Cognition 255 (2025) 105990



Contents lists available at ScienceDirect

Cognition

journal homepage: www.elsevier.com/locate/cognit



Full length article



# Exploring the hierarchical structure of human plans via program generation

Carlos G. Correa [a,*], Sophia Sanborn [b], Mark K. Ho [c,d], Frederick Callaway [c], Nathaniel D. Daw [a,c], Thomas L. Griffiths [c,d]

[a] Princeton Neuroscience Institute, Princeton University, USA
[b] Department of Ophthalmology, Stanford University, USA
[c] Department of Psychology, Princeton University, USA
[d] Department of Computer Science, Princeton University, USA

ARTICLE INFO

*Keywords:*
Hierarchical reinforcement learning
Program induction
Planning
Chunking

ABSTRACT

Human behavior is often assumed to be hierarchically structured, made up of abstract actions that can be decomposed into concrete actions. However, behavior is typically measured as a sequence of actions, which makes it difficult to infer its hierarchical structure. In this paper, we explore how people form hierarchically structured plans, using an experimental paradigm with observable hierarchical representations: participants create programs that produce sequences of actions in a language with explicit hierarchical structure. This task lets us test two well-established principles of human behavior: utility maximization (i.e. using fewer actions) and minimum description length (MDL; i.e. having a shorter program). We find that humans are sensitive to both metrics, but that both accounts fail to predict a qualitative feature of human-created programs, namely that people prefer programs with *reuse* over and above the predictions of MDL. We formalize this preference for reuse by extending the MDL account into a generative model over programs, modeling hierarchy choice as the induction of a grammar over actions. Our account can explain the preference for reuse and provides better predictions of human behavior, going beyond simple accounts of compressibility to highlight a principle that guides hierarchical planning.

## 1. Introduction

Human behavior has rich hierarchical structure, in which actions are grouped into abstract, higher-level actions that are used to accomplish tasks (Klir & Simon, 1991; Miller, Galanter, & Pribram, 1960; Newell & Simon, 1972). However, these internal representations are not observable, so they are typically inferred from behavioral signatures of hierarchy (Rosenbaum, Kenny, & Derr, 1983). Consider the problem of *making tomato sauce for dinner*. The task can be made successively more concrete as *prepare the tomatoes* then *prepare one tomato* then *slice the tomato*. Some of these actions have highly repetitive structure, like preparing a tomato by repeatedly slicing it. While action hierarchies arise naturally from practiced behavior, they can still provide a benefit in novel settings by providing a compact representation for behavior and by radically decreasing the complexity of search for a solution.

What guides the generation of hierarchical plans and action hierarchies, particularly in novel settings? While past studies try to infer this structure from behavioral signatures of hierarchy, it is challenging to address this question directly because the hierarchical structure of behavior is not observable. For example, in the sequence-learning literature, hierarchies are inferred based on elevated response times or error rates when switching between abstract actions (Acuna et al., 2014; Rosenbaum et al., 1983; Verwey, 1996). Similar methods are applied when investigating hierarchical planning, though additional behavioral attributes can be measured such as the particular choice of action sequence or self-reports about behavioral hierarchy (Correa, Ho, Callaway, Daw, & Griffiths, 2023; Huys et al., 2015; Solway et al., 2014; Tomov, Yagati, Kumar, Yang, & Gershman, 2020).

In order to make hierarchical plans observable, we use a process-tracing paradigm where participants are tasked with creating hierarchically structured plan-like *programs* in an environment called Lightbot.[1] In the task, participants create a program to guide a robot that starts in a specific location (Fig. 1) and must visit all unlit lights (blue squares) to activate them. The programs must contain simple actions (e.g., `Walk`, `Activate Light`, `Turn Right`), but can also have hierarchical structure by reusing sequences of actions as *subroutines*. These programs serve as an explicit representation of a hierarchical plan, where abstract actions correspond to subroutines. In previous studies, this experimental paradigm has been used to demonstrate that

* Corresponding author.
*E-mail address:* cgcorrea@princeton.edu (C.G. Correa).

[1] Lightbot (https://lightbot.com) is an educational game developed by Danny Yaroslavski, released as a Flash game in 2008 and a mobile application in 2013.

https://doi.org/10.1016/j.cognition.2024.105990
Received 23 November 2023; Received in revised form 9 August 2024; Accepted 16 October 2024
Available online 30 November 2024
0010-0277/© 2024 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

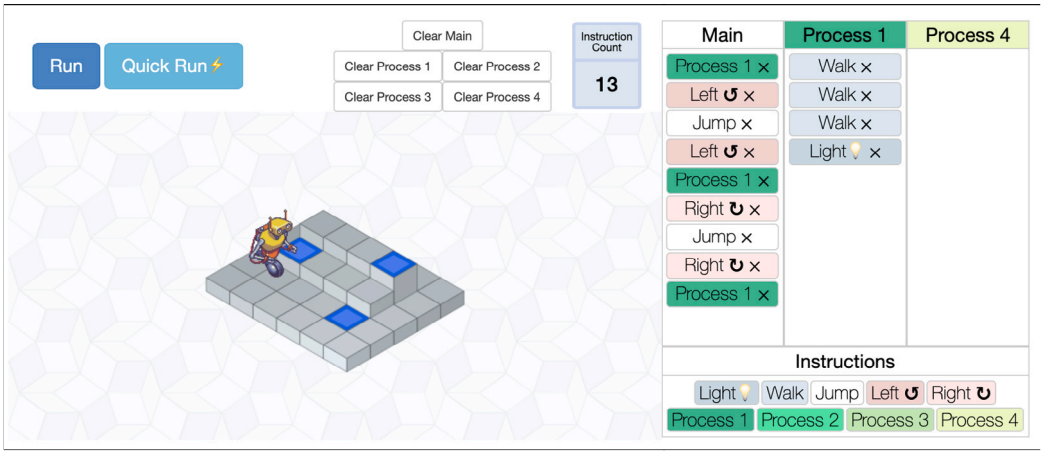



**Fig. 1.** The interface for Lightbot, a process-tracing experiment for complex, hierarchical plans. Participants create programs by dragging instructions from the lower right to define a program. The program is executed by the robot from the initial subroutine ("Main"), and the task is completed when all blue squares in the environment are activated. An example program that solves the task is shown. The program includes a single subroutine ("Process 1") with four actions (`Walk`, `Walk`, `Walk`, `Activate Light`). Participants can use up to four subroutines (referred to as processes in the experiment), but for brevity only two are pictured. The *Run* button executes the program with an animation of the robot taking each action. The program is executed without animation with the *Quick Run* button. The program length is displayed as an *Instruction Count*, and the white buttons at the top are used to clear all instructions from a subroutine for ease of program editing.

people prefer programs that are short or can be compressed to be short (Sanborn, Bourgin, Chang, & Griffiths, 2018).

This process-tracing paradigm facilitates the comparison of different accounts of hierarchy selection. For example, given the preference for compressibility in this domain (Sanborn et al., 2018), a natural theory is that hierarchy selection is based on the principle of minimum description length (MDL; Chater, 1999). MDL and related information-theoretic frameworks have been previously used to explain hierarchical representations in sequence learning (Planton et al., 2021) and planning (Lai, Huang, & Gershman, 2022; Maisto, Donnarumma, & Pezzulo, 2015; Solway et al., 2014). Another natural theory is that of utility maximization, e.g. solving the problem in the fewest steps. In this work, we find evidence that people favor both shorter programs and those with fewer actions. We highlight example programs that these theories cannot account for, but are consistent with a preference for reusing subroutines above and beyond the predictions of either account. We formalize this preference as a prior belief that subroutine use should be biased towards subroutines that have been used often in the current task. This prior is central to our computational-level theory (Marr, 1982) of human planning as *grammar induction*, an approach drawn from linguistics where it has been used to model word learning as joint inference of a grammar over words and sentences using those words (Goldwater, Griffiths, & Johnson, 2009; Johnson, Griffiths, & Goldwater, 2006). Inspired by this analogy, we use this approach to model planning as joint inference of a grammar over abstract actions and a plan using those actions. Here the preference for reuse arises from a clustering prior over the abstract (subroutine-level) actions (Johnson et al., 2006), resembling approaches used to model clustering and reuse in categorization (Anderson, 1991), reinforcement learning (Gershman, Blei, & Niv, 2010), and causal learning (Kemp, Goodman, & Tenenbaum, 2010).

In this article, we study how people select hierarchical plans by directly analyzing the programs they create in our process-tracing experiment. First, we develop our framework of plan inference using grammar induction and detail our generative model over hierarchical plans, showing that our theory is a natural approach to predict qualitative aspects of behavior missed by simpler accounts. Then, we analyze behavioral data from an online experiment using our process-tracing paradigm. We examine qualitative examples and perform model comparison, finding that participant programs are best predicted by our grammar induction model. Analyzing other behavioral signatures unique to our experiment, we also find that people prefer hierarchies that simplify task solution.

## 2. Background

Our approach to modeling complex, hierarchical planning draws on several distinct literatures, which we briefly summarize in this section. Common approaches for studying hierarchical structure in behavior have used paradigms like sequence learning or frameworks like hierarchical reinforcement learning. While these studies have often focused on characterizing hierarchy discovery following learning from experience, several have focused on the hierarchies used when planning, as we do in this paper. Theories of hierarchy discovery require specifying a space of possible hierarchies, which we accomplish through Bayesian program induction.

### 2.1. Sequence learning

A long-standing question is how sequences are learned by humans and other animals. An influential finding is that sequence memorization is vastly improved when information is grouped into larger *chunks* (Miller, 1956). Methodologically, mental representations of hierarchical structure are often inferred from behavior on the basis of elevated response times (Verwey, 1996), errors (Acuna et al., 2014; Rosenbaum et al., 1983), or decreased speed in motor trajectories (Ramkumar, Acuna, Berniker, Grafton, Turner, & Kording, 2016). Recent normative theories of sequence learning explain chunk selection through the optimization of a speed-accuracy trade-off (Dezfouli & Balleine, 2012; Wu, Éltetö, Dasgupta, & Schulz, 2023), MDL (Planton et al., 2021), and idealized search costs (Ramkumar et al., 2016). There are also theories of action segmentation that model reuse by using a similar framework (an Adaptor Grammar; Johnson et al., 2006) as our approach (Buchsbaum, Griffiths, Plunkett, Gopnik, & Baldwin, 2015). Another broad approach to modeling sequence learning is statistical learning (Perruchet & Pacton, 2006)—for example, modeling action chunking using a hierarchical non-parametric model of sequence statistics (Éltetö & Dayan, 2023; Éltetö, Nemeth, Janacsek, & Dayan, 2022).

### 2.2. Hierarchical reinforcement learning

Hierarchical reinforcement learning adapts the framework of reinforcement learning to a setting with hierarchically structured policies. In one prominent framework, this is formalized by augmenting a task with abstract actions, called "options", which each consist of a behavioral policy and conditions for initiation and termination (Botvinick,

Niv, & Barto, 2009; Stolle & Precup, 2002; Sutton, Precup, & Singh, 1999). Much of the research on option discovery has focused on *subgoal*-based options, with approaches including partitioning a task into subtasks (McNamee, Wolpert, & Lengyel, 2016; Solway et al., 2014; Tomov et al., 2020) or identifying "bottleneck" states within the state space (i.e. states that are commonly passed on the way to the ultimate goal; Şimşek & Barto, 2009).

Many approaches have been taken to understand how people choose behavioral hierarchies, including some based on task partitioning (Solway et al., 2014; Tomov et al., 2020), policy compression (Lai et al., 2022; Maisto et al., 2015; Solway et al., 2014), and minimizing planning costs (Correa et al., 2023; Huys et al., 2015). The approach we introduce in this paper based on an Adaptor Grammar (Johnson et al., 2006) is similar to a model used to examine planning behavior in prior work (Huys et al., 2015). Studies have investigated error-based learning of hierarchical policies, finding evidence in both neural data (Ribas-Fernandes, Solway, Diuk, McGuire, Barto, Niv, & Botvinick, 2011) and behavioral patterns (Eckstein & Collins, 2020).

### 2.3. Bayesian program induction

Bayesian program induction is an approach to inferring programs $\pi$ consistent with some observed data $d$ (Goodman, Tenenbaum, Feldman, & Griffiths, 2008; Lake, Salakhutdinov, & Tenenbaum, 2015; Piantadosi, Tenenbaum, & Goodman, 2012; Rule, Tenenbaum, & Piantadosi, 2020). Concretely, by specifying a *prior* probability distribution over programs $p(\pi)$, and the *likelihood* of the observed data under that program $p(d|\pi)$, Bayes' theorem can be used to define the *posterior* probability of programs conditioned on the data

$$p(\pi|d) \propto p(d|\pi)p(\pi).$$

This approach has been fruitfully used to model many aspects of cognition, including concept learning (Fränken, Theodoropoulos, & Bramley, 2022; Goodman, Tenenbaum, et al., 2008), number word learning (Piantadosi et al., 2012), writing characters with motor actions (Lake et al., 2015), and sequential decision making (Maisto et al., 2015; Wingate, Goodman, Roy, Kaelbling, & Tenenbaum, 2011). While the computational demands of inference usually restrict its application to simple domains, modern machine learning methods like neural networks have made it possible to scale these methods to more difficult domains, while still producing interpretable, human-like representations (Ellis et al., 2021; Poesia & Goodman, 2023). Another crucial strategy for making inference tractable is library learning (Ellis et al., 2021; Poesia & Goodman, 2023; Zhao, Lucas, & Bramley, 2023), where program fragments are reused in order to accelerate inference. Our grammar induction model builds upon this idea.

### 2.4. Minimum description length

Minimum description length (MDL) is an information-theoretic approach that focuses on minimizing the description length of mental representations and is formally related to Bayesian inference (Chater, 1999). In the setting of sequence learning, an MDL representation is one that provides a minimal, hierarchical compression of the sequence. This approach has been applied broadly in studying sequence learning. For example, Planton et al. (2021) propose that compressed, hierarchical representations of sequences support learning. Maisto et al. (2015) incorporate a prior over hierarchical policies related to the description length of plans under those hierarchies. Lai et al. (2022) extend a theory of policy compression to account for the information-theoretic savings from ignoring perceptual information while executing an action chunk. Solway et al. (2014) note their theory identifies hierarchies that minimize the description length of optimal behavior, even though it is primarily formulated as a Bayesian account.

## 3. Theories of program creation in Lightbot

Having introduced these literatures, we now turn to our approach. We draw on ideas from sequence learning and hierarchical reinforcement learning to study the hierarchical structure underlying action sequences, and propose a modeling framework based on principles from program induction and MDL.

Our approach to studying complex, hierarchical plans has two key components. First, we use a process-tracing experimental paradigm in order to make explicit the internal hierarchical representations participants use for planning. Instead of asking participants to simply solve the problem, we ask participants to create a program that will be interpreted by an agent in the experiment. Critically, this program may be hierarchically structured through the use of subroutines. Second, we model these plans with a framework based on models of grammar learning from computational linguistics (Goldwater et al., 2009; Johnson et al., 2006). In this section, we introduce the experiment and motivate our choice of modeling framework.

Our process-tracing paradigm is based on the popular Lightbot game. The experiment interface and an example program are shown in Fig. 1. In this task, research participants control a robot in a gridworld-like environment by supplying a program for it to execute, with the goal of having the robot light all the blue squares. The program is composed by dragging and dropping instructions in the interface shown in Fig. 1. Participants create programs that include five primitive actions (`Walk`, `Jump`, `Turn Left`, `Turn Right`, `Activate Light`) as well as calls to subroutines that can be defined and reused. Subroutines are defined in the same way (by dragging and dropping instructions), so they also consist of actions and subroutine calls, which makes it possible to submit programs that correspond to hierarchically structured plans. For simplicity, the number of subroutines is fixed to four, which means no action must be taken to create an empty subroutine. When participants submit a program for execution, the robot starts from the initial subroutine and runs instructions in order, so that primitive actions immediately take effect and subroutine calls execute the respective subroutine. Thus, executing a program results in a deterministic sequence of actions, which we refer to as the *execution trace*. A video of program creation and execution is included in the supplementary materials. We formally define the task, programs, and their execution in the next section (see Section 4).

In the example in Fig. 1, the robot is shown in the start state. The example program in Fig. 1 results in an S-shaped execution trace and accomplishes the goal of activating the lights, as shown in the top row of Fig. 2. Notably, the program has a subroutine that contains the instructions `Walk`, `Walk`, `Walk`, `Activate Light`. If the program were written without a subroutine, the program length would be 18, equivalent to the execution trace length. However, by using this subroutine, the program is much shorter, using only 13 instructions. In order to make program structure clearer in this article, we display programs as trees, as in the second column of Fig. 2, described further in the figure caption.

We formalize the process of planning as Bayesian program induction over hierarchical plans, so the posterior probability of a plan depends on the likelihood that it solves a task as well as the belief assigned to it *a priori*. By framing planning as an inference problem, we can examine different assumptions about program structure in the form of prior distributions and see how they influence the posterior distribution of programs that solve the task. This is a notable departure from non-hierarchical planning, which typically finds a sequence of actions that solve a task. By inferring a hierarchy, this also departs from accounts of hierarchical planning that use fixed hierarchies. A key component of the posterior is the likelihood that we assign to a program. Because participants are required to create programs that complete a task, we define this likelihood to require that execution of a program reaches the goal (i.e. having all lights activated).

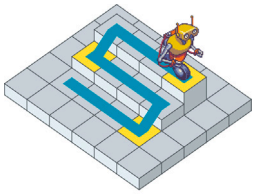
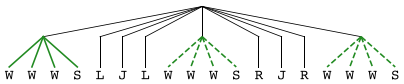
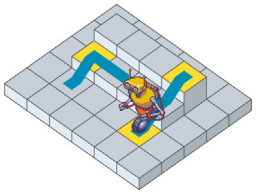
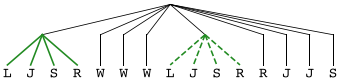
| Trace | Program | Step Count | Program Length |
|---|---|---|---|
|  |   | 18 | 13 |
|  |   | 15 | 13 |

**Fig. 2.** Two example programs that solve the task are shown in Fig. 1. The top row contains the same program as Fig. 1, demonstrating the execution trace, or sequence of resulting actions, in blue. Table rows correspond to distinct programs. Table columns show a program execution trace, a tree representation of the program, the step count (number of actions) resulting from program execution, and the program length. The tree representation of a program shows how the sequence of actions relates to the program structure. Actions are executed in sequence, from left to right. Subroutines are shown with green lines that connect the subroutine call to its constituent instructions. The first use of a subroutine has solid lines, while subsequent uses have dashed lines. In the experiment, participants only see an animation of the robot, not the execution trace in blue. Legend: S: `Activate Light`, W: `Walk`, J: `Jump`, R: `Turn Right`, L: `Turn Left`. .

A prior over programs is the other key component needed to define a posterior. Our modeling approach compares different priors over programs, which we introduce here. The simplest prior ignores program structure, preferring those that result in shorter execution traces. This model corresponds to a softmax choice rule (Luce, 1959) over plans based on the number of resulting actions, similar to standard modeling approaches in reinforcement learning. An alternative prior might prefer shorter programs, a reasonable approach given that program length directly corresponds to the number of motor actions participants take when writing programs. We quantify program length as the *number of instructions* used in the written description of the program, as opposed to the *number of actions* the program yields in the execution trace. We treat program length as description length, and consider this an MDL prior.

Both of these priors are sensible approaches to modeling plan inference in Lightbot, but are inconsistent with our experimental results (reported in detail below). For example, in Fig. 2 the top row features a commonly-written program that contains a single subroutine (`Walk`, `Walk`, `Walk`, `Activate Light`). This subroutine is used three times and mirrors the repeated environmental structure. The program in the bottom row, written less frequently, uses a subroutine only twice. Despite experimental participants preferring the top program over the bottom one, it would not be preferred by either of the priors outlined above.

We introduce a theory to explain this by having previous use of a subroutine in the current task inform future use—in other words, subroutine choice is biased to reuse subroutines. So, the top program in Fig. 2 is more intuitive because it has greater repeated use of the subroutine. Our theory of grammar induction is a prior over Lightbot programs, including subroutines, where each use of a subroutine increases the probability that it will be used again. By contrast, the MDL prior assigns the same probability to programs of equal length, equally preferring flat sequences of actions and highly nested programs. By accounting for this bias towards reuse, our prior can explain qualitative patterns in the use of subroutines in human behavioral data, as we explore further below.

## 4. Modeling framework

### 4.1. Task formalism

We formalize Lightbot as an undiscounted, deterministic Markov Decision Process (MDP), specified by a state set $\mathcal{S}$, initial state $s_0 \in \mathcal{S}$, action set $\mathcal{A}$, transition function $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, and reward function $R : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$. A Lightbot task is defined by a 2-dimensional grid of squares, which each has an associated integer height and type (light square or not). The state set $\mathcal{S}$ has elements that track the environment state (each light is either active or unactive) and the agent state (a location and orientation). The action set $\mathcal{A}$ consists of the five primitive actions (`Walk`, `Jump`, `Turn Left`, `Turn Right`, `Activate Light`). The goal states $\mathcal{G} \subset \mathcal{S}$ are those where all light squares are activated.

The transition function $T$ is straightforward: `Turn Left` and `Turn Right` change the agent orientation, by rotating either counterclockwise or clockwise, respectively; `Activate Light` activates a light if present at the agent location. A valid use of `Walk` and `Jump` both move the agent forward, changing the location based on the orientation. A valid use of `Walk` requires the current and next location to have matching heights, while a valid use of `Jump` requires the next location to either have a height greater by one unit or smaller by any amount. The agent does not move for invalid uses of either action.

The goal in Lightbot is to activate all light squares, so upon reaching a goal state, $g \in \mathcal{G}$, from a non-goal state, $s \notin \mathcal{G}$, the agent is rewarded, $R(s, g) = 1$. Otherwise, the agent receives no reward, $R(s, s') = 0$. The goal states $g \in \mathcal{G}$ are absorbing, so for all actions $a$, $T(g, a) = g$.

### 4.2. Program formalism

We now introduce notation for the programs that participants submit in our process-tracing paradigm. We define a *program* $\pi$ as a tuple of subroutines $(\rho^0, \rho^1, \ldots)$, where each subroutine $\rho^i$ consists of a sequence of instructions $(\rho^i_0, \rho^i_1, \ldots)$ and has length $|\rho^i|$. Each *instruction* $\rho^i_j$ is either an action in the MDP, so that $\rho^i_j \in \mathcal{A}$, or a *subroutine call* $\rho^i_j \in \{\rho^1, \rho^2, \ldots\}$. The program length is the length of all subroutines, $|\pi| = \sum_i |\rho^i|$.

To execute a program, we start at an initial state $s_0 \in \mathcal{S}$ and begin executing the initial subroutine $\rho^0$. Executing a subroutine $\rho^i$ in turn requires sequentially executing its constituent instructions $\rho^i_j$. When the current instruction is an action, so $\rho^i_j \in \mathcal{A}$, the action is taken, resulting in a state transition, so $s_{t+1} = T(s_t, a_t = \rho^i_j)$ and $t = t + 1$, and a corresponding reward, $R(s_t, s_{t+1})$. Otherwise, the instruction $\rho^i_j$ is a subroutine call, which does not directly result in a state transition but goes on to execute the referenced subroutine. When execution of the referenced subroutine is completed, execution returns to the original subroutine that made the call and continues on to the next instruction. To match the experimental interface, recursive calls are permitted however $\rho^0$ is excluded from the list of valid subroutine calls. The execution of a program can be visualized as a tree (Fig. 2), where leaf nodes correspond to executed actions, in order from left to right, and internal nodes correspond to subroutine calls.

Programs are executed until they reach a goal state, which results in an execution trace $\tau(\pi) = (a_0, a_1, \ldots, a_{T-1})$ and state trajectory $(s_0, s_1, \ldots, s_T)$, where the final state is a goal, $s_T \in \mathcal{G}$. This constraint ensures that programs that reach a goal always halt, even if the program is recursive. The value of a program is $V(\pi) = \sum_t R(s_t, s_{t+1})$. Since there is no reward until the goal is reached, $V(\pi) = 1$ when a program reaches the goal and $V(\pi) = 0$ otherwise. An algorithm for program evaluation is included in the appendix (see Algorithm 1).

### 4.3. Inference

We formulate the choice of programs as the result of Bayesian program induction, adapted to this sequential decision setting by recasting planning as inference (Levine, 2018; Toussaint & Storkey, 2006; Wingate et al., 2011). Concretely, we define a posterior distribution over programs that combines a prior distribution over programs $p(\pi)$ with a likelihood that indicates whether a program solves the task.

#### 4.3.1. Inferring programs

We first introduce a binary random variable $\Omega$ so that $\Omega = 1$ means a program solves the task. We condition on programs that solve the task in order to define a posterior distribution over programs, $p(\pi \mid \Omega = 1)$. Applying Bayes' theorem, we can define this posterior in terms of a likelihood that reflects whether a program solves the task, $p(\Omega = 1 \mid \pi)$, and a prior over programs, $p(\pi)$,

$$p(\pi \mid \Omega = 1) = \frac{p(\Omega = 1 \mid \pi)p(\pi)}{Z}$$

where $Z$ is a normalizing constant $Z = \sum_{\pi'} p(\Omega = 1 \mid \pi')p(\pi')$, with $\pi'$ ranging over all possible programs.

We define the likelihood to identify programs that solve the task, or equivalently, reach the goal. Since the value of a program is an indicator of reaching the goal, we can simply define our likelihood as

$$p(\Omega = 1 \mid \pi) = V(\pi) = \begin{cases} 1 & \text{if } \pi \text{ reaches the goal} \\ 0 & \text{otherwise} \end{cases}.$$

A key benefit of this inference-based formulation is that it can integrate prior beliefs about the structure of programs alongside the requirement that a program solves the task.

#### 4.3.2. Approximate inference

Computing the normalizing constant $Z$ requires the intractable task of enumerating all programs. As a tractable alternative, we approximate the normalization constant $Z$ using a large corpus of programs, intended to span a large space of compact programs with short traces. To generate these programs, we first search for a corpus of the shortest execution traces $(a_0, a_1, \ldots)$ that solve a given task. Then, we generate programs for each trace by taking each combination of non-trivial subroutines and rewriting the trace, assuming subroutines are used as much as possible. We describe trace search and program generation in more detail in the appendix.

Given a collection of programs $\Pi$, we can compute an approximation to the normalization constant $Z_\Pi$ by summing over the programs in $\Pi$, formally

$$Z_\Pi = \sum_{\pi' \in \Pi} p(\Omega = 1 \mid \pi')p(\pi').$$

This lets us define an approximate posterior

$$p(\pi \mid \Omega = 1) \approxeq Z_\Pi^{-1} p(\Omega = 1 \mid \pi)p(\pi). \tag{1}$$

We note that $Z_\Pi$ is a lower bound to $Z$, $Z_\Pi < Z$, since $Z$ is a sum of positive terms and $Z_\Pi$ is a sum of a subset of those terms.

Our approximation scheme is motivated by the difficulty of generating a large number of programs that solve the task. In particular, the number of traces grows exponentially with trace length. For example, the number of traces of length 18 (like the trace in the top row of Fig. 2) is $\prod^{18} 5 = 5^{18} \approx 3.8 \times 10^{12}$, or nearly 4 trillion. This large space of possible traces is what led us to our scheme to first identify traces that reach the goal, then to subsequently construct programs consistent with those traces.

Importantly, we are not claiming that people perform this approximation scheme when solving the task. Our model is a computational-level account (Marr, 1982) of program writing, intended to examine which factors guide how people select programs. By identifying the factors that influence program writing, we hope to facilitate future studies that could explore which algorithms people use to solve this challenging inference problem.

### 4.4. Baseline models

Having outlined our framework for Bayesian program induction, we formalize some simple priors over programs. As a baseline, we consider a model that minimizes trace length. This prior ignores program structure and penalizes the resulting trace length. Since this is equivalent to having a cost for every step, we refer to it as the *step cost* prior:

$$p(\pi) \propto \exp\{-|\tau(\pi)|\}.$$

Some prominent accounts of task decomposition have incorporated description length into their objectives to influence choice over subgoals. In one account, minimizing description length guides the choice among task decompositions, used to compress fixed, optimal policies (Solway et al., 2014). In another account, the description length of plans guides subgoal choice in the form of a prior over subgoals in an inference-based hierarchical planner (Maisto et al., 2015).

Based on this work, we additionally consider a prior that minimizes the description length of a program, which we refer to as the *MDL* prior. We formalize the description length (DL) as the program length, so $\mathrm{DL}(\pi) = |\pi|$. The description length is simple and idealized, assuming that all instructions have the same description length. We use this to define a prior, so that shorter lengths are preferred:

$$p(\pi) \propto \exp\{-|\pi|\}.$$

As noted above, the simple approaches introduced in this section struggle to explain intuitive solutions to Lightbot problems, as in Fig. 2. The critical missing piece is that these approaches are insensitive to differences in hierarchical structure observed when holding program length or trace length constant. We fill this explanatory gap with the model we introduce next.

### 4.5. Grammar induction

In this section, we recast hierarchical planning as grammar induction. Here, we propose a generative model in which new subroutines augment a grammar over actions. The key idea behind our model is that past use of subroutines should inform the probability assigned to future use of subroutines. We formalize this idea using Adaptor Grammars (Johnson et al., 2006).

To start, we define a prior over a single subroutine $\rho^i$ that is a sequence of low-level actions, so that $\rho^i_j \in \mathcal{A}$ for all $j$—that is, the subroutine does not contain other subroutines. We let the length of the sequence range dynamically in standard fashion, by having a constant probability that construction of the sequence terminates, $p_{end}$, and sampling actions from a uniform distribution, $p(\rho^i_j = a) = \frac{1}{|\mathcal{A}|}$. The overall probability of an action sequence is thus

$$p(\rho^i) = p_{end}\, p(\rho^i_0 = a) \prod_{j=1} (1 - p_{end}) p(\rho^i_j = a). \tag{2}$$

This constant probability of sequence termination means that action sequence lengths are geometrically-distributed with parameter $p_{end}$.

We augment this prior to generate subroutine calls in addition to actions by (1) sampling a subroutine call with probability $p_{call}$ or an action with probability $1 - p_{call}$ and then (2) either generating a new subroutine for the call or reusing one that was previously defined. Permitting the reuse of subroutines means that the resulting grammar is no longer *context-free*, since the probability of a subroutine varies based on previous subroutine calls. To formalize this kind of long-range dependency, we instead use an *Adaptor Grammar* (Huys et al., 2015; Johnson et al., 2006), which augments a standard context-free grammar by allowing the grammar's productions to be dependent on the history of previous productions. This provides a formal basis for a generative model of reuse, where previous productions can become more probable in the future. We use a Dirichlet Process (DP; Aldous, 1985) to adapt subroutine generation for our grammar over Lightbot programs. We briefly highlight some key aspects of this account of subroutine use: new subroutines are generated, subroutines can be reused, and neither the number of subroutines nor the subroutines themselves are predefined.

The DP is a probability distribution over clusters of data that incorporates a new element by either assigning it to (1) a new cluster or (2) an existing cluster, biased towards larger clusters. We likewise sample subroutine calls, as either (1) a new subroutine or (2) an existing subroutine, biased towards often-used subroutines. Formally, having already drawn $n$ subroutine calls to $m$ distinct subroutines with a total of $n_k$ calls to subroutine $k$, we define a distribution over the next subroutine call $z_{n+1}$, given the history of drawn subroutine calls $z_{1:n}$

$$p(z_{n+1} \mid z_{1:n}) = \begin{cases} \frac{\alpha}{n+\alpha} & \text{if } z_{n+1} = m+1 \\ \frac{n_k}{n+\alpha} & \text{if } 1 \le z_{n+1} = k \le m \end{cases} \tag{3}$$

The first case deals with a new subroutine call, which requires that a new subroutine be generated. In the second case, an existing subroutine is selected proportionally to how often its been used in the past, $n_k$. When an existing subroutine is selected, it is reused and generation of that subroutine is avoided. The DP has a single parameter, $\alpha \ge 0$, which is usually referred to as the concentration parameter since it controls the relative probability of creating a subroutine. As $\alpha \to 0$ reuse of an existing subroutine becomes more likely and as $\alpha \to \infty$ creation of a new subroutine becomes more likely. Since the most probable subroutines are those used most often in the past, the DP results in so-called *rich-get-richer* dynamics, where frequently used subroutines are most likely in the future. The rich-get-richer dynamics of the DP are a key prediction that differentiates our grammar induction account from the alternative priors—as noted above, this preference for reuse can account for the example in Fig. 2.

Having introduced the DP, we can define $p(\rho^i_j = a)$ to handle both low-level actions and subroutines as

$$p(\rho^i_j = a) = \begin{cases} (1 - p_{call})\frac{1}{|\mathcal{A}|} & \text{if } a \in \mathcal{A} \\ p_{call}\, p(z_{n+1} = a \mid z_{1:n}) & \text{if } a \in \{\rho^1, \rho^2, \ldots\}. \end{cases} \tag{4}$$

where we elide the dependence on past subroutine calls to simplify notation. Along with the above Eq. (2), we can define the prior probability of a Lightbot program as

$$p(\pi = \{\rho^1, \rho^2, \ldots\}) = \prod_i p(\rho^i) \tag{5}$$

Putting these pieces together, the grammar induction prior generates sequences of instructions, terminating with probability $p_{end}$ (Eq. (2)). Instructions are either subroutine calls with probability $p_{call}$ or actions with probability $1 - p_{call}$ (Eq. (4)). Subroutine calls are drawn from the Adaptor Grammar based on a DP, with new subroutines generated as an instruction sequence and existing subroutines reused in proportion to the number of past calls (Eq. (3)). An algorithm for sampling is included in the appendix (see Algorithm 2).

In the MDL account, using a subroutine more times can make a program shorter, so a preference for shorter programs can indirectly lead to a preference for subroutine reuse. The grammar induction model has an analogous preference because, in general, it assigns higher probability to programs with fewer instructions or subroutines. However, a distinctive prediction of the grammar induction model is the explicit preference to use subroutines based on how often they have been used in the past, which is above and beyond that of the implicit preference that comes from avoiding generation. While the MDL account only indirectly encourages hierarchical structure, the grammar induction model also does so in a more explicit way by assigning higher probability to hierarchies with greater reuse.

Adaptor grammars have been used to study linguistic phenomena like word segmentation (Goldwater et al., 2009) and inference of discourse structure (Luong, Frank, & Johnson, 2013), and have also been extended to study forms of generalization that require greater abstraction, like past tense and derivational morphemes in English (O'Donnell, 2015) and in non-linguistic topics like causal concept bootstrapping (Zhao et al., 2023). Adaptor grammars have also been used to model action sequences, by predicting action segmentation (Buchsbaum et al., 2015) and planning behavior (Huys et al., 2015). Adaptor grammars are also formally related to approaches used in probabilistic programming languages to express non-parametric distributions (Goodman, Mansinghka, Roy, Bonawitz, & Tenenbaum, 2008).

## 5. Experiment

### 5.1. Methods

#### 5.1.1. Procedure

Participants were given an extensive tutorial to ensure they learned how to control Lightbot using the five primitive actions, as well as to ensure they understood how to create and use programs. The tutorial described the functionality of each action, included demonstrations, and the opportunity to solve practice problems to ensure understanding. Screenshots of the entirety of the tutorial are included in the appendix.

In the remainder of the experiment, participants wrote programs to solve 10 Lightbot tasks and were incentivized to write short programs to encourage subroutine use. Participants received instructions about the incentive and were shown multiple examples of subroutine use. The program editing interface had a counter showing program length (Fig. 1). Importantly, our instructions focus on describing how the program length influences the received bonus, showing examples of how subroutines can result in both shorter and longer programs than the trajectory. Participants who wrote the shortest program for a task received the full bonus of $0.40 and those who wrote a longer program received

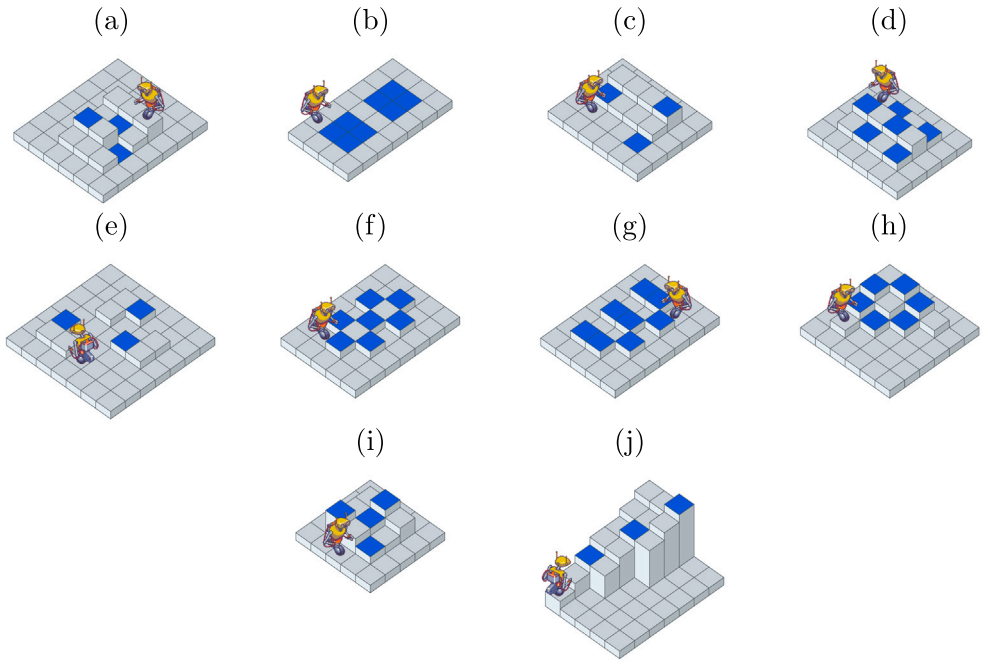



**Fig. 3.** The tasks participants completed in the experiment.

a proportionally smaller bonus, calculated after the experiment to ensure the average per-task bonus across participants was $0.20. Since participants solved 10 tasks, the total bonus ranged from $0.00 to $4.00, with an average bonus of $2.00. Since the task incentivizes short programs (by encouraging participants to minimize their program length), this could bias participants towards behavior more consistent with the MDL prior. However, because the instructions do not explicitly encourage subroutine reuse, we assume minimal experimentally induced bias towards the grammar induction prior (specifically, its preference for reuse).

After being instructed about the incentives, participants went on to write programs to solve each of 10 tasks (Fig. 3), with the program editing interface shown in Fig. 1. When starting a new task, the program editing interface was cleared of the program created for the previous task (i.e. subroutines from past tasks were not available for use). During each task, participants were given the option to skip the problem and forfeit a bonus for the current problem after 3.5 min elapsed. We recorded the programs submitted by participants, in addition to other measures related to task difficulty, like the number of program evaluations and time elapsed. In a closing survey, participants were asked about their programming experience and whether they had previously played games similar to the experiment.

#### 5.1.2. Participants

The experiment was run on the Prolific platform, requiring that participants were English-speakers, were located in the United States of America, had not participated in pilot studies for Lightbot, had an approval rate of 95%+, and had at least 25 past submissions on the platform. Participants consented to the experimental procedures beforehand, as approved by the Institutional Review Board of Princeton University. A total of 193 participants (age $M = 38.39$ $SD = 12.10$, range: 19–86, 72 female, 3 with demographic data missing) completed the entire study, in an average of 59.55 min ($SD = 25.63$, range: 17.20–125.56). 171 participants (89%) were included because they satisfied the inclusion criteria, having no more than 3 college courses related to computer programming. As noted, participants could skip tasks after 3.5 min. 148 participants (87%) skipped no tasks, 16 participants (9%) skipped 1 tasks, 3 participants (2%) skipped 2 tasks, and 4 participants (2%) skipped 3+ tasks. In the below analysis, we analyze all fully completed tasks, even from participants who may have skipped other tasks.

As noted above, participants were asked about their programming experience. In response to "How much experience do you have with computer programming?" 65% of participants responded with "None", 24% responded with "Between 1 and 3 college courses (or equivalent)", and 11% responded with "More than 3 college courses (or equivalent)". In response to "Have you played Lightbot or another similar programming game before?" 98% of participants responded with "No" and 2% responded with "Yes". In the appendix, we found that programming experience had little relationship to task performance, focusing on response times, number of program evaluations, and how often participants wrote the most common program.

An important question about our process-tracing paradigm is how it influences participant behavior. We quantify one aspect of this in the appendix, where we find that number of created subroutines and number of program evaluations are positively correlated.

#### 5.1.3. Parameter fitting

Given a behavioral dataset of Lightbot programs $\pi$ across all tasks and the number of times they were produced by research participants $n_\pi$, we want to optimize for parameters $\theta$ in the model based on the posterior probability they assign to our observed behavior

$$\mathcal{L}(\theta) = \prod_{\pi} p(\pi \mid \Omega = 1, \theta)^{n_\pi}.$$

For models with nuisance parameters $\psi$, we marginalize them out

$$\mathcal{L}(\theta) = \int_{\psi} \prod_{\pi} p(\pi \mid \Omega = 1, \theta, \psi)^{n_\pi} p(\psi) d\psi.$$

In both these equations, parameters are explicitly passed to the posterior for clarity, but were left implicit in Section 4.3. We estimate parameters by maximizing the probability assigned to observed behavior, so our objective for the parameters is

$$\arg\max_{\theta} \log\{\mathcal{L}(\theta)\}$$

Because of the intractability of computing the posterior, $p(\pi \mid \Omega = 1)$, we instead compute the approximation in Eq. (1).

**Table 1**
The steps taken in preprocessing participant programs. Shown in order, with descriptions and the number of programs modified by that preprocessing step.

| | Description | Number modified | Percent modified |
|---|---|---|---|
| Step 1 | Ensure subroutines are used as often as possible | 221/1668 | 13.25% |
| Step 2 | Remove instructions that never have an effect[a] or are never executed | 241/1668 | 14.45% |
| Step 3 | Remove subroutines that are never called | 0/1668 | 0.00% |
| Step 4 | Inline subroutines only called once | 461/1668 | 27.64% |
| Step 5 | Inline subroutines that only have a single instruction | 34/1668 | 2.04% |
| Step 6 | Order turns and lights to match constraints on program search (see Appendix) | 67/1668 | 4.02% |
| Step 7 | Ensure subroutines have a canonical ordering determined by their execution order | 338/1668 | 20.26% |
| | Modified by any step | 868/1668 | 52.04% |

[a] We remove instructions that are actions $\rho_j^i \in \mathcal{A}$ if for all $t$ where $a_t = \rho_j^i$ the action results in a transition to the same state, so $s_t = s_{t+1} = T(s_t, a_t)$.

To predict participant programs, we use six models. One baseline account is a null model of *random choice*, where the prior is uninformative, $p(\pi) \propto 1$, so uniform probability is assigned to programs. We also test the *step cost*, *MDL*, and *grammar induction* priors introduced above. Finally, given the efficacy of the step cost prior in explaining behavior, we combine it with the other two priors for the *grammar induction* + *step cost* and *MDL* + *step cost* models. These combined models define a new composite prior that is the product of the two original priors.

Parameters are displayed with fitted values in Table 2, but are briefly summarized here. A relative weight for the prior is fit, which is used to scale the log prior, $\log p(\pi)$, and are named as follows: $\beta_{\text{StepCost}}$, $\beta_{\text{MDL}}$, $\beta_{\text{GrammarInduction}}$. For the combined models that include the step cost prior, we also fit a weight of $\beta_{StepCost}$. The grammar induction model additionally fits the concentration parameter of the DP, $\alpha$, and the probability of a subroutine, $p_{call}$. In addition, the grammar induction model has the nuisance parameter $\psi = \{p_{end}\}$, which we approximately marginalize for the values $p_{end} \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, reflecting a uniform prior.

Parameter fitting was performed once from an initialization using a default value, and 50 times with sampled values. The parameters with best fit from these 51 parameter fitting executions are used to make predictions below. Probabilities (e.g., $p_{call}$) were constrained to be in $(0, 1)$ and either fit from an initial default value of $\frac{1}{2}$ or a value sampled uniformly from $(0, 1)$. All other parameters were constrained to be positive and either fit from an initial default value of 1 or a value sampled from an $Exponential(2)$ distribution.

#### 5.1.4. Preprocessing of participant programs

In order to facilitate analysis, participant programs are preprocessed in order to ensure they are present in our corpus of generated programs as often as possible. Preprocessing steps and the number of modified programs are reported in Table 1. To generate programs for our approximate posterior, we focus on structurally different programs, while avoiding consideration of variation that could arise from certain process-level details (like whether a subroutine was used in every possible location). This informs how we preprocess participant programs, in order to ensure that participant-generated programs can be matched to model-predicted programs as often as possible. For example, model-generated programs always make maximal use of subroutines, so we preprocess participant programs to similarly make maximal use of existing subroutines. While we recognize that these features can provide signatures of the process that participants use to solve the task, we leave further analyses to future studies. Before preprocessing, only 41% of participant programs were generated by our methods. After preprocessing, this increased to 78%. Focusing on programs written by at least 2 participants, preprocessing helped increase this coverage from 82% to 94%.

#### 5.1.5. Data availability

Code for the experiment is available at https://github.com/cgc/cocosci-lightbot/tree/v0.4. The data and analysis code are available at https://github.com/cgc/lightbot-grammar-induction.

### 5.2. Results

#### 5.2.1. Participant programs are inconsistent with alternative accounts

In this section, we qualitatively analyze some example participant programs in Figs. 4–6, comparing common programs to other programs that were uncommon. The programs in these figures are examples that rule out a straightforward account of human choice based on solely optimizing step cost or program length. In particular, these examples can be viewed as evidence of errors, since participants are generating programs that are in conflict with objectives they might have (i.e. minimizing program length or step cost). We illustrate how our model aligns with these patterns of choice behavior, but defer statistical comparisons to the next section where parameters are fit and models are compared. We primarily focus on whether the priors assign higher probability to participant behavior, but avoid consideration of the magnitudes of probability since our models in the next section fit a parameter that scales the log prior.

We first examine Fig. 4, which revisits the programs introduced in Fig. 2. Both programs have the same length, but differ in their cost. In particular, the most common program (left) has a large cost, while the uncommon program (right) has a smaller cost. This pattern of choice can not be explained solely on the basis of either program length or cost, but is consistent with the predictions of the grammar induction prior: The common program has greater subroutine use, which has higher probability under the grammar induction prior, due to the preference for reuse in the DP.

We now examine cases where the trace is held constant, but the program varies. By examining a fixed trace, we can directly compare the structural features of programs while holding action-related features constant, such as step cost. For example, Fig. 5 shows three programs that generate the most common trace for the problem shown. The common program (left) has a single subroutine. Notably, this program can be written in several ways since two of the three uses of a

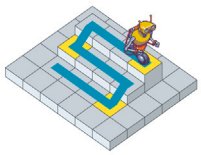
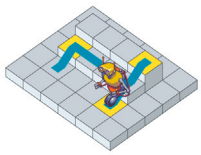
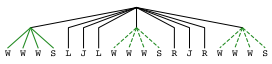
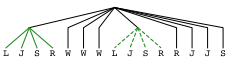
| | | |
|---|---|---|
| trace |  |  |
| program |  |  |
| participant count | 59 | 1 |
| step count | 18 | 15 |
| program length | 13 | 13 |
| grammar induction prior (log) | -31.97 | -33.17 |

**Fig. 4.** Comparing a common participant program (left) to an uncommon one (right) for one task, revisiting the example from Fig. 2. The two programs have equivalent program lengths. The more common program has a larger step count. The grammar induction prior assigns higher probability to the left program (it is 3.32 times more likely), so it can explain the overall trend of participant preferences. Subroutines are shown with green lines that connect the subroutine call to its constituent instructions. First use of a subroutine has solid lines, while subsequent uses have dashed lines. The log of the grammar induction prior is shown, with the values for its three parameters set to $\alpha = 1$, $p_{end} = \frac{1}{10}$, and $p_{call} = \frac{1}{2}$.

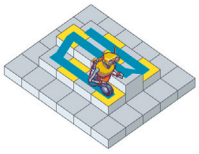
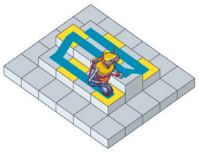
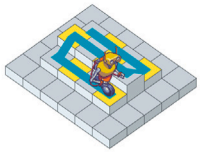
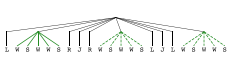
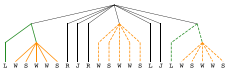
| | | | |
|---|---|---|---|
| trace |  |  |  |
| program |  |  |  |
| part. count | 50 | 3 | 3 |
| step count | 22 | 22 | 22 |
| prog. length | 15 | 15 | 18 |
| grammar induction prior (log) | -36.78 | -39.45 | -45.21 |

**Fig. 5.** Examining a common program (left) to two uncommon programs (middle, right) that all share the same trace. Here, programs are matched by step count, since they share a trace. Participant choice is not explained by program length, but can be explained by the grammar induction model. See Fig. 4 for more detail about the figure.

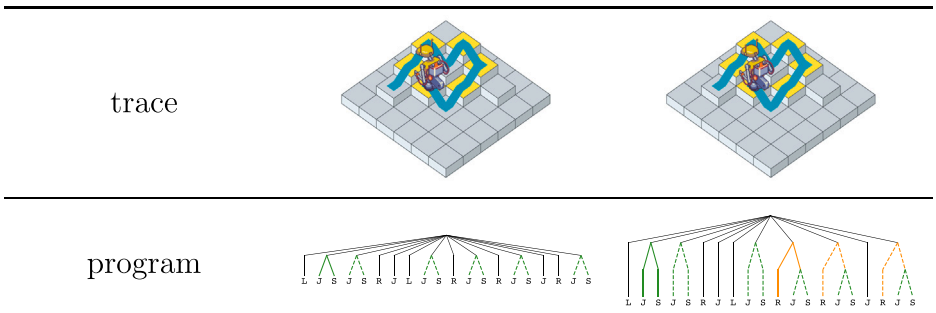
subroutine are preceded by the instruction `Turn Left`. This preceding instruction can be incorporated by creating an additional subroutine (middle). This program has the same program length, but is written by fewer participants. Under the grammar induction prior, this program is less probable because it has less reuse. An intermediate program between these two is also shown (right), where the original subroutine is instead rewritten to include the preceding instruction, reducing the number of places the subroutine can be called.

Fig. 6 shows a similar example of programs that generate the most common trace, for a different task. As in the previous example, the common program (left) has a shorter subroutine that appears more often, while the uncommon program (right) has more subroutines that are used less often. Notably, in this case the common program is actually *longer*. The grammar induction prior can capture this pattern for the same reason as above—by way of the DP, it prefers greater reuse of a single subroutine.

Taken together, these qualitative results challenge a simple account of choice solely on the basis of cost or program length. This is particularly notable since participants have a monetary incentive to minimize the length of their programs in the experiment. Distinctive about the grammar induction account is that, above and beyond the length savings associated with the reuse of a subroutine, we expect additional preference for reuse of subroutines, as predicted by the rich-get-richer dynamics of reuse from the DP. Next, we move to a quantitative comparison of models in predicting behavior.

### 5.2.2. *Predicting participant programs*

To more comprehensively examine human behavior, we fit models to the full set of programs people wrote, estimating parameters with maximum likelihood estimation using the procedures described above (see Table 2 for fitted parameters). We compare models by using the Bayesian information criterion (BIC; Schwarz, 1978), which penalizes




| | | |
|---|---|---|
| participant count | 19 | 3 |
| step count | 20 | 20 |
| program length | 16 | 15 |
| grammar induction prior (log) | -35.06 | -37.49 |

**Fig. 6.** A case where a common participant program is inconsistent with the predictions of program length. As in Fig. 5, the programs have the same trace, so they are matched by step cost. In contrast, now the uncommon figure has a shorter program length. The grammar induction prior can predict this pattern of behavior because of the bias towards reuse. See Fig. 4 for more detail about the figure.

**Table 2**
Table of fitted models, with log likelihood, BIC, parameter count, and fitted parameters.

| | Log likelihood | BIC | Param. count | Parameters |
|---|---|---|---|---|
| random choice | −21709.7 | 43419.4 | 0 | |
| MDL | −16787.3 | 33582.0 | 1 | $\beta_{\text{MDL}} = 0.98$ |
| grammar induction | −13482.3 | 26986.9 | 3 | $\alpha = 4.23$<br>$\beta_{\text{GrammarInduction}} = 0.48$<br>$p_{call} = 0.12$ |
| step cost | −15880.5 | 31768.5 | 1 | $\beta_{\text{StepCost}} = 1.31$ |
| MDL + step cost | −15267.2 | 30549.3 | 2 | $\beta_{\text{MDL}} = 0.53$<br>$\beta_{\text{StepCost}} = 0.85$ |
| grammar induction + step cost | −12825.2 | 25680.1 | 4 | $\alpha = 2.52$<br>$\beta_{\text{GrammarInduction}} = 0.38$<br>$p_{call} = 0.09$<br>$\beta_{\text{StepCost}} = 0.52$ |

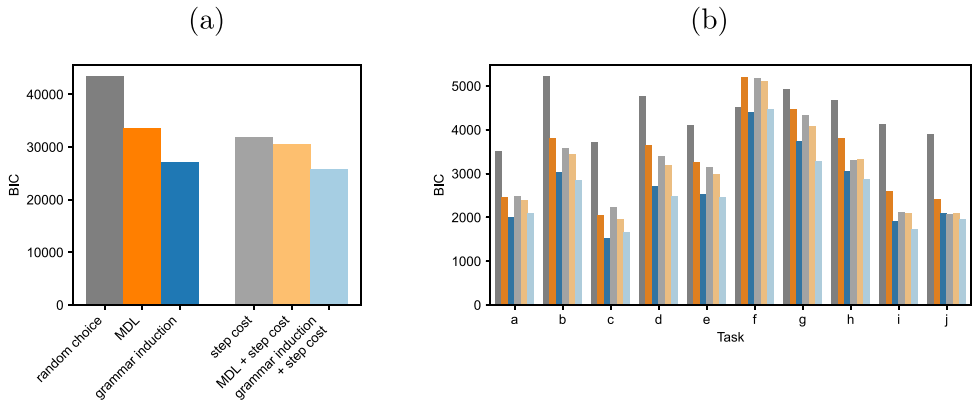



**Fig. 7.** Model comparison of accounts of participant program creation. (a) Plot of BIC of experimental data as predicted by each of the models, after parameter fitting. Models with smaller BIC are a better account of behavior. (b) BIC of data under each model, but split by task. Parameters are the same as in (a), so they are the best fit for all tasks. The color of the models is also the same as in (a). Task letter is a reference to the subfigure in Fig. 3.

models based on their number of fit parameters. We first compare the three priors, based on the model BICs shown in Fig. 7(a). All improve on a random choice model (Likelihood-ratio test for step cost: $\chi^2(1) = 11658.3$, $p < .001$, MDL: $\chi^2(1) = 9844.8$, $p < .001$, grammar induction: $\chi^2(3) = 16454.7$, $p < .001$), and the best fit to behavior is the grammar induction model based on the BIC. Surprisingly, the step cost model is a better fit to behavior than the MDL model—this could mean that participants first identified short traces and then compressed

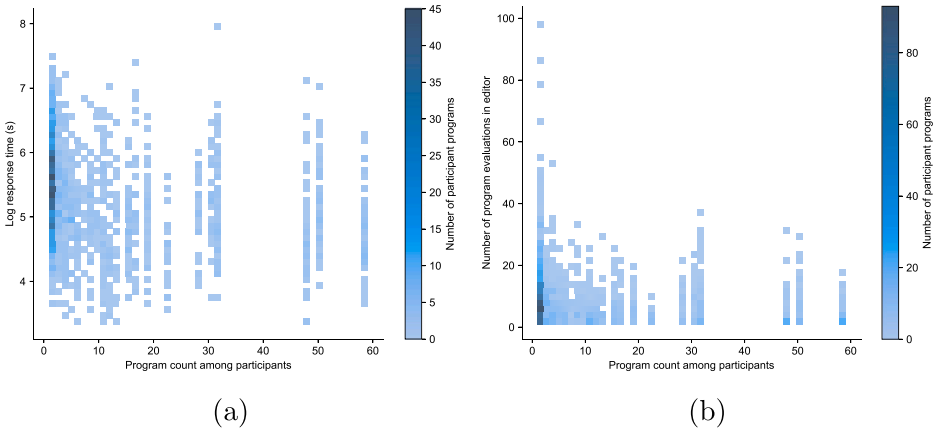



**Fig. 8.** Participants write common programs faster and with fewer program evaluations. Bivariate histogram of task responses for each participant on each task, with responses binned along horizontal axis by program count and along vertical axis by either (a) log-transformed response times (s) or (b) the number of program evaluations. A related regression analysis is reported in the main text, which uses program count to predict response times and number of program evaluations while controlling for program length and cost, and finds results consistent with these plots.

them. In order to control for the influence of this kind of strategy, we also include the combined models described above, in order to compare the MDL and grammar induction models. Compared to a baseline step cost model, we still find an improvement in fit in the MDL + step cost (Likelihood-ratio test, $\chi^2(1) = 1226.6$, $p < .001$) and grammar induction + step cost (Likelihood-ratio test, $\chi^2(3) = 6110.6$, $p < .001$) models. Among all tested models, the grammar induction + step cost model is the most predictive of behavior.

While our inclusion of step costs was primarily meant to ensure our results hold after controlling for step costs, we also found that their inclusion improved upon baseline accounts (Likelihood-ratio test for adding step cost to MDL: $\chi^2(1) = 3040.1$, $p < .001$; to grammar induction: $\chi^2(1) = 1314.1$, $p < .001$). We explore several qualitative examples demonstrating the importance of step costs in participant programs in the appendix.

Using these parameters (which were fit across all tasks), we examine the model's predictions for each task in Fig. 7(b), finding patterns that are broadly consistent with the group summary. In particular, we find that one of the grammar induction models is still the best predictor for each individual task, compared to the alternative accounts.

We test for the influence of two confounds in the Appendix: (1) Does programming experience have any influence on model comparison? (2) Does program preprocessing have any influence on model comparison? We find that controlling for either produces qualitatively similar results as reported here.

These analyses have focused on predicting behavior for all participants and tasks. However, an important direction for future studies is to better understand differences for individual participants and tasks. We report several analyses in the appendix that provide interesting directions for future study. On a per-task basis, we find that task difficulty is related to higher solution variability. We also develop a theoretical measure that is related to solution variability, based on the idea that competing objectives (i.e. step cost versus grammar induction) could lead to higher variability. We also quantify individual differences between participants by relating subroutine use and program length, finding variability in how participants trade them off.

#### 5.2.3. Common programs are easier to write

The following analyses explore how judicious use of hierarchy can make learning and planning easier. We used hierarchical linear mixed-effects models to test whether people who wrote common programs had an easier time completing the task. We fit models with `lmer` and used a baseline model with fixed effects for the intercept, program cost, and program length and random effects for per-participant and per-task intercepts. To test whether common programs were easier to write, we used a likelihood-ratio test to see if the addition of a regressor for program count would improve model fit when added to a baseline model.

One way to quantify the difficulty of the task is the length of time it takes participants to complete it, a measure that is not specific to our process-tracing paradigm. We find that participants who wrote more common programs were faster at writing their programs ($\beta = -.008$, $\chi^2(1) = 69.73$, $p < .001$, Fig. 8(a)). This suggests that people prefer hierarchies that are simpler to use for planning, consistent with prior findings. Another measure of difficulty, which is uniquely measurable in our paradigm, is the number of times participants test the execution of their program in the editor. We found that, in the process of writing their programs, those with more common programs executed their program fewer times ($\beta = -.053$, $\chi^2(1) = 16.48$, $p < .001$, Fig. 8(b)). In order to rule out any effect of programming experience, we also ran these analyses in the subset of participants with no programming experience. The results are similar as above, and reported in the appendix. So, people prefer hierarchies that are easier to reason about—in this case, people writing common programs can avoid explicit program execution in the editor, perhaps by instead relying on mental simulation.

## 6. Discussion

In this paper, we studied the human ability to construct complex, hierarchical plans. In order to do so, we used a process-tracing experimental paradigm to collect data about hierarchical plans and developed a framework for program inference to analyze these plans. We ran a behavioral experiment and found that our model of grammar induction was better at predicting behavior compared to alternative accounts that simply assumed participants were efficient as measured by program length or trace length. Key to the grammar induction model is the idea that previous use of subroutines should inform future use. In addition, we used our process-tracing paradigm to examine how the choice of hierarchy can simplify planning. We found that participants who wrote common programs were faster at solving the task and required less use of program execution in the editor.

Our model was motivated by the idea that previous usage should bias reuse. But when is this bias towards reuse sensible? It might be a natural solution when tasks typically have repeated subtasks, since it might lead to more efficient planning or learning (Wen, Precup, Ibrahimi, Barreto, Va. Roy, & Singh, 2020). Another explanation could rest on the theoretical observation that the DP is well-approximated by a particular kind of information-theoretic cost (Dasgupta & Griffiths, 2022)—this would correspond to a representational cost for the distribution over subroutine calls. One final rationale relates to the

idealized computational complexity of search, which is influenced by two factors: the possible choices at each state and the depth of a solution. Abstract actions can accelerate search by making it possible to rapidly reach deeper parts of the search tree. However, they might negatively impact the complexity of search by increasing the number of possible choices. A bias towards reuse could mitigate this increase in complexity by focusing attention on promising subroutines (c.f. Éltetö & Dayan, 2023). What psychological mechanisms might implement a bias towards reuse? Our theory could be cognitively implemented by some kind of learning or memorization—it may reflect a simple bias towards recently considered subroutines or instead reflect an online process of inferring appropriate subroutines based on prior history. We hope future research can explore how these rationalizations and potential mechanisms relate to our account, and how they can be developed into theories of reuse that our model could be quantitatively compared to. For example, future accounts could develop a generative model of tasks to explore how repeated substructure relates to participant solutions, inspired by the approach in Wen et al. (2020).

While we focused on reuse in the context of a single task, future research could examine patterns of reuse across tasks. Some existing research has examined this in how people learn program-based causal relations, showing that a compositional concept is difficult to learn without an appropriate curriculum (Rule, Schulz, Piantadosi, & Tenenbaum, 2018) and the concept learned from experience can be dependent on the sequence of experienced tasks (Zhao et al., 2023). Lightbot could be used to examine similar questions by permitting the transfer of subroutines between tasks. Having explicit representations of subroutines as they are transferred, adapted, and used in future tasks might provide an exciting process-tracing approach for testing theories of the influence of curriculum and experience on learning.

However, adapting our account to study across-task reuse would introduce issues that we have not yet considered. In particular, an ever-expanding library of subroutines would grow unwieldy, making the cost of search high since many subroutines would need to be considered at each step. The rich-get-richer dynamics of our model also pose an issue because a heavily-used subroutine might remain probable long after it is useful. These issues are addressable by decaying use counts over time, or ensuring that contextual information (like the current task or task state) is used to prioritize consideration of subroutines. These ideas could be incorporated into our model by generalizing our approach to subroutine sampling to be distance dependent (Blei & Frazier, 2011), so that subroutine use would be driven by similarities to past uses. Through appropriate definition of similarity between subroutine uses, like temporal recency or task state similarity, this formalism could be used to test complex hypotheses about how time, task, and task state influence the consideration of subroutines. Other ideas could be drawn from past work, which has identified that people prioritize possibilities that have been probable or effective in the past (Bear, Bensinger, Jara-Ettinger, Knobe, & Cushman, 2020; Mattar & Daw, 2018; Morris, Phillips, Huang, & Cushman, 2021).

Since our paradigm is a process-tracing paradigm, we cannot be sure about the relationship between the explicit representation we can measure (the hierarchical plans people submit) and the internal representations people use to act. Hierarchical structure has often been studied by using paradigms like sequence learning, where participants perform a fixed sequence many times. By contrast, participants never perform the sequence of actions directly in our paradigm, and instead must simulate it mentally or watch it be executed by the robot. In addition, participants are not trained on a fixed sequence; they instead solve the task a single time and simply submit their program. Because behavior in our experiment is not driven by bottom-up statistics in the same way as sequential learning, our behavioral paradigm might isolate a distinct aspect of behavior, namely structural biases of hierarchical planning.

It is possible that the explicitly hierarchical interface of the task encouraged participants to use hierarchical structure in a way that they would not in real-world planning problems. While contemporary studies using laboratory tasks have identified behavioral and neural signatures of hierarchical plan representations in navigation tasks (Balaguer, Spiers, Hassabis, & Summerfield, 2016; Huys et al., 2015; Solway et al., 2014), the structure of Lightbot could encourage subroutine reuse. Although the explicit incentives of the task (both bonus payment and number of clicks necessary) encourage minimizing total program length, the mere existence of explicitly reusable subroutines could push people towards reuse at the expense of longer programs, perhaps through a demand effect. Testing our account in a more naturalistic setting is thus a critical—although empirically challenging—direction for future work.

Our approach is a computational-level account (Marr, 1982), which naturally leads to the question of the algorithmic processes people use to approximately implement our theory, particularly because of the intractability of Bayesian inference. We hope that our experimental findings justify further investigation into possible process-level accounts and behavioral signatures that could be used to distinguish among them. Many existing approaches to inference explore how varying the parameterization of inference algorithms can lead to a human-like bias. For example, some studies have explored how low-capacity algorithms might recapitulate behavior (Daw & Courville, 2007; Lake & Piantadosi, 2020) or how different proposal distributions explain auto-correlation in participant hypotheses (Fränken et al., 2022). These results are often interpreted as evidence that human-like biases might arise from rational use of cognitive resources. Future research might pursue process models that are more consistent with findings in the sequence learning literature—for example, it has been observed that larger action sequences might form through concatenation of existing chunks (Tosatto, Fagot, Nemeth, & Rey, 2022), which can be used to guide the model of program creation. In general, future work could extend our model to investigate how process-level and algorithmic changes can better align with observed behavior.

Our grammar induction prior, as a generative model for programs, raises the straightforward possibility of an algorithmic implementation based on inference. In contrast, the MDL account we compare to is not formulated as a probabilistic model, so it requires non-trivial effort to adapt it to an inference-based algorithm. Either account could be implemented by a process of retrospectively compressing an identified action sequence, by first planning and then compressing. With appropriate weighting, this could form a valid algorithm for the models we have introduced. However, a more interesting direction for the future could examine the bias induced with varied path-finding (e.g., what heuristic is used to guide the search for solutions?) and compression algorithms (e.g., do cognitive constraints influence the set of programs that can be considered?). More specific to the grammar induction prior are sequential inference algorithms (like sequential Monte Carlo samplers; Doucet, D. Freitas, Gordon, et al., 2001), which suggest a process in which subroutines are generated prospectively, as part of the sequential generation of a program. This suggests an interesting algorithmic question for future studies, which could look for behavioral indications about whether participants generate hierarchical programs prospectively, or instead retrospectively compress already-identified plans. However, despite the access to representations that our paradigm offers, it seems difficult to distinguish between these two accounts with our reported experiments, since the program participants produce does not immediately indicate whether the trajectory or hierarchy was constructed first.

A related concern that is agnostic to the inference algorithm is the computational cost of evaluating the terms that comprise the unnormalized posterior, namely the prior and likelihood. In particular, the likelihood in our model requires evaluating a program in the task. Evaluating a policy in the general case of stochastic environments requires integrating over all possible outcomes which can be computationally costly, driving people to use heuristics when estimating utilities (Lieder, Griffiths, & Hsu, 2018), like relying on individual memories of past

trials (Duncan & Shohamy, 2016). While participants have extensive training on the effect of instructions in Lightbot and have access to a program simulator, effectively using mental resources and available time requires judicious use of simulation (Hamrick, Smith, Griffiths, & Vul, 2015; Ullman & Wang, 2023). We found that the most common programs participants wrote required less program evaluation, which suggests that the computational cost of evaluating programs may influence how participants choose among programs. A previous study used the Lightbot domain to examine how execution-related properties of programs influence prior beliefs about programs (Ho, Sanborn, Callaway, Bourgin, & Griffiths, 2018)—a natural extension of that analysis could examine whether these computational properties influence the process of search directly, or only by way of influencing program evaluation. Future work could use explicit representations of hierarchy, like in our experiment, in order to finely probe how differences in hierarchy influence mental simulation.

Hierarchy is a key organizational strategy that humans use to structure their behavior in order to make planning and learning efficient (Botvinick et al., 2009; Newell & Simon, 1972), but is difficult to study, requiring indirect measures to infer internal hierarchical representations (Huys et al., 2015; Rosenbaum et al., 1983; Verwey, 1996). In this article, we used a process-tracing paradigm to observe the hierarchical representations used to solve the task, which allowed us to identify that people have a bias towards reuse captured by our generative model of grammar induction. We hope our approach can inspire future efforts to externalize the representations underlying complex mental processes such as planning.

## CRediT authorship contribution statement

**Carlos G. Correa:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Sophia Sanborn:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Formal analysis, Conceptualization. **Mark K. Ho:** Writing – review & editing, Methodology, Formal analysis, Conceptualization. **Frederick Callaway:** Writing – review & editing, Methodology, Formal analysis, Conceptualization. **Nathaniel D. Daw:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Formal analysis, Conceptualization. **Thomas L. Griffiths:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Formal analysis, Conceptualization.

## Acknowledgments

This research was supported by John Templeton Foundation grant 61454 awarded to TLG and NDD (https://www.templeton.org/), U.S. Air Force Office of Scientific Research grant FA 9550-18-1-0077 awarded to TLG (https://www.afrl.af.mil/AFOSR/), and U.S. Army Research Office grant ARO W911NF-16-1-0474 awarded to NDD (https://www.arl.army.mil/who-we-are/directorates/aro/). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

## Appendix A

### A.1. Algorithms

In this section, we include an algorithm for program execution (see Algorithm 1) and a generative algorithm that returns samples from the grammar induction prior (see Algorithm 2).

**Algorithm 1** A recursive algorithm for program execution.

**Input:**
  Current subroutine $\rho^i$
  Start time $t$
**Output:** End time $t$
  **for** $\rho^i_j$ in $\rho^i$ **do**
    **if** $\rho^i_j \in \mathcal{A}$ **then** ▷ Instruction $\rho^i_j$ is the action $a_t$
      $s_{t+1} \leftarrow T(s_t, \rho^i_j)$
      **if** $s_{t+1} \in \mathcal{G}$ **then**
        Halt program execution because a goal has been reached
      **end if**
      $t \leftarrow t + 1$
    **else** ▷ Instruction $\rho^i_j$ is some subroutine $\rho^k$
      $t \leftarrow recurse(\rho^i_j, t)$ ▷ Recursively execute the subroutine
    **end if**
  **end for**

**Algorithm 2** Algorithm for sampling from the grammar induction prior. For simplicity, the programs generated by this algorithm can have an unlimited number of subroutines, though we only examine programs with four subroutines in the text.

**Output:** Program $\pi = (\rho^0, \rho^1, \ldots)$
**function** INSTRUCTION()
  **if** $true \sim Bernoulli(p_{call})$ **then**
    $k \sim p(z_{n+1} \mid z_{1:n})$ ▷ Sample subroutine, Eq. (3)
    **if** $\rho^k$ is not defined **then** ▷ Generate if new subroutine
      define $\rho^k$ ▷ Defining since recursive program is possible
      $\rho^k \sim$ SUBROUTINE()
    **end if**
    **return** $\rho^k$
  **else**
    $a \sim Uniform(\mathcal{A})$ ▷ Sample from actions $\mathcal{A}$
    **return** $a$
  **end if**
**end function**
**function** SUBROUTINE()
  Initialize $\rho$ as an empty subroutine
  $j \leftarrow 0$
  **repeat**
    $\rho_j \sim$ INSTRUCTION()
    $j \leftarrow j + 1$
  **until** $true \sim Bernoulli(p_{end})$
  **return** $\rho$
**end function**
$\rho^0 \sim$ SUBROUTINE() ▷ Initiate sampling of program

### A.2. Trace search

Our strategy for trace search features three components: (1) find a large corpus of traces (2) by using heuristic search methods, (3) while avoiding trivial traces.

To minimize bias in the programs we generate, we search for the shortest traces, taking a minimum of $m = 1000$ traces. To avoid any effects of tie-breaking, we continue searching to ensure we find all traces of equivalent cost to the $m$th trace.

We search in the space of traces using a variant of A* search (Hart, Nilsson, & Raphael, 1968) that is adapted to find the best $m$ traces and continue searching to avoid tie-breaking effects. Our algorithm closely resembles m-A* search (Flerova, Marinescu, & Dechter, 2016), which finds the top $m$ solutions for a problem. To ensure search efficiency, we use a heuristic based on shortest-path lengths. In particular, while the number of traces grows exponentially in trace length, the number of task states is finite (though exponential in the number of lights),

making it reasonable to compute the shortest path to a goal from any state. So, the heuristic cost function simply returns the shortest path length from the trace's final state to any goal. By design, the heuristic is monotone (which we also empirically verify), so this ensures the results are optimal given the assumptions of A* search (Flerova et al., 2016; Russell & Norvig, 2021).

In order to accelerate search, we avoid traces with trivial action sequences. We required traces to consist of actions that resulted in changes to the state, so $T(s_t, a_t) \neq s_t$. We also excluded certain sequences that are redundant or symmetric: three left turns, three right turns, a right turn followed by or preceding a left turn, and light instructions after turns.

### A.3. Program generation

We expand an execution trace into many possible programs that generate the trace by rewriting the trace using all combinations of possible subroutines. Subroutines are only considered if they are called in at least two places in the resulting program and contain at least two instructions. To match the experimental interface, programs could only contain at most four subroutines. Trace rewriting uses a greedy algorithm that rewrites to maximize subroutine use, considering one subroutine at a time from longest to shortest.

We add special cases to generate programs consistent with those written by some participants. One case optionally adds post-goal actions to the trace that are consistent with some candidate subroutine. Though post-goal actions are extraneous, when produced by a subroutine, they can still result in an overall reduction in program length. A second case involves recursive subroutines—subroutines that call themselves repeatedly until the goal is reached. We generate these by proposing all trace suffixes as subroutines. A final case combines these: recursive programs with post-goal actions.

Since these programs are used to approximate the posterior, we minimize bias by ensuring that participant programs are in this corpus of programs as much as possible. We do so by searching for traces in order of ascending trace length, taking a minimum of $n = 1000$ traces but continuing to search for traces of equivalent cost to the $n$th trace. Using A* search to find multiple solutions requires searching over state trajectories, as opposed to simply searching over states. The cost of the $n$th trace is the maximum trace cost we consider, so participants with programs of greater cost have programs that are not in this set. Using these methods, 78% of participant programs were generated and 94% of participant programs written by at least one other participant were generated. Missing programs largely correspond to programs excluded for the purpose of efficiency, as noted above: greater than maximum trace cost, use of actions that result in no state change, use of redundant instruction sequences, and post-goal actions due to a subroutine used only once in the pre-goal program.

### A.4. Examining how step cost influences program creation

The model that is the best fit to behavior in the text (grammar induction + step cost) incorporates the step cost prior, which minimizes trace length. While the qualitative examples in the main text focused extensively on providing support for the reuse-based grammar induction account, we did not provide examples showing support for the step cost prior. In this section, we review some examples that show how step costs inform participant choices, leading them to underuse a subroutine.

The first two columns of Fig. A.1 show programs with a shared subroutine (`Turn Left`, `Jump`, `Activate Light`), where the program in the first column was created by participants and the one in the second column was found by our program search methods. The two programs differ very slightly: the one participants wrote has a subroutine used three times (instead of four), a longer program length, but a shorter trace length. Looking closely at the resulting trace, the programs are identical until they diverge in their approach to the final light. Participants take a more direct route to the final light (`Walk`, `Turn Right`, `Jump`, `Activate Light`), instead of taking a longer route that requires one additional turn but can use the subroutine (`Turn Right`, `Walk`, `Turn Left`, `Jump`, `Activate Light`). Participant preferences between these programs are inconsistent with the MDL or grammar induction accounts, but can be explained by the step cost prior.

The third and fourth columns of Fig. A.1 provide a very similar example. The two programs share a subroutine (`Walk`, `Turn Left`, `Jump`, `Activate Light`) and only differ in their approach to the final light. As above, the program that participants created has three subroutine uses (instead of four), a longer program, and a shorter trace. This example can also be explained by the step cost prior.

We also include a fifth column which has the simplest subroutine (`Jump`, `Activate Light`), to show that participants will readily use a subroutine four times in this task.

These examples seem to suggest that the process of subroutine use might be informed by step costs, particularly since participants never create the shorter programs (in the second and fourth columns).

While our examples in the main text either have entirely different traces (Fig. 4) or identical traces (Fig. 5, Fig. 6), this example highlights differences in participant choices near the end of a program. We think a promising direction for future experiments and analyses could focus on examples like this, where the trace length and grammar induction/MDL models make different predictions.

### A.5. Controlling for effects of program preprocessing

Our primary analysis in the main text compares models based on how well they predict programs. As reported, these programs have been preprocessed. Because preprocessing modifies programs in a way that could favor the grammar induction account, in particular since it maximizes the use of existing subroutines, we run a variant of the analysis that avoids any preprocessing of programs.

Our results are very similar to those reported in the main text. The BIC, log likelihood, and fitted parameters for each model are reported in Table A.1. We found that all models were more predictive of behavior than a random choice model (Likelihood-ratio test for step cost: $\chi^2(1) = 10090.8$, $p < .001$, MDL: $\chi^2(1) = 6558.9$, $p < .001$, grammar induction: $\chi^2(3) = 12007.5$, $p < .001$) or a step cost model (Likelihood-ratio test for MDL: $\chi^2(1) = 301.4$, $p < .001$, grammar induction: $\chi^2(3) = 3519.8$, $p < .001$). We found that the grammar induction model (with or without step costs) was most predictive of behavior (Fig. A.2(a)). These qualitative results generally held on a task-specific basis (Fig. A.2(b)).

Of note, however, is that the $\alpha$ parameter is much higher for the grammar induction models than the analysis reported in the main text. While a bias towards reuse among existing subroutines is generally present in models due to the rich-get-richer dynamics of subroutine sampling, a high $\alpha$ leads to a preference for subroutine creation. This could be driven by the number of subroutines used only once, which are inlined by program preprocessing (Step 4 in Table 1), but present in this dataset.

### A.6. Controlling for programming experience

In our model comparison in the main text, we included participants with programming experience equivalent to between 1 and 3 college courses. In order to control for any influence of this prior experience, we run our primary analyses in the subset of participants ($N = 125$) that have no programming experience.

We found extremely similar results as our analysis in the main text. Table A.2 lists the BIC, log likelihood, and fit parameters for each model. We found that models were an improvement over random choice (Likelihood-ratio test for step cost: $\chi^2(1) = 9416.8$, $p < .001$, MDL: $\chi^2(1) = 6636.6$, $p < .001$, grammar induction: $\chi^2(3) = 12160.8$, $p <$

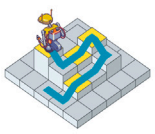
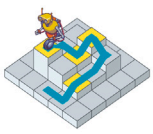
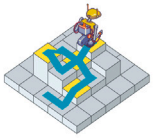
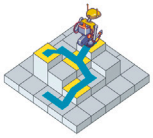
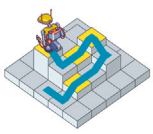
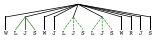
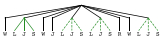
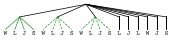
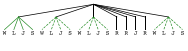
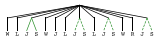
| | | | | | |
|---|---|---|---|---|---|
| trace |  |  |  |  |  |
| program |  |  |  |  |  |
| part. count | 10 | 0 | 5 | 0 | 9 |
| step count | 16 | 17 | 18 | 20 | 16 |
| prog. length | 13 | 12 | 13 | 12 | 14 |
| grammar induction prior (log) | -31.97 | -28.24 | -31.97 | -28.24 | -33.05 |

**Fig. A.1.** Example programs demonstrating the influence of step costs on participant programs. The most common hierarchical program (first column) uses a subroutine (`Turn Left`, `Jump`, `Activate Light`) three times. A program discovered by program search (second column) has the same subroutine and uses it four times. This pattern of choice is inconsistent with MDL and grammar induction, but is explained by step count. Third and fourth columns are a similar example, with the subroutine `Walk`, `Turn Left`, `Jump`, `Activate Light`. Fifth column shows that participants will use a shorter subroutine (`Jump`, `Activate Light`) four times. See Fig. 4 for more detail about this figure.

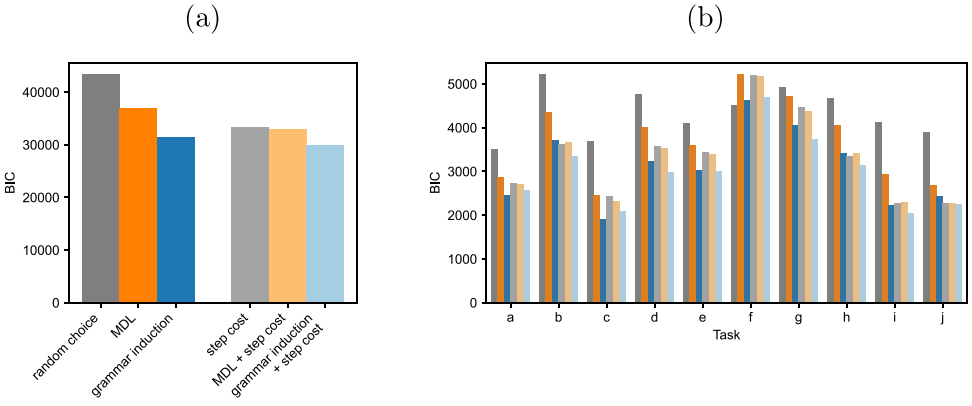



**Fig. A.2.** Model comparison of accounts, without program preprocessing. (a) Plot of BIC of experimental data as predicted by each of the models, after parameter fitting. Models with smaller BIC are a better account of behavior. (b) BIC of data under each model, but split by task. Parameters are the same as in (a), so they are the best fit for all tasks. The color of the models is also the same as in (a). Task letter is a reference to the subfigure in Fig. 3.

**Table A.1**
Table of fitted models to data without program preprocessing, with log likelihood, BIC, parameter count, and fitted parameters.

| | Log likelihood | BIC | Param. count | Parameters |
|---|---|---|---|---|
| random choice | −21710.4 | 43420.7 | 0 | |
| MDL | −18430.9 | 36869.3 | 1 | $\beta_{\text{MDL}} = 0.82$ |
| grammar induction | −15706.6 | 31435.5 | 3 | $\alpha = 68.12$<br>$\beta_{\text{GrammarInduction}} = 0.38$<br>$p_{call} = 0.28$ |
| step cost | −16665.0 | 33337.4 | 1 | $\beta_{\text{StepCost}} = 1.18$ |
| MDL + step cost | −16514.3 | 33043.4 | 2 | $\beta_{\text{MDL}} = 0.26$<br>$\beta_{\text{StepCost}} = 0.96$ |
| grammar induction + step cost | −14905.1 | 29839.8 | 4 | $\alpha = 45.48$<br>$\beta_{\text{GrammarInduction}} = 0.26$<br>$p_{call} = 0.17$<br>$\beta_{\text{StepCost}} = 0.59$ |

.001) and also an improvement over a baseline of step cost (Likelihood-ratio test for MDL + step cost: $\chi^2(1) = 377.8$, $p < .001$, grammar induction + step cost: $\chi^2(3) = 4110.4$, $p < .001$). Judged by BIC, the grammar induction model (with or without step cost) was the most predictive of behavior (Fig. A.3(a)). These patterns generally held when examined on a per-task basis (Fig. A.3(b)).

### *A.7. Does programming experience predict task performance?*

Another question of interest is whether programming experience has an influence on measures of task performance, like how quickly tasks were completed. In order to test these questions, we analyze all 193 participants that completed the task, which included the 22 participants with programming experience equivalent to > 3 college courses, who were excluded from analyses in the main text. For each measure of task

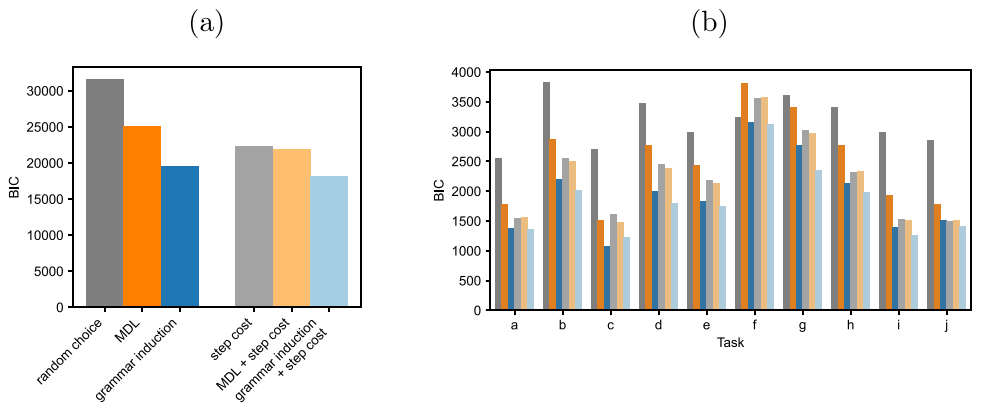



**Fig. A.3.** Model comparison of accounts, controlling for past programming experience. (a) Plot of BIC of experimental data as predicted by each of the models, after parameter fitting. Models with smaller BIC are a better account of behavior. (b) BIC of data under each model, but split by task. Parameters are the same as in (a), so they are the best fit for all tasks. The color of the models is also the same as in (a). Task letter is a reference to the subfigure in Fig. 3.

**Table A.2**
Table of fitted models to data excluding participants with programming experience, with log likelihood, BIC, parameter count, and fitted parameters.

| | Log likelihood | BIC | Param. count | Parameters |
|---|---|---|---|---|
| random choice | −15840.5 | 31680.9 | 0 | |
| MDL | −12522.2 | 25051.5 | 1 | $\beta_{\text{MDL}} = 0.95$ |
| grammar induction | −9760.1 | 19541.5 | 3 | $\alpha = 4.15$<br>$\beta_{\text{GrammarInduction}} = 0.48$<br>$p_{call} = 0.10$ |
| step cost | −11132.1 | 22271.3 | 1 | $\beta_{\text{StepCost}} = 1.38$ |
| MDL + step cost | −10943.2 | 21900.6 | 2 | $\beta_{\text{MDL}} = 0.36$<br>$\beta_{\text{StepCost}} = 1.07$ |
| grammar induction + step cost | −9076.9 | 18182.2 | 4 | $\alpha = 2.16$<br>$\beta_{\text{GrammarInduction}} = 0.35$<br>$p_{call} = 0.06$<br>$\beta_{\text{StepCost}} = 0.64$ |

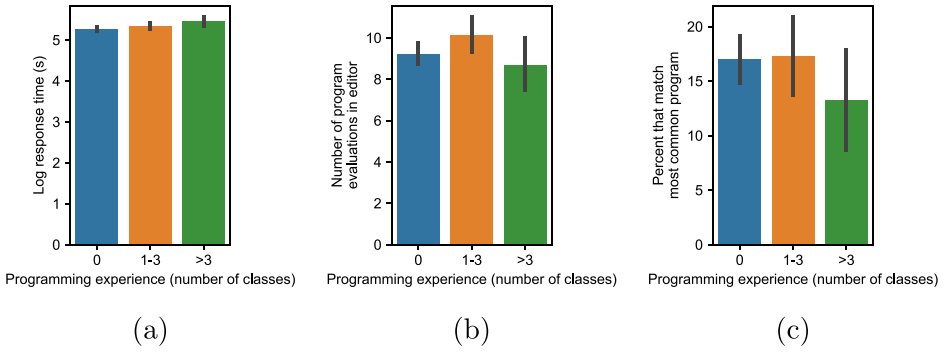



**Fig. A.4.** Plotting measures of task performance for different levels of programming experience. Values are (a) log-transformed response times (s), (b) the number of program evaluations, and (c) the percentage of programs that match the most common program. Error bars show the 95% confidence interval of the mean, estimated by bootstrapping.

performance, we used a likelihood-ratio test to see whether adding a fixed effect for programming experience would significantly improve upon a null model, which predicted the measure with a fixed intercept and random per-participant intercepts. We found that programming experience did not improve predictions of task performance for any measures we considered, which consisted of response times ($\chi^2(2) = 3.04$, $p = .218$; Fig. A.4(a)), number of program evaluations ($\chi^2(2) = 0.94$, $p = .625$; Fig. A.4(b)), and percentage of programs that match the common program ($\chi^2(2) = 2.08$, $p = .353$; Fig. A.4(c)). These findings suggest that prior programming instruction or experience may have minimal impact on task performance in Lightbot.

Another test that could be impacted by our inclusion of those with programming experience is our analysis of performance characteristics of those who wrote common programs (in Fig. 8). In particular, in the main text we found that participants, when writing common programs, completed the task more quickly and with fewer program evaluations. Running the same analysis with the 125 participants that had no programming experience, we found similar results of faster responses ($\beta = -0.011$, $\chi^2(1) = 43.77$, $p < .001$, Fig. A.5(a)) and fewer program evaluations ($\beta = -0.054$, $\chi^2(1) = 6.34$, $p = .012$, Fig. A.5(b)).

### *A.8. Examining variance in task solutions*

We examine the variability in how participants solve tasks, focusing on two questions: How does this variability relate to task performance? Can we use our models to predict which tasks will elicit greater

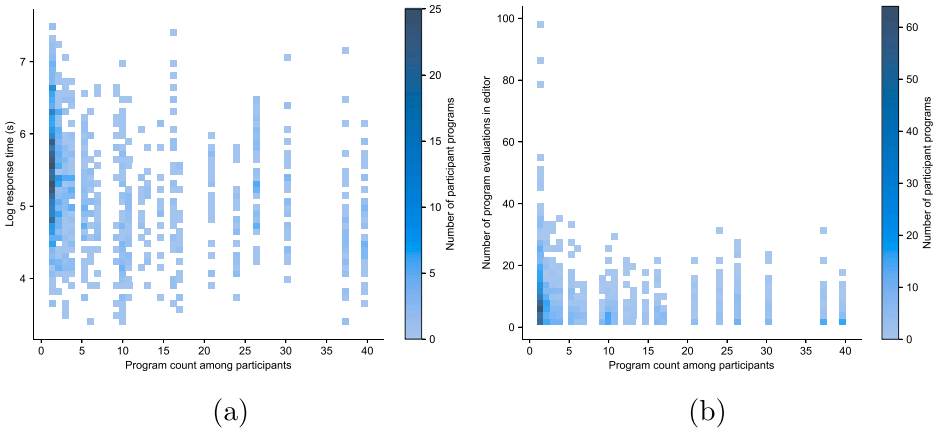



**Fig. A.5.** Participants without programming experience write common programs faster and with fewer program evaluations. Bivariate histogram of task responses for each participant on each task, with responses binned along horizontal axis by program count and along vertical axis by either (a) log-transformed response times (s) or (b) the number of program evaluations. A related regression analysis is reported in the text, which uses program count to predict response times and number of program evaluations while controlling for program length and cost, and finds results consistent with these plots.

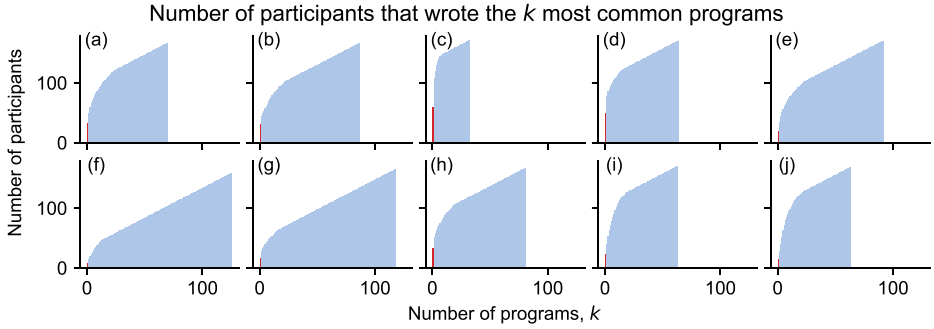



**Fig. A.6.** Plotting the cumulative number of participants that wrote the $k$ most common programs. The bar corresponding to the most common program is the leftmost bar in each subfigure, at left and in red. Task label is a reference to subfigure in Fig. 3.

**Table A.3**
Measures of task properties, difficulty, and solution variance. Each column is a different task, with letter referencing a subfigure in Fig. 3. Rows report task properties (light count), difficulty (mean across participants of log response time, mean across participants of number of program evaluations, number of times task was skipped), and solution variance (number of unique solutions, and number of participants that wrote the most common program).

| Task | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| Light count | 3 | 8 | 3 | 6 | 3 | 7 | 8 | 6 | 4 | 3 |
| Mean log response time (seconds) | 5.69 | 5.71 | 4.87 | 5.41 | 5.27 | 5.72 | 5.50 | 5.22 | 4.88 | 4.61 |
| Mean program evaluations | 9.31 | 12.55 | 5.42 | 9.92 | 8.69 | 14.80 | 12.24 | 9.60 | 7.14 | 5.60 |
| Times skipped | 5 | 5 | 0 | 1 | 1 | 13 | 7 | 5 | 2 | 3 |
| Number of unique solutions | 69 | 86 | 32 | 63 | 91 | 125 | 117 | 80 | 62 | 63 |
| Number of participants with most common program | 32 (19%) | 31 (19%) | 59 (35%) | 50 (29%) | 19 (11%) | 8 (5%) | 16 (10%) | 32 (19%) | 23 (14%) | 15 (9%) |

variability? We measure variance in problem solutions in two ways: the number of unique solutions, and the number of participants that wrote the most common program. The quantities we analyze in this section are reported in Table A.3 and a histogram of program frequencies is in Fig. A.6.

We first examine whether solution variability can be linked to measures of task difficulty previously examined: response times and number of program evaluations (Fig. A.7). We report Spearman's rank correlation coefficient between measures of difficulty and variance, using a permutation test to evaluate statistical significance. We found that unique solution count had a positive relationship with increased task difficulty (program evaluation count: $\rho = 0.78$, $p = .01$, $N = 10$, response times: $\rho = 0.73$, $p = .021$, $N = 10$). We found a negative relationship (that did not reach statistical significance) between task difficulty and the number of participants who wrote the most common program (program evaluation count: $\rho = -0.29$, $p = .414$, $N = 10$, response times: $\rho = -0.20$, $p = .572$, $N = 10$). These results both suggest that tasks with greater solution variability also tend to be more difficult.

Can we predict the empirical variability we see, using the predictions of our models? The variability could arise from the competing objectives that people are balancing. For example, our best fitting model combines two objectives: the grammar induction prior and the step cost prior. In cases where these two priors disagree, differences in their relative weight could lead to considerably different model predictions. So any variability in their relative weight in our research participants could be a source of solution variability.

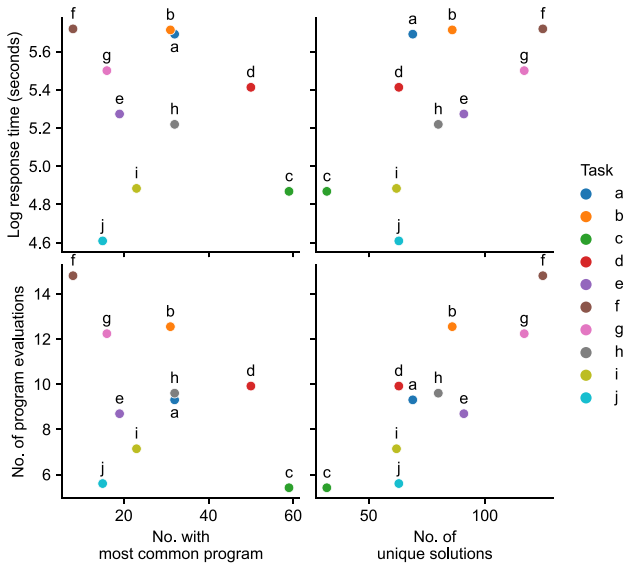



**Fig. A.7.** Plotting measures of task difficulty (response times, number of program evaluations) and solution variability (unique solution count, number of participants with most common program). Task label is a reference to subfigure in Fig. 3.

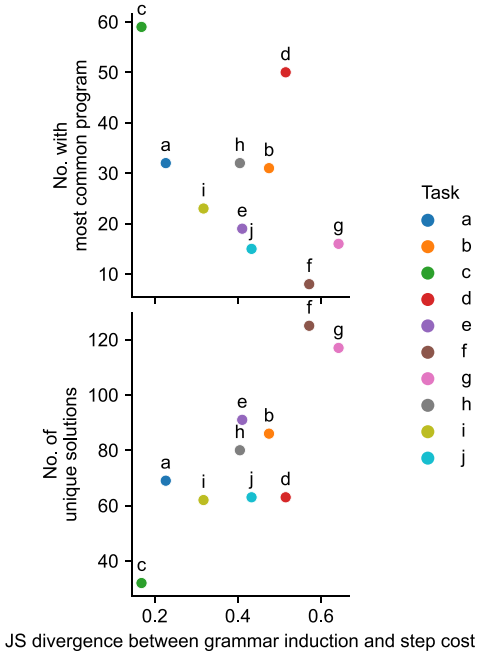



**Fig. A.8.** Plotting a measure of theoretical solution variability (the JS divergence between the grammar induction and step cost priors, using fitted parameters from Table 2) against two measures of empirical solution variability (number of unique solutions, and number of participants that wrote the most common program). Task label is a reference to subfigure in Fig. 3.

We test this idea by seeing whether our measures of empirical solution variance are related to a theoretical measure about the level of disagreement between our theories of program writing. We quantify the disagreement between two theories as the Jensen–Shannon (JS) divergence of their distributions over programs, using the fitted parameters in Table 2. The JS divergence is

$$D_{JS}(p \,||\, q) = \frac{1}{2}\left[D_{KL}(p \,||\, m) + D_{KL}(q \,||\, m)\right]$$

where $m(\rho) = \frac{1}{2}\left[p(\rho) + q(\rho)\right]$ is a mixture of the two distributions and $D_{KL}(p \,||\, m) = \sum_{\rho} p(\rho) \log\left(\frac{p(\rho)}{m(\rho)}\right)$ is the Kullback–Leibler divergence.

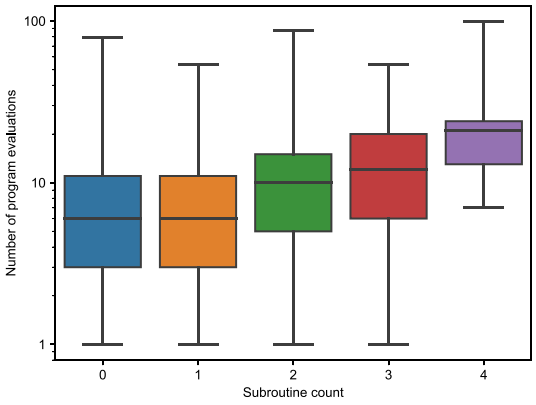



**Fig. A.9.** Plotting the number of program evaluations for trials with varying numbers of subroutines. The box shows data quartiles, while whiskers show range of the data.

The JS divergence is a non-negative measurement of how different two distributions are, taking the value of 0 when the two distributions are the same.

Since our best model combines the grammar induction and step cost priors, we used the JS divergence between these two priors as a measure of theoretical disagreement, and compared them to our measures of empirical solution variability (Fig. A.8). We found a positive relationship between the JS divergence and the number of unique solutions ($\rho = 0.69$, $p = .034$, $N = 10$) and a negative (but not statistically significant) relationship between the JS divergence and the count of participants that wrote the most common program ($\rho = -0.58$, $p = .083$, $N = 10$). The direction of these correlations is consistent with the idea that disagreement between priors could be a source of increased solution variance.

### *A.9. The influence of program evaluation on subroutine creation*

In contrast to typical studies of planning, where a plan might be updated in the course of its execution, our task permits the testing and revision of a plan. A natural question is whether the ability to evaluate programs in Lightbot has an influence on the programs people write. One simple test is to see whether the number of subroutines participants used was associated with the number of times they evaluated their program. A positive relationship might suggest that the use of subroutines requires validating the subroutine has the expected result. A negative relationship might suggest that participants avoid validating subroutines because it is easy to reason about their expected results. We find a positive relationship between program evaluations and subroutine counts ($\rho = 0.21$, $p < .001$, $N = 1668$, Fig. A.9), which holds even after excluding outlier trials where participants evaluated programs more than 40 times ($\rho = 0.20$, $p < .001$, $N = 1649$). While we avoid any causal interpretation of these results, we hope future research can continue to examine the influence of process-tracing paradigms on behavior.

### *A.10. Individual differences in subroutine use*

Our analyses have largely focused on characterizing behavior at the group level. However, there is considerable variety in the programs that people write. Are these programs merely superficially different, with similar structural characteristics, or do participants have different structural preferences? We investigate this at the individual level (averaging over tasks) by plotting program length against attributes related to subroutines—the number of subroutines (Fig. A.10(a)), and their average length (Fig. A.10(b)). Testing the relationship between these variables, we find a significant negative correlation between program length and subroutine count ($\rho = -0.79$, $p < .001$, $N =$

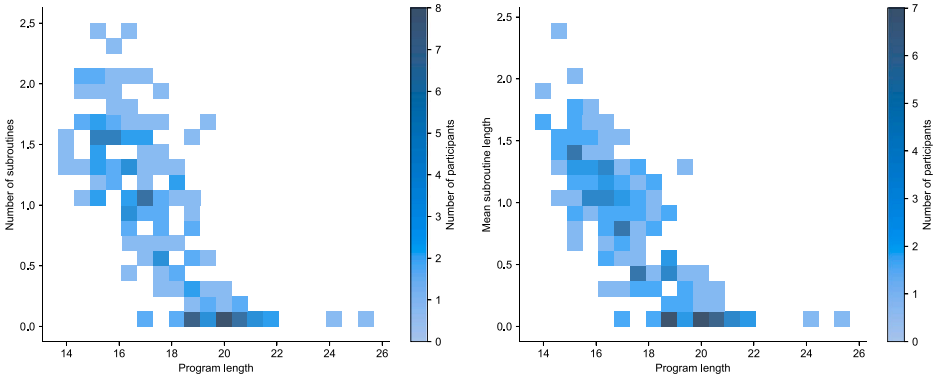



**Fig. A.10.** Showing individual differences between participants by plotting program length against (a) number of created subroutines and (b) the average length of created subroutines. Participant-level values are averaged across tasks.

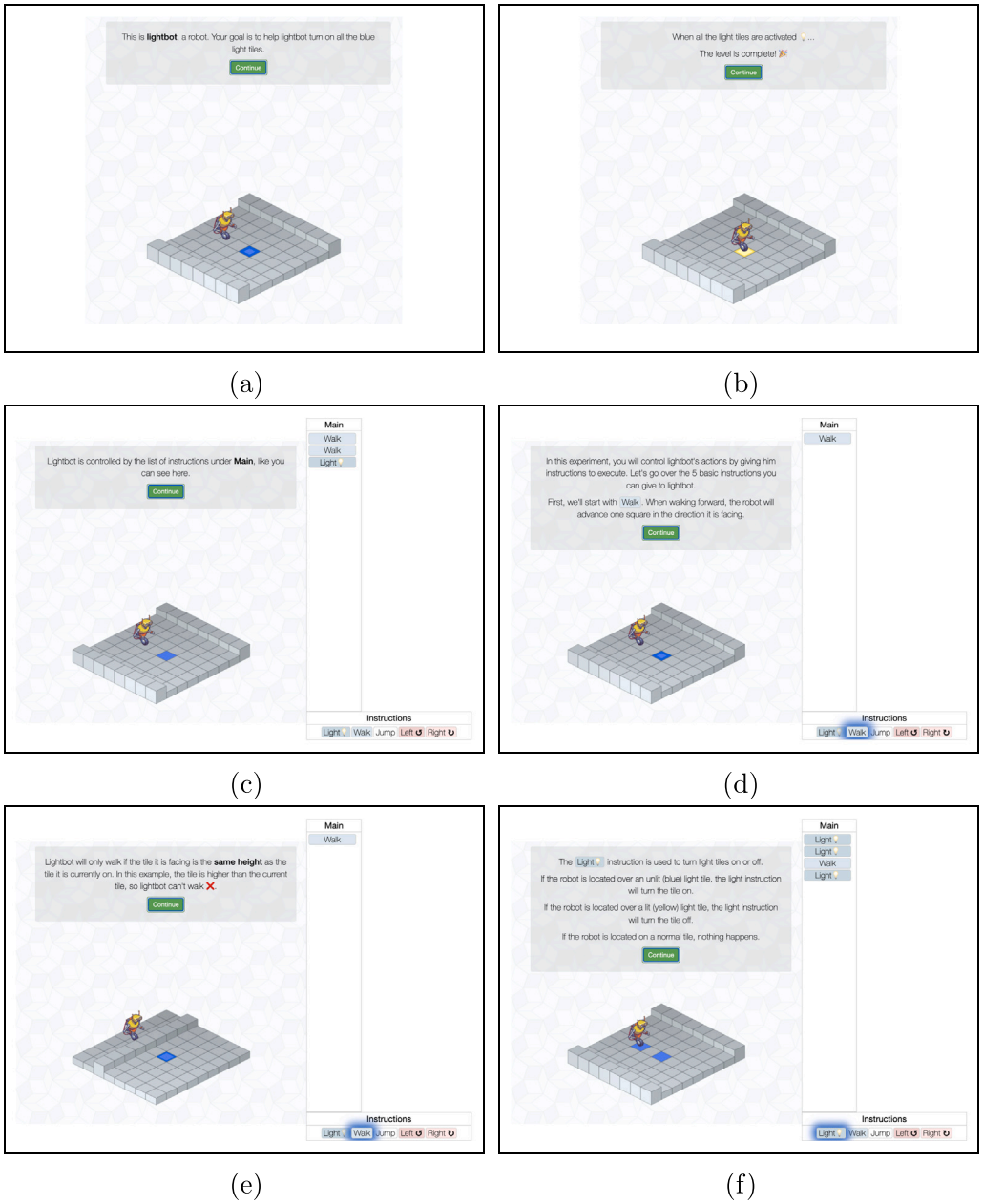



**Fig. A.11.** Screenshots of the experiment tutorial, in the same order as presented to research participants.

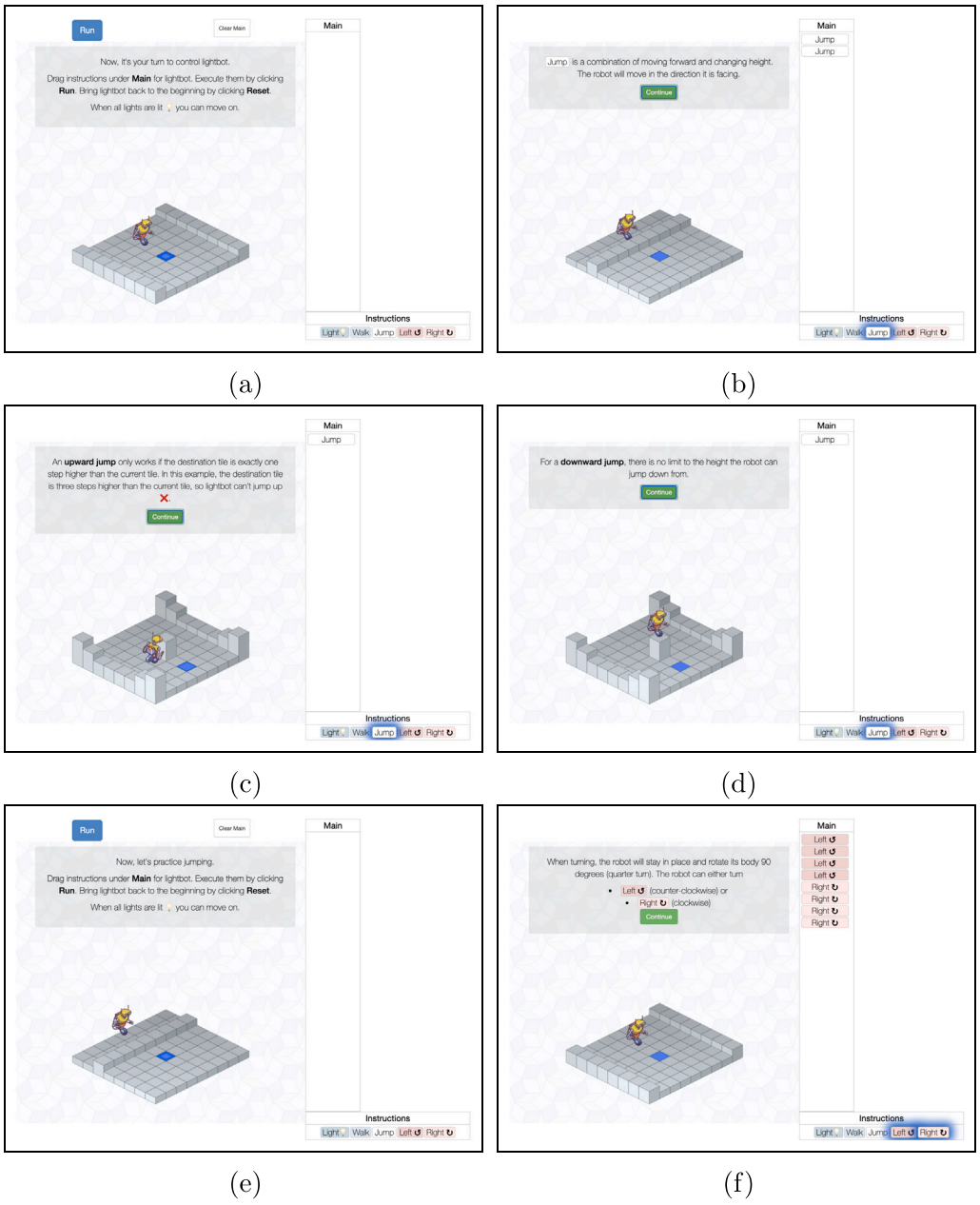



**Fig. A.12.** Screenshots of the experiment tutorial, continued from Fig. A.11.

171), as well as between program length and mean subroutine length ($\rho = -0.83$, $p < .001$, $N = 171$). These findings suggest that the variety in participant programs is fit well by a single dimension, with short programs and more/longer subroutines at one extreme, and long programs with fewer/shorter subroutines at the other.

### *A.11. Experiment tutorial*

The complete experiment tutorial is presented in Fig. A.11 to Fig. A.14. It explains the task (Figs. A.11(a)–A.11(c)), how to use each of the program instructions (Figs. A.11(d)–A.13(a)), and how to use and create subroutines (Figs. A.13(b)–A.13(c)). Screenshots with a *Run* button at upper left require the solution of a task (e.g., Fig. A.12(a)), while others are simply informational. Then, participants complete a more complex practice problem (Fig. A.13(d)). Finally, instructions for the incentive for minimizing instruction count (i.e. program length) are shown in Fig. A.14. It shows a number of example programs for a given trace, in order to demonstrate how program length varies with the choice of subroutines.

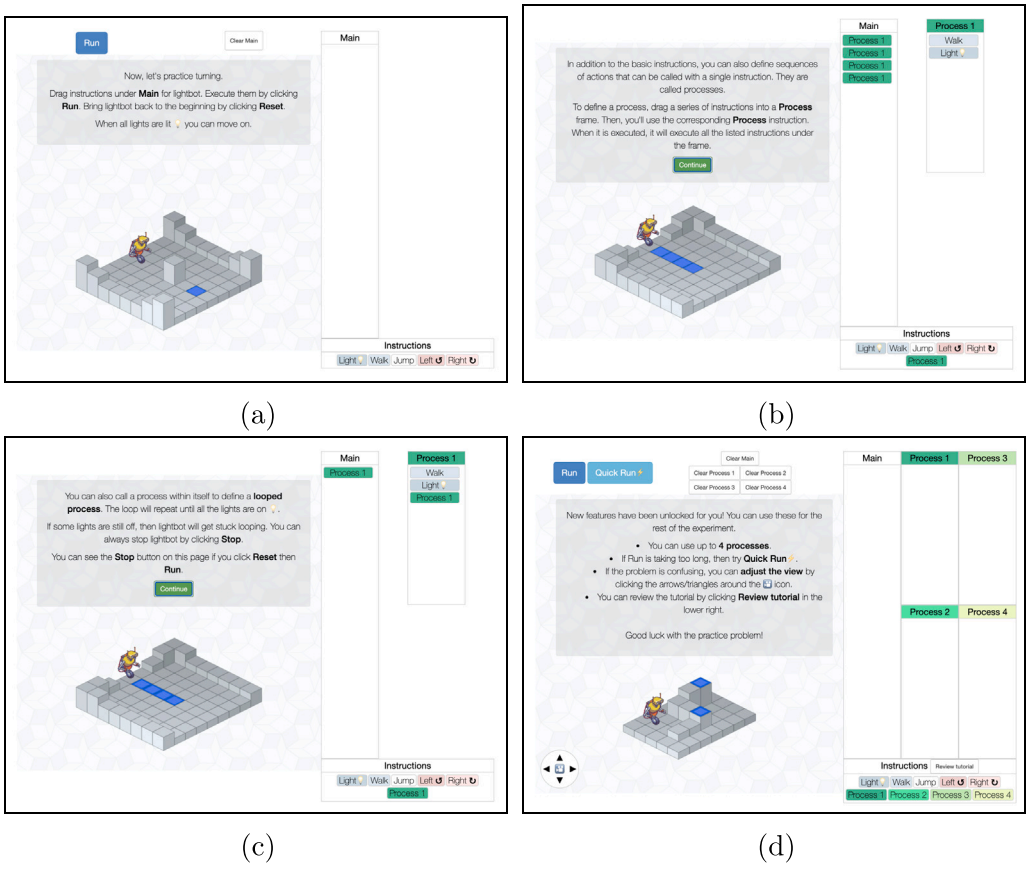



**Fig. A.13.** Screenshots of the experiment tutorial, continued from Fig. A.12.

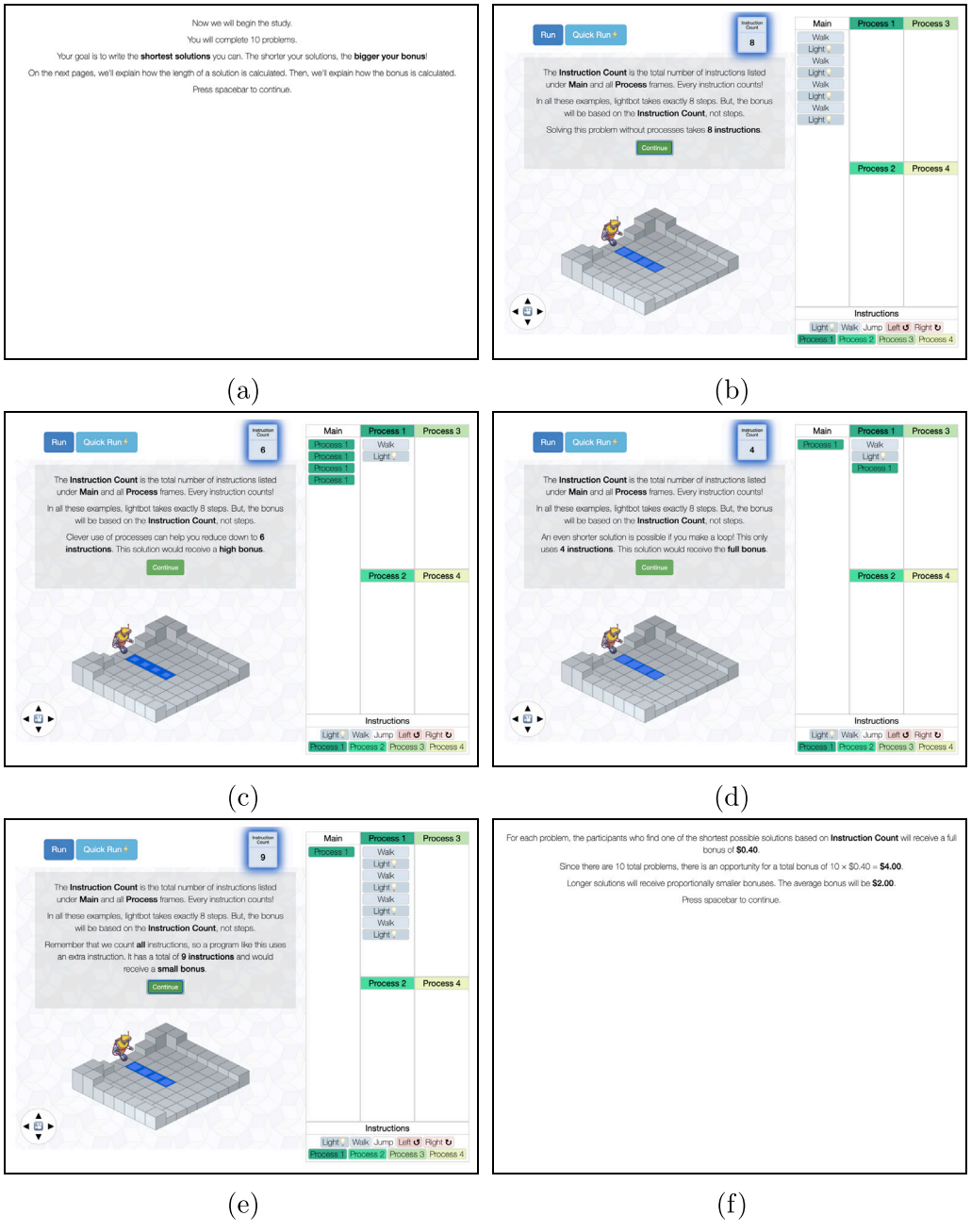



**Fig. A.14.** Screenshots of the experiment tutorial, continued from Fig. A.13.

## Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cognition.2024.105990.

## Data availability

See the Data Availability section in the manuscript.

## References


Acuna, D. E., Wymbs, N. F., Reynolds, C. A., Picard, N., Turner, R. S., Strick, P. L., et al. (2014). Multifaceted aspects of chunking enable robust algorithms. *Journal of Neurophysiology*, *112*, 1849–1856.

Aldous, D. J. (1985). Exchangeability and related topics. In D. J. Aldous, I. A. Ibragimov, J. Jacod, & P. L. Hennequin (Eds.), *École d'Été de Probabilités de Saint-Flour XIII—1983* (pp. 1–198). Berlin, Heidelberg: Springer.

Anderson, J. R. (1991). The adaptive nature of human categorization. *Psychological Review*, *98*, 409–429.

Balaguer, J., Spiers, H., Hassabis, D., & Summerfield, C. (2016). Neural mechanisms of hierarchical planning in a virtual subway network. *Neuron*, *90*, 893–903.

Bear, A., Bensinger, S., Jara-Ettinger, J., Knobe, J., & Cushman, F. (2020). What comes to mind? *Cognition*, *194*, Article 104057.

Blei, D. M., & Frazier, P. I. (2011). Distance dependent chinese restaurant processes. *Journal of Machine Learning Research*, *12*, 2461–2488.

Botvinick, M. M., Niv, Y., & Barto, A. G. (2009). Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition*, *113*, 262–280.

Buchsbaum, D., Griffiths, T. L., Plunkett, D., Gopnik, A., & Baldwin, D. (2015). Inferring action structure and causal relationships in continuous sequences of human action. *Cognitive Psychology*, *76*, 30–77.

Chater, N. (1999). The search for simplicity: A fundamental cognitive principle? *The Quarterly Journal of Experimental Psychology Section A*, *52*, 273–302.

Correa, C. G., Ho, M. K., Callaway, F., Daw, N. D., & Griffiths, T. L. (2023). Humans decompose tasks by trading off utility and computational cost. *PLoS Computational Biology*, *19*, 1–31.

Dasgupta, I., & Griffiths, T. L. (2022). Clustering and the efficient use of cognitive resources. *Journal of Mathematical Psychology*, *109*, Article 102675.

Daw, N. D., & Courville, A. C. (2007). The pigeon as particle filter. In *Proceedings of the 20th international conference on neural information processing systems* (pp. 369–376).

Dezfouli, A., & Balleine, B. W. (2012). Habits, action sequences and reinforcement learning. *European Journal of Neuroscience*, *35*, 1036–1051.

Doucet, A., D. Freitas, N., Gordon, N. J., et al. (2001). *Sequential Monte Carlo methods in practice*. Springer.

Duncan, K. D., & Shohamy, D. (2016). Memory states influence value-based decisions. *Journal of Experimental Psychology: General*, *145*, 1420–1426.

Eckstein, M. K., & Collins, A. G. E. (2020). Computational evidence for hierarchically structured reinforcement learning in humans. *Proceedings of the National Academy of Sciences*, *117*, 29381–29389.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., et al. (2021). Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation* (pp. 835–850). New York, NY, USA: Association for Computing Machinery.

Éltetö, N., & Dayan, P. (2023). Habits of mind: reusing action sequences for efficient planning. cs.

Éltetö, N., Nemeth, D., Janacsek, K., & Dayan, P. (2022). Tracking human skill learning with a hierarchical bayesian sequence model. *PLoS Computational Biology*, *18*, 1–28.

Flerova, N., Marinescu, R., & Dechter, R. (2016). Searching for the m best solutions in graphical models. *Journal of Artificial Intelligence Research*, *55*, 889–952.

Fränken, J. P., Theodoropoulos, N. C., & Bramley, N. R. (2022). Algorithms of adaptation in inductive inference. *Cognitive Psychology*, *137*, Article 101506.

Gershman, S. J., Blei, D. M., & Niv, Y. (2010). Context, learning, and extinction. *Psychological Review*, *117*, 197–209.

Goldwater, S., Griffiths, T. L., & Johnson, M. (2009). A bayesian framework for word segmentation: Exploring the effects of context. *Cognition*, *112*, 21–54.

Goodman, N. D., Mansinghka, V. K., Roy, D., Bonawitz, K., & Tenenbaum, J. B. (2008). Church: A language for generative models. In *Proceedings of the twenty-fourth conference on uncertainty in artificial intelligence* (pp. 220–229). AUAI Press.

Goodman, N. D., Tenenbaum, J. B., Feldman, J., & Griffiths, T. L. (2008). A rational analysis of rule-based concept learning. *Cognitive Science*, *32*, 108–154.

Hamrick, J. B., Smith, K. A., Griffiths, T. L., & Vul, E. (2015). Think again? the amount of mental simulation tracks uncertainty in the outcome. In *Proceedings of the 37th annual conference of the cognitive science society*.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*, 100–107.

Ho, M. K., Sanborn, S., Callaway, F., Bourgin, D., & Griffiths, T. (2018). Human priors in hierarchical program induction. In *Proceedings of the 2018 conference on cognitive computational neuroscience*.

Huys, Q. J. M., Lally, N., Faulkner, P., Eshel, N., Seifritz, E., Gershman, S. J., et al. (2015). Interplay of approximate planning strategies. In *Proc. of the National Academy of Sciences: vol. 112*, (pp. 3098–3103).

Johnson, M., Griffiths, T. L., & Goldwater, S. (2006). Adaptor grammars: A framework for specifying compositional nonparametric bayesian models. In *Advances in neural information processing systems*.

Kemp, C., Goodman, N. D., & Tenenbaum, J. B. (2010). Learning to learn causal models. *Cognitive Science*, *34*, 1185–1243.

Klir, G. J., & Simon, H. A. (1991). *The architecture of complexity*. Springer.

Lai, L., Huang, A. Z., & Gershman, S. J. (2022). Action chunking as policy compression. URL osf.io/preprints/psyarxiv/z8yrv.

Lake, B. M., & Piantadosi, S. T. (2020). People infer recursive visual concepts from just a few examples. *Computational Brain & Behavior*, *3*, 54–65.

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

Levine, S. (2018). Reinforcement learning and control as probabilistic inference: tutorial and review. arXiv:1805.00909.

Lieder, F., Griffiths, T. L., & Hsu, M. (2018). Overrepresentation of extreme events in decision making reflects rational use of cognitive resources. *Psychological Review*, *125*, 1–32.

Luce, R. D. (1959). On the possible psychophysical laws. *Psychological Review*, *66*, 81–95.

Luong, M. T., Frank, M. C., & Johnson, M. (2013). Parsing entire discourses as very long strings: Capturing topic continuity in grounded language learning. *Transactions of the Association for Computational Linguistics*, *1*, 315–326.

Maisto, D., Donnarumma, F., & Pezzulo, G. (2015). Divide et impera: subgoaling reduces the complexity of probabilistic inference and problem solving. *Journal of the Royal Society Interface*, *12*, Article 20141335.

Marr, D. (1982). *Vision*. W.H. Freeman.

Mattar, M. G., & Daw, N. D. (2018). Prioritized memory access explains planning and hippocampal replay. *Nature Neuroscience*, *21*, 1609–1617.

McNamee, D., Wolpert, D. M., & Lengyel, M. (2016). Efficient state-space modularization for planning: theory, behavioral and neural signatures. In *Advances in neural information processing systems*.

Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, *63*, 81–97.

Miller, G., Galanter, E., & Pribram, K. (1960). *Plans and the structure of behavior*. Henry Holt and Co..

Morris, A., Phillips, J., Huang, K., & Cushman, F. (2021). Generating options and choosing between them depend on distinct forms of value representation. *Psychological Science*, *32*, 1731–1746.

Newell, A., & Simon, H. A. (1972). *Human problem solving*. Prentice-Hall.

O'Donnell, T. J. (2015). *Productivity and reuse in language: a theory of linguistic computation and storage*. The MIT Press.

Perruchet, P., & Pacton, S. (2006). Implicit learning and statistical learning: one phenomenon, two approaches. *Trends in Cognitive Sciences*, *10*, 233–238.

Piantadosi, S. T., Tenenbaum, J. B., & Goodman, N. D. (2012). Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition*, *123*, 199–217.

Planton, S., van Kerkoerle, T., Abbih, L., Maheu, M., Meyniel, F., Sigman, M., et al. (2021). A theory of memory for binary sequences: Evidence for a mental compression algorithm in humans. *PLoS Computational Biology*, *17*, 1–43.

Poesia, G., & Goodman, N. D. (2023). Peano: learning formal mathematical reasoning. *Philosophical Transactions of the Royal Society of London A (Mathematical and Physical Sciences)*, *381*, Article 20220044.

Ramkumar, P., Acuna, D. E., Berniker, M., Grafton, S. T., Turner, R. S., & Kording, K. P. (2016). Chunking as the result of an efficiency computation trade-off. *Nature Communications*, *7*, 1–11.

Ribas-Fernandes, J. J., Solway, A., Diuk, C., McGuire, J. T., Barto, A. G., Niv, Y., et al. (2011). A neural signature of hierarchical reinforcement learning. *Neuron*, *71*, 370–379.

Rosenbaum, D. A., Kenny, S. B., & Derr, M. A. (1983). Hierarchical control of rapid movement sequences. *Journal of Experimental Psychology. Human Perception and Performance*, *9*, 86–102.

Rule, J., Schulz, E., Piantadosi, S. T., & Tenenbaum, J. B. (2018). Learning list concepts through program induction. In *Proceedings of the annual meeting of the cognitive science society*.

Rule, J. S., Tenenbaum, J. B., & Piantadosi, S. T. (2020). The child as hacker. *Trends in Cognitive Sciences*, *24*, 900–915.

Russell, S. J., & Norvig, P. (2021). *Artificial intelligence: a modern approach*. Pearson Education, Inc..

Sanborn, S., Bourgin, D. D., Chang, M., & Griffiths, T. L. (2018). Representational efficiency outweighs action efficiency in human program induction. In *Proceedings of the 40th annual conference of the cognitive science society*.

Schwarz, G. (1978). Estimating the Dimension of a Model. *The Annals of Statistics*, *6*, 461–464.

Şimşek, Ö., & Barto, A. G. (2009). Skill characterization based on betweenness. In *Advances in neural information processing systems*.

Solway, A., Diuk, C., Córdova, D., Barto, A. G., Niv, Y., & Botvinick, M. M. (2014). Optimal behavioral hierarchy. *PLoS Computational Biology*, *10*, 1–10.

Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. In *Abstraction, Reformulation, and Approximation: 5th International Symposium, SARA 2002 Kananaskis, Alberta, Canada August 2–4, 2002 Proceedings*: *vol. 5*, (pp. 212–223). Springer.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Tomov, M. S., Yagati, S., Kumar, A., Yang, W., & Gershman, S. J. (2020). Discovery of hierarchical representations for efficient planning. *PLoS Computational Biology*, *16*, 1–42.

Tosatto, L., Fagot, J., Nemeth, D., & Rey, A. (2022). The evolution of chunks in sequence learning. *Cognitive Science*, *46*, Article e13124.

Toussaint, M., & Storkey, A. (2006). Probabilistic inference for solving discrete and continuous state markov decision processes. In *Proceedings of the 23rd international conference on machine learning* (pp. 945–952). New York, NY, USA: Association for Computing Machinery.

Ullman, T. D., & Wang, Y. (2023). Resource bounds on mental simulations: evidence from a fluid-reasoning task. osf.io/preprints/psyarxiv/z8yrv.

Verwey, W. B. (1996). Buffer loading and chunking in sequential keypressing. *Journal of Experimental Psychology. Human Perception and Performance*, *22*, 544–562.

Wen, Z., Precup, D., Ibrahimi, M., Barreto, A., Va. Roy, B., & Singh, S. (2020). On efficiency in hierarchical reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 6708–6718). Curran Associates, Inc..

Wingate, D., Goodman, N. D., Roy, D. M., Kaelbling, L. P., & Tenenbaum, J. B. (2011). Bayesian policy search with policy priors. In *Proceedings of the twenty-second international joint conference on artificial intelligence*.

Wu, S., Éltetö, N., Dasgupta, I., & Schulz, E. (2023). Chunking as a rational solution to the speed-accuracy trade-off in a serial reaction time task. *Scientific Reports*, *13*, 7680.

Zhao, B., Lucas, C. G., & Bramley, N. R. (2023). A model of conceptual bootstrapping in human cognition. *Nature Human Behaviour*, 1–12.