the stage for a new generation of models framing learning as hacking in an effort to provide a computationally precise account of the richness and power of human learning. Still, all of the work here is just a first step. We are far from anything resembling a computationally precise account of learning and have only taken initial steps toward using hacking to improve the richness of our models. The rest of this final chapter considers the implications of these findings for future computational models of learning, the child as hacker hypothesis, and our understanding of learning more generally.

7.1 Representations

The model in this thesis treats the LOT as a term rewriting system. It does so as a way of combining the psychological processes of conceptual change and the development of distinct conceptual systems with the hacker's practice of developing domain-specific languages. Unlike common alternatives such as probabilistic context-free grammars, lambda calculus, or combinatory logic, term rewriting defines a sort of meta-grammar, a uniform interface in which individual expressions define the syntax and semantics of entire programming languages, modeling learning not as the acquisition of individual programs but as the evolution of the entire LOT in response to data. It natively supports a number of tools like symbolic pattern matching and case-based reasoning which appear important for human-like concept learning.

A good model of mental representations, however, must be able to naturally represent both the products of learning and the mechanisms by which learning occurs. Many, if not most, of our values and learning mechanisms are likely to be learned. They are frequently woven into the very fabric of the conceptual systems to which they are most relevant[1]. For

[1]This is arguably why metaphor is so important to human-like cognition (Dreyfus, 1992; Baum, 2004). Many useful tools for learning are perhaps embedded in domain-specific conceptual systems; metaphor and analogy are necessary to usefully transfer them to other domains.

example, the sensitivity to counting information people demonstrated in Chapter 5 is almost certainly learned, because it explicitly requires knowledge about number and the count routine, which are themselves learned. If so, these values and mechanisms are the output of learning, which means that they are program-like mental representations. Moreover, they are program-like representations that are then later used to pull apart, analyze, and reason about other program-like representations. The very system by which we learn is itself a product of learning.

In this, term rewriting shares a basic problem with other common formalisms. It, like lambda calculus or combinatory logic, is extensional. For a programming formalism, this means that two programs which behave identically cannot be distinguished within the formalism, even if they are implemented differently. More to the point, the internals of program structure cannot be reasoned about except in so far as they affect behavior. Extensional formalisms have no facilities for pulling apart, analyzing, and reconfiguring their own programs. This means, for example, that there is no term rewriting system which can determine something as simple as whether two other term rewriting systems are syntactically equal. If learning is a system of programs that pulls apart, analyzes, and revises itself, however, then an accurate model of learning may require an intensional formalism.

Barry Jay and colleagues have developed a new family of intensional formalisms that, like lambda calculus or combinatory logic, are extremely minimal (Jay, 2009; Jay & Given-Wilson, 2011; Jay, 2016; Jay & Vergara, 2017; Jay, 2019). The most sophisticated of these is SF calculus (Jay & Given-Wilson, 2011), a system which behaves very similarly to SK combinatory logic but supports intensional computation and can be easily extended to support many of the advantages of term rewriting, including: variable binding, symbolic pattern matching, type systems, and case-based reasoning. An exciting path for future work would be to reformulate the approach taken here in terms of a formalism like SF calculus. The ultimate effect would be to have an entire LOT-learning system specialized for the development of DSLs that is implemented in the language which it itself uses for learning. Future work could then investigate not only more accurate mechanisms for learning conceptual systems, but it could investigate how the learning mechanisms themselves are learned. The similarity between SF calculus and SK calculus would also lend itself to adapting the work of Pianta-

dosi (2016), providing a plausible pathway from high-level symbolic reasoning to a neural implementation.

7.2 Objectives

Hacking is driven by the active management of a rapidly evolving set of internally generated goals. Our approach takes a step in this direction by adopting two objective functions deployed for different tasks during learning. Together, they allow a learner to entertain and revise suboptimal hypotheses while still being aware of their inadequacies as a final explanation of the data. Moreover, these objectives are sensitive to the contributions of several dimensions of value including: accuracy, simplicity, discoverability, and the ability to produce well-formed guesses for novel inputs. These dimensions help the model to decouple the process of generating a program from the structure of the program itself while still being aware of both. Each can be represented by a separate term in the objective function and combined in complex ways, such as favoring either a simple process or a simple program during search while favoring the simplest process that generates the simplest program after search concludes.

Still, this is a far cry from an agent that: chooses its own values; uses its current values to select goals; actively manages its goals by deciding when to abandon, narrow, broaden, or set a goal aside for later reconsideration; adopts new goals, possibly from an unboundedly large space of possibilities; and connects its goals to specifically relevant learning mechanisms. Each of these abilities represents a significant, even daunting, challenge. How do we operationalize values like clarity, modularity, or elegance? How do we define the space of possible values? How do we connect values to goals and combine both into a meaningful objective function? Even so, these questions represent relatively unexplored territory, both computationally and empirically, and even first steps down any of these paths are likely to provide more faithful models than those currently available. Moreover, the active goal management we see in hacking is likely deeply connected to the role of goals in curiosity (Kidd & Hayden, 2015) and play (Chu & Schulz, 2020). Efforts to articulate these connections may also inform our models of learning.

7.3 Learning mechanisms

The diversity of learning mechanisms which people bring to bear in daily life is a hallmark of human learning, particularly the use of hypothesis-and-goal-driven search to enable learning by thinking. The diversity of techniques which hackers bring to bear in revising code is similarly a hallmark of hacking. Even so, the bulk of computational models in the learning as programming tradition rely on a single learning mechanism. Our work in HL extends traditional single-technique models of learning by recasting learning as the iterative combination of a toolkit of distinct learning techniques. It chains these techniques in context-sensitive ways to form meta-programs describing the generative process by which individual programs (in our case, term rewriting systems) can be constructed. This effectively decouples the complexity of expressing how to create a program from the complexity of the program itself.

Learners (Siegler, 1996; Xu, 2019) and hackers (Fowler, 2018) make use of hundreds of learning mechanisms, a much wider variety than the eleven which HL implements. These eleven have not themselves been tied to specific learning mechanisms known to operate in people. We therefore do not know whether these are among the most important mechanisms to be studying, or how well they capture the individual dynamics of specific mechanisms people use during learning. Detailed empirical study will be necessary to better understand which kinds of revision are most useful for modeling human learning.

It is also almost certain that models which incorporate a broader set of known hacking techniques—and which enrich the mechanisms for which HL has provided initial implementations—will be necessary to accurately model human learning. A rich set of learning mechanisms has the potential to make learning possible for large programs incorporating large sets of primitives by moving search from the level of individual symbols in individual expressions to the level of the structured generative processes by which entire expressions are created. Recasting search in this way makes it imperative that we also associate each mechanism with strong inductive biases that dramatically narrow the range of mechanisms which are likely to be considered in any given context. Ellis and colleagues (Ellis et al., 2020; Ellis et al., 2019; Ellis et al., 2018) have had great success in coupling symbolic learners with deep networks that reweight search options in context-sensitive ways. This has also been key in other

applications of Monte Carlo tree search algorithms like that used by HL (Silver et al., 2016; Simmons-Edler et al., 2018). Ideally, however, our models are able to learn more structured and more interpreatble sorts of biases that express the sorts of symbolic knowledge that humans bring to these tasks. In the list function domain, for example, this might include recognizing what distinguishes mapping (e.g. the same number of elements with the same function applied to each) from filtering (e.g. potentially fewer elements in the same order as in the original list sharing some common feature) from sorting (e.g. the same elements in a potentially different order based on some ordering relation). This type of information can be encoded in expressive type systems (Pierce, 2002) and learning algorithms exist which can make use of this sort of information (Polikarpova et al., 2016; Osera & Zdancewic, 2015). Learning how to identify these sorts of structured relations in the first place could provide a path toward the strong, symbolic inductive biases necessary for searching in a space of complex learning mechanisms. The key question here, then, is how we might build more powerful learning mechanisms.

More generally, the approach taken by HL can been seen as a way to reparameterize sampling. Rather than sampling at the level of programs, HL samples meta-programs, which can be used to construct programs. This strict separation, however, may be a false one. The `SampleRule` move, for example, simply lifts the program level to the meta-program level, making it possible for meta-programs to transparently encode programs. Moreover, there are even higher-level structures that are likely to be useful, such as learning to combine two moves or to favor specific combinations of parameters within a move. By implementing moves and meta-moves in a common framework, such as the intensional SF calculus discussed earlier, it may be possible to remove the program/meta-program distinction and frame both as expressions in a common language. This would then make learning inductive biases over the primitive symbols and learning inductive biases over learning mechanisms the same kind of learning, since each learning mechanism would be associated with a primitive symbol and implemented in terms of other primitives. This would provide a significantly different kind of LOT than has been explored in most learning as programming models to date. These differences, however, would permit the complex interactions between data, learning mechanisms, hypotheses, and objectives that we have argued are necessary to capture the

richness of human learning.

7.4 Conceptual systems

The models in this thesis framed the list function task as one of learning generative models for output lists conditioned on input lists. We also focused exclusively on lists of natural numbers. Humans, however, are likely capable of learning much more complex generative models, such as that in Figure 3-2. Extending the work done here to test and model this kind of learning in humans, as in Lake et al. (2015), would greatly strengthen the conclusions of this thesis. What can people learn about lists of lists of numbers, or lists containing different kinds of objects? What could they learn about trees, grids, or graphs? What other kinds of operations are easy for people to learn? Which are difficult? Can people learn to generate informative inputs for a concept? Can they generate new concepts that are similar to ones they have seen, or which are entirely novel? Can they learn to identify families of related concepts?

The tasks here also permit us to ask new kinds of questions, such as whether people can learn different strategies for when rewrite rules are allowed to apply (e.g. Can all rules apply at once? Can the same rule apply in multiple places? If not, how do we choose which place to rewrite?), the extent to which they are sensitive to the number of rewrite steps required to produce an output, and how learning varies with the number of rules to be learned. Extending our work in this way would allow us to unify it with work on learning other families of rule-based systems, such as fractal Lindenmeyer systems (Lake & Piantadosi, 2020).

Framing the explicit output of learning as an LOT also suggests the potential for large-scale simulations exploring the way the LOT evolves over the course of multiple tasks. This work would be very much in the style of other work on iterated learning (Dechter et al., 2013; Ellis et al., 2018; Ellis et al., 2020) which explores the way an agent might learn to self-select a curriculum of concepts that enables it to solve a large body of tasks. Work in which the output of learning is an entire LOT could be importantly different, however, in that it could learn entirely new data structures. That is, the ability to add new primitives and use them as

placeholders would allow it to associate latent structure with observed symbols. Over time these associations could become familiar data structures like numbers or lists. Long-term, this work could evolve into an exploration of how distinct domain-specific languages develop within the same overall LOT (e.g. lists and numbers) and the ways in which these distinct domain-specific languages evolve to interact toward solving complex problems (e.g. functions over lists of numbers). It could also permit the study of questions which have largely eluded computational models of development thus far, such as how a true concept of natural number might develop in a learner who is not initially equipped with such a concept.

7.5 Hacking

It is perhaps surprising that we hypothesize hacking as a means for explaining human learning, when there has been much more study of learning than there has been of hacking. Despite a great deal of practical literature written by hackers and software engineers about hacking (e.g. Fowler, 2018, 2010; Martin, 2009; Thomas & Hunt, 2019), hacking is relatively poorly understood as a cognitive activity. There are promising signs of increased scientific interest in hacking (Fedorenko et al., 2019; Ikutani et al., 2020; Vuculescu et al., 2020), and the semi-formal insights of hackers are ripe for transformation into an empirical theory of how people make code better. It is one of our deep hopes that this thesis would highlight the potential for progress in our psychological understanding of hacking and the critical role which that progress could play in our understanding of learning. Hacking provides a space where many of the relevant representations are explicitly available as code and modified using mechanisms similar to those hypothesized to underlie mental representations. Moreover, it is a place where humans routinely and simultaneously navigate complex spaces of values, goals, techniques, and programs, and some of these spaces are known to be Turing-complete. As has been argued throughout this thesis, a detailed understanding of hacking provides a clear path toward more capable algorithms for program synthesis and better models of human learning and cognitive development.

7.6 Developmental phenomena

This thesis is deeply inspired by developmental psychology and the breathtaking powerhouse that is a learning child. It is therefore sad that we have not directly grappled with specifically developmental data: our contributions in Chapter 2 were largely speculative and illustrative. Even so, the ideas discussed here have all been developed while trying to hold in tension what we currently know how to do computationally with what children know how to do effortlessly. The modeling tools we have developed are readily applicable to a number of central case studies in development, including kinship, counting and the development of natural number, arithmetic learning, the concept of infinity, intuitive biology, natural kinds, and seriation. These critical cognitive achievements have been carefully studied empirically, and many are associated with multiple computational models. None to our knowledge, however, has a computational theory that fully accounts for all the known phenomena. We are not so naive as to think the child as hacker will provide such a model anytime soon. It provides a path, however, that if pursued diligently, will gradually lead us toward models which likely can live up to that vision.

The list functions domain also provides a tractable way to examine the core computational dynamics of several key milestones in children's cognitive development. As discussed in Chapters 3 and 6, some of the list functions we study already mimic basic tasks relevant to the development of counting, natural number, arithmetic, and seriation. Moreover, the way HL frames learning as the development of an LOT is a natural fit for the processes of conceptual change proposed to underlie many of these conceptual achievements (Carey, 2009; Barner & Baron, 2016). By systematically exploring different model LOTs and combinations of learning mechanisms in HL, we may be able to rapidly develop even more compelling models of the acquisition of the cardinal principle (Piantadosi et al., 2012; Carey, 2015), the successor principle (Cheung et al., 2017), early arithmetic (Shrager & Siegler, 1998), or seriation (Mareschal & Shultz, 1999; Schultz & Vogel, 2004; McGonigle-Chalmers & Kusel, 2019).

To help guide this sort of modeling work, we have begun empirical work to better characterize children's algorithmic abilities. For example, children's transitions between algorithms

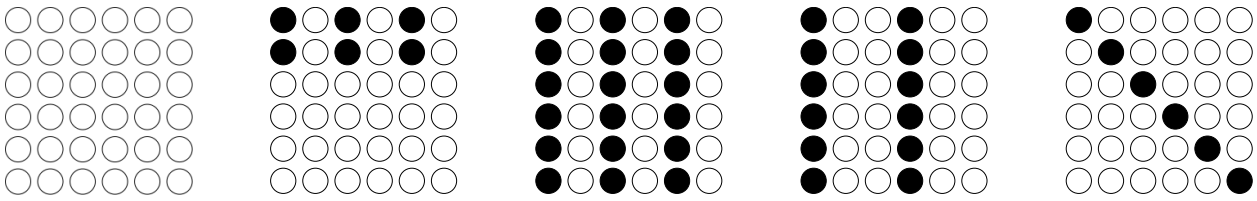


Figure 7-1: Othello paradigm: (a) the initially presented grid of binary chips; (b) a partial demonstration for (c); (c–e) three complete patterns.

in small number addition, as described in Chapter 2, are remarkable in that they change the *procedures* (the steps taken) without changing their *meaning* (the function computed). Recall, for instance, that both the *sum* and *min* algorithms generate correct answers, but children spontaneously transition between them. This suggests a general ability to refine algorithms according to criteria such as efficiency or robustness to error, and possibly even to reason about their behavior (e.g. determining patterns of output without running the procedure).

Our work is still in its formative stages, but we have started to look at how children infer simple procedures from examples and how they modify these procedures to make them better (e.g. more robust to noise, more efficient). In pilot work, children observed a uniform 6×6 grid of *Othello* chips (i.e. black on one side, white on the other; Figure 7-1a). After familiarization with the chips, children and an experimenter played a series of games in which children observed a partial demonstration of a deliberately inefficient procedure for constructing a pattern. Each partial demonstration processed two of the six rows in the grid (e.g. Figure 7-1b). Children observed demonstrations for each of: count and flip *every second* chip, count and flip *every third* chip, and count and flip *every seventh* chip. Critically, each algorithm generated a simple visual pattern that could be more quickly constructed by counting out just the first row and then directly flipping chips in the appropriate rows or diagonals (Figures 7-1c–e). Children then both predicted the appearance of the complete pattern and constructed the complete pattern in the experimenter's absence, being asked simply to "keep going" with the pattern.

Pilot results suggest that children can synthesize, analyze, and revise procedures for an

arbitrary domain of visual reasoning over patterns of Othello chips[2]. By age 6 or 7, they can revise these algorithms quickly, even from a short observation of an inefficient algorithm running. However, their skill depends on the complexity of the algorithm (e.g. every-other vs. every-seventh). We intend to more deeply investigate children's sensitivity to a broader range of semantic features of algorithms and the learning mechanisms they bring to bear during these kinds of tasks. In addition to their empirical value, these results can inform the creation of increasingly accurate and hacker-like models of learning.

7.7 Final thoughts

This thesis develops a metaphor in which hacking plays a critical role in explaining the representations, objectives, and mechanisms underlying human learning and cognitive development. It has translated core aspects of this metaphor into a computational theory of learning. This model attempts to go beyond standard approaches tying learning performance to description length in a mental lexicon and captures richer aspects of human cognition. It has also introduced a new domain for investigating inductive learning that captures more of the richness of humans' algorithmic abilities in a formally tractable way. These have been jointly assessed in a large scale investigation of human and machine learning. The results show that humans are deeply sensitive to structural cues of semantic content and that a model which takes a hacker-like approach to learning—iteratively exploring revisions to a programming language in ways that sometimes require entertaining hypotheses with known deficiencies—is better able to account for human performance than alternative models. These results support the conclusion that the child as hacker hypothesis can productively contribute to our understanding of learning. They also set the stage for a new generation of models framing learning as hacking in an effort to provide a computationally precise account of the richness and power of human learning.

[2]While developed independently, this domain is a simplified version of the ARC domain (Chollet, 2019) that uses a smaller number of elements. It also, like ARC, shares deep connections with list functions, since the grid can be modeled as a list of lists of 0s and 1s.

Appendix A

List Functions

The following table describes all 250 list functions used in Chapter 5, sorted according to mean human performance ($\boldsymbol{\mu}$). Also listed are the description length ($\boldsymbol{\mathcal{L}}$) of the program in the model LOT described in Table 5.2, the function's ID, a natural language description, the model LOT program, and the first five examples shown to participants. Some functions may appear twice if tested both with the numbers 0–99 and just with 0–9. The model comparison of Chapter 6 considered concepts c001–c100.

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.903	2	c102	*the input* `(λ x x)` [4, 72, 68, 63, 97] → [4, 72, 68, 63, 97] [79, 50, 92, 5, 8, 91, 27, 2, 43] → [79, 50, 92, 5, 8, 91, 27, 2, 43] [26, 86, 51] → [26, 86, 51] [0, 75, 58, 55] → [0, 75, 58, 55] [36, 57, 94, 1, 87, 38] → [36, 57, 94, 1, 87, 38]
0.874	12	c170	*remove all but element 1 and last element* `(λ x (cons (first x) (singleton (last x))))` [15, 4, 87, 8, 64, 14] → [15, 14] [90, 35, 8, 1, 5, 6, 21, 70, 48, 51] → [90, 51] [57, 74, 80, 40, 60, 25, 0, 52] → [57, 52] [44, 3, 19, 58, 50, 38, 29, 39, 2] → [44, 2] [56, 72, 9, 32, 7, 11, 30] → [56, 30]
0.873	6	c121	*remove all but last element* `(λ x (singleton (last x)))` [90, 80, 31, 14, 50] → [50] [11, 79, 83] → [83] [17, 59, 64, 22] → [22] [65, 43, 10, 73, 3, 51, 56, 8, 0, 2] → [2] [26, 24, 7, 85, 54, 52] → [52]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.868	2	c045	*the input* `(λ x x)` [1, 1, 2, 0] → [1, 1, 2, 0] [0] → [0] [8, 8] → [8, 8] [ ] → [ ] [5, 7, 9, 1, 3, 6, 4, 8, 2] → [5, 7, 9, 1, 3, 6, 4, 8, 2]
0.855	15	c072	*repeat every element 2 times in order of appearance* `(λ x (flatten (map (λ y (cons y (singleton y))) x)))` [1, 3, 3, 7] → [1, 1, 3, 3, 3, 3, 7, 7] [6, 9, 2, 8, 0, 5] → [6, 6, 9, 9, 2, 2, 8, 8, 0, 0, 5, 5] [9] → [9, 9] [4, 4, 4] → [4, 4, 4, 4, 4, 4] [5, 6, 4, 8, 9, 7, 3] → [5, 5, 6, 6, 4, 4, 8, 8, 9, 9, 7, 7, 3, 3]
0.853	13	c151	*repeat each element, M, M times, in order of appearance* `(λ x (flatten (map (λ y (repeat y y)) x)))` [2, 1, 3, 5, 0] → [2, 2, 1, 3, 3, 3, 5, 5, 5, 5, 5] [3, 4, 2, 0] → [3, 3, 3, 4, 4, 4, 4, 2, 2] [3] → [3, 3, 3] [0, 1, 1] → [1, 1] [1, 0] → [1]
0.841	4	c080	*elements in reverse order* `(λ x (reverse x))` [0, 5, 5, 5] → [5, 5, 5, 0] [6, 7, 9, 1, 4, 8, 2, 0, 2, 3] → [3, 2, 0, 2, 8, 4, 1, 9, 7, 6] [6, 3, 1] → [1, 3, 6] [0, 7] → [7, 0] [9, 5, 3, 0, 7, 4, 7, 1, 6] → [6, 1, 7, 4, 7, 0, 3, 5, 9]
0.836	6	c061	*remove all but the last element* `(λ x (singleton (last x)))` [9, 3, 2, 4] → [4] [1, 7, 5, 6, 9, 8] → [8] [4, 3] → [3] [2, 0, 1] → [1] [6, 6] → [6]
0.833	12	c189	*count from the smallest element to the largest element* `(λ x (range (min x) 1 (max x)))` [2, 7, 6, 8, 4] → [2, 3, 4, 5, 6, 7, 8] [69, 65, 65, 65] → [65, 66, 67, 68, 69] [98, 98, 98, 98] → [98] [10, 5, 10, 9, 4, 6, 4] → [4, 5, 6, 7, 8, 9, 10] [0, 0, 4, 3, 1, 5, 0, 1] → [0, 1, 2, 3, 4, 5]
0.823	8	c050	*prepend element 1* `(λ x (cons (first x) x))` [2, 4, 9, 3] → [2, 2, 4, 9, 3] [0, 4, 8, 4, 0] → [0, 0, 4, 8, 4, 0] [6, 6, 9, 7, 5, 9] → [6, 6, 6, 9, 7, 5, 9] [3, 7] → [3, 3, 7] [5] → [5, 5]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.818	6	c048	*remove all but element 1* `(λ x (take 1 x))` [6, 4, 7, 9] → [6] [4, 8, 6] → [4] [3, 3, 3] → [3] [2, 2] → [2] [1, 9, 9, 5, 5] → [1]
0.811	16	c147	*each element, followed by its original index* `(λ x (flatten (mapi (λ y (λ z (cons z (singleton y)))) x)))` [88, 93, 73, 54, 79] → [88, 1, 93, 2, 73, 3, 54, 4, 79, 5] [11, 0, 85, 98] → [11, 1, 0, 2, 85, 3, 98, 4] [62, 53, 21] → [62, 1, 53, 2, 21, 3] [90, 33] → [90, 1, 33, 2] [68, 49, 92, 75, 8, 17, 40] → [68, 1, 49, 2, 92, 3, 75, 4, 8, 5, 17, 6, 40, 7]
0.806	6	c120	*remove all but first element* `(λ x (singleton (first x)))` [74, 1, 93, 44, 5] → [74] [52, 27, 13, 3, 0, 60, 51, 80, 21] → [52] [19, 54] → [19] [46, 7, 84, 59, 89, 6, 2] → [46] [62, 4, 98, 65, 42, 22] → [62]
0.793	6	c127	*remove last element* `(λ x (droplast 1 x))` [74, 12, 59, 87, 7] → [74, 12, 59, 87] [9, 28, 91] → [9, 28] [30, 36, 65, 95, 2, 4, 23, 93, 6, 73] → [30, 36, 65, 95, 2, 4, 23, 93, 6] [45, 71, 78, 34, 3, 89, 67, 10, 96] → [45, 71, 78, 34, 3, 89, 67, 10] [90, 83, 81, 1, 58, 88] → [90, 83, 81, 1, 58]
0.792	4	c100	*reversed input* `(λ x (reverse x))` [31, 0, 51, 90] → [90, 51, 0, 31] [6, 1, 9, 13, 70, 66, 8, 40, 7] → [7, 40, 8, 66, 70, 13, 9, 1, 6] [5, 2] → [2, 5] [64, 64, 97] → [97, 64, 64] [75, 4, 7, 5, 33] → [33, 5, 7, 4, 75]
0.779	9	c145	*replace every element with element 1* `(λ x (map (λ y (first x)) x))` [45, 30, 33, 4, 64] → [45, 45, 45, 45, 45] [70, 43, 11, 75] → [70, 70, 70, 70] [51, 46, 52, 74, 5, 72, 9] → [51, 51, 51, 51, 51, 51, 51] [55, 22, 7, 94, 24, 60, 79, 97, 67] → [55, 55, 55, 55, 55, 55, 55, 55, 55] [54, 2, 10, 8, 6, 95] → [54, 54, 54, 54, 54, 54]
0.778	6	c079	*sum of elements* `(λ x (singleton (sum x)))` [0, 4, 1, 3] → [8] [5, 0] → [5] [1, 1, 7] → [9] [3, 3] → [6] [ ] → [0]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.777	20	c043	*the list [8, 2, 7, 0, 3]* `(λ x (cons 8 (cons 2 (cons 7 (cons 0 (singleton 3))))))` [2, 2, 2, 2] → [8, 2, 7, 0, 3] [5, 5, 5, 5, 5] → [8, 2, 7, 0, 3] [9, 3, 7, 8, 2, 1, 9, 0] → [8, 2, 7, 0, 3] [5, 0, 6, 4, 5, 5, 9, 6, 4, 1] → [8, 2, 7, 0, 3] [4, 1, 6, 4, 6, 1, 6, 3, 4] → [8, 2, 7, 0, 3]
0.773	6	c038	*append 9* `(λ x (append x 9))` [4, 2, 2, 2] → [4, 2, 2, 2, 9] [1, 0] → [1, 0, 9] [6] → [6, 9] [7, 9, 5] → [7, 9, 5, 9] [8, 6, 4, 5, 1, 9, 8, 3] → [8, 6, 4, 5, 1, 9, 8, 3, 9]
0.77	8	c042	*the list [5, 2]* `(λ x (cons 5 (singleton 2)))` [9, 3, 8, 0] → [5, 2] [1, 1, 0, 7, 7] → [5, 2] [6, 7, 3] → [5, 2] [9, 8] → [5, 2] [4, 4] → [5, 2]
0.766	23	c223	*swap the digits of each element* `(λ x (map (λ y (+ (* (% y 10) 10) (/ y 10))) x))` [4, 69, 95, 9, 49] → [40, 96, 59, 90, 94] [68, 99, 24] → [86, 99, 42] [54, 62, 9, 7, 32, 57, 45, 92, 47] → [45, 26, 90, 70, 23, 75, 54, 29, 74] [15, 17, 25, 19] → [51, 71, 52, 91] [18, 3, 6, 11, 58, 48] → [81, 30, 60, 11, 85, 84]
0.764	6	c137	*remove all occurrences of 3* `(λ x (cut_vals 3 x))` [8, 3, 1, 3, 3] → [8, 1] [73, 19, 51, 99, 67, 5, 47, 4, 3] → [73, 19, 51, 99, 67, 5, 47, 4] [5, 3, 34, 63, 38, 3] → [5, 34, 63, 38] [44, 66, 3, 46, 2, 6, 88, 75] → [44, 66, 46, 2, 6, 88, 75] [68, 76, 1, 3, 8, 12, 42, 0, 6, 18] → [68, 76, 1, 8, 12, 42, 0, 6, 18]
0.76	17	c238	*keep only elements that appear exactly once* `(λ x (filter (λ y (== 1 (count (== y) x))) x))` [8, 0, 5, 12, 0, 2] → [8, 5, 12, 2] [8, 19, 7, 8, 8, 8, 7, 7, 7, 7] → [19] [0, 1, 18, 9, 9, 0, 15, 6, 1] → [18, 15, 6] [0, 17, 4, 8, 4, 10, 1] → [0, 17, 8, 10, 1] [5, 3, 1, 6, 6, 3, 4, 4] → [5, 1]
0.758	6	c108	*sum of elements* `(λ x (singleton (sum x)))` [9, 6, 15, 3, 43] → [76] [ ] → [0] [12, 15, 7, 10, 8, 29] → [81] [39, 23, 34] → [96] [24, 46] → [70]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.758	6	c126	*remove element 1* `(λ x (drop 1 x))` [39, 52, 17, 56, 10] → [52, 17, 56, 10] [42, 5, 82, 8, 16, 19, 99, 3] → [5, 82, 8, 16, 19, 99, 3] [80, 0, 88, 30, 7, 15, 55] → [0, 88, 30, 7, 15, 55] [36, 73, 54] → [73, 54] [4, 24, 81, 64, 14, 70, 13, 32, 2, 9] → [24, 81, 64, 14, 70, 13, 32, 2, 9]
0.756	10	c187	*concatenate input with itself, separated by 0* `(λ x (concat x (cons 0 x)))` [83, 90, 11, 35, 5] → [83, 90, 11, 35, 5, 0, 83, 90, 11, 35, 5] [53, 73] → [53, 73, 0, 53, 73] [77, 7, 22] → [77, 7, 22, 0, 77, 7, 22] [89, 50, 2, 95] → [89, 50, 2, 95, 0, 89, 50, 2, 95] [16] → [16, 0, 16]
0.752	8	c022	*insert a 5 as element 2* `(λ x (insert 5 2 x))` [6, 5, 3, 3] → [6, 5, 5, 3, 3] [8, 4, 4, 4, 8, 4] → [8, 5, 4, 4, 4, 8, 4] [1, 1] → [1, 5, 1] [0, 2, 6] → [0, 5, 2, 6] [1, 9, 0, 9, 1] → [1, 5, 9, 0, 9, 1]
0.75	18	c212	*insert 3 at index 3, 3 times* `(λ x (splice (cons 3 (cons 3 (singleton 3))) 3 x))` [5, 9, 7, 80, 82] → [5, 9, 3, 3, 3, 7, 80, 82] [6, 54, 74, 26, 8, 95] → [6, 54, 3, 3, 3, 74, 26, 8, 95] [59, 96, 98, 25, 87, 86, 4] → [59, 96, 3, 3, 3, 98, 25, 87, 86, 4] [72, 15, 39] → [72, 15, 3, 3, 3, 39] [2, 65, 53, 68] → [2, 65, 3, 3, 3, 53, 68]
0.748	42	c101	*the list [11,19, 24, 33, 42, 5, 82, 0, 64, 9]* `(λ x (cons 11 (cons 19 (cons 24 (cons 33 (cons 42 (cons 5 (cons 82 (cons 0 (cons 64 (cons 9 empty))))))))))` [2, 67, 32, 46, 12] → [11, 19, 24, 33, 42, 5, 82, 0, 64, 9] [77, 51, 8, 27, 39, 7, 4, 92, 2, 71] → [11, 19, 24, 33, 42, 5, 82, 0, 64, 9] [36, 86, 78, 66, 6, 1, 70, 72] → [11, 19, 24, 33, 42, 5, 82, 0, 64, 9] [37, 47, 3, 74, 20, 20, 3] → [11, 19, 24, 33, 42, 5, 82, 0, 64, 9] [31, 80, 97, 98, 85, 60] → [11, 19, 24, 33, 42, 5, 82, 0, 64, 9]
0.744	8	c021	*insert an 8 as element 2* `(λ x (insert 8 2 x))` [1, 6, 9, 3] → [1, 8, 6, 9, 3] [7, 4] → [7, 8, 4] [0, 2, 3] → [0, 8, 2, 3] [7, 9, 1, 2, 6, 3, 0, 4, 7, 5] → [7, 8, 9, 1, 2, 6, 3, 0, 4, 7, 5] [0, 8, 6, 4, 0] → [0, 8, 8, 6, 4, 0]
0.744	16	c105	*concatenate all but the last element with all but the first element* `(λ x (splice (drop 1 (droplast 1 x)) 2 x))` [31, 40, 16, 44, 73] → [31, 40, 16, 44, 40, 16, 44, 73] [5, 1, 10, 24] → [5, 1, 10, 1, 10, 24] [26, 25, 85, 7, 94, 46] → [26, 25, 85, 7, 94, 25, 85, 7, 94, 46] [45, 62, 96, 4, 79, 57] → [45, 62, 96, 4, 79, 62, 96, 4, 79, 57] [6, 35, 75] → [6, 35, 35, 75]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.74	4	c041	*the list [9]* `(λ x (singleton 9))` [7, 3, 6, 4] → [9] [8, 7, 5, 5, 1, 6] → [9] [ ] → [9] [2, 2] → [9] [0] → [9]
0.738	6	c070	*concatenate input with itself* `(λ x (concat x x))` [3, 2, 0, 9] → [3, 2, 0, 9, 3, 2, 0, 9] [7, 1, 1, 2, 1, 2] → [7, 1, 1, 2, 1, 2, 7, 1, 1, 2, 1, 2] [7] → [7, 7] [0, 8, 4, 3, 6, 8, 4] → [0, 8, 4, 3, 6, 8, 4, 0, 8, 4, 3, 6, 8, 4] [5, 5] → [5, 5, 5, 5]
0.736	8	c052	*repeat element 1 ten times* `(λ x (repeat (first x) 10))` [9, 8, 7, 1] → [9, 9, 9, 9, 9, 9, 9, 9, 9, 9] [2, 5, 5] → [2, 2, 2, 2, 2, 2, 2, 2, 2, 2] [4, 0, 6, 7, 3, 5, 1, 6, 3] → [4, 4, 4, 4, 4, 4, 4, 4, 4, 4] [6, 7, 1] → [6, 6, 6, 6, 6, 6, 6, 6, 6, 6] [0] → [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
0.723	40	c044	*the list [1, 9, 4, 3, 2, 5, 8, 0, 4, 9]* `(λ x (cons 1 (cons 9 (cons 4 (cons 3 (cons 2 (cons 5 (cons 8 (cons 0 (cons 4 (singleton 9)))))))))))` [2, 7, 7, 3] → [1, 9, 4, 3, 2, 5, 8, 0, 4, 9] [6, 6, 6] → [1, 9, 4, 3, 2, 5, 8, 0, 4, 9] [0, 0, 1, 6, 1, 6, 1] → [1, 9, 4, 3, 2, 5, 8, 0, 4, 9] [ ] → [1, 9, 4, 3, 2, 5, 8, 0, 4, 9] [7] → [1, 9, 4, 3, 2, 5, 8, 0, 4, 9]
0.717	12	c190	*count by 2 from the first element to the last element* `(λ x (range (first x) 2 (last x)))` [46, 19, 96, 58, 58] → [46, 48, 50, 52, 54, 56, 58] [2, 8, 1, 14, 7, 4, 61, 0, 6] → [2, 4, 6] [66, 3, 6, 46, 82, 88, 17, 1, 93, 76] → [66, 68, 70, 72, 74, 76] [8, 92, 0, 2, 94, 41, 14] → [8, 10, 12, 14] [24, 4, 32] → [24, 26, 28, 30, 32]
0.717	6	c037	*append 3* `(λ x (append x 3))` [2, 0, 6, 0] → [2, 0, 6, 0, 3] [9, 9] → [9, 9, 3] [3, 1, 7] → [3, 1, 7, 3] [4, 4, 5, 5, 4, 4, 5, 4] → [4, 4, 5, 5, 4, 4, 5, 4, 3] [2, 6, 9, 6, 7, 1] → [2, 6, 9, 6, 7, 1, 3]
0.716	9	c222	*replace each element with the input length* `(λ x (map (λ y (length x)) x))` [34, 83, 11, 82, 31] → [5, 5, 5, 5, 5] [74, 59, 14, 50] → [4, 4, 4, 4] [25] → [1] [58, 80] → [2, 2] [22, 92, 28] → [3, 3, 3]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.714	4	c107	*unique elements* `(λ x (unique x))` [87, 87, 17, 17, 17, 87] → [87, 17] [3, 92, 18, 6, 49, 49, 1, 38, 80] → [3, 92, 18, 6, 49, 1, 38, 80] [68, 68, 68, 68, 68, 68, 68, 68, 68, 68] → [68] [89, 89, 89, 36, 55, 14, 7, 14] → [89, 36, 55, 14, 7] [81, 69, 85, 81, 69, 74, 0, 24, 74, 61] → [81, 69, 85, 74, 0, 24, 61]
0.713	8	c011	*elements 2 through 4* `(λ x (slice 2 4 x))` [6, 1, 3, 0, 4, 9] → [1, 3, 0] [7, 2, 4, 2, 4, 7, 2, 2, 4] → [2, 4, 2] [6, 8, 6, 9, 9, 8, 8, 6] → [8, 6, 9] [1, 1, 4, 0, 3, 1, 3, 5, 0] → [1, 4, 0] [5, 8, 5, 8, 3, 8, 7, 3, 7] → [8, 5, 8]
0.712	6	c104	*maximum element* `(λ x (singleton (max x)))` [5, 26, 31, 76, 89] → [89] [9, 42, 54, 18, 83, 7, 34] → [83] [4, 24, 58, 93, 28, 60, 2, 0, 22, 8] → [93] [48, 47] → [48] [63, 46, 6] → [63]
0.706	11	c192	*replace each element, M, with its tens digit* `(λ x (map (λ y (/ y 10)) x))` [31, 14, 3, 18, 32] → [3, 1, 0, 1, 3] [61, 40, 77, 2] → [6, 4, 7, 0] [92, 47, 62] → [9, 4, 6] [13] → [1] [45, 58] → [4, 5]
0.706	16	c182	*keep every third element* `(λ x (filteri (λ y (λ z (== (% y 3) 0))) x))` [64, 6, 85, 21, 47, 46, 60, 4, 7] → [85, 46, 7] [56, 12, 39, 29, 71, 2, 1, 55, 9, 93] → [39, 2, 9] [20, 92, 79, 36, 7, 34, 26, 25, 1] → [79, 34, 1] [41, 67, 38, 84, 14, 80, 99, 91, 23, 8] → [38, 80, 23] [42, 77, 3, 57, 5, 0, 79, 13, 83, 4] → [3, 0, 83]
0.701	6	c006	*the first 2 elements* `(λ x (take 2 x))` [7, 8, 5, 7] → [7, 8] [3, 1, 1, 1, 3, 1, 1] → [3, 1] [8, 0, 8, 7, 4, 0, 4] → [8, 0] [2, 0, 4, 6, 5] → [2, 0] [9, 9, 9] → [9, 9]
0.701	6	c046	*prepend 7* `(λ x (cons 7 x))` [8, 8, 5, 5] → [7, 8, 8, 5, 5] [ ] → [7] [9] → [7, 9] [1, 4] → [7, 1, 4] [3, 8, 6, 7, 0, 3, 4] → [7, 3, 8, 6, 7, 0, 3, 4]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.7	7	c106	*elements in ascending order* `(λ x (sort (λ y y) x))` [36, 86, 2, 97, 10] → [2, 10, 36, 86, 97] [50, 94, 0, 83, 77, 71, 5, 3, 57, 8] → [0, 3, 5, 8, 50, 57, 71, 77, 83, 94] [85, 26, 7, 23, 48, 39] → [7, 23, 26, 39, 48, 85] [89, 4, 21, 35, 78, 96, 11, 90, 47] → [4, 11, 21, 35, 47, 78, 89, 90, 96] [68, 1, 44, 93] → [1, 44, 68, 93]
0.699	8	c114	*prepend the last element* `(λ x (cons (last x) x))` [86, 84, 60, 20, 21] → [21, 86, 84, 60, 20, 21] [10, 4, 51, 57] → [57, 10, 4, 51, 57] [67, 72] → [72, 67, 72] [35, 49, 2, 45, 46, 92, 9, 6, 58] → [58, 35, 49, 2, 45, 46, 92, 9, 6, 58] [55, 5, 56] → [56, 55, 5, 56]
0.694	8	c093	*repeat element 1 ten times* `(λ x (repeat (first x) 10))` [94, 36, 57, 91] → [94, 94, 94, 94, 94, 94, 94, 94, 94, 94] [93, 7, 37, 90, 0, 99, 6, 6] → [93, 93, 93, 93, 93, 93, 93, 93, 93, 93] [62] → [62, 62, 62, 62, 62, 62, 62, 62, 62, 62] [19, 5, 2, 76, 61, 5, 43, 1, 20] → [19, 19, 19, 19, 19, 19, 19, 19, 19, 19] [4, 63, 0, 58, 61, 9, 0, 1, 85, 8] → [4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
0.693	32	c195	*element 1, followed by 23, 68, 42, 99, 71, followed by last element* `(λ x (cons (first x) (cons 23 (cons 68 (cons 42 (cons 99 (cons 71 (singleton (last x))))))))))` [1, 72, 24, 78, 31] → [1, 23, 68, 42, 99, 71, 31] [6, 61, 57, 53, 51, 95, 26] → [6, 23, 68, 42, 99, 71, 26] [79, 7, 54, 1, 38, 84] → [79, 23, 68, 42, 99, 71, 84] [64, 41, 89, 75, 63, 40, 2, 43, 21, 9] → [64, 23, 68, 42, 99, 71, 9] [56, 55, 17, 46, 94, 50, 29, 5, 0] → [56, 23, 68, 42, 99, 71, 0]
0.693	30	c196	*concatenate [17, 38, 82], input, and [1, 55, 27]* `(λ x (concat (cons 17 (cons 38 (singleton 82))) (concat x (cons 1 (cons 55 (singleton 27))))))` [90, 0, 19, 94, 8] → [17, 38, 82, 90, 0, 19, 94, 8, 1, 55, 27] [24, 49, 53] → [17, 38, 82, 24, 49, 53, 1, 55, 27] [86, 77] → [17, 38, 82, 86, 77, 1, 55, 27] [51, 52, 91, 9] → [17, 38, 82, 51, 52, 91, 9, 1, 55, 27] [15] → [17, 38, 82, 15, 1, 55, 27]
0.692	10	c095	*remove the first and last elements* `(λ x (drop 1 (droplast 1 x)))` [8, 97, 65, 9, 54, 97] → [97, 65, 9, 54] [16, 51, 51, 16, 16, 0, 0, 85, 51, 9] → [51, 51, 16, 16, 0, 0, 85, 51] [6, 21, 6, 59, 6, 59, 6, 12, 12] → [21, 6, 59, 6, 59, 6, 12] [56, 39, 5, 5, 2, 24, 24] → [39, 5, 5, 2, 24] [46, 46, 46, 46, 32, 32, 32, 46] → [46, 46, 46, 32, 32, 32]
0.689	8	c016	*replace element 2 with an 8* `(λ x (replace 2 8 x))` [1, 1, 0, 2] → [1, 8, 0, 2] [6, 1, 2, 4, 3, 5, 3, 9, 0, 7] → [6, 8, 2, 4, 3, 5, 3, 9, 0, 7] [5, 5, 5] → [5, 8, 5] [8, 1, 9, 6, 0, 7, 5] → [8, 8, 9, 6, 0, 7, 5] [9, 2] → [9, 8]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.687	16	c224	*the last element, followed by element 1, followed by the second to last element, followed by element 2, and so on* `(λ x (fold (λ y (λ z (cons z (reverse y)))) empty x))` [80, 31, 6, 69, 38] → [38, 6, 80, 31, 69] [29, 17, 49, 99, 41, 93, 0, 2, 5, 3] → [3, 2, 93, 99, 17, 29, 49, 41, 0, 5] [68, 4, 34, 17, 24, 85, 82, 7, 52] → [52, 82, 24, 34, 68, 4, 17, 85, 7] [87, 73, 92, 8] → [8, 73, 87, 92] [10, 65, 16, 45, 97, 22, 30] → [30, 97, 16, 10, 65, 45, 22]
0.682	12	c116	*reflect the input on itself* `(λ x (concat (reverse (drop 1 x)) x))` [52, 72, 4, 18, 70] → [70, 18, 4, 72, 52, 72, 4, 18, 70] [48, 47, 27] → [27, 47, 48, 47, 27] [67, 23, 25, 54] → [54, 25, 23, 67, 23, 25, 54] [31, 2, 68, 11, 5, 65, 81, 28] → [28, 81, 65, 5, 11, 68, 2, 31, 2, 68, 11, 5, 65, 81, 28] [64, 66] → [66, 64, 66]
0.682	8	c140	*replace element 2 with a 9* `(λ x (replace 2 9 x))` [75, 78, 54, 76, 56] → [75, 9, 54, 76, 56] [35, 24, 0, 8, 51, 42, 60, 20, 4] → [35, 9, 0, 8, 51, 42, 60, 20, 4] [16, 31, 77, 74, 38, 23] → [16, 9, 77, 74, 38, 23] [7, 2, 0, 6, 67, 64, 5, 30, 95, 70] → [7, 9, 0, 6, 67, 64, 5, 30, 95, 70] [25, 48, 96, 89] → [25, 9, 96, 89]
0.679	6	c049	*remove element 1* `(λ x (drop 1 x))` [3, 3, 3, 3] → [3, 3, 3] [7, 1, 4, 1, 0, 8] → [1, 4, 1, 0, 8] [1, 0, 9, 0, 2] → [0, 9, 0, 2] [4, 9, 7, 6, 6, 4, 5] → [9, 7, 6, 6, 4, 5] [2] → [ ]
0.678	11	c071	*add 2 to every element* `(λ x (map (λ y (+ 2 y)) x))` [0, 0, 7, 0] → [2, 2, 9, 2] [6, 7, 7, 6, 1, 4, 2, 6, 5] → [8, 9, 9, 8, 3, 6, 4, 8, 7] [6] → [8] [5, 5] → [7, 7] [1, 4, 3, 6, 0] → [3, 6, 5, 8, 2]
0.678	22	c161	*replace each element, M, with M + the input length - M's index* `(λ x (mapi (λ y (λ z (+ z (- (length x) y)))) x))` [75, 25, 38, 55, 91, 26] → [80, 29, 41, 57, 92, 26] [5, 30, 2, 9, 3, 19, 92, 15] → [12, 36, 7, 13, 6, 21, 93, 15] [38, 10, 66, 49, 50, 8, 61, 59, 64] → [46, 17, 72, 54, 54, 11, 63, 60, 64] [11, 19, 0, 31, 40, 16, 78] → [17, 24, 4, 34, 42, 17, 78] [89, 4, 7, 8, 82, 3, 9, 45, 38, 94] → [98, 12, 14, 14, 87, 7, 12, 47, 39, 94]
0.675	24	c068	*concatenate input and [7, 3, 8, 4, 3]* `(λ x (concat x (cons 7 (cons 3 (cons 8 (cons 4 (singleton 3)))))))` [8, 0, 8, 0] → [8, 0, 8, 0, 7, 3, 8, 4, 3] [5, 5, 4, 7, 4, 7, 5, 4] → [5, 5, 4, 7, 4, 7, 5, 4, 7, 3, 8, 4, 3] [0] → [0, 7, 3, 8, 4, 3] [ ] → [7, 3, 8, 4, 3] [6, 2, 1, 6, 2, 1] → [6, 2, 1, 6, 2, 1, 7, 3, 8, 4, 3]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.675	6	c103	*input length* `(λ x (singleton (length x)))` [38, 51, 18, 72, 13] → [5] [ ] → [0] [83] → [1] [73, 91, 96, 60, 61, 42, 2, 6, 33] → [9] [90, 93, 81, 3, 57, 69, 21] → [7]
0.673	6	c001	*remove all but element 3* `(λ x (singleton (third x)))` [2, 4, 3, 2] → [3] [9, 6, 9, 8, 6] → [9] [0, 0, 0, 0, 0] → [0] [8, 1, 8] → [8] [5, 7, 5, 7, 5] → [5]
0.67	20	c090	*the list [18, 42, 77, 20, 36]* `(λ x (cons 18 (cons 42 (cons 77 (cons 20 (singleton 36))))))` [79, 85, 85, 85] → [18, 42, 77, 20, 36] [33, 33] → [18, 42, 77, 20, 36] [92, 2, 2, 7, 9, 84, 52, 5, 6, 41] → [18, 42, 77, 20, 36] [2, 34, 96, 49, 83, 41, 35, 4, 39, 97] → [18, 42, 77, 20, 36] [89, 68, 4, 3, 68, 76, 80, 6] → [18, 42, 77, 20, 36]
0.667	38	c097	*concatenate [11, 21, 43, 19], input, and [7, 89, 0, 57]* `(λ x (concat (cons 11 (cons 21 (cons 43 (singleton 19)))) (concat x (cons 7 (cons 89 (cons 0 (singleton 57)))))))` [6, 59, 33, 33] → [11, 21, 43, 19, 6, 59, 33, 33, 7, 89, 0, 57] [2, 87, 9, 99, 62, 4] → [11, 21, 43, 19, 2, 87, 9, 99, 62, 4, 7, 89, 0, 57] [39, 87] → [11, 21, 43, 19, 39, 87, 7, 89, 0, 57] [91] → [11, 21, 43, 19, 91, 7, 89, 0, 57] [ ] → [11, 21, 43, 19, 7, 89, 0, 57]
0.665	23	c142	*every digit in order of appearance* `(λ x (flatten (map (λ y (cons (/ y 10) (singleton (% y 10)))) x)))` [77, 63, 83, 97, 58] → [7, 7, 6, 3, 8, 3, 9, 7, 5, 8] [47, 54] → [4, 7, 5, 4] [95, 60, 15, 52] → [9, 5, 6, 0, 1, 5, 5, 2] [33] → [3, 3] [70, 2, 92, 13, 22, 19] → [7, 0, 0, 2, 9, 2, 1, 3, 2, 2, 1, 9]
0.661	40	c091	*the list [81, 99, 41, 23, 22, 75, 68, 30, 24, 69]* `(λ x (cons 81 (cons 99 (cons 41 (cons 23 (cons 22 (cons 75 (cons 68 (cons 30 (cons 24 (singleton 69)))))))))))` [3, 88, 88] → [81, 99, 41, 23, 22, 75, 68, 30, 24, 69] [ ] → [81, 99, 41, 23, 22, 75, 68, 30, 24, 69] [6] → [81, 99, 41, 23, 22, 75, 68, 30, 24, 69] [61, 1, 59, 4, 5, 35, 48, 27, 9] → [81, 99, 41, 23, 22, 75, 68, 30, 24, 69] [72, 58, 56, 49, 40, 7, 25, 1] → [81, 99, 41, 23, 22, 75, 68, 30, 24, 69]
0.659	8	c034	*swap elements 2 and 3* `(λ x (swap 2 3 x))` [1, 9, 1, 4] → [1, 1, 9, 4] [7, 1, 8, 6, 3, 5, 9, 4] → [7, 8, 1, 6, 3, 5, 9, 4] [6, 1, 2, 7, 4] → [6, 2, 1, 7, 4] [9, 0, 2] → [9, 2, 0] [3, 1, 4, 2, 6, 5, 9, 7, 0, 8] → [3, 4, 1, 2, 6, 5, 9, 7, 0, 8]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.657	13	c244	*number of 3s* `(λ x (singleton (count (λ y (== 3 y)) x)))` [52, 3, 3, 3, 3, 3, 52] → [5] [3, 3, 3, 3, 3, 14, 14, 3, 14] → [6] [28, 79, 1, 3, 55, 42, 70, 60, 7, 67] → [1] [92, 11, 94, 7, 2, 1, 18, 8, 89, 5] → [0] [3, 43, 3, 3, 27, 3, 7, 69] → [4]
0.655	26	c171	*cumulative sum of elements* `(λ x (drop 1 (fold (λ y (λ z (append y (+ (last y) z)))) (singleton 0) x)))` [2, 9, 17, 9, 17, 4] → [2, 11, 28, 37, 54, 58] [7, 7, 8, 3, 4, 4, 5, 0] → [7, 14, 22, 25, 29, 33, 38, 38] [5, 7, 2, 6, 1, 0, 9] → [5, 12, 14, 20, 21, 21, 30] [5, 0, 4, 15, 5, 7, 6, 15, 2, 7] → [5, 5, 9, 24, 29, 36, 42, 57, 59, 66] [7, 7, 2, 7, 11, 0, 16, 16, 7] → [7, 14, 16, 23, 34, 34, 50, 66, 73]
0.651	26	c172	*cumulative product of elements* `(λ x (drop 1 (fold (λ y (λ z (append y (* (last y) z)))) (singleton 1) x)))` [4, 2, 2, 2] → [4, 8, 16, 32] [5, 2, 7] → [5, 10, 70] [4, 1, 1, 4, 1, 4, 1] → [4, 4, 4, 16, 16, 64, 64] [7, 9] → [7, 63] [2, 2, 2, 3, 3, 1] → [2, 4, 8, 24, 72, 72]
0.65	6	c062	*remove the last element* `(λ x (droplast 1 x))` [2, 5, 2, 7] → [2, 5, 2] [8] → [ ] [7, 6, 0, 7, 3] → [7, 6, 0, 7] [9, 9] → [9] [1, 3, 8, 5, 7, 6, 0, 9, 2, 4] → [1, 3, 8, 5, 7, 6, 0, 9, 2]
0.647	10	c096	*prepend 98 and append 37* `(λ x (cons 98 (append x 37)))` [20, 70, 38, 80] → [98, 20, 70, 38, 80, 37] [3, 3] → [98, 3, 3, 37] [ ] → [98, 37] [8, 8, 1, 89, 85, 7, 49] → [98, 8, 8, 1, 89, 85, 7, 49, 37] [6] → [98, 6, 37]
0.644	6	c081	*remove all but element 3* `(λ x (singleton (third x)))` [40, 50, 76, 47, 39] → [76] [9, 81, 6, 81, 6] → [6] [9, 91, 70, 48, 59, 83, 43] → [70] [27, 4, 38, 83, 5, 3, 15, 4, 5, 83] → [38] [45, 45, 45] → [45]
0.641	6	c109	*product of elements* `(λ x (singleton (product x)))` [5, 1, 1, 1, 3] → [15] [6, 6] → [36] [ ] → [1] [4, 3, 2, 3] → [72] [7, 14, 1] → [98]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.636	18	c002	*remove all but element 3* `(λ x (if (> 3 (length x)) empty (singleton (third x))))` [0, 5] → [ ] [5, 6, 1, 3, 2, 0, 7, 8, 9, 4] → [1] [6, 7, 8, 1, 4, 3, 0, 5, 9, 2] → [8] [7, 4, 3, 9, 5, 8, 2, 1, 6] → [3] [4, 6] → [ ]
0.636	6	c132	*remove element 3* `(λ x (cut_idx 3 x))` [8, 0, 3, 56, 95] → [8, 0, 56, 95] [93, 5, 51, 24, 11, 7, 44] → [93, 5, 24, 11, 7, 44] [27, 20, 19, 63, 13, 64] → [27, 20, 63, 13, 64] [15, 57, 70, 35] → [15, 57, 35] [1, 43, 23, 65, 4, 6, 28, 2, 10, 40] → [1, 43, 65, 4, 6, 28, 2, 10, 40]
0.636	24	c153	*each unique element followed by its number of occurrences, in order of appearance* `(λ x (flatten (map (λ y (append (take 1 y) (length y))) (group (λ z z) x))))` [23, 23, 23, 27, 27] → [23, 3, 27, 2] [1, 95, 95, 41, 41, 1, 35, 95, 35] → [1, 2, 95, 3, 41, 2, 35, 2] [39, 39] → [39, 2] [0, 0, 97, 97, 25, 25] → [0, 2, 97, 2, 25, 2] [78, 99, 50, 5, 90, 5, 78, 90, 50, 99] → [78, 2, 99, 2, 50, 2, 5, 2, 90, 2]
0.63	12	c051	*prepend element 1 five times* `(λ x (concat (repeat (first x) 5) x))` [9, 9, 9, 9] → [9, 9, 9, 9, 9, 9, 9, 9, 9] [4, 1] → [4, 4, 4, 4, 4, 4, 1] [5, 7, 0, 7, 3, 5, 8] → [5, 5, 5, 5, 5, 5, 7, 0, 7, 3, 5, 8] [8] → [8, 8, 8, 8, 8, 8] [2, 4, 5, 1, 3, 6, 8, 0] → [2, 2, 2, 2, 2, 2, 4, 5, 1, 3, 6, 8, 0]
0.63	10	c225	*remove first and last two elements* `(λ x (drop 2 (droplast 2 x)))` [8, 28, 97, 66, 46] → [97] [53, 95, 39, 49, 62, 74, 5, 4] → [39, 49, 62, 74] [11, 87, 44, 41, 6, 27] → [44, 41] [2, 40, 29, 81, 54, 48, 76, 15, 8, 80] → [29, 81, 54, 48, 76, 15] [13, 38, 91, 64, 16, 0, 5] → [91, 64, 16]
0.629	22	c067	*swap the first and last elements* `(λ x (cons (last x) (append (drop 1 (droplast 1 x)) (first x))))` [4, 8, 9, 9] → [9, 8, 9, 4] [5, 0, 7, 6, 6, 0, 6, 0] → [0, 0, 7, 6, 6, 0, 6, 5] [4, 7] → [7, 4] [2, 1, 6, 3, 4, 0, 9, 8, 7, 4] → [4, 1, 6, 3, 4, 0, 9, 8, 7, 2] [1, 3, 2, 8, 8, 5, 5] → [5, 3, 2, 8, 8, 5, 1]
0.626	10	c112	*count from 1 to the last element* `(λ x (range 1 1 (last x)))` [25, 0, 22, 48, 7] → [1, 2, 3, 4, 5, 6, 7] [2, 66, 71, 42, 29, 99, 95, 81, 19, 3] → [1, 2, 3] [5, 26, 75, 4, 97, 32, 73, 59, 1] → [1] [9, 10] → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] [76, 61, 15, 74, 77, 6, 2] → [1, 2]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.624	6	c030	*remove the last two elements* `(λ x (droplast 2 x))` [6, 4, 8, 1, 0] → [6, 4, 8] [9, 7, 4, 3, 8, 1] → [9, 7, 4, 3] [2, 5, 7, 8, 2, 2, 9, 3, 1, 0] → [2, 5, 7, 8, 2, 2, 9, 3] [5, 4, 5, 0] → [5, 4] [6, 2, 9, 7, 8, 3, 8, 5, 0] → [6, 2, 9, 7, 8, 3, 8]
0.622	17	c149	*replace each element, M, with M * element 1; remove element 1* `(λ x (map (λ y (* y (first x))) (drop 1 x)))` [5, 2, 13, 6, 3] → [10, 65, 30, 15] [3, 33] → [99] [9] → [ ] [8, 2, 6, 9] → [16, 48, 72] [6, 3, 1] → [18, 6]
0.615	12	c148	*count from 1 to each original element, in order of appearance* `(λ x (flatten (map (range 1 1) x)))` [2, 5, 1, 4, 1] → [1, 2, 1, 2, 3, 4, 5, 1, 1, 2, 3, 4, 1] [3, 3, 2] → [1, 2, 3, 1, 2, 3, 1, 2] [5, 4] → [1, 2, 3, 4, 5, 1, 2, 3, 4] [3, 0, 5, 2] → [1, 2, 3, 1, 2, 3, 4, 5, 1, 2] [5] → [1, 2, 3, 4, 5]
0.609	8	c017	*replace element 2 with an 8 if there is an element 2* `(λ x (replace 2 8 x))` [9, 1, 7, 7] → [9, 8, 7, 7] [9, 2, 1, 6, 4, 0] → [9, 8, 1, 6, 4, 0] [0] → [0] [1, 4, 2, 5, 3, 9, 7, 2] → [1, 8, 2, 5, 3, 9, 7, 2] [0, 2, 2] → [0, 8, 2]
0.604	22	c092	*concatenate [92, 63, 34, 18, 55] with input* `(λ x (cons 92 (cons 63 (cons 34 (cons 18 (cons 55 x))))))` [97, 45, 97, 8] → [92, 63, 34, 18, 55, 97, 45, 97, 8] [7, 87, 87, 87, 5, 11] → [92, 63, 34, 18, 55, 7, 87, 87, 87, 5, 11] [8, 54, 84, 7, 9, 94, 3, 40, 6] → [92, 63, 34, 18, 55, 8, 54, 84, 7, 9, 94, 3, 40, 6] [1] → [92, 63, 34, 18, 55, 1] [66, 66] → [92, 63, 34, 18, 55, 66, 66]
0.601	6	c077	*input length* `(λ x (singleton (length x)))` [1, 7, 2, 0] → [4] [8, 6, 6] → [3] [2] → [1] [8, 3, 9, 5, 7] → [5] [ ] → [0]
0.6	6	c098	*add the index to every element* `(λ x (mapi + x))` [40, 52, 40, 50] → [41, 54, 43, 54] [0, 0, 8, 8, 8, 8] → [1, 2, 11, 12, 13, 14] [62, 4, 8, 85, 68, 9, 62, 85] → [63, 6, 11, 89, 73, 15, 69, 93] [18, 5, 79, 21, 0, 47, 91] → [19, 7, 82, 25, 5, 53, 98] [87, 56, 7, 56, 72, 33, 36, 57, 87, 7] → [88, 58, 10, 60, 77, 39, 43, 65, 96, 17]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.6	42	c248	*elements after the last 0*

```
(λ x (first (fold (λ y (λ z (if (== z 0) (cons empty y) (cons (append (first y) z) (drop 1 y))))) (singleton empty) x)))
```

[9, 0, 4, 8, 0, 2, 46, 96, 30] → [2, 46, 96, 30]
[97, 0, 46, 8, 89, 0, 17, 3, 17, 7] → [17, 3, 17, 7]
[0, 1, 5, 1, 32, 5, 41, 5, 0, 87] → [87]
[70, 6, 11, 0, 37, 9, 9, 0, 13, 54] → [13, 54]
[86, 7, 99, 0, 99, 0, 3, 4, 38] → [3, 4, 38]

0.599	21	c199	*elements in ascending order, each preceded by its rank*

```
(λ x (flatten (zip (range 1 1 (length x)) (sort (λ y y) x))))
```

[59, 22, 86, 64, 25] → [1, 22, 2, 25, 3, 59, 4, 64, 5, 86]
[72, 69, 74, 27] → [1, 27, 2, 69, 3, 72, 4, 74]
[6, 99, 46, 0, 96, 49, 77] → [1, 0, 2, 6, 3, 46, 4, 49, 5, 77, 6, 96, 7, 99]
[12, 81] → [1, 12, 2, 81]
[42] → [1, 42]

0.597	6	c007	*remove all but the first 2 elements*

```
(λ x (take 2 x))
```

[0, 2, 2, 0] → [0, 2]
[ ] → [ ]
[1] → [1]
[3, 3, 1, 9, 8] → [3, 3]
[7, 8, 5] → [7, 8]

0.591	21	c176	*sums of each consecutive pair of elements, in order of appearance*

```
(λ x (map (λ y (sum y)) (zip (droplast 1 x) (drop 1 x))))
```

[22, 1, 6, 8, 51, 26] → [23, 7, 14, 59, 77]
[8, 2, 65, 9, 81, 16, 79, 3, 80, 5] → [10, 67, 74, 90, 97, 95, 82, 83, 85]
[8, 5, 28, 36, 58, 40, 0] → [13, 33, 64, 94, 98, 40]
[31, 29, 3, 19, 5, 50, 0, 76] → [60, 32, 22, 24, 55, 50, 76]
[20, 71, 5, 1, 38, 4, 93, 2, 50] → [91, 76, 6, 39, 42, 97, 95, 52]

0.588	12	c066	*left-rotate by 1*

```
(λ x (append (drop 1 x) (first x)))
```

[5, 6, 5, 8] → [6, 5, 8, 5]
[1, 6, 6] → [6, 6, 1]
[8, 2, 4, 7, 3, 0] → [2, 4, 7, 3, 0, 8]
[7, 9, 8, 2, 5, 1, 2, 4] → [9, 8, 2, 5, 1, 2, 4, 7]
[3, 7] → [7, 3]

0.587	16	c053	*replace element 2 with element 1*

```
(λ x (concat (repeat (first x) 2) (drop 2 x)))
```

[8, 9, 6, 4] → [8, 8, 6, 4]
[6, 5, 8, 9, 1, 3, 4, 1, 0] → [6, 6, 8, 9, 1, 3, 4, 1, 0]
[9, 3, 7, 0, 1, 5, 5, 0] → [9, 9, 7, 0, 1, 5, 5, 0]
[5] → [5, 5]
[2, 0, 2, 0, 2] → [2, 2, 2, 0, 2]

0.584	22	c047	*concatenate [9, 6, 3, 8, 5] and input*

```
(λ x (cons 9 (cons 6 (cons 3 (cons 8 (cons 5 x))))))
```

[8, 3, 7, 9] → [9, 6, 3, 8, 5, 8, 3, 7, 9]
[4, 6, 7, 0, 7, 7, 1, 9] → [9, 6, 3, 8, 5, 4, 6, 7, 0, 7, 7, 1, 9]
[6] → [9, 6, 3, 8, 5, 6]
[ ] → [9, 6, 3, 8, 5]
[5, 2, 0, 2, 4, 0, 3, 2, 4] → [9, 6, 3, 8, 5, 5, 2, 0, 2, 4, 0, 3, 2, 4]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.584	10	c064	*remove the first and last elements* `(λ x (drop 1 (droplast 1 x)))` [4, 5, 0, 0] → [5, 0] [3, 8, 3, 8, 3] → [8, 3, 8] [4, 9, 4, 9, 4, 9, 9, 9, 9, 9] → [9, 4, 9, 4, 9, 9, 9, 9] [5, 7, 7, 9, 8, 1, 4, 0, 6] → [7, 7, 9, 8, 1, 4, 0] [2, 1, 1, 2, 2, 7, 2, 7] → [1, 1, 2, 2, 7, 2]
0.584	6	c078	*maximum element* `(λ x (singleton (max x)))` [2, 3, 0, 1] → [3] [2, 7, 9, 5, 4, 0, 8, 1, 3, 6] → [9] [1, 0] → [1] [7, 1, 5, 4, 0, 2, 8, 6] → [8] [2, 5, 3, 0, 6, 4, 1] → [6]
0.584	32	c119	*prepend or append 8, if necessary, so the list begins and ends with 8* `(λ x ((λ y (concat (y first) (concat x (y last)))) (λ z (if (== (z x) 8) empty (singleton 8)))))` [8, 87, 23, 25, 34] → [8, 87, 23, 25, 34, 8] [63] → [8, 63, 8] [47, 68, 84, 8] → [8, 47, 68, 84, 8] [46, 77] → [8, 46, 77, 8] [52, 7, 2, 91, 5, 75, 58, 3, 8] → [8, 52, 7, 2, 91, 5, 75, 58, 3, 8]
0.573	6	c135	*remove the first 7* `(λ x (cut_val 7 x))` [7, 99, 63, 7, 7] → [99, 63, 7, 7] [7, 7] → [7] [47, 14, 45, 7, 6, 59, 48, 50, 5, 19] → [47, 14, 45, 6, 59, 48, 50, 5, 19] [8, 38, 3, 42, 7, 78, 71] → [8, 38, 3, 42, 78, 71] [85, 46, 84, 79, 44, 7, 1, 2, 7] → [85, 46, 84, 79, 44, 1, 2, 7]
0.571	11	c198	*unique elements in descending order* `(λ x (reverse (sort (λ y y) (unique x))))` [62, 86, 85, 62, 29, 8, 85, 29] → [86, 85, 62, 29, 8] [26, 26, 26, 26, 10, 55, 6, 10, 55, 6] → [55, 26, 10, 6] [18, 65, 98, 98, 98, 18, 18, 98, 65] → [98, 65, 18] [7, 5, 5, 69, 69, 30, 30, 7, 5] → [69, 30, 7, 5] [73, 66, 24, 52, 24, 43, 73, 24, 66, 52] → [73, 66, 52, 43, 24]
0.568	14	c219	*keep only every other element, starting from the end of the list* `(λ x (reverse (filteri (λ y (λ z (is_odd y))) (reverse x))))` [42, 0, 46, 12, 8, 58, 50] → [42, 46, 8, 50] [7, 93, 99, 86, 30, 97, 60, 62, 57, 17] → [93, 86, 97, 62, 17] [3, 52, 78, 9, 94, 54, 90, 89, 37, 14] → [52, 9, 54, 89, 14] [65, 2, 66, 47, 28, 71, 9, 5] → [2, 47, 71, 5] [81, 22, 85, 82, 36, 59, 16, 8, 45] → [81, 85, 36, 16, 45]
0.566	10	c075	*remove every element with an even index* `(λ x (filteri (λ y (λ z (is_odd y))) x))` [6, 5, 2, 9] → [6, 2] [7, 3] → [7] [0, 1, 1, 9, 2, 0, 1, 0, 9, 2] → [0, 1, 2, 1, 9] [8, 0, 4, 2, 8] → [8, 4, 8] [7, 4, 5, 2, 5, 5, 2] → [7, 5, 5, 2]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.559	10	c177	*interleave the input and the reversed input* `(λ x (flatten (zip x (reverse x))))` [64, 98, 27, 26, 32] → [64, 32, 98, 26, 27, 27, 26, 98, 32, 64] [1, 15, 28, 4] → [1, 4, 15, 28, 28, 15, 4, 1] [80, 2, 61, 24, 37, 8] → [80, 8, 2, 37, 61, 24, 24, 61, 37, 2, 8, 80] [93, 50, 81] → [93, 81, 50, 50, 81, 93] [59, 90] → [59, 90, 90, 59]
0.553	6	c025	*remove element 2* `(λ x (cut_idx 2 x))` [1, 1, 4, 5] → [1, 4, 5] [2, 6] → [2] [1, 6, 6, 0, 8, 3, 9, 0, 7, 9] → [1, 6, 0, 8, 3, 9, 0, 7, 9] [3, 0, 4, 9, 5] → [3, 4, 9, 5] [7, 2, 5, 0, 8, 4, 1] → [7, 5, 0, 8, 4, 1]
0.552	21	c125	*keep only elements whose tens digit equals the tens digit of element 1* `(λ x (filter (λ y (== (/ (first x) 10) (/ y 10))) x))` [41, 6, 41, 27, 55, 66, 42, 3] → [41, 41, 42] [61, 62, 9, 3, 56, 85, 64, 82, 49] → [61, 62, 64] [5, 2, 90, 75, 57, 1, 7, 19, 8, 84] → [5, 2, 1, 7, 8] [32, 32, 4, 3, 32, 7, 30, 96, 5] → [32, 32, 32, 30] [99, 99, 97, 79, 16, 75, 75, 97, 16, 8] → [99, 99, 97, 97]
0.545	22	c211	*reverse input; insert elements 4 and 5 so they are fourth and third from last, respectively* `(λ x (splice (slice 4 5 x) (- (length x) 2) (reverse x)))` [22, 1, 7, 65, 21, 77] → [77, 21, 65, 65, 21, 7, 1, 22] [94, 72, 79, 4, 47, 46] → [46, 47, 4, 4, 47, 79, 72, 94] [97, 59, 0, 39, 8, 48, 53, 4] → [4, 53, 48, 8, 39, 39, 8, 0, 59, 97] [12, 5, 18, 62, 78, 28, 31, 68] → [68, 31, 28, 78, 62, 62, 78, 18, 5, 12] [49, 6, 23, 92, 45, 36, 75] → [75, 36, 45, 92, 92, 45, 23, 6, 49]
0.54	8	c087	*swap elements 2 and 3* `(λ x (swap 2 3 x))` [36, 77, 25, 3, 1] → [36, 25, 77, 3, 1] [21, 94, 56, 2, 0, 0, 21, 94] → [21, 56, 94, 2, 0, 0, 21, 94] [26, 71, 7, 31, 5, 21, 55, 4, 87] → [26, 7, 71, 31, 5, 21, 55, 4, 87] [72, 88, 72, 88, 45, 88, 72, 85, 45, 79] → [72, 72, 88, 88, 45, 88, 72, 85, 45, 79] [0, 9, 0, 68, 68, 9] → [0, 0, 9, 68, 68, 9]
0.539	8	c013	*elements 3 through 7* `(λ x (slice 3 7 x))` [8, 5, 5, 8, 8, 5, 5, 5] → [5, 8, 8, 5, 5] [9, 9, 4, 4, 9, 3, 1, 1, 9] → [4, 4, 9, 3, 1] [6, 4, 2, 4, 0, 0, 8, 7, 5] → [2, 4, 0, 0, 8] [3, 2, 0, 3, 4, 4, 6, 6] → [0, 3, 4, 4, 6] [9, 8, 7, 4, 1, 3, 2, 0, 6, 5] → [7, 4, 1, 3, 2]
0.534	38	c069	*concatenate [9, 3, 4, 0], input, and [7, 2, 9, 1]* `(λ x (concat (cons 9 (cons 3 (cons 4 (singleton 0)))) (concat x (cons 7 (cons 2 (cons 9 (singleton 1)))))))` [0, 5, 5, 5] → [9, 3, 4, 0, 0, 5, 5, 5, 7, 2, 9, 1] [ ] → [9, 3, 4, 0, 7, 2, 9, 1] [8, 4, 7] → [9, 3, 4, 0, 8, 4, 7, 7, 2, 9, 1] [8] → [9, 3, 4, 0, 8, 7, 2, 9, 1] [7, 1] → [9, 3, 4, 0, 7, 1, 7, 2, 9, 1]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.534	14	c117	*left-rotate by three elements* `(λ x (concat (drop 3 x) (take 3 x)))` [77, 90, 44, 9, 84] → [9, 84, 77, 90, 44] [57, 0, 17, 95, 1, 94, 68, 31, 46] → [95, 1, 94, 68, 31, 46, 57, 0, 17] [20, 6, 21, 56, 79, 8] → [56, 79, 8, 20, 6, 21] [23, 5, 42, 7, 52, 74, 25] → [7, 52, 74, 25, 23, 5, 42] [2, 81, 92, 80] → [80, 2, 81, 92]
0.531	18	c228	*replace each element with the number of occurrences of that element so far in the input* `(λ x (mapi (λ y (λ z (count (== z) (take y x)))) x))` [42, 22, 22, 42, 42] → [1, 1, 2, 2, 3] [11, 11, 6, 11, 11, 6, 6, 6] → [1, 2, 1, 3, 4, 2, 3, 4] [80, 80, 80, 80] → [1, 2, 3, 4] [84, 84, 84] → [1, 2, 3] [58, 58] → [1, 2]
0.53	13	c197	*replace each element with the number of occurrences of that element* `(λ x (map (λ y (count (== y) x)) x))` [43, 43, 19, 72, 73] → [2, 2, 1, 1, 1] [23, 53, 46, 79, 41, 0, 51, 41, 16, 93] → [1, 1, 1, 1, 2, 1, 1, 2, 1, 1] [70, 70, 70, 70, 42, 42] → [4, 4, 4, 4, 2, 2] [27, 74, 27, 74, 64, 74, 27, 74, 74] → [3, 5, 3, 5, 1, 5, 3, 5, 5] [8, 80, 80] → [1, 2, 2]
0.527	10	c065	*prepend 9 and append 7* `(λ x (cons 9 (append x 7)))` [1, 0, 0, 8] → [9, 1, 0, 0, 8, 7] [2, 2, 8] → [9, 2, 2, 8, 7] [ ] → [9, 7] [7] → [9, 7, 7] [1, 3, 1, 5, 6, 4, 4, 3, 8] → [9, 1, 3, 1, 5, 6, 4, 4, 3, 8, 7]
0.524	14	c156	*reverse the input and add each element to its new index* `(λ x (mapi (λ y (λz (+ z y))) (reverse x)))` [38, 61, 56, 17, 51, 4] → [5, 53, 20, 60, 66, 44] [28, 7, 69, 5, 55, 18, 83, 71, 46] → [47, 73, 86, 22, 60, 11, 76, 15, 37] [2, 33, 39, 42, 8, 7, 1, 64] → [65, 3, 10, 12, 47, 45, 40, 10] [37, 23, 94, 5, 27, 72, 0] → [1, 74, 30, 9, 99, 29, 44] [6, 48, 13, 78, 18, 88, 30, 86, 62, 21] → [22, 64, 89, 34, 93, 24, 85, 21, 57, 16]
0.523	11	c110	*three largest elements in ascending order* `(λ x (takelast 3 (sort (λ y y) x)))` [97, 7, 13, 2, 55] → [13, 55, 97] [18, 35, 7, 70, 96, 74, 37, 45] → [70, 74, 96] [45, 92, 5, 40, 3, 78, 81, 50, 4, 76] → [78, 81, 92] [85, 89, 8, 7, 9, 44, 30, 68, 69] → [69, 85, 89] [57, 15, 51, 31, 33, 61, 6] → [51, 57, 61]
0.521	11	c233	*number of occurrences of each unique element, in order of appearance* `(λ x (map length (group (λ y y) x)))` [2, 82, 82, 52, 87, 41, 87] → [1, 2, 1, 2, 1] [4, 63, 9, 68, 62, 67, 9, 22, 56] → [1, 1, 2, 1, 1, 1, 1, 1] [34, 34, 34, 34, 34, 34, 34, 34, 34, 34] → [10] [71, 17, 71, 71, 71, 17, 71, 17] → [5, 3] [33, 55, 84, 84, 64, 33, 55, 64, 6, 18] → [2, 2, 2, 2, 1, 1]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.519	8	c003	*remove all but element 7* `(λ x (singleton (nth 7 x)))` [7, 2, 3, 9, 0, 4, 5, 6] → [5] [6, 1, 6, 4, 4, 7, 0, 4, 6, 1] → [0] [0, 5, 7, 4, 9, 1, 3, 6, 7, 8] → [3] [6, 8, 5, 0, 2, 9, 8, 2] → [8] [1, 0, 8, 6, 7, 2, 9, 3, 5, 4] → [9]
0.519	12	c230	*count by 2 from the smallest element to the largest element* `(λ x (range (min x) 2 (max x)))` [8, 6, 7, 12, 2] → [2, 4, 6, 8, 10, 12] [44, 48] → [44, 46, 48] [5, 1, 7] → [1, 3, 5, 7] [16, 6, 9, 4] → [4, 6, 8, 10, 12, 14, 16] [78, 86] → [78, 80, 82, 84, 86]
0.515	8	c122	*remove all but penultimate element* `(λ x (singleton (second (reverse x))))` [22, 46, 27, 2, 89] → [2] [77, 4, 57] → [4] [80, 70] → [80] [9, 26, 65, 71, 33, 5, 67, 3, 40, 56] → [40] [98, 55, 32, 41, 8, 93, 11, 44, 88] → [44]
0.514	10	c020	*replace the first element with the last element* `(λ x (replace 1 (last x) x))` [7, 7, 7, 9] → [9, 7, 7, 9] [8, 7, 6, 8, 9, 4, 3] → [3, 7, 6, 8, 9, 4, 3] [9, 9, 2, 3, 3, 3, 2, 1, 1] → [1, 9, 2, 3, 3, 3, 2, 1, 1] [8, 9, 7, 2, 7, 0] → [0, 9, 7, 2, 7, 0] [8, 5] → [5, 5]
0.506	46	c235	*count up and down between elements* `(λ x (fold (λ y (λ z (concat y (drop 1 (range (last y) (if (> z (last y)) 1 -1) z))))) (take 1 x) (drop 1 x)))` [8, 7, 5, 6, 15] → [8, 7, 6, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] [46, 43, 42] → [46, 45, 44, 43, 42] [0, 4, 7, 6] → [0, 1, 2, 3, 4, 5, 6, 7, 6] [1, 5, 7] → [1, 2, 3, 4, 5, 6, 7] [6, 9] → [6, 7, 8, 9]
0.503	19	c157	*each element followed by 0 if even or 1 if odd, in order of appearance* `(λ x (flatten (map (λ y (cons y (singleton (% y 2)))) x)))` [32, 3, 5, 81, 16, 6] → [32, 0, 3, 1, 5, 1, 81, 1, 16, 0, 6, 0] [63, 18, 24, 92, 44, 89, 30] → [63, 1, 18, 0, 24, 0, 92, 0, 44, 0, 89, 1, 30, 0] [74, 71, 15, 46, 26, 59] → [74, 0, 71, 1, 15, 1, 46, 0, 26, 0, 59, 1] [78, 94, 6, 90, 7, 77, 79] → [78, 0, 94, 0, 6, 0, 90, 0, 7, 1, 77, 1, 79, 1] [96, 28, 95, 6, 4, 57, 9] → [96, 0, 28, 0, 95, 1, 6, 0, 4, 0, 57, 1, 9, 1]
0.5	26	c185	*replace the sublist between the first and last elements with its sum* `(λ x (cons (first x) (cons (sum (drop 1 (droplast 1 x))) (takelast 1 x))))` [41, 9, 5, 45, 30, 89] → [41, 89, 89] [65, 0, 2, 23, 7, 21, 5, 3, 74] → [65, 61, 74] [16, 19, 36, 12, 4, 6, 8, 2, 0, 23] → [16, 87, 23] [53, 3, 1, 4, 26, 41, 35] → [53, 75, 35] [81, 29, 31, 15, 7, 9, 4, 47] → [81, 95, 47]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.5	11	c213	*three smallest elements in ascending order*

```
(λ x (take 3 (sort (λ y y) x)))
```

```
[16, 25, 95, 44, 39]                → [16, 25, 39]
[58, 7, 48, 1, 43, 8, 3, 84, 53, 0] → [0, 1, 3]
[62, 5, 41, 7, 68, 65, 3, 32, 18]   → [3, 5, 7]
[83, 23, 2, 52, 61, 59]             → [2, 23, 52]
[86, 19, 21, 17, 40, 45, 13]        → [13, 17, 19]
```

0.495	16	c184	*keep only odd elements with an even index*

```
(λ x (filteri (λ y (λ z (and (is_even y) (is_odd z)))) x))
```

```
[6, 91, 0, 77, 18, 25]                → [91, 77, 25]
[19, 45, 91, 1, 70, 33, 0, 81, 47, 73] → [45, 1, 33, 81, 73]
[82, 7, 87, 9, 2, 31, 67]             → [7, 9, 31]
[54, 97, 49, 5, 6, 35, 2, 1, 78]      → [97, 5, 35, 1]
[4, 75, 8, 63, 18, 3, 51, 7]          → [75, 63, 3, 7]
```

0.492	12	c155	*multiply each element by its index*

```
(λ x (mapi (λ y (λ z (* z y))) x))
```

```
[2, 0, 6, 0, 3, 2]             → [2, 0, 18, 0, 15, 12]
[3, 7, 9, 8, 0, 6, 1, 2]       → [3, 14, 27, 32, 0, 36, 7, 16]
[5, 5, 10, 5, 6, 4, 5]         → [5, 10, 30, 20, 30, 24, 35]
[2, 4, 2, 1, 0, 2, 9, 1, 7, 6] → [2, 8, 6, 4, 0, 12, 63, 8, 63, 60]
[9, 3, 3, 5, 8, 0, 7, 4, 2]    → [9, 6, 9, 20, 40, 0, 49, 32, 18]
```

0.49	18	c082	*remove all but element 3*

```
(λ x (if (> 3 (length x)) empty (singleton (third x))))
```

```
[10, 6, 2, 99, 0]                → [2]
[15]                             → [ ]
[18, 79, 7, 5, 3, 7, 3, 5, 79]   → [7]
[96, 40]                         → [ ]
[91, 75, 3, 6, 8, 42, 11, 4, 1, 60] → [3]
```

0.487	16	c231	*the number of occurrences of each unique element in ascending order*

```
(λ x (sort (λ y y) (map length (group (λ z z) x))))
```

```
[80, 5, 98, 32, 98, 80, 98]          → [1, 1, 2, 3]
[92, 92, 92, 92, 21, 21, 92, 92]     → [2, 6]
[78, 34, 78, 78, 34, 34, 4, 4, 34, 4] → [3, 3, 4]
[33, 31, 71, 33, 59, 64, 31, 31, 59, 7] → [1, 1, 1, 2, 2, 3]
[38, 38, 38, 38, 38, 38, 38, 38, 38] → [9]
```

0.485	11	c099	*remove all elements whose value < 50*

```
(λ x (filter (λ y (> y 49)) x))
```

```
[8, 87, 2, 64, 48, 79]            → [87, 64, 79]
[1, 7, 8, 25, 0, 40, 53, 94]      → [53, 94]
[78, 18, 92, 42, 95, 3, 98]       → [78, 92, 95, 98]
[7, 28, 13, 8, 0, 28, 8, 28]      → [ ]
[86, 5, 86, 86, 11, 99, 99, 99, 5] → [86, 86, 86, 99, 99, 99]
```

0.485	44	c247	*elements before the first 0*

```
(λ x (first (reverse (fold (λ y (λ z (if (== z 0) (cons empty y) (cons (append (first y) z))
(drop 1 y)))) (singleton empty) x))))
```

```
[5, 26, 7, 23, 0, 93, 0, 18, 1]     → [5, 26, 7, 23]
[1, 71, 89, 0, 71, 46, 8, 87, 0, 7] → [1, 71, 89]
[9, 86, 0, 60, 69, 53, 1, 0, 6, 60] → [9, 86]
[4, 0, 80, 87, 87, 0, 45, 2, 4]     → [4]
[84, 0, 20, 0, 63, 63, 20, 20, 20]  → [84]
```

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.476	6	c008	*the first 6 elements* `(λ x (take 6 x))` [1, 3, 9, 8, 1, 6, 7] → [1, 3, 9, 8, 1, 6] [9, 2, 2, 6, 9, 9, 2, 9] → [9, 2, 2, 6, 9, 9] [3, 7, 7, 0, 3, 8, 5] → [3, 7, 7, 0, 3, 8] [0, 4, 4, 0, 9, 0, 4] → [0, 4, 4, 0, 9, 0] [4, 0, 5, 6, 5, 0, 5] → [4, 0, 5, 6, 5, 0]
0.473	14	c055	*swap elements 1 and 3 and elements 2 and 4* `(λ x (swap 1 3 (swap 2 4 x)))` [4, 8, 7, 9] → [7, 9, 4, 8] [0, 2, 6, 1, 9, 5, 6, 5, 3] → [6, 1, 0, 2, 9, 5, 6, 5, 3] [8, 1, 7, 0, 5] → [7, 0, 8, 1, 5] [9, 4, 4, 3, 9, 9, 3, 4] → [4, 3, 9, 4, 9, 9, 3, 4] [3, 9, 7, 2, 0, 8, 5] → [7, 2, 3, 9, 0, 8, 5]
0.462	6	c029	*remove the first two elements* `(λ x (drop 2 x))` [8, 1, 9, 9] → [9, 9] [2, 2] → [ ] [5, 1, 0, 0, 5, 8, 1, 8, 3, 0] → [0, 0, 5, 8, 1, 8, 3, 0] [6, 6, 6] → [6] [7, 0, 4, 1, 7] → [4, 1, 7]
0.459	14	c168	*count from 1 to 10, skipping the input's length* `(λ x (cut_val (length x) (range 1 1 10)))` [66, 74, 88, 49, 15] → [1, 2, 3, 4, 6, 7, 8, 9, 10] [96, 25, 43, 86, 50, 44, 13, 87, 2, 84] → [1, 2, 3, 4, 5, 6, 7, 8, 9] [21, 85] → [1, 3, 4, 5, 6, 7, 8, 9, 10] [98, 30, 27, 53, 7, 45, 0] → [1, 2, 3, 4, 5, 6, 8, 9, 10] [65, 1, 69, 76, 33, 16] → [1, 2, 3, 4, 5, 7, 8, 9, 10]
0.45	6	c009	*remove all but the first 6 elements* `(λ x (take 6 x))` [6, 2, 4, 4, 4, 8, 8] → [6, 2, 4, 4, 4, 8] [7] → [7] [5, 8, 8, 9, 9, 5, 8, 5] → [5, 8, 8, 9, 9, 5] [6, 7, 0] → [6, 7, 0] [1, 1, 1, 1] → [1, 1, 1, 1]
0.442	8	c133	*remove elements 2 through 5* `(λ x (cut_slice 2 5 x))` [17, 65, 41, 49, 9, 5] → [17, 5] [85, 50, 30, 14, 6, 89, 57, 77] → [85, 89, 57, 77] [73, 3, 2, 70, 21, 87, 86, 23, 76] → [73, 87, 86, 23, 76] [11, 2, 74, 41, 1, 10, 0] → [11, 10, 0] [31, 47, 82, 96, 52, 98, 3, 4, 68, 61] → [31, 98, 3, 4, 68, 61]
0.435	27	c146	*absolute difference of each consecutive pair, in order of appearance* `(λ x (map (λ y (- (max y) (min y))) (zip (droplast 1 x) (drop 1 x))))` [87, 67, 47, 9, 44] → [20, 20, 38, 35] [80, 98, 4, 25] → [18, 94, 21] [2, 53, 31, 23, 49, 6, 16] → [51, 22, 8, 26, 43, 10] [58, 86, 12, 66, 90, 20, 45, 64] → [28, 74, 54, 24, 70, 25, 19] [8, 34, 17, 82, 4, 93, 5, 18, 41, 11] → [26, 17, 65, 78, 89, 88, 13, 23, 30]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.434	20	c004	*remove all but element 7*

```
(λ x (if (> 7 (length x)) empty (singleton (nth 7 x))))
```

```
[7, 4]                    → [ ]
[0, 3, 2, 9, 4, 6, 8, 4, 8] → [8]
[5, 6, 0, 2, 9, 7, 3, 2, 1, 8] → [3]
[9, 9]                    → [ ]
[5, 9, 8, 8, 5, 0, 0, 2]  → [0]
```

0.433	15	c240	*number of elements equal to the input length*

```
(λ x (singleton (count (λ y (== (length x) y)) x)))
```

```
[21, 7, 7, 7, 83, 21, 29]          → [3]
[8, 65, 5, 34, 8, 59, 18, 4]       → [2]
[60, 24, 51, 8, 72, 9, 98, 2, 65, 1] → [0]
[0, 5, 95, 3, 7, 91, 7]            → [2]
[68, 3, 94, 22, 99, 16, 93, 2, 9]  → [1]
```

0.43	14	c239	*number of repetitions in the input*

```
(λ x (singleton (- (length x) (length (unique x)))))
```

```
[86, 27, 88, 71, 54, 86, 71, 54]    → [3]
[25, 37, 28, 43, 14, 14, 6, 25, 51] → [2]
[13, 84, 48, 60, 84, 60, 21, 21, 96] → [3]
[52, 44, 0, 70, 7, 77, 18, 79, 44, 67] → [1]
[75, 75, 22, 31, 9, 22, 62, 31, 12, 4] → [3]
```

0.429	8	c246	*number of unique elements*

```
(λ x (singleton (length (unique x))))
```

```
[76, 5, 19, 22, 19, 3, 9]          → [6]
[98, 64, 57, 6, 45, 79, 2, 59, 92] → [9]
[99, 77, 42, 26, 75, 7, 90, 38]    → [8]
[ ]                                → [0]
[37, 87, 33, 24, 18, 2, 17, 71, 47, 66] → [10]
```

0.428	8	c138	*remove all occurrences of element 1*

```
(λ x (cut_vals (first x) x))
```

```
[7, 32, 7, 32, 87]                 → [32, 32, 87]
[27, 38, 68, 75, 79, 8, 22, 0, 44, 1] → [38, 68, 75, 79, 8, 22, 0, 44, 1]
[34, 34, 19, 34, 35, 34]           → [19, 35]
[17, 17, 17]                       → [ ]
[92, 31, 45, 92, 49, 26, 11, 3, 97] → [31, 45, 49, 26, 11, 3, 97]
```

0.427	8	c018	*replace element 6 with a 3*

```
(λ x (replace 6 3 x))
```

```
[7, 7, 7, 7, 7, 7, 7]              → [7, 7, 7, 7, 7, 3, 7]
[8, 8, 6, 8, 5, 1, 4, 0, 5]        → [8, 8, 6, 8, 5, 3, 4, 0, 5]
[5, 3, 2, 8, 4, 6]                 → [5, 3, 2, 8, 4, 3]
[9, 9, 3, 0, 0, 9, 0, 0, 9, 3]     → [9, 9, 3, 0, 0, 3, 0, 0, 9, 3]
[9, 3, 1, 8, 3, 9, 3, 3, 1]        → [9, 3, 1, 8, 3, 3, 3, 3, 1]
```

0.424	8	c012	*remove all but elements 2 through 4*

```
(λ x (slice 2 4 x))
```

```
[3, 3, 4, 7, 6]    → [3, 4, 7]
[7, 8, 2, 0, 4, 2] → [8, 2, 0]
[6]                → [ ]
[2, 9, 4]          → [9, 4]
[6, 8]             → [8]
```

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.419	12	c232	*mean value of the input* `(λ x (singleton (/ (sum x) (length x))))` [72, 14, 69, 77, 8] → [48] [94, 60, 0, 89, 41] → [56] [18, 10, 16, 4] → [12] [13, 22, 68] → [34] [52, 75, 71] → [66]
0.417	14	c094	*swap elements 1 and 3 and elements 2 and 4* `(λ x (swap 1 3 (swap 2 4 x)))` [4, 1, 99, 66, 4] → [99, 66, 4, 1, 4] [6, 35, 46, 67, 7, 7] → [46, 67, 6, 35, 7, 7] [68, 90, 68, 31, 68, 58, 90] → [68, 31, 68, 90, 68, 58, 90] [96, 35, 63, 96, 63, 52, 96, 95, 37, 95] → [63, 96, 96, 35, 63, 52, 96, 95, 37, 95] [73, 73, 5, 5, 73, 5, 5, 5, 5] → [5, 5, 73, 73, 73, 5, 5, 5, 5]
0.414	8	c086	*swap elements 1 and 4* `(λ x (swap 1 4 x))` [72, 14, 74, 16, 27] → [16, 14, 74, 72, 27] [56, 0, 49, 15, 49, 80, 18, 80, 18] → [15, 0, 49, 56, 49, 80, 18, 80, 18] [1, 35, 35, 9, 8, 41, 1, 85, 41, 7] → [9, 35, 35, 1, 8, 41, 1, 85, 41, 7] [1, 5, 1, 5, 50, 1, 50] → [5, 5, 1, 1, 50, 1, 50] [69, 3, 39, 51, 8, 51, 3, 3] → [51, 3, 39, 69, 8, 51, 3, 3]
0.413	15	c193	*input in ascending order, minus an occurrence of both the smallest and largest elements* `(λ x (drop 1 (droplast 1 (sort (λ y y) x))))` [37, 24, 71, 8, 0, 39, 6, 25, 90] → [6, 8, 24, 25, 37, 39, 71] [6, 5, 7, 34, 78, 29, 23, 26, 1, 28] → [5, 6, 7, 23, 26, 28, 29, 34] [53, 60, 91, 61, 12, 3, 68, 8, 79, 96] → [8, 12, 53, 60, 61, 68, 79, 91] [88, 46, 2, 10, 34, 56, 45, 4, 64] → [4, 10, 34, 45, 46, 56, 64] [57, 77, 16, 17, 27, 44, 0, 42, 1] → [1, 16, 17, 27, 42, 44, 57]
0.4	6	c056	*remove element 5* `(λ x (cut_idx 5 x))` [2, 1, 2, 6, 7, 2] → [2, 1, 2, 6, 2] [5, 6, 9, 6, 6, 5, 9] → [5, 6, 9, 6, 5, 9] [7, 6, 7, 8, 6, 0, 7, 6, 0, 5] → [7, 6, 7, 8, 0, 7, 6, 0, 5] [8, 8, 8, 8, 8, 8, 8, 8, 8] → [8, 8, 8, 8, 8, 8, 8, 8] [1, 1, 1, 1, 1, 1, 1, 1, 1] → [1, 1, 1, 1, 1, 1, 1, 1]
0.396	20	c028	*remove element 2 if element 1 > element 2, else remove element 3* `(λ x (cut_idx (if (> (first x) (second x)) 2 3) x))` [1, 2, 0, 6] → [1, 2, 6] [3, 2, 5, 8, 0, 9, 5] → [3, 5, 8, 0, 9, 5] [5, 7, 1, 9, 0, 6, 2, 8, 4, 7] → [5, 7, 9, 0, 6, 2, 8, 4, 7] [3, 1, 4] → [3, 4] [3, 0, 7, 6, 9, 4] → [3, 7, 6, 9, 4]
0.395	21	c220	*if input length is even, double each element, else triple it* `(λ x (map (λ y (* y (if (is_even (length x)) 2 3))) x))` [4, 8, 3, 29, 15] → [12, 24, 9, 87, 45] [5, 39] → [10, 78] [19] → [57] [1, 4, 0, 9, 25, 6, 2, 28, 7] → [3, 12, 0, 27, 75, 18, 6, 84, 21] [43, 23, 11, 5, 8, 30, 41, 2] → [86, 46, 22, 10, 16, 60, 82, 4]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.394	8	c241	*number of even elements* `(λ x (singleton (count is_even x)))` [49, 87, 13, 67, 4, 5, 8] → [2] [0, 45, 84, 90, 72, 8, 68, 72, 30] → [8] [5, 53, 61, 57, 7, 63, 12, 3] → [1] [22, 22, 10, 6, 28, 26, 26, 16, 22, 26] → [10] [3, 44, 36, 82, 4, 6, 2, 14, 32, 32] → [9]
0.392	22	c060	*elements 3, 2, 1, the number 4, then elements 5 and 7, in that order* `(λ x (swap 3 1 (replace 4 4 (cut_idx 6 (take 7 x)))))` [7, 9, 0, 2, 6, 8, 3, 5, 1] → [0, 9, 7, 4, 6, 3] [1, 7, 8, 2, 5, 6, 0, 4, 3, 9] → [8, 7, 1, 4, 5, 0] [6, 7, 1, 3, 2, 0, 8, 9, 4, 5] → [1, 7, 6, 4, 2, 8] [9, 2, 0, 5, 8, 7, 6, 4, 1, 3] → [0, 2, 9, 4, 8, 6] [9, 2, 5, 1, 3, 4, 7, 0, 6, 8] → [5, 2, 9, 4, 3, 7]
0.391	12	c169	*second largest element* `(λ x (singleton (max (cut_vals (max x) x))))` [26, 68, 87, 84, 58, 10] → [84] [8, 24, 51, 14, 2, 69, 9, 28, 48] → [51] [5, 32, 76, 7, 90, 53, 65, 54] → [76] [36, 99, 8, 9, 16, 67, 94, 0, 4, 40] → [94] [3, 13, 1, 95, 17, 20, 12] → [20]
0.39	8	c057	*insert a 4 as element 7* `(λ x (insert 4 7 x))` [3, 3, 3, 3, 3, 3, 3] → [3, 3, 3, 3, 3, 3, 4, 3] [2, 7, 8, 4, 0, 6, 5, 1] → [2, 7, 8, 4, 0, 6, 4, 5, 1] [2, 3, 9, 7, 6, 0, 0, 8] → [2, 3, 9, 7, 6, 0, 4, 0, 8] [2, 9, 9, 2, 9, 9, 2, 2, 2] → [2, 9, 9, 2, 9, 9, 4, 2, 2, 2] [5, 9, 3, 7, 2, 1, 6, 0, 8] → [5, 9, 3, 7, 2, 1, 4, 6, 0, 8]
0.388	31	c150	*count from element 1 up to each element, in order of appearance* `(λ x (flatten (map (λ y (if (> y (first x)) (range (first x) 1 y) (singleton y))) x)))` [37, 20, 47, 8, 5] → [37, 20, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 8, 5] [28, 29, 2] → [28, 28, 29, 2] [83, 19, 7, 32, 9, 86] → [83, 19, 7, 32, 9, 83, 84, 85, 86] [13, 1, 18] → [13, 1, 13, 14, 15, 16, 17, 18] [24, 15, 27, 4] → [24, 15, 24, 25, 26, 27, 4]
0.386	8	c033	*swap elements 1 and 4* `(λ x (swap 1 4 x))` [8, 1, 8, 5, 5, 2] → [5, 1, 8, 8, 5, 2] [7, 9, 1, 5, 6, 3, 0] → [5, 9, 1, 7, 6, 3, 0] [7, 8, 6, 5, 6, 7, 7, 5, 6, 5] → [5, 8, 6, 7, 6, 7, 7, 5, 6, 5] [1, 9, 6, 2] → [2, 9, 6, 1] [7, 8, 4, 6, 1, 2, 9, 5] → [6, 8, 4, 7, 1, 2, 9, 5]
0.386	17	c218	*keep only elements 2 and following equal to element 1* `(λ x (filter (λ y (== y (first x))) (drop 1 x)))` [93, 93, 52, 93, 99, 99, 71, 52] → [93, 93] [90, 61, 9, 9, 37, 90, 30, 3, 3] → [90] [55, 2, 55, 6, 6, 19, 53, 8, 55, 55] → [55, 55, 55] [15, 25, 73, 6, 7, 20, 67, 13, 23] → [ ] [45, 27, 1, 45, 1, 27, 45, 45, 45, 1] → [45, 45, 45, 45]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.385	8	c014	*remove all but elements 3 through 7* `(λ x (slice 3 7 x))` [6, 3, 0, 4] → [0, 4] [9, 9, 1] → [1] [0, 7, 9, 6, 1, 8, 2, 3] → [9, 6, 1, 8, 2] [3, 0, 5, 7, 7, 5, 3, 0, 5] → [5, 7, 7, 5, 3] [9] → [ ]
0.376	8	c136	*remove the first occurrence of the largest element* `(λ x (cut_val (max x) x))` [95, 95, 73, 95, 23] → [95, 73, 95, 23] [35, 22, 46, 94, 94, 52] → [35, 22, 46, 94, 52] [2, 90, 8, 90, 72, 14, 13, 1, 80] → [2, 8, 90, 72, 14, 13, 1, 80] [92, 92] → [92] [61, 79, 89, 71, 74, 20, 30, 62, 67, 3] → [61, 79, 71, 74, 20, 30, 62, 67, 3]
0.352	8	c083	*remove all but element 7* `(λ x (singleton (nth 7 x)))` [47, 27, 63, 1, 28, 26, 82, 1] → [82] [67, 3, 20, 9, 50, 72, 29, 9, 35, 82] → [29] [3, 4, 23, 57, 46, 60, 70, 7] → [70] [6, 4, 54, 54, 5, 5, 4, 68] → [4] [0, 6, 89, 5, 13, 89, 21, 21, 3, 30] → [21]
0.351	8	c019	*replace element 6 with a 3 if there is an element 6* `(λ x (replace 6 3 x))` [4, 5, 5, 0, 4, 2] → [4, 5, 5, 0, 4, 3] [6, 6] → [6, 6] [3, 7, 2, 0, 4, 9, 8] → [3, 7, 2, 0, 4, 3, 8] [1, 1, 7, 4, 0, 8, 9, 0] → [1, 1, 7, 4, 0, 3, 9, 0] [4, 3, 5, 8, 3, 2, 7, 9, 1] → [4, 3, 5, 8, 3, 3, 7, 9, 1]
0.347	14	c203	*the first M positive multiples of the smallest element, where M is the input's length* `(λ x (mapi (λ y (λ z (* (min x) y))) x))` [7, 3, 5, 2, 1] → [1, 2, 3, 4, 5] [37, 82] → [37, 74] [23, 92, 84] → [23, 46, 69] [67, 64, 29, 99, 8, 62, 22, 81, 44] → [8, 16, 24, 32, 40, 48, 56, 64, 72] [36, 78, 19, 89] → [19, 38, 57, 76]
0.331	20	c115	*the unique elements, prepended and appended by their sum* `(λ x (cons (sum (unique x)) (append (unique x) (sum (unique x)))))` [7, 17, 45, 17, 12] → [81, 7, 17, 45, 12, 81] [4, 4, 31, 38, 38, 31, 38, 38] → [73, 4, 31, 38, 73] [24, 3, 24, 1, 3, 2, 42, 2, 1, 42] → [72, 24, 3, 1, 2, 42, 72] [27, 14, 14, 14, 14, 27, 27, 27, 27] → [41, 27, 14, 41] [ ] → [0, 0]
0.329	12	c207	*element-wise sum of the input and the reversed input* `(λ x (map sum (zip x (reverse x))))` [7, 6, 9, 79, 46] → [53, 85, 18, 85, 53] [14, 8, 22] → [36, 16, 36] [1, 31, 84, 4, 68, 89] → [90, 99, 88, 88, 99, 90] [19, 1, 97, 62] → [81, 98, 98, 81] [5, 13, 51] → [56, 26, 56]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.325	28	c088	*swap elements 1 and 4 if element 2 = element 3, else swap elements 2 and 3* `(λ x (if (== (second x) (third x)) (swap 1 4 x) (swap 2 3 x)))` [19, 1, 99, 19, 20, 99, 20] → [19, 99, 1, 19, 20, 99, 20] [95, 31, 31, 4, 39, 5, 32, 0] → [4, 31, 31, 95, 39, 5, 32, 0] [15, 53, 53, 95] → [95, 53, 53, 15] [84, 86, 3, 84, 3, 89] → [84, 3, 86, 84, 3, 89] [17, 41, 41, 85, 25, 2, 17, 25, 17, 10] → [85, 41, 41, 17, 25, 2, 17, 25, 17, 10]
0.322	30	c040	*append 3 if the list contains a 3, else append 9 if the list contains a 9* `(λ x (if (is_in x 3) (append x 3) (if (is_in x 9) (append x 9) x)))` [4, 8, 3, 7, 8] → [4, 8, 3, 7, 8, 3] [5, 8, 2, 9, 0, 0] → [5, 8, 2, 9, 0, 0, 9] [3, 8, 5] → [3, 8, 5, 3] [9, 1, 1, 5, 1, 6, 5, 6] → [9, 1, 1, 5, 1, 6, 5, 6, 9] [7, 0] → [7, 0]
0.32	14	c085	*remove all but element N + 1, N = element 1* `(λ x (singleton (nth (first x) (drop 1 x))))` [4, 39, 4, 48, 46, 48] → [46] [2, 67, 52, 72, 93, 9, 67, 5, 68] → [52] [5, 4, 3, 80, 36, 66, 75] → [66] [4, 28, 5, 26, 29, 6, 94, 3] → [29] [1, 4, 4] → [4]
0.316	10	c111	*repeat the largest element N times, where N is the smallest element* `(λ x (repeat (max x) (min x)))` [38, 76, 18, 31, 5] → [76, 76, 76, 76, 76] [90, 28, 72, 2, 5, 94, 85, 88, 68] → [94, 94] [52, 75, 83, 30, 3, 93] → [93, 93, 93] [1, 82, 56, 49, 1, 60, 60, 90] → [90] [0, 24, 65] → [ ]
0.316	6	c073	*add the index to every element* `(λ x (mapi + x))` [4, 4, 4, 4] → [5, 6, 7, 8] [1] → [2] [5, 5, 5, 2, 2, 2] → [6, 7, 8, 6, 7, 8] [1, 3, 3, 5, 3, 0, 0, 1, 0] → [2, 5, 6, 9, 8, 6, 7, 9, 9] [3, 3] → [4, 5]
0.315	20	c027	*remove element 2 if element 1 = element 2, else remove element 3* `(λ x (cut_idx (if (== (first x) (second x)) 2 3) x))` [0, 0, 9, 0] → [0, 9, 0] [8, 8, 4, 1, 8, 4, 1, 4] → [8, 4, 1, 8, 4, 1, 4] [6, 4, 6, 2, 3, 3] → [6, 4, 2, 3, 3] [1, 2, 5, 1, 3, 2, 5] → [1, 2, 1, 3, 2, 5] [9, 9, 7, 7, 7, 6, 6, 7, 2] → [9, 7, 7, 7, 6, 6, 7, 2]
0.312	18	c234	*repetitions in order of first appearance* `(λ x (flatten (map (λ y (drop 1 y)) (group (λ z z) x))))` [66, 0, 66, 58, 25, 0, 25, 92] → [66, 0, 25] [48, 1, 6, 94, 96, 65, 82, 45, 90, 5] → [ ] [7, 26, 84, 6, 62, 77, 2, 84, 10, 80] → [84] [3, 22, 5, 24, 1, 22, 21, 19, 5, 2] → [22, 5] [4, 16, 85, 16, 8, 16, 4, 85, 85] → [4, 16, 16, 85, 85]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.307	8	c165	*reverse the input and keep only even elements* `(λ x (filter is_even (reverse x)))` [68, 9, 86, 13, 57, 14, 72, 25, 69] → [72, 14, 86, 68] [77, 5, 56, 4, 34, 22, 65, 94, 20, 3] → [20, 94, 22, 34, 4, 56] [42, 7, 59, 1, 92, 48, 24, 63, 5, 8] → [8, 24, 48, 92, 42] [53, 5, 0, 7, 78, 43, 45, 39, 2] → [2, 78, 0] [54, 0, 97, 79, 99, 50, 6, 93, 3, 84] → [84, 6, 50, 0, 54]
0.306	17	c191	*repeat each element N times, where N is its tens digit, in order of appearance* `(λ x (flatten (map (λ y (repeat y (/ y 10))) x)))` [2, 0, 7, 30, 26] → [30, 30, 30, 26, 26] [37] → [37, 37, 37] [1, 12, 7, 9, 8, 23, 0, 6, 15] → [12, 23, 23, 15] [55, 5, 7, 20, 1, 27] → [55, 55, 55, 55, 55, 20, 20, 27, 27] [3, 4, 2] → [ ]
0.303	6	c026	*remove element 3* `(λ x (cut_idx 3 x))` [4, 2, 2, 5] → [4, 2, 5] [0, 7, 7, 4, 6] → [0, 7, 4, 6] [3, 9, 8, 1, 6, 2, 0, 4, 5, 7] → [3, 9, 1, 6, 2, 0, 4, 5, 7] [1, 1, 3, 2, 2, 3, 3] → [1, 1, 2, 2, 3, 3] [8, 3, 9] → [8, 3]
0.3	8	c059	*swap elements 4 and 8* `(λ x (swap 4 8 x))` [0, 5, 3, 8, 1, 9, 4, 6, 2] → [0, 5, 3, 6, 1, 9, 4, 8, 2] [6, 1, 8, 5, 2, 3, 7, 9, 0, 4] → [6, 1, 8, 9, 2, 3, 7, 5, 0, 4] [1, 0, 7, 8, 6, 4, 2, 6, 1, 9] → [1, 0, 7, 6, 6, 4, 2, 8, 1, 9] [9, 5, 5, 9, 3, 7, 6, 3, 9, 3] → [9, 5, 5, 3, 3, 7, 6, 9, 9, 3] [7, 3, 4, 0, 1, 6, 8, 1, 5] → [7, 3, 4, 1, 1, 6, 8, 0, 5]
0.298	11	c074	*remove every element whose value < 8* `(λ x (filter (λ y (> y 7)) x))` [8, 2, 7, 6, 8, 6] → [8, 8] [9, 2, 0, 5, 7, 5, 2, 3, 4, 7] → [9] [8, 8, 8, 2, 7, 7, 2] → [8, 8, 8] [0] → [ ] [8, 9, 9, 1, 1, 9, 8, 8, 9, 1] → [8, 9, 9, 9, 8, 8, 9]
0.295	14	c173	*replace element N with the largest element in elements 1 through N* `(λ x (mapi (λ y (λ z (max (take y x)))) x))` [2, 6, 74, 86, 1, 89] → [2, 6, 74, 86, 86, 89] [8, 9, 19, 2, 67, 83, 53, 4, 56, 95] → [8, 9, 19, 19, 67, 83, 83, 83, 83, 95] [3, 5, 9, 50, 7, 62, 78, 0] → [3, 5, 9, 50, 50, 62, 78, 78] [9, 11, 21, 25, 55, 48, 7, 1, 70] → [9, 11, 21, 25, 55, 55, 55, 55, 70] [47, 66, 81, 0, 1, 99, 4] → [47, 66, 81, 81, 81, 99, 99]
0.294	34	c175	*keep only elements larger than any preceding element* `(λ x (fold (λ y (λ z (if (> z (last y)) (append y z) y))) (take 1 x) (drop 1 x)))` [45, 58, 87, 48, 31, 34] → [45, 58, 87] [7, 8, 39, 95, 11, 1, 72] → [7, 8, 39, 95] [2, 44, 50, 62, 65, 9, 3, 8, 88, 91] → [2, 44, 50, 62, 65, 88, 91] [2, 25, 39, 51, 16, 5, 66, 7] → [2, 25, 39, 51, 66] [4, 31, 40, 6, 63, 1, 3, 66, 88] → [4, 31, 40, 63, 66, 88]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.294	29	c204	*keep only elements followed by 0* `(λ x (map first (filter (λ y (== (second y) 0)) (zip (droplast 1 x) (drop 1 x)))))` [9, 34, 0, 0, 96] → [34, 0] [4, 0, 29, 3, 0, 5, 6, 51] → [4, 3] [29, 28, 0, 2, 0, 64] → [28, 2] [13, 53, 88, 0, 6, 65, 21, 0, 9] → [88, 21] [28, 4, 97, 34, 14, 0, 0, 1, 0, 88] → [14, 0, 1]
0.293	20	c031	*remove whichever are equal: the two elements or the last two* `(λ x (if (== (first x) (second x)) drop droplast 2 x))` [6, 6, 6, 8] → [6, 8] [0, 0, 5, 5, 0, 5, 5, 0] → [5, 5, 0, 5, 5, 0] [1, 4, 4] → [1] [4, 4, 8, 3, 4, 9, 9, 9, 3, 6] → [8, 3, 4, 9, 9, 9, 3, 6] [4, 0, 2, 4, 2, 2, 7, 9, 9] → [4, 0, 2, 4, 2, 2, 7]
0.292	15	c236	*even elements divided by 2* `(λ x (map (λ y (/ y 2)) (filter is_even x)))` [18, 37, 3, 50, 13, 95, 9] → [9, 25] [25, 24, 7, 5, 38, 52, 74, 94] → [12, 19, 26, 37, 47] [92, 84, 9, 23, 7, 87, 73, 28, 90] → [46, 42, 14, 45] [64, 80, 8, 20, 2, 7, 6, 0, 44, 12] → [32, 40, 4, 10, 1, 3, 0, 22, 6] [86, 53, 99, 5, 30, 65, 72, 93, 43] → [43, 15, 36]
0.286	16	c054	*replace elements 1 and 2 with element 3* `(λ x (concat (repeat (third x) 3) (drop 3 x)))` [5, 5, 9, 5] → [9, 9, 9, 5] [7, 5, 2, 5, 7, 7, 2] → [2, 2, 2, 5, 7, 7, 2] [4, 1, 1, 1, 1, 4, 4, 1, 3, 3] → [1, 1, 1, 1, 1, 4, 4, 1, 3, 3] [9, 5, 9, 2, 3, 8, 2, 3, 8] → [9, 9, 9, 2, 3, 8, 2, 3, 8] [9, 3, 6, 7, 0] → [6, 6, 6, 7, 0]
0.267	12	c227	*interleave input and reversed input; keep only unique elements in order of appearance* `(λ x (unique (flatten (zip x (reverse x)))))` [9, 66, 10, 0, 43, 66, 9] → [9, 66, 10, 43, 0] [85, 39, 0, 33, 26, 27, 83, 18] → [85, 18, 39, 83, 0, 27, 33, 26] [40, 75, 49, 75, 40, 49, 49, 68, 49] → [40, 49, 75, 68] [55, 17, 20, 89, 22, 5, 20, 6, 65, 1] → [55, 1, 17, 65, 20, 6, 89, 22, 5] [65, 99, 86, 86, 28, 42, 7, 42, 53, 86] → [65, 86, 99, 53, 42, 7, 28]
0.265	20	c024	*insert as element 2: 8 if element 1 > 5 else 5* `(λ x (insert (if (> 5 (first x)) 8 5) 2 x))` [8, 7, 4, 1] → [8, 5, 7, 4, 1] [0, 8, 3] → [0, 8, 8, 3] [9, 9, 4, 9, 6, 1] → [9, 5, 9, 4, 9, 6, 1] [7, 6, 0, 7, 6] → [7, 5, 6, 0, 7, 6] [5, 2] → [5, 5, 2]
0.264	16	c200	*tens digits of the elements in ascending order* `(λ x (sort (λ y y) (map (λ z (/ z 10)) x)))` [49, 0, 24, 33, 92] → [0, 2, 3, 4, 9] [5, 54, 41, 72] → [0, 4, 5, 7] [68, 5, 91, 59, 36, 18, 71] → [0, 1, 3, 5, 6, 7, 9] [14, 89, 46, 34, 79, 0] → [0, 1, 3, 4, 7, 8] [87, 90, 16] → [1, 8, 9]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.256	12	c010	*elements 2 through N + 1, N = element 1*

```
(λ x (take (first x) (drop 1 x)))
```

```
[2, 3, 2, 7, 6]            → [3, 2]
[3, 9, 8, 6, 5, 1]         → [9, 8, 6]
[1, 2, 4, 5, 0, 8, 9, 7, 8] → [2]
[5, 5, 5, 1, 1, 5]         → [5, 5, 1, 1, 5]
[0, 2]                     → [ ]
```

0.256	10	c123	*remove all but element N, N = last element*

```
(λ x (singleton (nth (last x) x)))
```

```
[28, 48, 57, 36, 4]                → [36]
[90, 54, 16, 3]                    → [16]
[22, 9, 14, 87, 71, 3]             → [14]
[72, 1]                            → [72]
[69, 63, 50, 8, 86, 17, 0, 80, 19, 7] → [0]
```

0.256	20	c023	*insert as element 2: 8 if the list length > 5 else 5*

```
(λ x (insert (if (> 5 (length x)) 8 5) 2 x))
```

```
[2, 0, 5, 4]                  → [2, 8, 0, 5, 4]
[0, 2, 7, 9, 5, 8, 6, 3, 0, 9] → [0, 5, 2, 7, 9, 5, 8, 6, 3, 0, 9]
[9, 7, 6, 1, 2]               → [9, 5, 7, 6, 1, 2]
[8, 8]                        → [8, 8, 8]
[7, 0, 2]                     → [7, 8, 0, 2]
```

0.256	16	c194	*reverse the input; prepend and append the input's length*

```
(λ x (cons (length x) (append (reverse x) (length x))))
```

```
[76, 62, 80, 54, 23]       → [5, 23, 54, 80, 62, 76, 5]
[81, 43]                   → [2, 43, 81, 2]
[1, 63, 21, 16]            → [4, 16, 21, 63, 1, 4]
[92, 51, 35, 20, 9, 0, 18] → [7, 18, 0, 9, 20, 35, 51, 92, 7]
[39, 90, 8]                → [3, 8, 90, 39, 3]
```

0.25	19	c245	*number of times element 1 appears in elements 2 and following*

```
(λ x (singleton (count (λ y (== (first x) y)) (drop 1 x))))
```

```
[5, 5, 5, 5, 5, 41, 5]              → [5]
[67, 23, 84, 30, 18, 80, 1, 69, 28] → [0]
[1, 40, 1, 3, 51, 9, 91, 1, 2, 1]   → [3]
[59, 87, 59, 91, 53, 0, 2, 62, 76, 61] → [1]
[18, 49, 72, 7, 71, 8, 27, 97]      → [0]
```

0.242	20	c032	*remove two elements: the first two if element 1 > last element, else the last two*

```
(λ x (if (> (first x) (last x)) drop droplast 2 x))
```

```
[0, 1, 7, 9, 3]               → [0, 1, 7]
[7, 6, 4, 4, 1, 8, 3]         → [4, 4, 1, 8, 3]
[6, 3, 2, 9, 9, 2, 6, 2]      → [2, 9, 9, 2, 6, 2]
[0, 9, 4, 6, 8, 2, 5, 7, 1]   → [0, 9, 4, 6, 8, 2, 5]
[3, 7, 0, 5, 1, 4, 8, 6, 2, 9] → [3, 7, 0, 5, 1, 4, 8, 6]
```

0.242	6	c058	*remove the first 7 elements*

```
(λ x (drop 7 x))
```

```
[7, 2, 9, 5, 3, 6, 4, 5]       → [5]
[9, 8, 9, 8, 9, 8, 5, 0, 2, 5] → [0, 2, 5]
[3, 0, 1, 4, 8, 2, 7]          → [ ]
[2, 8, 6, 3, 9, 5, 7, 6, 4]    → [6, 4]
[9, 2, 4, 8, 0, 5, 3, 1, 7]    → [1, 7]
```

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.242	10	c174	*keep the first N elements, where N is the number of unique elements* `(λ x (take (length (unique x)) x))` [49, 32, 85, 49, 32, 2] → [49, 32, 85, 49] [29, 0, 77, 35, 50, 7, 53, 35, 8, 82] → [29, 0, 77, 35, 50, 7, 53, 35, 8] [66, 71, 9, 72, 11, 86, 91, 9] → [66, 71, 9, 72, 11, 86, 91] [67, 1, 24, 37, 5, 18, 67] → [67, 1, 24, 37, 5, 18] [9, 52, 96, 27, 83, 4, 42, 98, 4] → [9, 52, 96, 27, 83, 4, 42, 98]
0.238	10	c229	*keep only first N elements of the reversed input, N = element 1* `(λ x (take (first x) (reverse x)))` [7, 58, 5, 9, 21, 22, 51] → [51, 22, 21, 9, 5, 58, 7] [4, 89, 16, 33, 53, 3, 6, 1, 76] → [76, 1, 6, 3] [5, 18, 99, 7, 7, 99, 81, 11] → [11, 81, 99, 7, 7] [1, 3, 49, 5, 2, 15, 77, 68, 27, 13] → [13] [9, 2, 72, 56, 1, 0, 26, 69, 95, 86] → [86, 95, 69, 26, 0, 1, 56, 72, 2]
0.237	15	c162	*replace each element, M, with 3 * M + 7* `(λ x (map (λ y (+ 7 (* 3 y))) x))` [8, 0, 17, 5, 5, 0] → [31, 7, 58, 22, 22, 7] [3, 9, 1, 7, 2, 4, 8, 0, 15, 5] → [16, 34, 10, 28, 13, 19, 31, 7, 52, 22] [18, 1, 5, 11, 2, 1, 18] → [61, 10, 22, 40, 13, 10, 61] [8, 7, 3, 9, 5, 1, 5, 1, 4] → [31, 28, 16, 34, 22, 10, 22, 10, 19] [19, 3, 4, 1, 6, 2, 0, 9] → [64, 16, 19, 10, 25, 13, 7, 34]
0.231	34	c039	*append 3 if the list length is 3, else append 9 if the list length is 9* `(λ x (if (== (length x) 3) (append x 3) (if (== (length x) 9) (append x 9) x)))` [9, 3, 6] → [9, 3, 6, 3] [2, 1, 0, 1, 7, 8, 1, 8, 7] → [2, 1, 0, 1, 7, 8, 1, 8, 7, 9] [9, 1, 4] → [9, 1, 4, 3] [0, 5, 6, 5, 5] → [0, 5, 6, 5, 5] [4, 5, 8, 4, 0, 2, 8, 7, 2] → [4, 5, 8, 4, 0, 2, 8, 7, 2, 9]
0.23	26	c181	*reverse the order of elements with even indices* `(λ x (flatten (zip (filteri (λ y (λ z (is_odd y)) x)) (reverse (filteri (λ u (λ v (is_even u))) x)))))` [24, 99, 36, 61, 55, 6] → [24, 6, 36, 61, 55, 99] [1, 53, 21, 2, 57, 48, 74, 7] → [1, 7, 21, 48, 57, 2, 74, 53] [16, 97, 40, 26, 35, 65, 63, 59] → [16, 59, 40, 65, 35, 26, 63, 97] [4, 19, 51, 96, 33, 3] → [4, 3, 51, 96, 33, 19] [39, 50, 8, 82, 68, 52, 1, 89, 14, 5] → [39, 5, 8, 89, 68, 52, 1, 82, 14, 50]
0.23	15	c163	*replace each element, M, with 2 * M - 10* `(λ x (map (λ y (- (* y 2) 10)) x))` [7, 9, 8, 24, 23] → [4, 8, 6, 38, 36] [6, 8, 47, 6] → [2, 6, 84, 2] [7, 11, 7, 5] → [4, 12, 4, 0] [9, 5, 22, 18] → [8, 0, 34, 26] [33, 19, 29, 8] → [56, 28, 48, 6]
0.22	19	c154	*triple each even element* `(λ x (map (λ y (if (is_even y) (* 3 y) y)) x))` [5, 93, 14, 73, 4, 8] → [5, 93, 42, 73, 12, 24] [77, 0, 6, 8, 35, 7, 22, 21] → [77, 0, 18, 24, 35, 7, 66, 21] [81, 23, 89, 6, 9, 2, 1, 5, 55] → [81, 23, 89, 18, 9, 6, 1, 5, 55] [71, 75, 8, 1, 99, 6, 4] → [71, 75, 24, 1, 99, 18, 12] [6, 3, 16, 53, 20, 47, 69, 5, 33, 0] → [18, 3, 48, 53, 60, 47, 69, 5, 33, 0]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.215	21	c166	*unique elements in ascending order by the sum of their digits*

```
(λ x (sort (λ y (+ (% y 10) (/ y 10))) (unique x)))
```

[43, 20, 1, 20, 17, 55] → [1, 20, 43, 17, 55]
[92, 24, 11, 25, 21, 53, 25, 21] → [11, 21, 24, 25, 53, 92]
[86, 7, 63, 81, 9, 97, 41, 86, 3] → [3, 41, 7, 63, 81, 9, 86, 97]
[70, 70, 50, 70, 8, 50, 8, 50, 8, 8] → [50, 70, 8]
[58, 58, 58, 82, 58, 82, 58] → [82, 58]

0.209	20	c084	*remove all but element 7*

```
(λ x (if (> 7 (length x)) empty (singleton (nth 7 x))))
```

[2, 42] → [ ]
[90, 8, 5, 34, 79, 65, 8, 48, 79, 9] → [8]
[94, 0, 58, 62, 7, 3, 1, 55, 88] → [1]
[5, 5] → [ ]
[85, 18, 85, 91, 91, 18, 91] → [91]

0.2	46	c249	*use 0s to delimit sublists; give element 1 of each sublist*

```
(λ x (map first (reverse (fold (λ y (λ z (if (== z 0) (cons empty y) (cons (append (first y) z) (drop 1 y))))) (singleton empty) x))))
```

[64, 0, 50, 1, 50, 1, 0, 1, 64] → [64, 50, 1]
[8, 0, 4, 68, 68, 78, 0, 68, 25, 68] → [8, 4, 68]
[94, 9, 3, 5, 5, 0, 5, 0, 95] → [94, 5, 95]
[2, 3, 67, 0, 44, 0, 6, 91, 76] → [2, 44, 6]
[42, 37, 80, 0, 47, 13, 80, 0, 80, 42] → [42, 47, 80]

0.199	18	c118	*left-rotate by N elements, N = last element*

```
(λ x (concat (drop (last x) x) (take (last x) x)))
```

[24, 8, 57, 44, 4] → [4, 24, 8, 57, 44]
[97, 28, 30, 5, 48, 7, 2, 76, 9, 1] → [28, 30, 5, 48, 7, 2, 76, 9, 1, 97]
[18, 96, 25, 71, 99, 1] → [96, 25, 71, 99, 1, 18]
[39, 85, 2] → [2, 39, 85]
[5, 6, 22, 44, 90, 11, 3] → [44, 90, 11, 3, 5, 6, 22]

0.199	19	c152	*replace each element with the product of its digits*

```
(λ x (map (λ y (* (/ y 10) (% y 10))) x))
```

[37, 98, 4, 19, 82] → [21, 72, 0, 9, 16]
[87, 5, 77, 7, 54, 67, 8] → [56, 0, 49, 0, 20, 42, 0]
[59, 47, 46] → [45, 28, 24]
[65, 25] → [30, 10]
[88] → [64]

0.197	32	c237	*cumulative sum of unique elements*

```
(λ x (fold (λ y (λ z (append y (+ (last y) z)))) (take 1 (unique x)) (drop 1 (unique x))))
```

[14, 1, 3, 2, 14, 3] → [14, 15, 18, 20]
[7, 15, 2, 1, 0, 20, 20, 0, 4] → [7, 22, 24, 25, 25, 45, 49]
[1, 4, 1, 7, 0, 3, 2, 0] → [1, 5, 12, 12, 15, 17]
[0, 7, 4, 3, 8, 16, 16, 6, 2, 0] → [0, 7, 11, 14, 22, 38, 44, 46]
[6, 7, 8, 5, 7, 9, 7] → [6, 13, 21, 26, 35]

0.196	8	c243	*number of odd elements*

```
(λ x (singleton (count is_odd x)))
```

[6, 92, 34, 20, 69, 34, 0] → [1]
[19, 5, 4, 33, 17, 3, 19, 45, 93, 7] → [9]
[24, 21, 15, 94, 32, 47, 2, 3] → [4]
[2, 25, 18, 8, 46, 29, 38, 99] → [3]
[75, 8, 55, 5, 42, 62, 67, 89, 43] → [6]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.194	21	c188	*replace each element with 1 if it is divisible by 3, else 0*

```
(λ x (map (λ y (if (== (% y 3) 0) 1 0)) x))
```

```
[84, 9, 10, 11, 4]                → [1, 1, 0, 0, 0]
[79, 3, 48, 13, 53, 41, 22, 2, 5] → [0, 1, 1, 0, 0, 0, 0, 0, 0]
[95, 28, 86, 2, 21, 6, 40, 55]    → [0, 0, 0, 0, 1, 1, 0, 0]
[90, 60]                          → [1, 1]
[38, 51, 9]                       → [0, 1, 1]
```

0.192	14	c139	*remove all occurrences of the smallest and largest elements*

```
(λ x (cut_vals (max x) (cut_vals (min x) x)))
```

```
[28, 97, 22, 97, 22]                → [28]
[0, 85, 96, 30]                     → [85, 30]
[20, 45, 76, 66, 92, 52, 7, 8]      → [20, 45, 76, 66, 52, 8]
[0, 4, 38, 1, 88, 88, 62, 2, 81, 87] → [4, 38, 1, 62, 2, 81, 87]
[14, 13, 91, 43, 13, 17]            → [14, 43, 17]
```

0.189	28	c036	*swap elements 2 and 3 if element 2 > element 3, else swap elements 1 and 4*

```
(λ x (if (> (second x) (third x)) (swap 2 3 x) (swap 1 4 x)))
```

```
[1, 4, 0, 4]                  → [1, 0, 4, 4]
[6, 0, 8, 2, 5, 2, 1, 7, 3, 4] → [2, 0, 8, 6, 5, 2, 1, 7, 3, 4]
[9, 6, 9, 6, 9, 6]            → [6, 6, 9, 9, 9, 6]
[5, 8, 5, 8, 3, 6, 1]         → [5, 5, 8, 8, 3, 6, 1]
[0, 4, 1, 9, 7]               → [0, 1, 4, 9, 7]
```

0.186	28	c201	*elements > element 1, followed by elements < element 1*

```
(λ x (concat (filter (λ y (< (first x) y)) x) (filter (λ z (> (first x) z)) x)))
```

```
[9, 0, 73, 25, 4]                  → [73, 25, 0, 4]
[57, 62, 34, 54, 3, 6, 75, 8, 91, 99] → [62, 75, 91, 99, 34, 54, 3, 6, 8]
[51, 20, 95, 39, 52, 53, 78, 2]    → [95, 52, 53, 78, 20, 39, 2]
[40]                               → [ ]
[43, 41, 22, 48, 77, 82, 18]       → [48, 77, 82, 41, 22, 18]
```

0.182	11	c217	*elements in ascending order by ones digits*

```
(λ x (sort (λ y (% y 10)) x))
```

```
[34, 33, 80, 4, 79]          → [80, 33, 34, 4, 79]
[36, 72, 45, 67, 50, 90]     → [50, 90, 72, 45, 36, 67]
[66, 52, 15, 32]             → [52, 32, 15, 66]
[83, 30, 28, 38, 21, 0, 5]   → [30, 0, 21, 83, 5, 28, 38]
[63, 42, 1, 69, 61, 75, 46]  → [1, 61, 42, 63, 75, 46, 69]
```

0.178	14	c005	*remove all but element N + 1, N = element 1*

```
(λ x (singleton (nth (first x) (drop 1 x))))
```

```
[2, 1, 9, 6, 7, 0, 4, 5, 3]    → [9]
[7, 2, 1, 8, 0, 6, 3, 5, 9, 4] → [5]
[5, 1, 7, 6, 9, 8, 2, 0, 3, 4] → [8]
[9, 1, 6, 4, 7, 5, 3, 8, 2, 0] → [0]
[4, 1, 9, 6, 3, 2, 5, 0, 8, 7] → [3]
```

0.178	21	c209	*elements in ascending order; insert sum of smallest and largest elements at index 3*

```
(λ x (insert (+ (max x) (min x)) 3 (sort (λ y y) x)))
```

```
[34, 2, 3, 96, 64]       → [2, 3, 98, 34, 64, 96]
[87, 76, 1, 38, 85, 83]  → [1, 38, 88, 76, 83, 85, 87]
[9, 67, 94, 5]           → [5, 9, 99, 67, 94]
[39, 86, 23, 8, 7, 31]   → [7, 8, 93, 23, 31, 39, 86]
[25, 72, 49]             → [25, 49, 97, 72]
```

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.174	12	c063	*remove the first N + 1 elements, N = element 1* `(λ x (drop (first x) (drop 1 x)))` [2, 6, 2, 6, 6, 6] → [6, 6, 6] [4, 1, 1, 4, 1, 1, 3] → [1, 3] [3, 3, 3, 3, 3, 3, 3, 3, 3, 3] → [3, 3, 3, 3, 3, 3] [1, 7, 9, 9, 8, 4, 1, 7, 8] → [9, 9, 8, 4, 1, 7, 8] [0, 3, 9, 4, 6, 6, 7, 8, 2] → [3, 9, 4, 6, 6, 7, 8, 2]
0.161	16	c141	*replace element M + 2 with N, M = element 1, N = element 2; remove elements 1 and 2* `(λ x (replace (first x) (second x) (drop 2 x)))` [4, 3, 12, 6, 67, 1, 9] → [12, 6, 67, 3, 9] [3, 1, 2, 10, 90, 6, 0, 76] → [2, 10, 1, 6, 0, 76] [5, 7, 33, 0, 71, 9, 78, 4, 2, 66] → [33, 0, 71, 9, 7, 4, 2, 66] [1, 0, 1, 39, 49, 14, 90, 57, 0, 99] → [0, 39, 49, 14, 90, 57, 0, 99] [2, 4, 59, 62, 5, 6, 36, 45, 64] → [59, 4, 5, 6, 36, 45, 64]
0.155	28	c089	*swap elements 2 and 3 if element 2 > element 3, else swap elements 1 and 4* `(λ x (if (> (second x) (third x)) (swap 2 3 x) (swap 1 4 x)))` [90, 7, 14, 59, 53, 24, 25] → [59, 7, 14, 90, 53, 24, 25] [28, 99, 6, 43, 41, 28] → [28, 6, 99, 43, 41, 28] [96, 8, 51, 44, 20] → [44, 8, 51, 96, 20] [58, 65, 32, 63] → [58, 32, 65, 63] [78, 97, 77, 93, 64, 3, 38, 18, 0, 24] → [78, 77, 97, 93, 64, 3, 38, 18, 0, 24]
0.148	14	c124	*element M, M = element N, N = element 1* `(λ x (singleton (nth (nth (first x) x) x)))` [2, 5, 0, 82, 9] → [9] [4, 93, 98, 3, 1, 96] → [98] [3, 27, 5, 2, 8, 7, 97, 84, 42] → [8] [8, 25, 16, 35, 6, 1, 49, 3, 2, 55] → [16] [7, 61, 0, 56, 8, 92, 4, 2] → [56]
0.141	11	c216	*elements in ascending order by tens digits* `(λ x (sort (λ y (/ y 10)) x))` [77, 74, 26, 9, 31] → [9, 26, 31, 77, 74] [54, 86, 4, 66, 25, 13, 84] → [4, 13, 25, 54, 66, 86, 84] [91, 20, 3, 82] → [3, 20, 82, 91] [62, 32, 78, 53, 42, 8, 6] → [8, 6, 32, 42, 53, 62, 78] [22, 38, 58, 30, 92, 14] → [14, 22, 38, 30, 58, 92]
0.14	21	c143	*replace every occurrence of the largest element with the smallest element* `(λ x (map (λ y (if (== y (max x)) (min x) y)) x))` [9, 0, 49, 49, 3] → [9, 0, 0, 0, 3] [8, 82, 71, 25, 24, 31, 90, 34, 69, 6] → [8, 82, 71, 25, 24, 31, 6, 34, 69, 6] [9, 83, 0, 56, 18, 48, 61, 5, 12] → [9, 0, 0, 56, 18, 48, 61, 5, 12] [74, 8, 2, 74, 22, 4] → [2, 8, 2, 2, 22, 4] [52, 44, 16, 1, 6, 5, 52, 23] → [1, 44, 16, 1, 6, 5, 1, 23]
0.139	16	c215	*replace element M + 1 with the length of the input, M = element 1; drop element 1* `(λ x (replace (first x) (length x) (drop 1 x)))` [3, 59, 55, 17, 3] → [59, 55, 5, 3] [8, 4, 0, 43, 95, 70, 1, 34, 0, 4] → [4, 0, 43, 95, 70, 1, 34, 10, 4] [5, 74, 35, 35, 36, 78, 36, 35] → [74, 35, 35, 36, 8, 36, 35] [5, 31, 60, 84, 7, 89, 96] → [31, 60, 84, 7, 7, 96] [6, 25, 8, 72, 79, 27, 24, 2, 94] → [25, 8, 72, 79, 27, 9, 2, 94]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.136	37	c144	*replace every occurrence of the largest or smallest element with their absolute difference*

```
(λ x (map (λ y (if (or (== y (max x)) (== y (min x))) (- (max x) (min x)) y)) x))
```

[60, 87, 71, 94, 35] → [60, 87, 71, 59, 59]
[11, 90, 4, 16] → [11, 86, 86, 16]
[14, 52, 5] → [14, 47, 47]
[23, 77] → [54, 54]
[22] → [0]

0.129	15	c128	*elements in ascending order after removing elements 1, 2, and 5*

```
(λ x (sort (λ y y) (cut_idx 3 (drop 2 x))))
```

[14, 46, 41, 44, 85, 5] → [5, 41, 44]
[34, 74, 96, 40, 9, 7, 58, 2] → [2, 7, 40, 58, 96]
[69, 8, 38, 91, 83, 54, 10, 17, 51] → [10, 17, 38, 51, 54, 91]
[3, 1, 28, 0, 6, 93, 90, 9, 56, 45] → [0, 9, 28, 45, 56, 90, 93]
[26, 49, 8, 4, 21, 16, 63] → [4, 8, 16, 63]

0.127	8	c208	*keep only the last N elements, where N is the last element*

```
(λ x (takelast (last x) x))
```

[3, 1, 5, 35, 65, 7, 84, 4] → [65, 7, 84, 4]
[28, 69, 5, 58, 51, 66, 16, 9, 7] → [5, 58, 51, 66, 16, 9, 7]
[23, 9, 22, 0, 76, 3, 5, 75, 2, 1] → [1]
[2, 24, 92, 14, 95, 54, 79, 42, 71, 8] → [92, 14, 95, 54, 79, 42, 71, 8]
[94, 8, 68, 86, 7, 95, 56, 0, 6] → [86, 7, 95, 56, 0, 6]

0.124	28	c035	*swap elements 1 and 4 if element 2 = element 3, else swap elements 2 and 3*

```
(λ x (if (== (second x) (third x)) (swap 1 4 x) (swap 2 3 x)))
```

[0, 5, 0, 0] → [0, 0, 5, 0]
[7, 1, 7, 7, 1, 7, 1] → [7, 7, 1, 7, 1, 7, 1]
[3, 3, 3, 2, 2, 6] → [2, 3, 3, 3, 2, 6]
[9, 9, 8, 9, 8] → [9, 8, 9, 9, 8]
[5, 5, 4, 5, 6, 6, 4, 4, 4, 6] → [5, 4, 5, 5, 6, 6, 4, 4, 4, 6]

0.118	12	c214	*remove elements 1 and N+1, where N is element 1*

```
(λ x (cut_idx (first x) (drop 1 x)))
```

[4, 47, 54, 54, 4, 4] → [47, 54, 54, 4]
[7, 6, 30, 64, 8, 41, 41, 36, 56] → [6, 30, 64, 8, 41, 41, 56]
[2, 7, 23, 63, 63, 23, 93, 78] → [7, 63, 63, 23, 93, 78]
[3, 3, 21, 32, 21, 21, 98] → [3, 21, 21, 21, 98]
[9, 27, 19, 1, 64, 61, 61, 67, 27, 65] → [27, 19, 1, 64, 61, 61, 67, 27]

0.116	12	c130	*elements 2 through N + 1, N = element 1*

```
(λ x (take (first x) (drop 1 x)))
```

[5, 0, 24, 4, 41, 18, 9] → [0, 24, 4, 41, 18]
[1, 75, 48, 90, 5, 9, 4, 21, 59] → [75]
[0, 96, 6, 62, 83, 96, 43, 25, 52, 43] → [ ]
[3, 81, 0, 8, 87, 62, 6, 10] → [81, 0, 8]
[7, 9, 6, 4, 15, 45, 88, 83, 26, 92] → [9, 6, 4, 15, 45, 88, 83]

0.115	24	c159	*element N counts the number of occurrences of N, up to the largest element*

```
(λ x (map (λ y (count (λ z (== y z)) x)) (range 1 1 (max x))))
```

[8, 8, 8, 6, 5, 5, 5, 6] → [0, 0, 0, 0, 3, 2, 0, 3]
[3, 4, 3, 5, 4, 4, 5, 5, 4, 3] → [0, 0, 3, 4, 3]
[7, 6, 6, 6, 6, 2, 2, 7, 6] → [0, 2, 0, 0, 0, 5, 2]
[4, 4, 7, 7, 9, 4, 9, 2, 2] → [0, 2, 0, 3, 0, 0, 2, 0, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1] → [10]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.114	19	c205	*product of elements divisible by 4* `(λ x (singleton (product (filter (λ y (== (% y 4) 0)) x))))` [4, 1, 9, 20, 3, 11, 70] → [80] [99, 46, 7, 5, 2, 8, 5, 12] → [96] [15, 73, 98, 8, 53, 1, 95, 9, 4] → [32] [67, 19, 23, 7, 2, 12, 8, 23, 97, 6] → [96] [83, 37, 16, 21, 0, 6, 2, 87, 7] → [0]
0.112	30	c076	*the maximum, last element, length, first element, and minimum, in that order* `(λ x (cons (max x) (cons (last x) (cons (length x) (cons (first x) (singleton (min x)))))))` [5, 7, 9, 4] → [9, 4, 4, 5, 4] [0, 6, 1, 9, 7, 8, 4, 2, 5, 3] → [9, 3, 10, 0, 0] [6] → [6, 6, 1, 6, 6] [8, 2] → [8, 2, 2, 8, 2] [3, 5, 4, 0, 8, 7, 1] → [8, 1, 7, 3, 0]
0.107	48	c250	*use 0s to delimit sublists; concatenate reverse of each sublist* `(λ x (flatten (map reverse (reverse (fold (λ y (λ z (if (== z 0) (cons empty y) (cons (append (first y) z) (drop 1 y))))) (singleton empty) x)))))` [38, 0, 38, 3, 60, 60, 0, 3, 38] → [38, 60, 60, 3, 38, 38, 3] [4, 8, 0, 77, 0, 25, 66, 77, 40, 66] → [8, 4, 77, 66, 40, 77, 66, 25] [7, 27, 0, 8, 5, 0, 86, 24, 4, 4] → [27, 7, 5, 8, 4, 4, 24, 86] [6, 14, 75, 0, 1, 75, 75, 0, 0] → [75, 14, 6, 75, 75, 1] [2, 0, 62, 80, 54, 49, 0, 6, 4] → [2, 49, 54, 80, 62, 4, 6]
0.103	13	c186	*keep only elements greater than element 1* `(λ x (filter (λ y (> y (first x))) x))` [6, 71, 97, 98, 0, 64, 60, 3, 0] → [71, 97, 98, 64, 60] [79, 7, 32, 2, 8, 86, 93, 51, 95, 5] → [86, 93, 95] [31, 30, 53, 74, 9, 12, 12, 5, 31, 31] → [53, 74] [77, 20, 26, 99, 12, 81, 27, 90, 90, 77] → [99, 81, 90, 90] [72, 17, 4, 50, 80, 17, 37, 29, 57] → [80]
0.098	25	c242	*take the largest unique element, append the second largest unique element, prepend the third largest unique element, append the fourth largest unique element, and so on* `(λ x (fold (λ y (λ z (append (reverse y) z))) empty (reverse (unique (sort (λ u u) x)))))` [43, 43, 43, 17, 17, 4, 17, 17] → [17, 43, 4] [32, 81, 53, 32, 1, 32, 81, 53, 81, 1] → [32, 81, 53, 1] [66, 0, 6, 66, 31, 66, 28, 28] → [6, 31, 66, 28, 0] [60, 92, 9, 18, 35, 9, 10, 60, 10, 92] → [10, 35, 92, 60, 18, 9] [51, 46, 74, 46, 75, 48, 89, 89, 51] → [48, 74, 89, 75, 51, 46]
0.091	11	c160	*replace each element, M, with 99 - M* `(λ x (map (λ y (- 99 y)) x))` [55, 82, 57, 4, 59, 6] → [44, 17, 42, 95, 40, 93] [1, 13, 34, 6, 36, 79, 21, 11, 64, 46] → [98, 86, 65, 93, 63, 20, 78, 88, 35, 53] [32, 70, 51, 41, 5, 69, 28] → [67, 29, 48, 58, 94, 30, 71] [9, 72, 7, 75, 74, 1, 79, 84, 2] → [90, 27, 92, 24, 25, 98, 20, 15, 97] [8, 66, 3, 43, 2, 73, 48, 74] → [91, 33, 96, 56, 97, 26, 51, 25]
0.073	15	c167	*keep only elements divisible by 3* `(λ x (filter (λ y (== (% y 3) 0)) x))` [18, 78, 3, 5, 62, 8] → [18, 78, 3] [27, 70, 0, 21, 74, 33, 87, 12, 22] → [27, 0, 21, 33, 87, 12] [31, 91, 84, 30, 38, 10, 66, 0, 98, 94] → [84, 30, 66, 0] [90, 50, 3, 1, 99, 0, 93] → [90, 3, 99, 0, 93] [45, 8, 2, 69, 39, 9, 44, 4] → [45, 69, 39, 9]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.068	6	c202	*the indices of every even number* `(λ x (find is_even x))` [31, 98, 55, 14, 50] → [2, 4, 5] [71, 7, 23, 20, 13, 66, 26, 42, 52] → [4, 6, 7, 8, 9] [91, 78, 5, 46, 15, 33, 0, 62] → [2, 4, 7, 8] [73, 41, 58, 24] → [3, 4] [90, 60, 25] → [1, 2]
0.056	16	c129	*elements M + 2 through N + 2, M = element 1, N = element 2* `(λ x (slice (first x) (second x) (drop 2 x)))` [2, 3, 75, 17, 8, 6] → [17, 8] [1, 4, 99, 5, 4, 14, 73, 28] → [99, 5, 4, 14] [4, 6, 37, 8, 1, 31, 7, 69, 62, 67] → [31, 7, 69] [4, 5, 20, 66, 61, 19, 16] → [19, 16] [6, 6, 70, 27, 86, 99, 7, 30, 66] → [30]
0.048	10	c221	*sum of even elements* `(λ x (singleton (sum (filter is_even x))))` [46, 91, 4, 9, 1, 67, 5] → [50] [65, 44, 3, 1, 91, 7, 41, 43, 20] → [64] [55, 26, 34, 95, 19, 6, 0, 79, 8, 53] → [74] [81, 21, 0, 32, 16, 4, 2, 28, 33] → [82] [6, 13, 7, 10, 47, 75, 80, 93] → [96]
0.045	13	c131	*keep only elements whose tens digit is even* `(λ x (filter (λ y (is_even (/ y 10))) x))` [68, 93, 10, 24, 5] → [68, 24, 5] [54, 99, 84, 58] → [84] [3, 38, 7, 16, 78, 51, 41, 12, 67, 77] → [3, 7, 41, 67] [95, 37, 73, 97, 97, 56, 97, 97] → [ ] [94, 0, 9, 5, 6, 33, 4, 59, 54] → [0, 9, 5, 6, 4]
0.045	21	c180	*take the smallest element, append the second smallest, prepend the third smallest, append the fourth smallest, and so on* `(λ x (fold (λ y (λ z (append (reverse y) z))) empty (sort (λ u u) x)))` [43, 6, 18, 33, 8, 38] → [38, 18, 6, 8, 33, 43] [11, 7, 56, 84, 4, 60, 5] → [60, 11, 5, 4, 7, 56, 84] [1, 67, 23, 63, 59, 36, 45, 21, 5] → [63, 45, 23, 5, 1, 21, 36, 59, 67] [93, 3, 48, 34, 21, 83, 14, 66, 65, 80] → [83, 66, 48, 21, 3, 14, 34, 65, 80, 93] [92, 94, 57, 2, 98, 89, 13, 28] → [94, 89, 28, 2, 13, 57, 92, 98]
0.042	16	c015	*elements M + 2 through N + 2, M = element 1, N = element 2* `(λ x (slice (first x) (second x) (drop 2 x)))` [3, 4, 5, 5, 4, 3, 3] → [4, 3] [1, 3, 9, 2, 0, 5, 7, 5, 7, 1] → [9, 2, 0] [4, 5, 6, 6, 4, 6, 9, 9] → [6, 9] [1, 1, 3, 2, 3, 4, 5, 6, 5, 1] → [3] [1, 5, 8, 3, 6, 2, 4, 8, 0] → [8, 3, 6, 2, 4]
0.04	23	c179	*take the largest element, append the second largest, prepend the third largest, append the fourth largest, and so on* `(λ x (fold (λ y (λ z (append (reverse y) z))) empty (reverse (sort (λ u u) x))))` [80, 68, 56, 38, 97, 92] → [56, 80, 97, 92, 68, 38] [3, 81, 25, 8, 5, 7, 41, 75, 39] → [5, 8, 39, 75, 81, 41, 25, 7, 3] [46, 85, 95, 0, 38, 4, 66, 35] → [4, 38, 66, 95, 85, 46, 35, 0] [53, 27, 30, 13, 1, 18, 55] → [13, 27, 53, 55, 30, 18, 1] [11, 4, 36, 71, 19, 2, 90, 6, 10, 86] → [4, 10, 19, 71, 90, 86, 36, 11, 6, 2]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0.04	14	c226	*remove first M and last N elements, M = element 1, N = last element* `(λ x (drop (first x) (droplast (last x) x)))` [1, 15, 15, 1, 15, 1, 1] → [15, 15, 1, 15, 1] [6, 67, 67, 67, 6, 67, 1, 1] → [1] [5, 64, 64, 5, 83, 83, 83, 83, 8, 5] → [ ] [1, 14, 61, 49, 2, 12, 98, 4, 4] → [14, 61, 49, 2] [1, 5, 3, 96, 37, 35, 68, 5, 23, 0] → [5, 3, 96, 37, 35, 68, 5, 23, 0]
0.034	12	c134	*remove elements M through N, M = element 1, N = element 2* `(λ x (cut_slice (first x) (second x) x))` [3, 4, 9, 6, 91] → [3, 4, 91] [2, 9, 3, 29, 19, 61, 23, 59, 66, 76] → [2, 76] [3, 5, 31, 85, 37, 9, 4] → [3, 5, 9, 4] [1, 8, 0, 65, 95, 28, 3, 7] → [ ] [2, 4, 0, 7] → [2]
0.03	15	c164	*replace each element, M, with M / 4 + 5* `(λ x (map (λ y (+ (/ y 4) 5)) x))` [22, 14, 26, 39, 26, 13] → [10, 8, 11, 14, 11, 8] [6, 50, 18, 72, 7, 84, 94, 0, 46, 8] → [6, 17, 9, 23, 6, 26, 28, 5, 16, 7] [31] → [12] [55, 3, 92, 85, 63, 58, 33, 67, 48] → [18, 5, 28, 26, 20, 19, 13, 21, 17] [8, 2, 42, 59, 95, 97, 3] → [7, 5, 15, 19, 28, 29, 5]
0.02	27	c178	*keep only elements followed by an even number* `(λ x (map first (filter (λ y (is_even (second y))) (zip (droplast 1 x) (drop 1 x)))))` [27, 4, 9, 71, 45, 69] → [27] [10, 68, 80, 5, 29, 23, 9, 33, 69] → [10, 68] [73, 32, 70, 0, 22, 2, 46, 8, 7, 92] → [73, 32, 70, 0, 22, 2, 46, 7] [19, 81, 1, 53, 85, 3, 97] → [ ] [40, 2, 91, 28, 61, 0, 55, 4] → [40, 91, 61, 55]
0.018	17	c113	*keep only elements whose ones digit is greater than element 1* `(λ x (filter (λ y (> (first x) (% y 10))) x))` [3, 91, 59, 91, 60, 6, 44, 2] → [91, 91, 60, 2] [7, 35, 37, 74, 73, 22, 85, 8, 68, 7] → [35, 74, 73, 22, 85] [7, 38, 1, 29, 40, 48, 45, 81] → [1, 40, 45, 81] [2, 93, 68, 36, 41, 8, 27, 20, 8, 50] → [41, 20, 50] [4, 82, 5, 52, 83, 7, 5, 4, 9] → [82, 52, 83]
0.013	31	c206	*keep only elements whose value is between the first two elements* `(λ x (filter (λ y (and (> (max (take 2 x)) y) (> y (min (take 2 x))))) x))` [43, 3, 80, 40, 29, 31] → [40, 29, 31] [1, 63, 2, 7, 48, 9, 97, 4] → [2, 7, 48, 9, 4] [39, 80, 95, 9, 44, 77, 2, 33, 75, 6] → [44, 77, 75] [6, 90, 25, 9, 18, 0, 7] → [25, 9, 18, 7] [87, 5, 71, 7, 3, 19, 8, 22, 56] → [71, 7, 19, 8, 22, 56]
0.01	14	c210	*unique elements with last element inserted at index M, where M is element 1* `(λ x (insert (last x) (first x) (unique x)))` [2, 9, 57, 6, 9, 6] → [2, 6, 9, 57, 6] [4, 0, 50, 4, 19, 34, 50, 34, 4, 19] → [4, 0, 50, 19, 19, 34] [8, 79, 23, 60, 74, 49, 71, 0, 76] → [8, 79, 23, 60, 74, 49, 71, 76, 0, 76] [2, 99, 87, 2, 87, 99, 87] → [2, 87, 99, 87] [7, 65, 3, 68, 73, 66, 9, 5] → [7, 65, 3, 68, 73, 66, 5, 9, 5]

μ	$\mathcal{L}$	ID	Description, Program, & Examples
0	18	c158	*replace each element with 1 if element N equals N, else 0* `(λ x (mapi (λ y (λ z (if (== z y) 1 0))) x))` [1, 45, 3, 4, 23, 55] → [1, 0, 1, 1, 0, 0] [4, 68, 3, 68, 24, 8, 7, 8] → [0, 0, 1, 0, 0, 0, 1, 1] [84, 2, 29, 7, 35, 2, 4, 8, 9, 31] → [0, 1, 0, 0, 0, 0, 0, 1, 1, 0] [9, 3, 8, 6, 7, 5, 2] → [0, 0, 0, 0, 0, 0, 0] [16, 20, 67, 4, 63, 1, 7, 6, 9, 4] → [0, 0, 0, 1, 0, 0, 1, 0, 1, 0]
0	14	c183	*list the index minus 1 of elements 2 and following equal to element 1* `(λ x (find (== (first x)) (drop 1 x)))` [3, 4, 3, 3, 3, 9, 5, 8] → [2, 3, 4] [8, 7, 2, 1, 1, 2, 8, 9, 6] → [6] [2, 2, 2, 2, 5, 2, 5, 4, 4, 5] → [1, 2, 3, 5] [6, 3, 6, 6, 9, 9, 3, 6, 6, 6] → [2, 3, 7, 8, 9] [2, 12, 2, 49, 8, 2, 65, 83, 36] → [2, 5]

References

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs*. MIT Press.

Abend, O., Kwiatkowski, T., Smith, N. J., Goldwater, S., & Steedman, M. (2017). Bootstrapping language acquisition. *Cognition*, *164*, 116–143.

Akiba, T., Imajo, K., Iwami, H., Iwata, Y., Kataoka, T., Takahashi, N., Moskal, M., & Swamy, N. (2013). *Calibrating research in program synthesis using 72,000 hours of programmer time* (tech. rep.). Microsoft Research.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., & Udupa, A. (2013). Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, 1–8.

Amalric, M., Wang, L., Pica, P., Figueira, S., Sigman, M., & Dehaene, S. (2017). The language of geometry: Fast comprehension of geometrical primitives and rules in human adults and preschoolers. *PLoS Computational Biology*. https://doi.org/10.1371/journal.pcbi.1005273

Andreessen, M. (2011). Why software is eating the world. *Wall Street Journal*, *20*(2011), C2.

Ashcraft, M. H. (1982). The development of mental arithmetic: A chronometric approach. *Developmental Review*, *2*(3), 213–236.

Ashcraft, M. H. (1987). Children's knowledge of simple arithmetic: A developmental model and simulation. In J. Bisanz, C. Brainerd, & R. Kail (Eds.), *Formal methods in developmental psychology* (pp. 302–338). Springer.

Baader, F., & Snyder, W. (2001). Unification theory. *Handbook of automated reasoning* (pp. 447–533). Gulf Professional.

Baader, F., & Nipkow, T. (1999). *Term rewriting and all that*. Cambridge University Press.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). Deepcoder: Learning to write programs. *Procedeedings of the Fifth International Conference on Learning Representations*.

Barendregt, H. P. et al. (1984). *The lambda calculus* (Vol. 3). North-Holland Amsterdam.

Barner, D. (2017). Language, procedures, and the non-perceptual origin of number word meanings. *Journal of Child Language*, *44*(3), 553–590.

Barner, D., & Baron, A. S. (2016). *Core knowledge and conceptual change*. Oxford University Press.

Baroody, A. J. (1984). The case of felicia: A young child's strategies for reducing memory demand during mental addition. *Cognition and Instruction*, *1*(1), 109–116.

Baroody, A. J., & Gannon, K. E. (1984). The development of the commutativity principle and economical addition strategies. *Cognition and Instruction*, *1*(3), 321–339.

Bartlett, F. (1958). *Thinking: An experimental and social study*. Basic Books.
Baum, E. B. (2004). *What is thought?* MIT Press.
Bezem, M., Klop, J. W., & de Vrijer, R. (Eds.). (2003). *Term rewriting systems*. Cambridge University Press.
Biermann, A. W. (1978). The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, *8*(8), 585–600.
Block, N. (1987). Advertisement for a semantics for psychology. *Midwest Studies in Philosophy*, *10*(1), 615–678. https://doi.org/10.1111/j.1475-4975.1987.tb00558.x
Block, N. (1997). Semantics, conceptual role. *The Routledge Encylopedia of Philosophy*.
Bloom, P., & Wynn, K. (1997). Linguistic cues in the acquisition of number words. *Journal of Child Language*, *24*(3), 511–533.
Bongard, M. M. (1967). *The problem of recogntion*.
Boole, G. (1854). *An investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities*. Dover Publications.
Bošnjak, M., Rocktäschel, T., Naradowsky, J., & Riedel, S. (2017). Programming with a differentiable forth interpreter. *International conference on machine learning*, 547–556.
Bratko, I. (2001). *Prolog programming for artificial intelligence*. Pearson education.
Briars, D., & Siegler, R. S. (1984). A featural analysis of preschoolers' counting knowledge. *Developmental Psychology*, *20*(4), 607–618.
Brodie, L. (2004). *Thinking forth*. Punchy Publishing.
Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1–43.
Bruner, J. S., Goodnow, J. J., & George, A. (1956). *A study of thinking*.
Bunel, R., Hausknecht, M., Devlin, J., Singh, R., & Kohli, P. (2018). Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*.
Burge, T. (2010). Steps toward origins of propositional thought. *Disputatio*, *4*(29), 1–29.
Calvo, P., & Symons, J. (2014). *The architecture of cognition: Rethinking fodor and pylyshyn's systematicity challenge*. MIT Press.
Carey, S. (1985). *Conceptual change in childhood*. MIT Press.
Carey, S. (2009). *The origin of concepts*. Oxford University Press.
Carey, S. (2015). Why theories of concepts should not ignore the problem of acquisition. In E. Margolis & S. Laurence (Eds.), *The conceptual mind: New directions in the study of concepts*. MIT Press.
Carey, S., & Bartlett, E. (1978). Acquiring a single new word.
Carey, S., & Spelke, E. (1994). Domain-specific knowledge and conceptual change. *Mapping the mind: Domain specificity in cognition and culture* (pp. 169–200). Cambridge University Press.
Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, *76*(1).

Carpenter, T. P., & Moser, J. M. (1984). The acquisition of addition and subtraction concepts in grades one through three. *Journal for Research in Mathematics Education*, *15*(3), 179–202.

Carpentier, A., & Valko, M. (2014). Extreme bandits. *Advances in Neural Information Processing Systems*, 1089–1097.

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive psychology*, *4*(1), 55–81.

Chater, N., & Oaksford, M. (2013). Programs as causal models: Speculations on mental programs and mental representation. *Cognitive Science*, *37*(6), 1171–1191.

Chater, N., & Vitányi, P. (2003). Simplicity: A unifying principle in cognitive science? *Trends in Cognitive Sciences*, *7*(1), 19–22.

Chen, X., Liu, C., & Song, D. (2019). Execution-guided neural program synthesis. *International Conference on Learning Representations*.

Cheung, P., Rubenson, M., & Barner, D. (2017). To infinity and beyond: Children generalize the successor function to all possible numbers years after learning to count. *Cognitive Psychology*, *92*, 22–36.

Cheyette, S., & Piantadosi, S. (2017). Knowledge transfer in a probabilistic language of thought. *Proceedings of the 39th Annual Conference of the Cognitive Science Society*.

Childs, B. E., Brodeur, J. H., & Kocsis, L. (2008). Transpositions and move groups in monte carlo tree search. *2008 IEEE Symposium On Computational Intelligence and Games*, 389–395.

Chollet, F. (2019). On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Chomsky, N. (1959). A review of bf skinner's verbal behavior. *Language*, *35*(1), 26–58.

Chomsky, N., Keyser, S. J. et al. (1988). *Language and problems of knowledge: The managua lectures* (Vol. 16). MIT press.

Chu, J., Cheung, P., Schneider, R. M., Sullivan, J., & Barner, D. (2020). Counting to infinity: Does learning the syntax of the count list predict knowledge that numbers are infinite?

Chu, J., Gauthier, J., Levy, R., Tenenbaum, J., & Schulz, L. (2019). Query-guided visual search. *Proceedings of the 41st Annual Conference of the Cognitive Science Society*.

Chu, J., & Schulz, L. (2020). Exploratory play, rational action, and efficient search [PsyArxiv preprint: 10.31234/osf.io/9yra2]. *Proceedings of the 42nd Annual Conference of the Cognitive Science Society*.

Church, A. (1932). A set of postulates for the foundation of logic. *The Annals of Mathematics*, *33*(2), 346. https://doi.org/10.2307/1968337

Cicirello, V. A., & Smith, S. F. (2005). The max k-armed bandit: A new model of exploration applied to search heuristic selection. *The Proceedings of the Twentieth National Conference on Artificial Intelligence*, *3*, 1355–1361.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to algorithms*. MIT Press.

Cropper, A., & Morel, R. (2020). Learning programs by learning from failures. *arXiv preprint arXiv:2005.02259*.

Cropper, A., Morel, R., & Muggleton, S. H. (2019). Learning higher-order logic programs. *arXiv preprint arXiv:1907.10953*.

Cropper, A., & Muggleton, S. H. (2015). Logical minimisation of meta-rules within meta-interpretive learning. *Inductive logic programming* (pp. 62–75). Springer.

Cropper, A., & Muggleton, S. H. (2016). Metagol system.

Cropper, A., & Muggleton, S. H. (2019). Learning efficient logic programs. *Machine Learning*, *108*(7), 1063–1083.

Csikszentmihalyi, M. (1990). *Flow: The psychology of happiness*. Harper & Row.

Curry, H. B. (1930). Grundlagen der kombinatorischen logik. *American Journal of Mathematics*, *52*(3), 509. https://doi.org/10.2307/2370619

Davidson, K., Eng, K., & Barner, D. (2012). Does learning to count involve a semantic induction? *Cognition*, *123*(1), 162–173.

Dechter, E., Malmaud, J., Adams, R. P., & Tenenbaum, J. B. (2013). Bootstrap learning via modular concept discovery. *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 1302–1309.

Depeweg, S., Rothkopf, C. A., & Jäkel, F. (2018). Solving bongard problems with a visual language and pragmatic reasoning. *arXiv preprint arXiv:1804.04452*.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., & Kohli, P. (2017). RobustFill: Neural program learning under noisy I/O. *Proceedings of the 34th International Conference on Machine Learning*.

Dreyfus, H. L. (1992). *What computers still can't do: A critique of artificial reason*. MIT press.

Ellerman, M. (2020). linux-fullhistory: Full history of Linux created by Yoann Padioleau and Rob Landley.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning libraries of subroutines for neurally–guided Bayesian program induction. *Advances in Neural Information Processing Systems*, 7816–7826.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., & Solar-Lezama, A. (2019). Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 9165–9174.

Ellis, K., Solar-Lezama, A., & Tenenbaum, J. (2016). Sampling for bayesian program learning. *Advances in Neural Information Processing Systems*, 1297–1305.

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

Erdogan, G., Yildirim, I., & Jacobs, R. A. (2015). From sensory signals to modality-independent conceptual representations: A probabilistic language of thought approach. *PLoS Computational Biology*. https://doi.org/10.1371/journal.pcbi.1004610

Ericsson, K. A. (2006). The influence of experience and deliberate practice on the development of superior expert performance. *The Cambridge handbook of expertise and expert performance* (pp. 683–704). Cambridge University Press. https://doi.org/10.1017/cbo9780511816796.038

Fedorenko, E., Ivanova, A., Dhamala, R., & Bers, M. U. (2019). The language of programming: A cognitive perspective. *Trends in Cognitive Sciences*, *23*(7), 525–528.

Feldman, J. (2000). Minimization of Boolean complexity in human concept learning. *Nature*, *407*(6804), 630–633.

Feldman, J. (2003). The simplicity principle in human concept learning. *Current Directions in Psychological Science*, *12*(6), 227–232.

Feser, J. K., Brockschmidt, M., Gaunt, A. L., & Tarlow, D. (2017). Neural functional programming.

Feser, J. K., Chaudhuri, S., & Dillig, I. (2015). Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, *50*(6), 229–239.

Feyerabend, P. (1962). *Knowledge without foundations*. Oberlin College.

Field, H. H. (1977). Logic, meaning, and conceptual role. *The Journal of Philosophy*, *74*(7), 379–409.

Flener, P., & Schmid, U. (2008). An introduction to inductive programming. *AI Review*, *29*(1), 45–62.

Fodor, J. (1975). *The Language of Thought*. Harvard University Press.

Fodor, J., & Pylyshyn, Z. (1988). Connectionism and cognitive architecture: A critical analysis, connections and symbols. *Cognition*, *28*, 3–71.

Fodor, J. A. (1980). On the impossibility of acquiring "more powerful" structures. In M. Piattelli-Palmarini (Ed.), *Language and learning: The debate between Jean Piaget and Noam Chomsky* (pp. 142–149). Harvard University Press.

Fodor, J. A. (1983). *The modularity of mind*. MIT press.

Fodor, J. A. (2008). *Lot 2: The language of thought revisited*. Oxford University Press on Demand.

Fodor, J. A., & Lepore, E. (1992). *Holism: A shopper's guide*. Blackwell.

Foundalis, H. E. (2006). *Phaeaco: A cognitive architecture inspired by bongard's problems.* (Doctoral dissertation). University of Indiana.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Frank, M., & Goodman, N. (2012). Predicting pragmatic reasoning in language games. *Science*, *336*(6084), 998.

Fuson, K. C. (1992). Research on whole number addition and subtraction. In D. A. Grouws (Ed.), *Handbook of research on mathematics teaching and learning* (pp. 243–275). Macmillan.

Fuson, K. C., Richards, J., & Briars, D. J. (1982). The acquisition and elaboration of the number word sequence. In C. J. Brainerd (Ed.), *Children's logical and mathematical cognition* (pp. 33–92). Springer-Verlag.

Fuson, K. (1988). *Children's counting and concepts of number*. Springer.

Gallistel, C. R. (2017). The neurobiological bases for the computational theory of mind. *Minds on language and thought: The status of cognitive science and its prospects*. Oxford University Press.

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., & Tarlow, D. (2016). Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*.

Gauthier, J., Levy, R. P., & Tenenbaum, J. B. (2019). A rational model of syntactic bootstrapping. *Proceedings of the 41st Annual Conference of the Cognitive Science Society*.

Geary, D. C., & Burlingham-Dubree, M. (1989). External validation of the strategy choice model for addition. *Journal of Experimental Child Psychology*, *47*(2), 175–192.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive science*, *7*(2), 155–170.

Gerstenberg, T., & Tenenbaum, J. B. (2017). Intuitive theories. *Oxford handbook of causal reasoning*, 515–548.

Gibbons, J. (2003). Origami programming. In J. Gibbons & O. de Moor (Eds.), *The fun of programming*. Macmillan Education.

Gick, M. L., & Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology*, *12*(3), 306–355.

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology*, *15*(1), 1–38.

Gigerenzer, G., & Murray, D. J. (1987). *Cognition as intuitive statistics*. Psychology Press.

Gleitman, L. R., Gleitman, H., & Shipley, E. F. (1977). The emergence of the child as grammarian. In M. H. Appel & L. S. Goldberg (Eds.), *Topics in cognitive development* (pp. 91–117). Springer.

Gold, E. M. (1967). Language identification in the limit. *Information and control*, *10*(5), 447–474.

Goldman, S. R., Mertz, D. L., & Pellegrino, J. W. (1989). Individual differences in extended practice functions and solution strategies for basic addition facts. *Journal of Educational Psychology*, *81*(4), 481–496.

Goodman, N., Tenenbaum, J., Feldman, J., & Griffiths, T. (2008). A rational analysis of rule-based concept learning. *Cognitive Science*, *32*(1), 108–154.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., & Tenenbaum, J. B. (2008). Church: A language for generative models. In D. McAllester & P. Myllymaki (Eds.), *Proceedings of the 24th conference conference on uncertainty in artificial intelligence*.

Goodman, N., Tenenbaum, J. B., & Gerstenberg, T. (2015). Concepts in a probabilistic language of thought. In E. Margolis & S. Laurence (Eds.), *The conceptual mind: New directions in the study of concepts* (pp. 623–654). MIT Press.

Goodman, N. D., & Frank, M. C. (2016). Pragmatic language interpretation as probabilistic inference. *Trends in Cognitive Sciences*, *20*, 818–829.

Goodman, N. D., & Lassiter, D. (2015). Probabilistic semantics and pragmatics: Uncertainty in language and thought. In S. Lappin & C. Fox (Eds.), *The handbook of contemporary semantic theory* (2nd). Wiley-Blackwell.

Goodman, N. D., Ullman, T. D., & Tenenbaum, J. B. (2011). Learning a theory of causality. *Psychological Review*, *118*(1), 110–119.

Goodwin, G. P., & Johnson-Laird, P. N. (2013). The acquisition of boolean concepts. *Trends in cognitive sciences*, *17*(3), 128–133.

Gopnik, A., & Meltzoff, A. (1997). *Words, thoughts, and theories*. MIT Press.

Gopnik, A. (1983). Conceptual and semantic change in scientists and children: Why there are no semantic universals. *Linguistics*, *21*(1), 163–180.

Gopnik, A. (1996). The scientist as child. *Philosophy of Science*, *63*(4), 485–514.

Gopnik, A. (2012). Scientific thinking in young children: Theoretical advances, empirical research, and policy implications. *Science*, *337*(6102), 1623–1627.

Gopnik, A., Glymour, C., Sobel, D. M., Schulz, L. E., Kushnir, T., & Danks, D. (2004). A theory of causal learning in children: Causal maps and bayes nets. *Psychological Review*, *111*(1), 1–30.

Gopnik, A., & Schulz, L. (2004). Mechanisms of theory formation in young children. *Trends in Cognitive Sciences*, *8*(8), 371–377.

Gopnik, A., & Tenenbaum, J. B. (2007). Bayesian networks, bayesian learning and cognitive development. *Developmental Science*, *10*(3), 281–287.

Gopnik, A., & Wellman, H. M. (2012). Reconstructing constructivism: Causal models, bayesian learning mechanisms, and the theory theory. *Psychological Bulletin*, *138*(6), 1085–1108.

Gottlieb, J., Oudeyer, P.-Y., Lopes, M., & Baranes, A. (2013). Information-seeking, curiosity, and attention: Computational and neural mechanisms. *Trends in Cognitive Sciences*, *17*(11), 585–593.

Graham, P. (1993). *On lisp: Advanced techniques for common lisp*. Prentice Hall.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Green, C. C., Waldinger, R. J., Barstow, D. R., Elschlager, R., Lenat, D. B., McCune, B. R., Shaw, D. E., & Steinberg, L. I. (1974). *Progress report on program-understanding systems* (tech. rep. AIM-240). Stanford Artificial Intelligence Laboratory.

Green, C. (1981). Application of theorem proving to problem solving. *Readings in artificial intelligence* (pp. 202–222). Elsevier.

Greenfield, P. M., Nelson, K., & Saltzman, E. (1972). The development of rulebound strategies for manipulating seriated cups: A parallel between action and grammar. *Cognitive psychology*, *3*(2), 291–310.

Griffiths, T. L., Lieder, F., & Goodman, N. D. (2015). Rational use of cognitive resources: Levels of analysis between the computational and the algorithmic. *Topics in Cognitive Science*, *7*(2), 217–229.

Groen, G., & Parkman, J. (1972). A chronometric analysis of simple addition. *Psychological Review*, *79*(4), 329–343.

Groen, G., & Resnick, L. B. (1977). Can preschool children invent addition algorithms? *Journal of Educational Psychology*, *69*(6), 645–652.

Grünwald, P. D. (2007). *The minimum description length principle*. MIT press.

Grünwald, P. D., & Vitányi, P. M. (2008). Algorithmic information theory.

Gulwani, S., Korthikanti, V. A., & Tiwari, A. (2011). Synthesizing geometry constructions. *ACM SIGPLAN Notices*, *46*(6), 50–61.

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

Haber, N., Mrowca, D., Wang, S., Fei-Fei, L. F., & Yamins, D. L. (2018). Learning to play with intrinsically-motivated, self-aware agents. *Advances in Neural Information Processing Systems*, 8388–8399.

Harman, G. (1975). Meaning and semantics. *Semantics and philosophy*. NYU Press.

Harman, G. (1982). Conceptual role semantics. *Notre Dame Journal of Formal Logic*, *23*(2), 242–256.

Harman, G. (1987). (non-solipsistic) conceptual role semantics. *New directions in semantics*. Academic Press.

Harper, R. (2016). *Practical foundations for programming languages*. Cambridge University Press.

Harris, P. L. (2012). The child as anthropologist. *Infancia y Aprendizaje*, *35*(3), 259–277.

Hartnett, P., & Gelman, R. (1998). Early understandings of numbers: Paths or barriers to the construction of new understandings? *Learning and Instruction*, *8*(4), 341–374.

Hartnett, P. M. (1991). *The development of mathematical insight: From one, two, three to infinity* (Doctoral dissertation). University of Pennsylvania.

Hewitt, L. B., Le, T. A., & Tenenbaum, J. B. (2020). Learning to infer program sketches. *Proceedings of the 36th Conference Conference on Uncertainty in Artificial Intelligence*.

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, *146*, 29–60.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hofmann, M., Kitzelmann, E., & Schmid, U. (2009). A unifying framework for analysis and evaluation of inductive programming systems. *Proceedings of the 2nd Conference on Artificiel General Intelligence (2009)*.

Hofstadter, D. R. (1979). *Gödel, escher, bach: An eternal golden braid* (Vol. 13). Basic Books.

Hofstadter, D. R. (1995). *Fluid concepts and creative analogies: Computer models of the fundamental mechanisms of thought*. Basic books.

Holland, J. (1975). *Adaptation in natural and artificial systems: An introductory analysis with application to biology*. University of Michigan Press.

Holyoak, K. J. (2012). Analogy and relational reasoning. *The oxford handbook of thinking and reasoning* (pp. 234–259). Oxford University Press.

Hutter, M. (2005). *Universal artificial intelligence*. Springer.

Ikutani, Y., Kubo, T., Nishida, S., Hata, H., Matsumoto, K., Ikeda, K., & Nishimoto, S. (2020). Expert programmers have fine-tuned cortical representations of source code. *bioRxiv*.

Inhelder, B., & Piaget, J. (1964). *The early growth of logic in the child: Classification and seriation*. Humanities Press.

ISO. (2016). *Sql — part 1: Framework*.

Ivanova, A. A., Srikant, S., Sueoka, Y., Kean, H. H., Dhamala, R., O'reilly, U.-M., Bers, M. U., & Fedorenko, E. (2020). Comprehension of computer code relies primarily on domain-general executive resources. *BioRxiv*.

James, S., Konidaris, G., & Rosman, B. (2017). An analysis of monte carlo tree search. *Thirty-First AAAI Conference on Artificial Intelligence*.

Jara-Ettinger, J., Gweon, H., Schulz, L. E., & Tenenbaum, J. B. (2016). The naïve utility calculus: Computational principles underlying commonsense psychology. *Trends in cognitive sciences*, *20*(8), 589–604.

Jara-Ettinger, J., Piantadosi, S., Spelke, E. S., Levy, R., & Gibson, E. (2017). Mastery of the logic of natural numbers is not the result of mastery of counting: Evidence from late counters. *Developmental Science*. https://doi.org/10.1111/desc.12459

Jay, B. (2009). *Pattern calculus*. Springer Nature. https://doi.org/10.1007/978-3-540-89185-7

Jay, B. (2016). Programs as data structures in λSF-calculus. *Electronic Notes in Theoretical Computer Science*.

Jay, B. (2019). A simpler lambda calculus. *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 1–9.

Jay, B., & Given-Wilson, T. (2011). A combinatory account of internal structure. *The Journal of Symbolic Logic*, *76*(3), 807–826.

Jay, B., & Vergara, J. (2017). Conflicting accounts of λ-definability. *Journal of Logical and Algebraic Methods in Programming*, *87*.

Jefferys, W. H., & Berger, J. O. (1992). Ockham's razor and bayesian analysis. *American Scientist*, *80*(1), 64–72.

Jha, S., Gulwani, S., Seshia, S. A., & Tiwari, A. (2010). Oracle-guided component-based program synthesis. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, *1*, 215–224.

Johnson-Laird, P. N. (1977). Procedural semantics. *Cognition*, *5*(3), 189–214.

Johnson-Laird, P. N. (1989). *Mental models.* MIT Press.

Johnson-Laird, P. N. (2012). Inference with mental models. *The oxford handbook of thinking and reasoning* (pp. 134–145). Oxford University Press.

Jones, R. M., & Van Lehn, K. (1994). Acquisition of children's addition strategies: A model of impasse-free, knowledge-level learning. *Machine Learning*, *16*(1-2), 11–36.

Joshi, R., Nelson, G., & Randall, K. (2002). Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices*, *37*(5), 304–314.

Joulin, A., & Mikolov, T. (2015). Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 190–198.

Kahneman, D., Slovic, S. P., Slovic, P., & Tversky, A. (1982). *Judgment under uncertainty: Heuristics and biases*. Cambridge university press.

Karmiloff-Smith, A. (1992). *Beyond modularity. a developmental perspective on cognitive science*. MIT Press.

Karmiloff-Smith, A. (1988). The child is a theoretician, not an inductivist. *Mind & Language*, *3*(3), 183–196.

Kashyap, R. L., & Oommen, B. J. (1984). Spelling correction using probabilistic methods. *Pattern Recognition Letters*, *2*(3), 147–154.

Katayama, S. (2013). Magichaskeller on the web: Automated programming as a service. *Haskell Symposium*.

Kaye, D. B., Post, T. A., Hall, V. C., & Dineen, J. T. (1986). Emergence of information-retrieval strategies in numerical cognition: A developmental study. *Cognition and Instruction*, *3*(2), 127–150.

Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory*. MIT press.

Kemp, C., Shafto, P., Berke, A., & Tenenbaum, J. B. (2007). Combining causal and similarity-based reasoning. *Advances in neural information processing systems*, 681–688.

Kemp, C., & Tenenbaum, J. B. (2008). The discovery of structural form. *Proceedings of the National Academy of Sciences*, *105*(31), 10687–10692.

Kemp, C., & Tenenbaum, J. B. (2009). Structured statistical models of inductive reasoning. *Psychological review*, *116*(1), 20.

Kemp, C., Tenenbaum, J. B., Niyogi, S., & Griffiths, T. L. (2010). A probabilistic model of theory formation. *Cognition*, *114*(2), 165–196.

Kidd, C., & Hayden, B. Y. (2015). The psychology and neuroscience of curiosity. *Neuron*, *88*(3), 449–460.

Kinzler, K. D., & Spelke, E. S. (2007). Core systems in human cognition. *Progress in brain research* (pp. 257–264). Elsevier BV. https://doi.org/10.1016/s0079-6123(07)64014-x

Kitzelmann, E. (2009). Inductive programming: A survey of program synthesis techniques. *International workshop on approaches and applications of inductive programming*, 50–73.

Knoth, T., Wang, D., Polikarpova, N., & Hoffmann, J. (2019). Resource-guided program synthesis. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 253–268.

Knuth, D. E. (1973). *The art of computer programming, volume 3: Searching and sorting*. Addison-Wesley Publishing Company.

Kohlberg, L. (1968). The child as a moral philosopher. *Psychology Today*, *2*(4), 25–30.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: Principles and techniques*. MIT Press.

Kolmogorov, A. N. (1963). On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A*, 369–376.

Koopman, P., Plasmeijer, R., & Jansen, J. M. (2014). Church encoding of data types considered harmful for implementations. *26th Symposium on Implementation and Application of Functional Languages (IFL)*.

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. *Proceedings of the International Joint Conference on Artifical Intelligence*, *89*, 768–774.

Koza, J. R., & Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. MIT press.

Kuhn, T. S. (1962). *The structure of scientific revolutions*. University of Chicago Press.

Labov, W. (1989). The child as linguistic historian. *Language Variation and Change*, *1*(1), 85–97.

Lafond, D., Lacouture, Y., & Mineau, G. (2007). Complexity minimization in rule-based category learning: Revising the catalog of boolean concepts and evidence for non-minimal rules. *Journal of Mathematical Psychology*, *51*(2), 57–74.

Lake, B., Ullman, T., Tenenbaum, J., & Gershman, S. (2017). Building machines that learn and think like people. *Behavioral and Brain Sciences*, *40*. https://doi.org/10.1017/S0140525X16001837

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

Lake, B. M., & Piantadosi, S. T. (2020). People infer recursive visual concepts from just a few examples. *Computational Brain & Behavior*, *3*(1), 54–65.

Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2019). The omniglot challenge: A 3-year progress report. *Current Opinion in Behavioral Sciences*, *29*, 97–104.

Langdon, W. B., & Poli, R. (2013). *Foundations of genetic programming*. Springer Science & Business Media.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Lehman, J., & Stanley, K. O. (2011a). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, *19*(2), 189–223.

Lehman, J., & Stanley, K. O. (2011b). Novelty search and the problem with objectives. *Genetic programming theory and practice ix* (pp. 37–56). Springer.

Lenat, D. (1976). *Am: An artificial intelligence approach to discovery in mathematics* (Doctoral dissertation). Stanford University.

Lenat, D. B. (1983). Eurisko: A program that learns new heuristics and domain concepts: The nature of heuristics iii: Program design and results. *Artificial Intelligence*, *21*(1-2), 61–98.

Levy, S. (1984). *Hackers: Heroes of the computer revolution*. Anchor/Doubleday.

Lewis, R. L., Howes, A., & Singh, S. (2014). Computational rationality: Linking mechanism and behavior through bounded utility maximization. *Topics in Cognitive Science*, *6*(2), 279–311.

Liang, P., Jordan, M. I., & Klein, D. (2010). Type-based mcmc. *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 573–581.

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

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J. B., & Muggleton, S. H. (2014). Bias reformulation for one-shot function induction. *Proceedings of the 21st European Conference on Artificial Intelligence*, 525–530.

Loar, B. (1982). Conceptual role and truth-conditions: Comments on Harman's paper:"Conceptual Role Semantics". *Notre Dame Journal of Formal Logic*, *23*(3), 272–283.

Locke, J. (1690). *An essay concerning human understanding* (1st ed.). Thomas Bassett.

Lombrozo, T. (2012). Explanation and abductive inference. *The oxford handbook of thinking and reasoning* (pp. 260–276). Oxford University Press.

Lombrozo, T. (2019). "learning by thinking" in science and in everyday life. In A. Levy & P. Godfrey-Smith (Eds.), *The scientific imagination* (pp. 230–249). Oxford University Press.

Lovett, M. C., & Anderson, J. R. (2005). Thinking as a production system. In K. J. Holyoak & R. G. Morrison (Eds.), *The Oxford handbook of thinking and reasoning* (pp. 401–429). Cambridge University Press.

Lucas, C. G., Griffiths, T. L., Xu, F., Fawcett, C., Gopnik, A., Kushnir, T., Markson, L., & Hu, J. (2014). The child as econometrician: A rational model of preference understanding in children. *PLoS One*. https://doi.org/10.1371/journal.pone.0092160

Lupyan, G., & Bergen, B. (2016). How language programs the mind. *Topics in cognitive science*, *8*(2), 408–424.

MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge University press.

Macnamara, J. (1999). *Through the rearview mirror: Historical reflections on psychology*. MIT Press.

Macnamara, J., & Reyes, G. E. (1994). *The logical foundations of cognition*. Oxford University Press.

Mahabal, A. A. (2010). *Seqsee: A concept-centered architecture for sequence perception* (Doctoral dissertation). Indiana University.

Manna, Z., & Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *2*(1), 90–121.

Mareschal, D., & Shultz, T. R. (1999). Development of children's seriation: A connectionist approach. *Connection Science*, *11*(2), 149–186.

Marr, D. (1982). *Vision*. W.H. Freeman.

Martelli, A., & Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *4*(2), 258–282.

Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Pearson Education.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, *3*(4), 184–195.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

McGinn, C. (1993). *Problems in philosophy. the limits of inquiry.*

McGonigle-Chalmers, M., & Kusel, I. (2019). The development of size sequencing skills: An empirical and computational analysis. *Monographs of the Society for Research in Child Development*, *84*(4), 7–202.

Meredith, M. J. (1986). *Seek-whence: A model of pattern perception* (Doctoral dissertation). Indiana University.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*(2), 81.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, *17*(3), 348–375.

Minsky, M. L. (1967). *Computation: Finite and infinite machines*. Prentice-Hall.

Mitchell, M. (1992). *Copycat: A computer model of high-level perception and conceptual slippage in analogy-making* (Doctoral dissertation). University of Michigan.

Mollica, F., & Piantadosi, S. (2019). Logical word learning: The case of kinship. https://doi.org/10.31234/osf.io/a7tnb

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, *19*, 629–679.

Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, *100*(1), 49–73.

Murphy, G. L., & Medin, D. L. (1985). The role of theories in conceptual coherence. *Psychological Review*, *92*(3), 289–316.

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT Press.

National Governors Association Center for Best Practices, Council of Chief State School Officers. (2010). *Common core state standards mathematics*. National Governors Association Center for Best Practices, Council of Chief State School Officers.

Neches, R. (1987). Learning through incremental refinement of procedures. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development* (pp. 163–219). MIT Press.

Newell, A., Shaw, J. C., & Simon, H. A. (1959). Report on a general problem solving program. *IFIP Congress*, *256*, 64.

Newell, A., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, *65*(3), 151.

Newell, A., & Simon, H. (1956). The logic theory machine–a complex information processing system. *IRE Transactions on Information Theory*, *2*(3), 61–79.

Nosofsky, R. M., Gluck, M. A., Palmeri, T. J., McKinley, S. C., & Glauthier, P. (1994). Comparing modes of rule-based classification learning: A replication and extension of shepard, hovland, and jenkins (1961). *Memory & cognition*, *22*(3), 352–369.

Nosofsky, R. M., Palmeri, T. J., & McKinley, S. C. (1994). Rule-plus-exception model of classification learning. *Psychological review*, *101*(1), 53.

Nowak, M., & Sigmund, K. (1993). A strategy of win-stay, lose-shift that outperforms tit-for-tat in the prisoner's dilemma game. *Nature*, *364*(6432), 56–58.

Nye, M., Hewitt, L., Tenenbaum, J., & Solar-Lezama, A. (2019). Learning to infer program sketches. *Proceedings of the 36th International Conference on Machine Learning*, 4861–4870.

Ohshima, Y., Amelang, D., Kaehler, T., Freudenberg, B., Lunzer, A., Kay, A., Piumarta, I., Yamamiya, T., Borning, A., Samimi, H., Victor, B., & Rose, K. (2012). *Steps toward the reinvention of programming, 2012 final report submitted to the national science foundation (nsf) october 2012* (tech. rep. TR-2012-001). Viewpoints Research Institute.

Okasaki, C. (1999). *Purely functional data structures*. Cambridge University Press.

Osera, P.-M., & Zdancewic, S. (2015). Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, *50*(6), 619–630.

Oudeyer, P. Y. (2018). Computational theories of curiosity-driven learning. In G. Gordon (Ed.), *The new science of curiosity*. Nova Science Publishers.

Overlan, M., Jacobs, R., & Piantadosi, S. (2017). Learning abstract visual concepts via probabilistic program induction in a language of thought. *Cognition*, *168*, 320–334.

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

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. Morgan Kaufmann.

Peterson, C. R., & Beach, L. R. (1967). Man as an intuitive statistician. *Psychological Bulletin*, *68*(1), 29–46.

Piaget, J. (1952). *The child's conception of number*. Routledge; Kegan Paul.

Piaget, J. (1955). *The child's construction of reality*. Routledge & Kegan Paul.

Piantadosi, S., & Jacobs, R. (2016). Four problems solved by the probabilistic language of thought. *Current Directions in Psychological Science*, *25*(1), 54–59.

Piantadosi, S., Tenenbaum, J., & Goodman, N. (2012). Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, *123*(2), 199–217.

Piantadosi, S., Tenenbaum, J., & Goodman, N. (2016). The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological Review*, *123*(4), 392–424.

Piantadosi, S. T. (2011). *Learning and the language of thought* (Doctoral dissertation). Massachusetts Institute of Technology.

Piantadosi, S. T. (2016). *The computational origin of representation and conceptual change* [unpublished draft].

Piantadosi, S. T. (2020). Fleet system.

Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A field guide to genetic programming*. Lulu.

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

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

Post, E. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, *65*(2), 197. https://doi.org/10.2307/2371809

Rahwan, I., Cebrian, M., Obradovich, N., Bongard, J., Bonnefon, J.-F., Breazeal, C., Crandall, J. W., Christakis, N. A., Couzin, I. D., Jackson, M. O., et al. (2019). Machine behaviour. *Nature*, *568*(7753), 477–486.

Rao, M. K. (2004). Inductive inference of term rewriting systems from positive data. *International Conference on Algorithmic Learning Theory*, 69–82.

Reed, S., & de Freitas, N. (2015). Neural programmer-interpreters.

Resnick, L. B., & Neches, R. (1984). Factors affecting individual differences in learning ability. In R. J. Sternberg (Ed.), *Advances in the psychology of human intelligence* (pp. 275–323). Lawrence Erlbaum Associates.

Rips, L. J. (1994). *The psychology of proof: Deductive reasoning in human thinking*. MIT Press.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, *12*(1), 23–41.

Romano, S., Salles, A., Amalric, M., Dehaene, S., Sigman, M., & Figueira, S. (2018). Bayesian validation of grammar productions for the language of thought. *PLoS One*. https://doi.org/10.1371/journal.pone.0200420

Rothe, A., Lake, B. M., & Gureckis, T. (2017). Question asking as program generation. *Advances in Neural Information Processing Systems*, 1046–1055.

Rule, J., Dechter, E., & Tenenbaum, J. B. (2015). Representing and learning a large system of number concepts with latent predicate networks. *Proceedings of the 37th Annual Conference of the Cognitive Science Society*.

Rule, J., Schulz, E., Piantadosi, S. T., & Tenenbaum, J. B. (2018). Learning list concepts through program induction. *Proceedings of the 40th Annual Conference of the Cognitive Science Society*.

Rule, J. S., Piantadosi, S. T., & Tenenbaum, J. B. (in press). The child as hacker. *Trends in Cognitive Sciences*.

Rumelhart, D. E., McClelland, J. L., & PDP Research Group. (1987). *Parallel distributed processing*. MIT Press.

Russell, S., & Norvig, P. (2002). *Artificial intelligence: A modern approach*. Pearson.

Saxe, G. B. (1988a). Candy selling and math learning. *Educational Researcher*, *17*(6), 14–21.

Saxe, G. B. (1988b). The mathematics of child street vendors. *Child Development*, *59*(5), 1415–1425.

Saxe, G. B., Guberman, S. R., & Gearhart, M. (1987). Social processes in early number development. *Monographs of the Society for Research in Child Development*, *52*(2).

Schkufza, E., Sharma, R., & Aiken, A. (2013). Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, *41*(1), 305–316.

Schmidhuber, J. (1987). *Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook* (Doctoral dissertation). Technische Universität München.

Schönfinkel, M. (1924). Über die bausteine der mathematischen logik. *Mathematische Annalen*, *92*(3), 305–316.

Schultz, T. R., & Vogel, A. (2004). A connectionist model of the development of transitivity. *Proceedings of the Annual Conference of the Cognitive Science Society*, *26*(26).

Schulz, E., Tenenbaum, J. B., Duvenaud, D., Speekenbrink, M., & Gershman, S. J. (2017). Compositional inductive biases in function learning. *Cognitive psychology*, *99*, 44–79.

Schulz, L. (2012a). Finding new facts; thinking new thoughts. *Advances in child development and behavior* (pp. 269–294). Elsevier.

Schulz, L. (2012b). The origins of inquiry: Inductive inference and exploration in early childhood. *Trends in Cognitive Sciences*, *16*(7), 382–389.

Secada, W. G., Fuson, K. C., & Hall, J. W. (1983). The transition from counting-all to counting-on in addition. *Journal for Research in Mathematics Education*, *14*(1), 47–57.

Selman, R. L. (1981). The child as a friendship philosopher. In S. R. Asher & J. M. Gottman (Eds.), *The development of children's friendships* (pp. 242–272). Cambridge University Press.

Shapiro, E. Y. (1983). *Algorithmic program debugging*. MIT Press.

Shaw, D. E., Swartout, W. R., & Green, C. C. (1975). Inferring lisp programs from examples. *IJCAI*, *75*, 260–267.

Shepard, R. N., Hovland, C. I., & Jenkins, H. M. (1961). Learning and memorization of classifications. *Psychological monographs: General and applied*, *75*(13), 1.

Shrager, J., & Siegler, R. (1998). Scads: A model of children's strategy choices and strategy discoveries. *Psychological Science*, *9*(5), 405–410.

Siegler, R., & Jenkins, E. (1989). *How children discover new strategies*. Erlbaum.

Siegler, R. S. (1996). *Emerging minds*. Oxford Univesity Press.

Siegler, R., & Shipley, C. (1995). Variation, selection, and cognitive change. In T. J. Simon & G. S. Halford (Eds.), *Developing cognitive competence: New approaches to process modeling* (pp. 31–76). Psychology Press.

Siegler, R., & Shrager, J. (1984). Strategy choices in addition and subtraction: How do children know what to do? In C. Sophian (Ed.), *Origins of cognitive skills* (pp. 229–293). Lawrence Erlbaum Associates.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Simmons-Edler, R., Miltner, A., & Seung, S. (2018). Program synthesis through reinforcement learning guided tree search. *arXiv preprint arXiv:1806.02932*.

Sipser, M. (2012). *Introduction to the theory of computation*. Cengage Learning.

Siskind, J. (1996). A computational study of cross-situational techniques for learning word-to-meaning mappings. *Cognition*, *61*, 31–91.

Slaughter, V., Itakura, S., Kutsuki, A., & Siegal, M. (2011). Learning to count begins in infancy: Evidence from 18 month olds' visual preferences. *Proceedings of the Royal Society of London B: Biological Sciences*, *278*(1720), 2979–2984.

Smith, D. R. (1984). The synthesis of lisp programs from examples: A survey. In A. W. Biermann, G. Guiho, & Y. Kodratoff (Eds.), *Automatic program construction techniques* (pp. 307–324). Macmillan.

Smith, K., Mei, L., Yao, S., Wu, J., Spelke, E., Tenenbaum, J. B., & Ullman, T. (2019). Modeling expectation violation in intuitive physics with coarse probabilistic object representations. *Advances in Neural Information Processing Systems*, 8983–8993.

Solar-Lezama, A. (2008). *Program synthesis by sketching* (Doctoral dissertation). University of California, Berkeley.

Solomonoff, R. J. (1964a). A formal theory of inductive inference, part i. *Information and Control*, *7*(1), 1–22.

Solomonoff, R. J. (1964b). A formal theory of inductive inference. part ii. *Information and control*, *7*(2), 224–254.

Spelke, E. S., & Kinzler, K. D. (2007). Core knowledge. *Developmental Science*, *10*(1), 89–96. https://doi.org/10.1111/j.1467-7687.2007.00569.x

Steffe, L., Von Glasersfeld, E., Richards, J., & Cobb, P. (1983). *Children's counting types: Philosophy, theory, and applications*. Praeger.

Sternberg, R. J., & Davidson, J. E. (1995). *The nature of insight.* The MIT Press.

Stuhlmuller, A., Tenenbaum, J. B., & Goodman, N. D. (2010). Learning structured generative concepts. *Proceedings of the Annual Conference of the Cognitive Science Society.*

Sussman, G. J. (1973). *A computational model of skill acquisition* (Doctoral dissertation). Massaschusetts Institute of Technology.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning*. MIT Press.

Svenson, O. (1975). Analysis of time required by children for simple additions. *Acta Psychologica*, *39*(4), 289–301.

Tenenbaum, J. B. (1999). Bayesian modeling of human concept learning. *Advances in Neural Information Processing Systems*, 59–68.

Tenenbaum, J. B. (2000). Rules and similarity in concept learning. *Advances in Neural Information Processing Systems*, 59–65.

Tenenbaum, J. B., Kemp, C., Griffiths, T. L., & Goodman, N. D. (2011). How to grow a mind: Statistics, structure, and abstraction. *Science*, *331*(6022), 1279–1285.

Thomas, D., & Hunt, A. (2019). *The pragmatic programmer*. Addison-Wesley Professional.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, *25*(3/4), 285–294.

Torlak, E., & Bodik, R. (2013). Growing solver-aided languages with rosette. *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 135–152.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, *2*(42), 230–265.

Turkle, S., & Papert, S. (1992). Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, *11*(1), 3–33.

Ullman, T., Goodman, N., & Tenenbaum, J. (2012). Theory learning as stochastic search in the language of thought. *Cognitive Development*, 455–480.

Ullman, T. D., Spelke, E., Battaglia, P., & Tenenbaum, J. B. (2017). Mind games: Game engines as an architecture for intuitive physics. *Trends in Cognitive Sciences*, *21*(9), 649–665.

Ullman, T. D., Stuhlmüller, A., Goodman, N. D., & Tenenbaum, J. B. (2018). Learning physical parameters from dynamic scenes. *Cognitive psychology*, *104*, 57–82.

Ullman, T. D., & Tenenbaum, J. B. (2020). Bayesian models of conceptual development: Learning as building models of the world. *Annual Review of Psychology*.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, *27*(11), 1134–1142.

Van Raamsdonk, F. (1999). Higher-order rewriting. *International Conference on Rewriting Techniques and Applications*, 220–239.

Vousden, W. D., Farr, W. M., & Mandel, I. (2015). Dynamic temperature selection for parallel tempering in markov chain monte carlo simulations. *Monthly Notices of the Royal Astronomical Society*, *455*(2), 1919–1937. https://doi.org/10.1093/mnras/stv2422

Vuculescu, O., Stausberg, N., Sergeev, R., & Ham, H. (2020). What do the experts know: An empirical investigation of adaptive search. *Proceedings of the 42nd Annual Conference of the Cognitive Science Society*.

W3C. (2017). *Html 5.2*. Retrieved August 3, 2020, from https://www.w3.org/TR/html52/

Wagner, K., Tillman, K., & Barner, D. (2016). Inferring number, time, and color concepts from core knowledge and linguistic structure. In D. Barner & A. S. Baron (Eds.), *Core knowledge and conceptual change*. Oxford University Press.

Wan, X., Nakatani, H., Ueno, K., Asamizuya, T., Cheng, K., & Tanaka, K. (2011). The neural basis of intuitive best next-move generation in board game experts. *Science*, *331*(6015), 341–346.

Wang, L., Amalric, M., Fang, W., Jiang, X., Pallier, C., Figueira, S., Sigman, M., & Dehaene, S. (2019). Representation of spatial sequences using nested rules in human prefrontal cortex. *NeuroImage*, *186*, 245–255.

Wason, P. C. (1960). On the failure to eliminate hypotheses in a conceptual task. *Quarterly journal of experimental psychology*, *12*(3), 129–140.

Wellman, H. M., & Gelman, S. A. (1992). Cognitive development: Foundational theories of core domains. *Annual Review of Psychology*, *43*(1), 337–375.

Wellman, H. M., & Gelman, S. A. (1998). Knowledge acquisition in foundational domains. In W. Damon (Ed.), *Handbook of child psychology: Vol. 2. cognition, perception, and language* (pp. 523–573). John Wiley & Sons Inc.

Wierzbicka, A. (1996). *Semantics: Primes and universals*. Oxford University Press.

Woods, W. A. (1981). Procedural semantics as a theory of meaning. In B. W. A. Joshi & I. Sag (Eds.), *Elements of discourse understanding*. Cambridge University Press.

Wynn, K. (1990a). Children's understanding of counting. *Cognition*, *36*(2), 155–193.

Wynn, K. (1990b). Children's understanding of counting. *Cognition*, *36*(2), 155–193.

Wynn, K. (1992a). Addition and subtraction by human infants. *Nature*, *358*(6389), 749.

Wynn, K. (1992b). Children's acquisition of the number words and the counting system. *Cognitive Psychology*, *24*(2), 220–251.

Xu, F. (2019). Towards a rational constructivist theory of cognitive development. *Psychological Review*, *126*(6), 841–864.

Xu, F., & Griffiths, T. L. (2011). Probabilistic models of cognitive development: Towards a rational constructivist approach to the study of learning and development. *Cognition*, *120*, 299–301.

Yang, C. (2016). The linguistic origin of the next number. *LingBuzz preprint lingbuzz/003824*.

Yildirim, I., & Jacobs, R. A. (2015). Learning multisensory representations for auditory-visual transfer of sequence category knowledge: A probabilistic language of thought approach. *Psychonomic Bulletin & Review*, *22*(3), 673–686.

Young, R. M. (1976). *Seriation by children: An artifical intelligence analysis of piagetian task*. Birkhäuser.

Zettlemoyer, L. S., & Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*, 658–666.

Zylberberg, A., Dehaene, S., Roelfsema, P. R., & Sigman, M. (2011). The human turing machine: A neural framework for mental programs. *Trends in Cognitive Sciences*, *15*(7), 293–300.