

---

# Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Learning

---

Anonymous Author(s)

Affiliation

Address

email

## Abstract

1 Successful approaches to program induction require a hand-engineered domain-  
2 specific language (DSL), constraining the space of allowed programs and imparting  
3 prior knowledge of the domain. We contribute a program induction algorithm  
4 called ECC that learns a DSL while jointly training a neural network to efficiently  
5 search for programs in the learned DSL. We use our model to synthesize functions  
6 on lists, edit strings, and solve symbolic regression problems, showing how the  
7 model learns a domain-specific library of program components for expressing  
8 solutions to problems in the domain.

## 9 1 Introduction

10 Imagine that you are asked to edit some text, and told that you should change the text “Nancy  
11 FreeHafer” to “Dr. Nancy”. From this example, you likely infer that “Pushmeet Kohli” should be  
12 changed to “Dr. Pushmeet”, drawing upon your prior knowledge of text, like that words are separated  
13 by spaces and that one commonly prepends titles like “Dr.” In this work, we consider the problem  
14 of building agents that solve few-shot learning tasks like these, and also the problem of acquiring  
15 the prior knowledge necessary to quickly solve these tasks (Figure 1). We think of solutions to these  
16 tasks as being well represented by programs, and so our problem can be stated as follows: how should  
17 an agent learn to write programs? We take inspiration from two sources: (1) Good software engineers  
18 compose libraries of reusable subroutines that are shared across related programming tasks. Returning  
19 to Figure 1, a good string processing library should support appending strings and splitting on spaces  
20 – exactly the prior knowledge needed to solve the task in Figure 1. (2) Skilled human programmers  
21 can quickly recognize what kinds of programming idioms and library routines would be useful for  
solving the task at hand, even if they cannot instantly work out the details. We combine these two

TASK	<div>Nancy FreeHafer → Dr. Nancy Pushmeet Kohli → ???</div>
PROGRAM	<div><math>f(s) = (f_0 \text{ "Dr." } (f_2 \text{ s " "}))</math></div>
LIBRARY (DSL)	<div><math>f_0(a,b) = (\text{fold } a \text{ b } (\lambda(x \ y) (\text{cons } x \ y)))</math> (<math>f_0</math>: Appends lists (of characters)) <math>f_1(s,c) = (\text{fold } s \ s (\lambda(x \ a) (\text{if } (= c \ x) \text{ nil } (\text{cons } x \ a))))</math> (<math>f_1</math>: Take characters from s until c reached)</div>

Figure 1: **Task:** Few-shot learning problem. Model solves tasks by writing **programs**, and jointly learns a **library** of reusable subroutines that are shared across multiple tasks, called a **Domain Specific Language (DSL)**. Program writing is guided by a neural network trained jointly with the library.

22  
23 ideas into a new algorithm called ECC, which takes as input a collection of programming **tasks**,  
24 and then jointly solves three problems: (1) Writing programs that solve the tasks; (2) Composing a


Domain	Example Task		Part of the learned DSL
Lists	[7 0 2]	→ [7 0 2 4]	(foldr nil (lambda (a b) (cons a b)))
	[3 9]	→ [3 9 4]	(appends lists)
Strings	Temple Anna H	→ TAH	(map (lambda (x) (if (= x a) b x)))
	Lara Gregori	→ LG	(replace occurrences of a w/ b)
Regression			(+ (* real x) real) (a linear function of x)

Figure 2: Examples of structure found in DSLs learned by our algorithm. ECC builds a new DSL by discovering and reusing useful subroutines.

library of domain-specific subroutines – which allow the agent to more compactly write programs in the domain, and (3) Training a neural network to recognize which library components are useful for which kinds of tasks. Together, the library and neural net encode the domain specific knowledge needed to quickly write programs.

We call the learned library of subroutines a **Domain-Specific Language (DSL)**, a term widely used in the program synthesis community [14]. Prior work on program learning has largely assumed a fixed, hand-engineered DSL, both in classic symbolic program learning approaches (e.g., Metagol: [1], FlashFill: [2]), neural approaches (e.g., RobustFill: [3]), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: [4], DeepCoder: [5]). The contribution of this work is an algorithm for learning DSLs while jointly training a neural net to search for programs in the DSL.

Because any model may be encoded as a (deterministic or probabilistic) program, we carefully delineate the scope of program learning problems considered here. We think of ECC as learning to solve the kinds of problems that humans can solve relatively quickly – once they acquire the relevant domain expertise. These correspond to short programs – if you have an expressive DSL. Even with a good DSL, program search may be intractable, so we amortize the cost of program search by training a neural network to assist the search procedure.

Our algorithm is called **Explore/Compress/Compile (ECC)**, because it iterates between three different steps: an **Explore** step uses the DSL to explore the space of programs, searching for ones that solve the tasks; a **Compress** step modifies the structure of the DSL by discovering regularities across programs found by the previous Explore step; and a **Compile** step, which improves the program search procedure by training a neural network to write programs in the current DSL, in the spirit of “amortized” or “compiled” inference [6]. We call the neural net a **recognition model** (c.f. Hinton 1995 [7]). The learned DSL distills commonalities across programs that solve tasks, helping the agent solve related program induction problems. The neural recognition model ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands.

We apply ECC to four domains: list processing; FlashFill-style [2] string editing; symbolic regression; and turtle graphics [? ]. For each of these we initially provide a generic set of programming primitives. Our algorithm then discovers its own domain-specific vocabulary for expressing solutions in the domain (Tbl. 2).

## 2 The ECC Algorithm

Our goal is to induce a DSL while finding programs solving each of the tasks. We take inspiration primarily from the Exploration-Compression algorithm for bootstrap learning [8]. Exploration-Compression alternates between exploring the space of solutions to a set of tasks, and compressing those solutions to suggest new search primitives for the next exploration stage. We extend these ideas into an inference strategy that iterates through three steps: an **Explore** step uses the current DSL and recognition model to search for programs that solve the tasks. The **Compress** and **Compile** steps update the DSL and the recognition model, respectively. Crucially, these steps synergistically bootstrap off each other:

**Exploration: Searching for programs.** Our program search is informed by both the DSL and the recognition model. When these improve, we can solve more tasks.

**Compression: Improving the DSL.** We induce the DSL from the programs found in the exploration

66 phase, aiming to maximally compress (or, raise the prior probability of) these programs. As we solve  
 67 more tasks, we hone in on DSLs that more closely match the domain.

68 **Compilation: Learning a neural recognition model.** We update the recognition model by training  
 69 on two data sources: samples from the DSL (as in the Helmholtz Machine’s “sleep” phase), and  
 70 programs found by the search procedure during exploration. As the DSL improves and as search  
 71 finds more programs, the recognition model gets more data to train on, and better data.

## 72 2.1 Hierarchical Bayesian Framing

73 ECC takes as input a set of *tasks*, written  $X$ , each of which is a program synthesis problem. It has at  
 74 its disposal a domain-specific *likelihood model*, written  $\mathbb{P}[x|p]$ , which scores the likelihood of a task  
 75  $x \in X$  given a program  $p$ .<sup>1</sup> Its goal is to solve each of the tasks by writing a program, and also to  
 76 infer a DSL, written  $\mathcal{D}$ . We equip  $\mathcal{D}$  with a real-valued weight vector  $\theta$ , and together  $(\mathcal{D}, \theta)$  define a  
 77 generative model over programs. We frame our goal as maximum a posteriori (MAP) inference of  
 78  $(\mathcal{D}, \theta)$  given  $X$ . Writing  $J$  for the joint probability of  $(\mathcal{D}, \theta)$  and  $X$ , we want the  $\mathcal{D}^*$  and  $\theta^*$  solving:

$$J(\mathcal{D}, \theta) \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_p \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$$

$$\mathcal{D}^* = \arg \max_{\mathcal{D}} \int J(\mathcal{D}, \theta) d\theta \quad \theta^* = \arg \max_{\theta} J(\mathcal{D}^*, \theta) \quad (1)$$

79 The above equations summarize the problem from the point of view of an ideal Bayesian learner.  
 80 However, Eq. 1 is wildly intractable because evaluating  $J(\mathcal{D}, \theta)$  involves summing over the infinite  
 81 set of all programs. In practice we will only ever be able to sum over a finite set of programs. So, for  
 82 each task, we define a finite set of programs, called a *frontier*, and only marginalize over the frontiers:  
 83 **Definition.** A *frontier of task  $x$* , written  $\mathcal{F}_x$ , is a finite set of programs s.t.  $\mathbb{P}[x|p] > 0$  for all  $p \in \mathcal{F}_x$ .

84 Using the frontiers we define the following intuitive lower bound on the joint probability, called  $\mathcal{L}$ :

$$J \geq \mathcal{L} \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \quad (2)$$

85 ECC does approximate MAP inference by maximizing this lower bound on the joint probability,  
 86 alternating maximization w.r.t. the frontiers (Exploration) and the DSL (Compression):

87 **Program Search: Maxing  $\mathcal{L}$  w.r.t. the frontiers.** Here  $(\mathcal{D}, \theta)$  is fixed and we want to find new  
 88 programs to add to the frontiers so that  $\mathcal{L}$  increases the most.  $\mathcal{L}$  most increases by finding programs  
 89 where  $\mathbb{P}[x, p|\mathcal{D}, \theta]$  is large.

90 **DSL Induction: Maxing  $\int \mathcal{L} d\theta$  w.r.t. the DSL.** Here  $\{\mathcal{F}_x\}_{x \in X}$  is held fixed, and so we can  
 91 evaluate  $\mathcal{L}$ . Now the problem is that of searching the discrete space of DSLs and finding one  
 92 maximizing  $\int \mathcal{L} d\theta$ . Once we have a DSL  $\mathcal{D}$  we can update  $\theta$  to  $\arg \max_{\theta} \mathcal{L}(\mathcal{D}, \theta, \{\mathcal{F}_x\})$ .

93 Searching for programs is hard because of the large combinatorial search space. We ease this  
 94 difficulty by training a neural recognition model,  $q(\cdot|\cdot)$ , during the compilation phase:  $q$  is trained to  
 95 approximate the posterior over programs,  $q(p|x) \approx \mathbb{P}[p|x, \mathcal{D}, \theta] \propto \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$ , thus amortizing  
 96 the cost of finding programs with high posterior probability.

97 **Neural recognition model: tractably maxing  $\mathcal{L}$  w.r.t. the frontiers.** Here we train  $q(p|x)$  to  
 98 assign high probability to programs  $p$  where  $\mathbb{P}[x, p|\mathcal{D}, \theta]$  is large, because including those programs  
 99 in the frontiers will most increase  $\mathcal{L}$ .

## 100 2.2 Exploration: Searching for Programs

101 Now our goal is to search for programs solving the tasks. We use the simple approach of enumerating  
 102 programs from the DSL in decreasing order of their probability, and then checking if a program  $p$   
 103 assigns positive probability to a task ( $\mathbb{P}[x|p] > 0$ ); if so, we incorporate  $p$  into the frontier  $\mathcal{F}_x$ .

104 To make this concrete we need to define what programs actually are and what form  $\mathbb{P}[p|\mathcal{D}, \theta]$  takes.  
 105 We represent programs as  $\lambda$ -calculus expressions.  $\lambda$ -calculus is a formalism for expressing functional

<sup>1</sup>For example, for string editing, the likelihood is 1 if the program predicts the observed outputs on the observed inputs, and 0 otherwise.

106 programs that closely resembles the Lisp programming language.  $\lambda$ -calculus includes variables,  
 107 function application, and the ability to create new functions. Throughout this paper we will write  $\lambda$ -  
 108 calculus expressions in Lisp syntax. Our programs are all strongly typed. We use the Hindley-Milner  
 109 polymorphic typing system [9] which is used in functional programming languages like OCaml and  
 110 Haskell. We now define DSLs:

111 **Definition:**  $(\mathcal{D}, \theta)$ . A DSL  $\mathcal{D}$  is a set of typed  $\lambda$ -calculus expressions. A weight vector  $\theta$  for a DSL  
 112  $\mathcal{D}$  is a vector of  $|\mathcal{D}| + 1$  real numbers: one number for each DSL element  $e \in \mathcal{D}$ , written  $\theta_e$  and  
 113 controlling the probability of  $e$  occurring in a program, and a weight controlling the probability of a  
 114 variable occurring in a program,  $\theta_{\text{var}}$ .

115 Together with its weight vector, a DSL defines a distribution over programs,  $\mathbb{P}[p|\mathcal{D}, \theta]$ . In the  
 116 supplement, we define this distribution by specifying a procedure for drawing samples from  $\mathbb{P}[p|\mathcal{D}, \theta]$ .  
 117 Care must be taken to ensure that programs are well-typed and that variable scoping rules are obeyed.  
 118 With this distribution in hand, we search for programs by enumerating  $\lambda$ -calculus expressions in  
 119 decreasing order of their probability under  $(\mathcal{D}, \theta)$ .

120 Why enumerate, when the program synthesis community has invented many sophisticated algorithms  
 121 that search for programs? [10, 11, 12, 13, 14]. We have two reasons: (1) A key point of our work is  
 122 that learning the DSL, along with a neural recognition model, can make program induction tractable,  
 123 even if the search algorithm is very simple. (2) Enumeration is a general approach that can be applied  
 124 to any program induction problem. Many of these more sophisticated approaches require special  
 125 conditions on the space of programs.

126 A drawback of using an enumerative search algorithm is that we have no efficient means of solving for  
 127 arbitrary constants that might occur in the program. In Sec. 4, we will show how to find programs with  
 128 real-valued constants by automatically differentiating through the program and setting the constants  
 129 using gradient descent.

### 130 2.3 Compilation: Learning a Neural Recognition Model

131 The purpose of training the recognition model is to amortize the cost of searching for programs. It  
 132 does this by learning to predict, for each task, programs with high likelihood according to  $\mathbb{P}[x|p]$   
 133 while also being probable under the prior  $(\mathcal{D}, \theta)$ . Concretely, the recognition model  $q$  predicts, for  
 134 each task  $x \in X$ , a weight vector  $q(x) = \theta^{(x)} \in \mathbb{R}^{|\mathcal{D}|+1}$ . Together with the DSL, this defines a  
 135 distribution over programs,  $\mathbb{P}[p|\mathcal{D}, \theta = q(x)]$ . We abbreviate this distribution as  $q(p|x)$ . The crucial  
 136 aspect of this framing is that the neural network leverages the structure of the learned DSL, so it is *not*  
 137 responsible for generating programs wholesale. We share this aspect with DeepCoder [5] and [15].

138 How should we get the data to train  $q$ ? This is nonobvious because ECC is only weakly supervised  
 139 (i.e., learns only from tasks and not from (program, task) pairs). One approach is to sample programs  
 140 from the DSL, run them to get their input/outputs, and then train  $q$  to predict the program from the  
 141 input/outputs. This approach is like how a Helmholtz machine trains its recognition model during its  
 142 “sleep” phase [17]. The advantage of “Helmholtz machine” training is that we can draw unlimited  
 143 samples from the DSL, training on a large amount of data. Another approach is self-supervised  
 144 learning, training  $q$  on the (program, task) pairs discovered by the algorithm so far. The advantage of  
 145 self-supervised learning is that the training data is much higher quality, because we are training on  
 146 the actual tasks. Due to these complementary advantages, we train on both these sources of data.

147 Formally,  $q$  should approximate the true posteriors over programs: minimizing the expected KL-  
 148 divergence,  $\mathbb{E}[\text{KL}(\mathbb{P}[p|x, \mathcal{D}, \theta] || q(p|x))]$ , equivalently maximizing  $\mathbb{E}[\sum_p \mathbb{P}[p|x, \mathcal{D}, \theta] \log q(p|x)]$ ,  
 149 where the expectation is taken over tasks. Taking this expectation over the empirical distribution of  
 150 tasks gives self-supervised training; taking it over samples from the generative model gives Helmholtz-  
 151 machine style training. The objective for a recognition model ( $\mathcal{L}_{\text{RM}}$ ) combines the Helmholtz machine  
 152 ( $\mathcal{L}_{\text{HM}}$ ) and self supervised ( $\mathcal{L}_{\text{SS}}$ ) objectives,  $\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{SS}} + \mathcal{L}_{\text{HM}}$ :

$$\mathcal{L}_{\text{HM}} = \mathbb{E}_{(p,x) \sim (\mathcal{D}, \theta)} [\log q(p|x)] \quad \mathcal{L}_{\text{SS}} = \mathbb{E}_{x \sim X} \left[ \sum_{p \in \mathcal{F}_x} \frac{\mathbb{P}[x, p|\mathcal{D}, \theta]}{\sum_{p' \in \mathcal{F}_x} \mathbb{P}[x, p'|\mathcal{D}, \theta]} \log q(p|x) \right]$$

153 Evaluating  $\mathcal{L}_{\text{HM}}$  involves sampling programs from the current DSL, running them to get their outputs,  
 154 and then training  $q$  to regress from the input/outputs to the program. Since these programs map

Example programs in frontiers	Proposed subexpression
<pre>(lambda (a b) (foldr b (cons " " a)   (lambda (x z) (cons x z)))) (lambda (a b) (foldr a b (lambda (x z) (cons x z))))</pre>	<pre>(foldr a b (lambda (x z)   (cons x z)))</pre>

Figure 3: The DSL induction algorithm proposes subexpressions of programs to add to the DSL. These subexpressions are taken from programs in the frontiers (left column), and can introduce new variables (right column: a and b). Here, the proposed subexpression appends two lists.

inputs to outputs, we need to sample the inputs as well. Our solution is to sample the inputs from the empirical observed distribution of inputs in  $X$ .

## 2.4 Compression: Learning a Generative Model (a DSL)

The purpose of the DSL is to offer a set of abstractions that allow an agent to easily express solutions to the tasks at hand. Intuitively, we want the algorithm to look at the frontiers and generalize beyond them, both so the DSL can better express the current solutions, and also so that the DSL might expose new abstractions which will later be used to discover more programs. Formally, we want the DSL maximizing  $\int \mathcal{L} d\theta$  (Sec. 2.1). We replace this marginal with an AIC approximation, giving the following objective for DSL induction:

$$\log \mathbb{P}[\mathcal{D}] + \arg \max_{\theta} \log \sum_{x \in X} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] + \log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 \quad (3)$$

We induce a DSL by searching locally through the space of DSLs, proposing small changes to  $\mathcal{D}$  until Eq. 3 fails to increase. The search moves work by introducing new  $\lambda$ -expressions into the DSL. We propose these new expressions by extracting subexpressions from programs already in the frontiers. These subexpressions are fragments of the original programs, and can introduce new variables (Fig. 3), which then become new functions in the DSL. The idea of storing and reusing fragments of expressions comes from Fragment Grammars [18] and Tree-Substitution Grammars [19].

To define the prior distribution over  $(\mathcal{D}, \theta)$ , we penalize the syntactic complexity of the  $\lambda$ -calculus expressions in the DSL, defining  $\mathbb{P}[\mathcal{D}] \propto \exp\left(-\lambda \sum_{p \in \mathcal{D}} \text{size}(p)\right)$  where  $\text{size}(p)$  measures the size of the syntax tree of program  $p$ , and place a symmetric Dirichlet prior over the weight vector  $\theta$ .

Putting all these ingredients together, Alg. 1 describes how we combine program search, recognition model training, and DSL induction.

### Algorithm 1 The ECC Algorithm

**Input:** Initial DSL  $\mathcal{D}$ , set of tasks  $X$ , iterations  $I$   
**Hyperparameters:** Enumeration timeout  $T$   
Initialize  $\theta \leftarrow$  uniform  
**for**  $i = 1$  **to**  $I$  **do**  
 $\mathcal{F}_x^\theta \leftarrow \{p | p \in \text{enum}(\mathcal{D}, \theta, T) \text{ if } \mathbb{P}[x|p] > 0\}$  (**Explore**)  
 $q \leftarrow$  train recognition model, maximizing  $\mathcal{L}_{\text{RM}}$  (**Compile**)  
 $\mathcal{F}_x^q \leftarrow \{p | p \in \text{enum}(\mathcal{D}, q(x), T) \text{ if } \mathbb{P}[x|p] > 0\}$  (**Explore**)  
 $\mathcal{D}, \theta \leftarrow \text{induceDSL}(\{\mathcal{F}_x^\theta \cup \mathcal{F}_x^q\}_{x \in X})$  (**Compress**)  
**end for**  
**return**  $\mathcal{D}, \theta, q$

## 3 Sequence manipulating programs

We apply ECC to list processing (Section 3.1) and text editing (Section 3.2). For both these domains we use a bidirectional GRU [21] for the recognition model, and initially provide the system with a generic set of list processing primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

### 3.1 List Functions

Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community [12]. We consider this problem within the context of learning functions that manipulate lists. We created 244 Lisp-style list manipulation tasks, each with 15 input/output examples (Tbl. ??). Our data set is challenging along two dimensions: many of the functions are very

---

```

f0(a,b) = (foldr a b (lambda (x y) (cons x y))))
  (f0: Appends lists (of characters))
f1(s,c) = (foldr s s (lambda (x a) (cdr (if (= c x) s a))))
  (f1: Drop first characters from s until c reached)
f2(s) = (unfold s empty? car (lambda (z) (f1 z SPACE)))
  (f2: Abbreviates a sequence of words)
f3(s,c) = (foldr s s (lambda (x a) (if (= c x) nil (cons x a))))
  (f3: Take characters from s until c reached)

```

---

Table 1: Some string editing learned subroutines

Temple Annalisa Haven 185 → TAH1	Nancy FreeHafer → Dr. Nancy
Lara Gregori Bradford → LGB	Andrew Cencici → Dr. Andrew
$f(s) = (f_2 \ s)$	$f(s) = (f_0 \ "Dr. \ " \ (f_3 \ s \ " \ "))$

Figure 4: Two string edit tasks (top) and the programs ECC writes for them (bottom).  $f_0$  and  $f_2$  are subroutines written by ECC, defined in Tbl. 1.

192 complicated, and the agent must learn to solve these complicated problems from only 244 tasks. Our  
193 data set primarily consists of arithmetic operations upon sequences, and so, in addition to the Lisp  
194 primitives provided for the text editing experiments, we additionally start the system out with the  
195 following primitives: `mod`, `*`, `>`, `is-square`, `is-prime`, 2, 3, 4, 5.

196 A complete repertoire of higher-order functions is a staple of functional programming standard  
197 libraries. Although we provided ECC with some of the standard higher-order functions, like `foldr`  
198 and `unfold`, we did not include others. When trained on these list functions, our system rediscovers  
199 and then reuses the higher-order function `filter` **Lucas: not sure if it actually does this! It would**  
200 **be cool if we can write something like this though.**

### 201 3.2 String Editing

202 Synthesizing programs that manipulate strings is a classic problem in the programming languages  
203 and AI literatures [15, 22], and algorithms that learn string editing programs ship in Microsoft  
204 Excel [2]. This prior work presumes a ready-made DSL, expertly crafted to suit string editing. We  
205 show ECC can instead start out with generic Lisp primitives and recover many of the higher-level  
206 building blocks that have made these other system successful. An obstacle here, however, is that our  
207 enumerative search procedure has no means of generating string constants, and so we incorporate  
208 string-valued parameters as a primitive, defining  $\mathbb{P}[x|p]$  by marginalizing out the values of the string  
209 via dynamic programming. In Section 4, we will use a similar trick to synthesize programs containing  
210 real numbers using gradient descent.

211 We automatically generated 109 string editing tasks (Fig. 4) and model strings as lists of characters.  
212 At first, ECC cannot find any correct programs for most of the tasks. It assembles a DSL (Tbl. 1) that  
213 lets it rapidly explore the space of programs and find solutions to all of the tasks.

214 How well does the learned DSL generalized to real text-editing scenarios? We tested, but did not  
215 train, our system on problems from the SyGuS [23] program synthesis competition. Before any  
216 learning, ECC solves 32/108 of the problems with an average search time of 11 minutes. After  
217 learning, it solves 80/108, and does so much faster, solving them in an average of 29 seconds. As of  
218 the 2017 SyGuS competition, the best-performing algorithm solves 86/108 problems, and does so  
219 with a *different hand-engineered DSL for each problem*. Here we learned a single DSL that applied  
220 generically to all of the tasks, and perform comparably to the best prior work.

## 221 4 Symbolic Regression: Programs from visual input

222 We apply ECC to symbolic regression problems. Here, the agent observes points along the curve of a  
223 function, and must write a program that fits those points. We initially equip our learner with addition,  
224 multiplication, and division, and task it with solving 100 symbolic regression problems, each either a

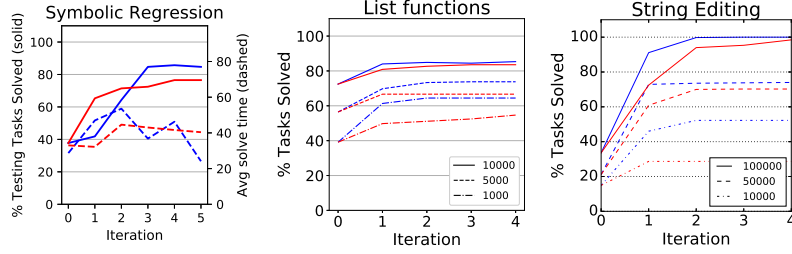


Figure 7: Learning curves for ECC both with (blue) and without (red) the recognition model as the frontier size is varied (solid/dashed/dotted lines).

polynomial of degree 1–4 or a rational function. The recognition model is a convolutional network that observes an image of the target function’s graph (Fig. 5) – visually, different kinds of polynomials and rational functions produce different kinds of graphs, and so the recognition model can learn to look at a graph and predict what kind of function best explains it. A key difficulty, however, is that these problems are best solved with programs containing real numbers. Our solution to this difficulty is to allow the system to write programs with real-valued parameters, and then fit those parameters by automatically differentiating through the programs the system writes and use gradient descent to fit the parameters. We define the likelihood model,  $\mathbb{P}[x|p]$ , by assuming a Gaussian noise model for the input/output examples, and penalize the use of real-valued parameters using the BIC [24].

ECC learns a DSL containing templates for polynomials of different orders, as well as ratios of polynomials (Fig. 6). The algorithm also discovers programs that minimize the number of continuous degrees of freedom. For example, it learns to represent linear functions with the program `(* real (+ x real))`, which has two continuous degrees of freedom, and represents quartic functions using the invented DSL primitive  $f_6$  in Tbl. 6 which has five continuous parameters. This phenomenon arises from our Bayesian framing – both the bias towards shorter programs and the likelihood model’s BIC penalty.

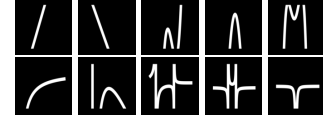


Figure 5: Recognition model input for symbolic regression. While the DSL learns subroutines for rational functions & polynomials, the recognition model jointly learns to look at a graph of the function (above) and predict which of those subroutines is appropriate for explaining the observation.

#### 4.1 Quantitative Results

We compare with four baselines on held-out tasks:

**Ours (no NN)**, which lesions the recognition model.

**RF/DC**, which holds the generative model  $(\mathcal{D}, \theta)$  fixed and learns a recognition model only from samples from the fixed generative model. This is equivalent to our algorithm with  $\lambda = \infty$  (Sec. 2.4) and  $\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{HM}}$  (Sec. 2.3). We call this baseline RF/DC because this setup is closest to how RobustFill [3] and DeepCoder [5] are trained. We can not compare directly with these systems, because they are engineered for one specific domain, and do not have publicly available code and datasets.

**PCFG**, which lesions the recognition model, learns  $\theta$ , and fixes  $\mathcal{D}$ . This is equivalent to ECC with  $q(x) = \theta$  and  $\lambda = \infty$ , and is like learning the parameters of a PCFG while not learning its structure.

**Enum**, which enumerates a frontier without any learning – equivalently, our first exploration cycle.

$f_0(x) = (+ \ x \ \text{real})$
$f_1(x) = (f_0 \ (* \ \text{real} \ x))$
$f_2(x) = (f_1 \ (* \ x \ (f_0 \ x))$
$(f_2: \text{quadratics})$
$f_3(x) = (/ \ (f_2 \ x) \ (f_0 \ x))$
$(f_3: \text{ratio of polynomials})$

Figure 6: Some learned subroutines for symbolic regression. System starts with addition, multiplication, division, and real numbers, and learns to build rational functions and polynomials up to 4th order.

For each domain, we are interested both in how many tasks the agent can solve and how quickly it can find those solutions. Tbl. 2 compares our model against these baselines. Our full model consistently improves on the baselines, sometimes dramatically (string editing and symbolic regression). The recognition model consistently increases the number of solved held-out tasks, and lesioning it also slows down the convergence of the algorithm, taking more iterations to reach a given number of tasks solved (Fig. 7). This supports a view of the recognition model as a way of amortizing the cost of searching for programs.

## 5 Related Work

Our work is far from the first for learning to learn programs, an idea that goes back to Solomonoff [25]:

*Deep learning:* Much recent work in the ML community has focused on creating neural networks that regress from input/output examples to programs [3, 26, 15, 5]. These neural networks are typically trained with strong supervision (i.e., with annotated ground-truth programs) on massive data sets (i.e., hundreds of millions [3]). Our work considers a weakly-supervised regime where ground truth programs are not provided and the agent must learn from a few hundred tasks.

*Inventing new subroutines for program induction:* Several program induction algorithms, most prominently the EC algorithm [8], take as their goal to learn new, reusable subroutines that are shared in a multitask setting. We find this work inspiring and motivating, and extend it along two dimensions: (1) we propose a new algorithm for inducing reusable subroutines, based on Fragment Grammars [18]; and (2) we show how to combine these techniques with bottom-up neural recognition models. Other instances of this related idea are [27], Schmidhuber’s OOPS model [28], and predicate invention in ILP [29].

Our work is an instance of Bayesian Program Learning (BPL; see [30, 8, 31, 27]). Previous BPL systems have largely assumed a fixed DSL (but see [27]), and our contribution here is a general way of doing BPL with less hand-engineering of the DSL.

	Ours	Ours (no NN)	RF/DC	PCFG	Enum
<i>List functions</i>					
% solved	<b>86%</b>	84%	60%	74%	70%
Solve time	0.8s	0.7s	1.0s	1.0s	1.1s
<i>String Editing</i>					
% solved	<b>75%</b>	%	33%	0%	30%
Solve time	29s	s	80s	–	–
<i>Symbolic Regression</i>					
% solved	<b>84%</b>	75%	38%	38%	37%
Solve time	24s	40s	31s	55s	29s

Table 2: % solved w/ 5 sec timeout. Solve time: averaged over solved tasks. RF/DC: trained like RobustFill/DeepCoder. PCFG: model w/o structure learning. Enum: model w/o any learning.

## 6 Contribution and Outlook

We contribute an algorithm, ECC, that learns to program by bootstrapping a DSL with new domain-specific primitives that the algorithm itself discovers, together with a neural recognition model that learns how to efficiently deploy the DSL on new tasks. We believe this integration of top-down symbolic representations and bottom-up neural networks – both of them learned – could help make program induction systems more generally useful for AI. Many directions remain open. Two immediate goals are to integrate more sophisticated neural recognition models [3] and program synthesizers [10], which may improve performance in some domains over the generic methods used here. Another direction is to explore DSL meta-learning: Can we find a *single* universal primitive set that could effectively bootstrap DSLs for new domains, including the four domains considered, but also many others?

## References

- [1] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

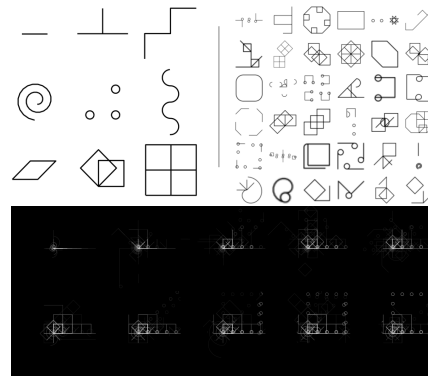


Figure 8: Future: dreams. Top left: hand crafted training targets. Top right: examples of discovered compiled new programs. Bottom: compiled program across iterations to highlight structure emergence.



- 319 [2] Sumit Gulwani. Automating string processing in spread-  
320 sheets using input-output examples. In *ACM SIGPLAN*  
321 *Notices*, volume 46, pages 317–330. ACM, 2011.
- 322 [3] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh  
323 Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv*  
324 *preprint arXiv:1703.07469*, 2017.
- 326 [4] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov,  
327 Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-  
328 guided deductive search for real-time program synthesis  
329 from examples. *ICLR*, 2018.
- 330 [5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Se-  
331 bastian Nowozin, and Daniel Tarlow. Deepcoder: Learning  
332 to write programs. *ICLR*, 2016.
- 333 [6] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. In-  
334 ference Compilation and Universal Probabilistic Program-  
335 ming. In *AISTATS*, 2017.
- 336 [7] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Rad-  
337 ford M Neal. The "wake-sleep" algorithm for unsupervised  
338 neural networks. *Science*, 268(5214):1158–1161, 1995.
- 339 [8] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B.  
340 Tenenbaum. Bootstrap learning via modular concept dis-  
341 covery. In *IJCAI*, 2013.
- 342 [9] Benjamin C. Pierce. *Types and programming languages*.  
343 MIT Press, 2002.
- 344 [10] Armando Solar Lezama. *Program Synthesis By Sketching*.  
345 PhD thesis, 2008.
- 346 [11] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic  
347 superoptimization. In *ACM SIGARCH Computer Architec-*  
348 *ture News*, volume 41, pages 305–316. ACM, 2013.
- 349 [12] John K Feser, Swarat Chaudhuri, and Isil Dillig. Syn-  
350 thesizing data structure transformations from input-output  
351 examples. In *PLDI*, 2015.
- 352 [13] Peter-Michael Osera and Steve Zdancewic. Type-and-  
353 example-directed program synthesis. In *ACM SIGPLAN*  
354 *Notices*, volume 50, pages 619–630. ACM, 2015.
- 355 [14] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A  
356 framework for inductive program synthesis. *ACM SIG-*  
357 *PLAN Notices*, 50(10):107–126, 2015.
- 358 [15] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lamp-  
359 son, and Adam Kalai. A machine learning framework for  
360 programming by example. In *ICML*, pages 187–195, 2013.
- 361 [16] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing  
362 the dimensionality of data with neural networks. *Science*,  
363 2006.
- 364 [17] Peter Dayan, Geoffrey E Hinton, Radford M Neal, and  
365 Richard S Zemel. The helmholtz machine. *Neural compu-*  
366 *tation*, 7(5):889–904, 1995.
- 367 [18] Timothy J. O'Donnell. *Productivity and Reuse in Lan-*  
368 *guage: A Theory of Linguistic Computation and Storage*.  
369 The MIT Press, 2015.

- 370 [19] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Induc-  
371 ing tree-substitution grammars. *JMLR*.
- 372 [20] J.D. Lafferty. *A Derivation of the Inside-outside Algorithm*  
373 *from the EM Algorithm*. Research report.
- 374 [21] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre,  
375 Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and  
376 Yoshua Bengio. Learning phrase representations using rnn  
377 encoder-decoder for statistical machine translation. *arXiv*  
378 *preprint arXiv:1406.1078*, 2014.
- 379 [22] Tessa Lau. *Programming by demonstration: a machine*  
380 *learning approach*. PhD thesis, 2001.
- 381 [23] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando  
382 Solar-Lezama. Sygus-comp 2016: results and analysis.  
383 *arXiv preprint arXiv:1611.07627*, 2016.
- 384 [24] Christopher M. Bishop. *Pattern Recognition and Machine*  
385 *Learning*. 2006.
- 386 [25] Ray J Solomonoff. A system for incremental learning  
387 based on algorithmic probability. Sixth Israeli Conference  
388 on Artificial Intelligence, Computer Vision and Pattern  
389 Recognition, 1989.
- 390 [26] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew  
391 Hausknecht, and Pushmeet Kohli. Neural program meta-  
392 induction. In *NIPS*, 2017.
- 393 [27] Percy Liang, Michael I. Jordan, and Dan Klein. Learning  
394 programs: A hierarchical bayesian approach. In *ICML*,  
395 2010.
- 396 [28] Jürgen Schmidhuber. Optimal ordered problem solver. *Ma-*  
397 *chine Learning*, 54(3):211–254, 2004.
- 398 [29] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenen-  
399 baum, and Stephen Muggleton. Bias reformulation for  
400 one-shot function induction. In *ECAI 2014*, 2014.
- 401 [30] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B  
402 Tenenbaum. Human-level concept learning through proba-  
403 bilistic program induction. *Science*, 350(6266):1332–1338,  
404 2015.
- 405 [31] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum.  
406 Sampling for bayesian program learning. In *Advances in*  
407 *Neural Information Processing Systems*, 2016.