LEARNING TO PROGRAM TURING MACHINES: FROM NONDETERMINISTIC TO DETERMINISTIC PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent results on reasoning with large language models show that autoregressive transformers are general-purpose machines that can be programmed via input prompts. The difficulty of verifying the correctness of such programs prompted us to investigate a different line of work. Instead of relying on an emergent machine, we design a general-purpose machine and learn programs for it that encode solutions to challenging problems through search. We hypothesize that searching in the space of nondeterministic programs to find deterministic ones can improve our system's ability to discover algorithmic solutions efficiently from small amounts of data. We introduce a learning system that starts with nondeterministic programs, represented by a learned policy over the machine's operators, and transitions to a deterministic program once learning is complete. We can then verify the resulting deterministic program for correctness. Experiments on challenging problems, including bit addition and bit multiplication, show that we can discover efficient and correct programs from only hundreds of input-output examples. Our results also show that searching in the space of nondeterministic programs is substantially more efficient than a baseline that searches directly in the space of deterministic programs, thus supporting our hypothesis. Compared to the Neural Turing Machines line of work (Graves et al., 2014), we can discover algorithms using over 200 times less data than existing approaches.

1 INTRODUCTION

Recent years have shown the impressive capabilities of Transformer-based models (Vaswani et al., 2017) on various tasks, including, but not limited to mathematical reasoning (Cobbe et al., 2021; Lewkowycz et al., 2022; Trinh et al., 2024) and code generation (Austin et al., 2021; Li et al., 2022; Rozière et al., 2024; Jimenez et al., 2024). This has, in part, been driven by a mechanism called chain of thought (CoT, Wei et al., 2022), which large language models (LLMs) use to generate intermediate steps as tokens. This procedure allows for spending inference-time computation to gain accuracy on problems that cannot be solved by a single forward pass of the network. Theoretical work has shown that CoT indeed increases expressivity (Merrill & Sabharwal, 2024) and can even simulate universal computation (Schuurmans et al., 2024).

Despite these benefits, CoT provides no guarantee of consistency and is sensitive to labels, order, and format of the prompt (Min et al., 2022; Sclar et al., 2024; Lu et al., 2022; Li et al., 2025; de Wynter, 2026). In addition, on computational and compositional tasks, LLMs often learn statistical shortcuts instead of the underlying algorithm (Liu et al., 2023; Dziri et al., 2023; Mirzadeh et al., 2025). These failure cases coupled with the results that CoT can represent universal computation demonstrate that *expressivity* is not an issue – these models do not lack capacity to solve these tasks. Therefore, the failure is in *discovering* the right computation. And even when the computation is found, it cannot be inspected; correctness can only be estimated empirically on held-out examples, but not verified.

We focus on the programming-by-example approach (PBE, Solar-Lezama et al., 2006; Gulwani, 2011; Ellis et al., 2021), where we are given input-output examples, and must discover a generalized program from them. Within PBE, the inspiration for our method comes from the Neural Turing Machines (NTM) line of work (Graves et al., 2014; Zaremba & Sutskever, 2016; Graves et al., 2016), which augment neural networks with external memory. There, much like in modern LLMs, a single neural network must learn two things: *which* operation to apply at each step, and *what* that operation

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

produces (Tanneberg et al., 2020). Both of these are conflated within the network's weights, so no program can be returned. Over time, this line of work has progressively made more and more of the machine discrete – the NTM and DNC are fully differentiable, using soft attention over memory (Graves et al., 2014; 2016), whereas the RL-NTM replaced soft memory access with discrete actions learned via reinforcement learning, leaving the controller the only neural component (Zaremba & Sutskever, 2016; see also Gulcehre et al., 2018). Our work is a continuation of this trend, making the entire machine non-neural, with a policy that guides a search for a deterministic, verifiable program.

Our method decouples the two mechanisms – we design a Turing-like **machine** with explicit operations for controlling input, output, and "reasoning" operations (Section 3); and a **controller**, which programs the machine (Section 4.1). The method relies on **search**, which produces execution traces for training the controller to operate the machine (Section 4.2). While the machine itself is deterministic, the controller defines a distribution over operations – a nondeterministic program; conditioning each choice on the machine state in the spirit of execution-guided synthesis (Chen et al., 2019). We evaluate our method on a set of algorithmic tasks, similar to those that the NTM (Graves et al., 2014) and Neural GPUs (Kaiser & Sutskever, 2016) were evaluated on. We find that our method discovers exact programs from a limited set of examples, which generalize in length much further beyond the training distribution (Section 5). Our contributions are:

- **Framework.** We introduce a framework for searching over traces of nondeterministic programs to train a policy allowing us to discover deterministic programs for computational tasks with small amounts of data. These programs are verifiably correct and generalize to any length.
- **Results.** We empirically show that searching over the space of nondeterministic program traces can more efficiently find programs than searching over a deterministic program space. Our method discovers solutions for *copy*, *reverse*, *bigram flip*, *addition*, and *multiplication* tasks while using 200 times less data than the previous state of the art.

2 Related work

LLMs as General-Purpose Machines. This line of work relies on the emergent machine implicit within a transformer, where the prompt represents the program (Pérez et al., 2021; Schuurmans, 2023; Qiu et al., 2025). Giannou et al. (2023) hand-design the weights of a Transformer and put it in a loop, showing it can emulate computing blocks and work as a general-purpose computer. Schuurmans (2023) instead keeps the weights frozen, showing that augmenting Transformers with a read-write mechanism allows simulating any algorithm under a fixed set of prompts. In both cases the machine is general-purpose, but the program is designed by hand. Gao et al. (2023) separate the roles – the Transformer generates a program, and an external interpreter executes it. However, the program is produced anew for each instance, and is not checked for correctness. The work by Nye et al. (2021) trains Transformers to emit intermediate computation steps, hypothesizing that they help generalize, although the intermediate steps are hand-designed. Within these works the program is either hand-designed or encoded in the weights; we instead discover a program, and return it.

Differentiable Neural Program Induction. Our work is closely related to neural program induction, which works by inferring an algorithm from the input-output examples and simulating it (Joulin & Mikolov, 2015; Kurach et al., 2016; Neelakantan et al., 2016; Li et al., 2017). Several architectures have been developed in this area, such as the Neural Turing Machine (NTM, Graves et al., 2014) and its successors (Graves et al., 2016; Zaremba & Sutskever, 2016). It introduced a differentiable neural controller with an external addressable memory, solving various algorithmic tasks such as associative recall, sorting, and routing. The most successful model on the tasks we consider is achieved by Neural GPUs (Kaiser & Sutskever, 2016) which train on short-length instances, extrapolating to long ones. This extrapolation does not guarantee correctness – Price et al. (2016) show that Neural GPUs still fail on highly symmetrical and atypical inputs, while still requiring large models trained on large amounts of data. A different approach by Reed & de Freitas (2016) relies on a dataset of execution traces instead of input-output examples, and Cai et al. (2017) add recursion, obtaining programs that provably generalize on a verification set. However, both works rely on imitating provided execution traces, whereas we discover them via search.

Program Synthesis. Our work is also related to the field of program synthesis (Jha et al., 2010; Alur et al., 2013; Osera & Zdancewic, 2015; Parisotto et al., 2017), which works by searching a

space defined by a fixed set of operators. Classical approaches restrict the search space to make it tractable, e.g. Sketch (Solar-Lezama et al., 2006) requires a partial implementation and a specification; component-based synthesis (Gulwani et al., 2011) restricts the search space to loop-free programs from a given library of components. Neither of these can have variable length loops, whereas we have a general-purpose machine, designed to support any kind of computation. A close work by Gaunt et al. (2016) studies different machines, such as a Turing machine or the assembly language, and compares gradient descent versus search and solver approaches; they find that search and solvers are more efficient than gradient descent, although every task is solved from scratch.

Guided Search. In synthesis Balog et al. (2017) make the search more efficient by learning which functions should appear in the program, and CrossBeam (Shi et al., 2022) trains a bottom-up search policy from its past searches. These systems are trained on a large dataset of synthetic program examples, and in all of them the learned model is only a guide. The same holds outside of synthesis. Levin Tree Search (LTS, Orseau et al., 2018) searches with a policy over actions and bounds the number of node expansions based on the policy; this bound works as a loss for training the policy itself (Orseau et al., 2023). LTS builds a search tree using the learnt policy for every instance – solving a new instance requires searching again. We instead take the resulting program, which is run on unseen inputs without search, and can be verified against the specification.

3 SETUP

A *task* is a computational problem with some unknown computation $p^\star$ that solves the problem. We are given $\mathcal{D}$ – a set of input-output examples $\{(x_i, y_i)\}_{i=1}^{|\mathcal{D}|}$ for a certain task (referred to as problem instances) where for all i the output is $y_i = p^\star(x_i)$. The goal is to find a program p that solves the task on new examples x of arbitrary size, such that $p(x) = p^\star(x)$.

We design a machine inspired by a traditional Turing machine, which is a tuple $(Q, \Sigma, \delta, \mathtt{b}, q_0)$, where Q is a set of states and $q_0 \in Q$ is the initial state; Σ is the set of symbols and $\mathtt{b} \in \Sigma$ is the blank symbol, and $\delta : Q \times \Sigma \times \Sigma \times \Sigma \to Q \times Op$ is the transition function, where Op is the set of operations available to the machine. A configuration of the machine is represented using the fields in Table 1. The operations control the register and handle head movement for both input and work tapes. See Appendix Table A.1 for a description of the operations. The set of symbols will usually be $\Sigma = \{0, 1, \mathtt{b}\}$ (for the addition task we include +, for multiplication we include $\star$ as the operand-separating symbol). We refer to the configurations observed by the machine as $\mathcal{O}_\mathrm{m} = Q \times \Sigma \times \Sigma \times \Sigma$, and the set of actions as $\mathcal{A} = Q \times Op$ (making $\delta : \mathcal{O}_\mathrm{m} \to \mathcal{A}$). Solving a problem instance requires finding a sequence of operations such that the correct answer is written to the work tape; a program *trace* is a sequence of $(o^\mathrm{m}, a) \in \mathcal{O}_\mathrm{m} \times \mathcal{A}$ pairs representing that solution. We search for a program p in the machine's transition function δ format such that it solves all problem instances. A verifier handles the recognition of a solved instance; the program is not responsible for halting.

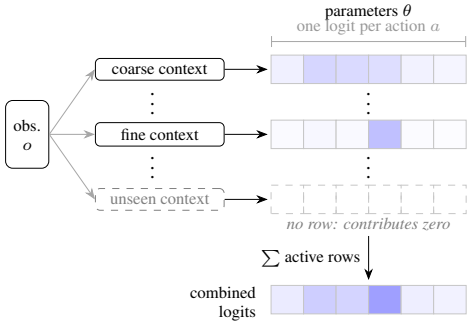

Figure 1: A context model. Each active context in observation o^π indexes one row of parameter table θ. Coarse contexts match many observations; fine contexts concentrate on few. Unseen contexts contribute zero.

Table 1: Overview of the machine configuration.

Field	Description
Input Tape	Holds the problem instance (read-only).
Work Tape	Holds the output; initially blank (read-write).
Input Head	Current read position on the input tape.
Work Head	Current read-write position on the work tape.
Register	Holds one symbol; initially blank (read-write).
State	Internal state $q \in Q$.

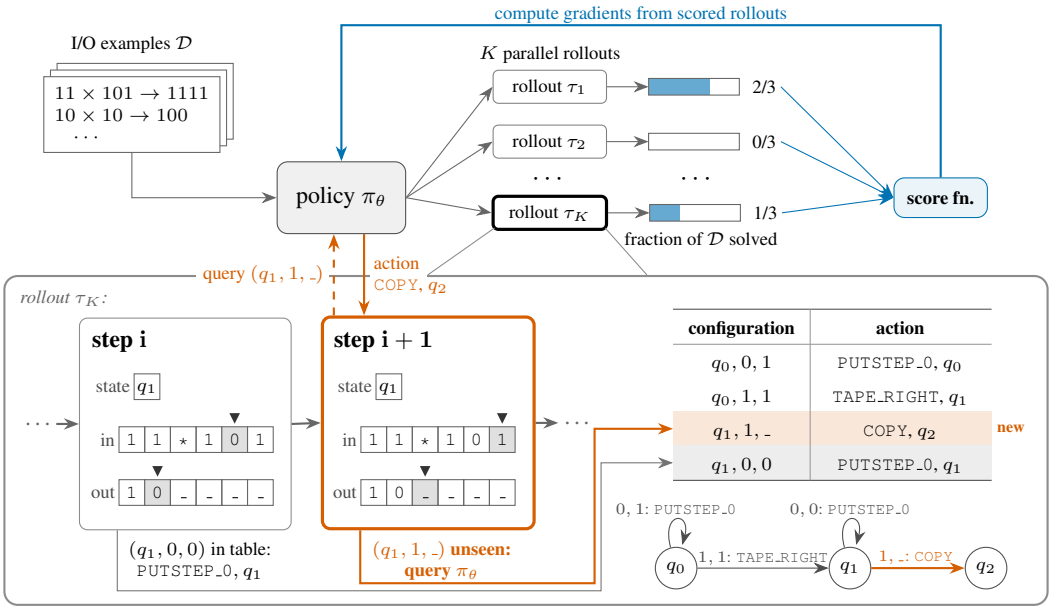

Figure 2: **Top:** Main training loop. We are supplied with I/O examples $\mathcal{D}$, and run K parallel rollouts (computational traces in the underlying Turing machine) on these examples, sampling actions based on the policy π_θ. We score each rollout based on the number of examples it solved, partial solutions, and the number of steps (Equation 1). We update the policy based on this score function, so future rollouts are more successful. **Bottom:** Behavior within a rollout. Whenever the machine reaches a configuration already present in the table (e.g., step i), it executes it. When it is not present (e.g., step $i + 1$), we query the policy for a new action, write a new row in the table, and execute it. A transition graph represents the same table, and the bold orange arrow indicates the action that was added by the policy from the new configuration.

4 Method

4.1 The Controller

The controller is responsible for choosing which actions to take in order to solve a problem instance. We model the controller as a policy $\pi_\theta(a_t \mid o_t^\pi)$, which maps the current observation $o_t^\pi \in \mathcal{O}_\pi$ at step t to a distribution over the action space $\mathcal{A}$. The policy observation space $\mathcal{O}_\pi$ is a richer observation, defined below in Section 4.2.1.

We use a context model (Orseau et al., 2023) as the policy. A context model parameterises the policy as a product of experts. Each context is a specific feature of the observation and holds a learned vector of action logits. To select an action, the policy simply sums the active feature vectors and applies a softmax, incurring a decision cost of just those vector additions rather than a full neural network forward pass. Parameter rows exist only for contexts observed on valid solution paths, therefore unseen contexts contribute zero logits, allowing the model to smoothly fall back to its remaining active experts. Contexts can be coarse observations, contributing a broad prior for a decision, while others can be fine, contributing a situation-specific opinion (see example in Figure 1). The same decision can be supported at several granularities at once, so the coarse experts can continue to guide the search even if observations not yet learned by the fine experts are encountered. This enhances the model's ability to generalize early, before it has been trained completely, which is beneficial for our bootstrap-style search method.

A configuration of the machine maps to a set of contexts; the specific contexts being used can be found in Appendix B. We use contexts that limit observations of the input and work tape to fixed windows, as well as encoding the offsets of the read/write heads instead of absolute positions. These offsets are bucketed as they grow further apart, giving a finite set of offsets that can be learned during the training phase.

4.2 SEARCH

An overview of the method is presented in Figure 2. At a high level, we use the policy to generate a program trace for solving a given problem. It can be difficult to find a correct trace blindly, so we search the space of nondeterministic program traces given by the policy. Finding a single program trace can be simple given an easy enough problem instance. However, it is unlikely that the program representing the first found trace would be able to generalize to other problem instances. Our approach constructs programs by generating program traces, keeps the best ones, and then trains a policy to construct similarly successful programs. We call the process of generating program traces *rollouts*.

4.2.1 CONSTRUCTING PROGRAMS WITH ROLLOUTS

Given M problem instances $x_1 \dots x_M$ and an empty program table representing the transition function δ, we attempt to solve all of them round-robin in a single rollout (Algorithm 1). A round-robin approach means taking a single action for each problem instance ($a_1^{x_1}$, then $a_1^{x_2}$, and so on) before taking the next set of actions ($a_2^{x_1}$, then $a_2^{x_2}$, and so on). This continues until all instances are solved or we reach a limit. Note that with this method generalization is structural; a single program must solve all M problem instances. Each rollout produces one program trace, and each trace can be condensed into one program table. To explore the search space more efficiently, we run K rollouts in parallel, each with its own random seed.

There are two kinds of observation spaces: $\mathcal{O}_\pi$ and $\mathcal{O}_\text{m}$. Each one is a representation of the configuration of the machine, but $\mathcal{O}_\text{m}$ presents a more limited view than $\mathcal{O}_\pi$. Programs are defined with observations $o^\text{m} \in \mathcal{O}_\text{m}$. The limited observation space makes it easy to define a compact, deterministic program for solving the given problem, and allows it to more easily generalize to other instances. The observation o^m is represented as a 4-tuple of (state, symbol under input head, symbol under work head, symbol in register). When the policy is required to make a decision, it receives the entire configuration of the machine as the observation $o^\pi \in \mathcal{O}_\pi$, allowing it to utilize all of the information available to it.

We assume that we have access to a verifier for each problem domain we apply this method to. That is, given a machine configuration, if the answer to the problem instance in question has been written to the work tape, we mark the instance as solved. This means that the program is not responsible for managing its own termination.

Algorithm 1 Round-Robin Rollout

Require: Set of problem instances $\{1, \dots, M\}$, step caps $\{\kappa_1, \dots, \kappa_M\}$
Require: Step counters $\{\text{steps}_1, \dots, \text{steps}_M\}$, solved states $\{\text{is_solved}_1, \dots, \text{is_solved}_M\}$
Require: Program table T (initially empty), max table size T_max, policy π

while $|T| < T_\text{max}$ **do**
  for each problem instance $i \in \{1, \dots, M\}$ **do**
    if is_solved_i **or** $\text{steps}_i \geq \kappa_i$ **then**
      continue
    $c \leftarrow \text{GetMachineConfiguration}(i)$
    $o^\pi, o^\text{m} \leftarrow \text{GetObservations}(c)$ ▷ Rich and limited views of c
    if $o^\text{m} \in T$ **then**
      $a \leftarrow T[o^\text{m}]$ ▷ Observation o^m already in table – retrieve associated action
    else
      $P(\cdot \mid o^\pi) \leftarrow \pi(o^\pi)$ ▷ Query the policy on the rich view
      $a \sim P(\cdot \mid o^\pi)$ ▷ Sample action from distribution
      $T[o^\text{m}] \leftarrow a$ ▷ Store mapping (o^m, a) in table
    $\text{ApplyAction}(i, a)$ ▷ Apply action a to machine instance i
    $\text{steps}_i \leftarrow \text{steps}_i + 1$
    $c' \leftarrow \text{GetMachineConfiguration}(i)$
    $\text{is_solved}_i \leftarrow \text{VerifySolved}(c')$ ▷ Check work tape for correct answer
  if all instances are solved or capped **then**
    break

To guard against an infinite rollout, we bound the rollout both with respect to the number of actions taken and the size of the table constructed. A program trace for a single problem instance is limited by a step cap, where each action taken for an instance is considered one step. The step cap for a specific instance of input length n is $an^2 + bn + c$. We tune the coefficients depending on the problem to match our expectation for the complexity of the problem domain. For example, addition problems are given a linear step cap, whereas multiplication problems are given a quadratic step cap. The entire rollout is limited by a decision cap, which is a bound on the number of rows in the program table. In addition to bounding the rollout, these limits also allow us to inject a slight bias for preventing inefficient solutions to the problem from being found.

4.2.2 Training the Policy

Given K programs, we rank all of them and train on the best one. A program's score is

$$J = M_{\text{solved}} - \lambda \log \left(\frac{\sum_{i=1}^{M} N_a^{(i)}}{\sum_{i=1}^{M} P_i} \right) \tag{1}$$

where $N_a^{(i)}$ is the number of actions the program trace takes and P_i is the progress of instance i: the length of the longest correct prefix of the answer written to the work tape. First, we reward programs that solve many problem instances, as that is our ultimate goal (M_{solved}). However, it is common for there to be many ties among programs for M_{solved}, so we need a differentiator. This is the role that the penalty on the number of actions plays; we prioritize programs that are efficient. The λ term serves to control how much weight we give to the penalty as opposed to M_{solved}. It might seem that the step cap in Algorithm 1 and the penalty here are playing similar roles, and they are. The penalty is important for shorter problem instances where a large enough difference between linear, quadratic, and exponential programs has not yet emerged.

After choosing the program that maximizes J from the K rollouts, we extract the sequence of (o^π, a) pairs from the program's trace. Then, we train the policy on them using supervised learning, maximizing the log likelihood of each action a taken at a given observation o^π.

4.2.3 Terminating the Search

Once a program solves all M instances, we run a verification pass on it. A program candidate must solve B sets of M problem instances to be accepted as the final program. If it is unable to, we continue the search until we hit a time limit, otherwise, we return the program and perform test-time verification on in-distribution and out-of-distribution problem instances. See Algorithm 2 for an overview of the full search and training loop.

Curriculum. For more complex problem domains like addition and multiplication, it becomes very important to use a strict curriculum on difficulty. The ideal curriculum slowly increases the number of rows a program table needs in order to encode a correct solution. With fewer rows, the search becomes easier as there are fewer decision points where the policy can make a mistake. It is also important that curriculum levels are composable: a program table that solves the second level should include all of the rows of a program table that solves the first level. To advance from one curriculum level to the next, the same verification process previously described is required.

Restarts. If we find that our best program fails to improve the top score J for a certain number of rounds, we either demote our curriculum level or reset the policy entirely. Demoting means returning to the previous curriculum level, or, if a given number of demotions have been accumulated or we are at the lowest curriculum level already, re-initializing the policy completely and returning to the start of the curriculum.

5 Experiments

5.1 Problem Domains

We run experiments on five binary domains: *copy*, *reverse*, *bigram flip*, *addition*, and *multiplication*. The choice of domains is intended to showcase a variety of methods of manipulating the machine, at varying degrees of difficulty. All are mentioned by Kaiser & Sutskever (2016), although they do not

Algorithm 2 Search and Training Loop

Require: Policy π_θ, dataset $\mathcal{D}$, number of rollouts K, verification target B
Require: Batch size M, learning rate η
Ensure: Final generalized program T^*

while time remaining **do**
  $\mathcal{P} \leftarrow \text{SampleBatch}(\mathcal{D}, M)$ ▷ Sample M problem instances
  for each rollout $\tau \in \{1, \ldots, K\}$ **in parallel do**
    $T_\tau \leftarrow \text{RoundRobinRollout}(\mathcal{P}, \pi_\theta)$ ▷ Algorithm 1
    $J_\tau \leftarrow \text{ComputeScore}(T_\tau, \mathcal{P})$ ▷ Equation (1)
  $T^* \leftarrow \text{argmax}_{T_\tau} \{J_\tau\}_{\tau=1}^{K}$
  $\mathcal{T} \leftarrow \text{ExtractTrajectory}(T^*)$ ▷ Retrieve state-action pairs (o^π, a)
  $\theta \leftarrow \theta + \eta \nabla_\theta \sum_{(o^\pi, a) \in \mathcal{T}} \log \pi_\theta(a \mid o^\pi)$ ▷ Supervised policy update
  if $\text{CountSolved}(T^*, \mathcal{P}) = M$ **then** ▷ Check if all instances in batch are solved
    verified $\leftarrow$ **true**
    for $b \leftarrow 1$ **to** $B - 1$ **do**
      $\mathcal{P}_{\text{eval}} \leftarrow \text{SampleBatch}(\mathcal{D}, M)$
      if $\text{CountSolved}(T^*, \mathcal{P}_{\text{eval}}) < M$ **then**
        verified $\leftarrow$ **false**
        break
    if verified **then**
      return T^*
return Failure

evaluate on bigram flip, which comes from Grefenstette et al. (2015). More detailed explanations of the problem domains can be found in Appendix C.

Copy. A binary string must be reproduced exactly.

Reverse. A binary string must be reversed.

Bigram Flip. Each adjacent pair's symbols in a binary string must be swapped.

Addition. Two binary strings are added to each other, least-significant bit first.

Multiplication. Two binary strings are multiplied, least-significant bit first. Multiplication is challenging, and we require a more advanced machine in order to see any success. The details of the machine can be found in Appendix Table A.2 and Appendix Table A.3. Notably, this machine adds composed operations, which are combinations of lower level operations. Composing actions reduces the number of decisions needed by the policy, making the chance of finding a correct program higher.

5.2 Baseline

A deterministic program is a table mapping observations to actions, and there is a finite set of deterministic programs $\mathcal{A}^{\mathcal{O}_m}$. This is a very large space, for addition on our machine it holds $117^{832} \approx 10^{1721}$ programs, but it is the natural space to search given that we are ultimately looking for a deterministic program.

Our baseline searches the space of deterministic programs directly, with no learning. We start with a randomly initialized program table and then run a beam search on permutations of the table until a generalizing program table is found or we run out of time. The baseline moves between deterministic, complete programs while our method never instantiates a complete table. Instead, we search the traces of a nondeterministic program, and the deterministic program is constructed using the trace afterwards. See Appendix D for more details.

5.3 Results

A comparison of our method against the baseline is shown in Table 2. We report the number of successful seeds across 10 total seeds where the returned program is able to solve all test instances. We also report the mean wall time it takes to find a generalizing program as both a measure of

Table 2: Success counts and mean wall time across 10 random seeds. †Mean wall times are computed only over successful runs and are not directly comparable across methods when success counts differ.

		Our Method		Baseline	
Domain	**Time Limit**	Successes	Mean Time (s)	Successes	Mean Time (s)
Base Machine					
Copy	24h	10	19	10	218
Reverse	24h	7	26588†	5	12707†
Bigram Flip	24h	10	3288†	1	3564†
Addition	24h	4	49244	0	—
Advanced Machine					
Addition	24h	8	2788	0	—
Multiplication	40h	1	129240	0	—

efficiency and a comparison for when both methods have the same success rate. Appendix E contains examples of generalizing programs discovered by our method.

Both approaches solve copy very quickly across all seeds. However, comparing wall time shows that our method is vastly more efficient at finding solutions. For reverse and bigram flip, we are able to find generalizable programs at a higher success rate than the baseline. Addition proves to be a step up in difficulty, but we are still able to see success with both the base machine and advanced machine, while the baseline no longer has any signal. Before being able to write a correct symbol to the work tape, a typical addition program requires multiple complex operations to be performed: copying, rewinding, carrying, and adding. Our method avoids this by using a curriculum, which teaches some of the simpler operations in advance. The baseline does not learn, and thus does not benefit from such a curriculum. Furthermore, by exploiting the context model policy's various context granularities, we are able to learn information that can be applied to unseen observations, giving us a better signal for traversing deep into the search space. We find generalizable multiplication programs very rarely, requiring a lot more time to find even a single one. Furthermore, this requires the use of the advanced machine; there was no signal of success with the base machine. However, the program we discover with just a small amount of data is verifiably correct.

Program Correctness. We prove that the programs we discover for all domains are correct using machine-checked Lean 4 (de Moura & Ullrich, 2021) proofs in Appendix F. Although our programs do not halt explicitly, we discovered that a halting indicator can be derived from whenever we reach a configuration that is not covered by the program table, which is what is used for the proofs. Due to a quirk in the generation of the addition training data distribution, addition ($a + b$) is only proved correct for $|a| \leq |b|$. The rest are proved correct for all inputs.

Bounded Dataset Size. To showcase our method's ability to find solutions using very small data, we pre-generate the dataset $\mathcal{D}$ before running the search, then sample problem instances from that dataset during search. This guarantees that we use no more than $|\mathcal{D}|$ problem instances during training. For copy, reverse, and bigram flip, $|\mathcal{D}| = 100$. For addition and multiplication, we use a limit of 100 for each curriculum level. That is, we generate a separate dataset for each curriculum level respecting the restrictions of the level, with max size 100. In some cases, given the restrictions of a level, there are fewer than 100 possible unique problem instances, but the total dataset size $|\mathcal{D}|$ will always be bounded by $100 \times L$, where L is the number of curriculum levels. In this case, addition uses $|\mathcal{D}| = 386$ and multiplication uses $|\mathcal{D}| = 870$. We have not investigated smaller dataset sizes thoroughly, but it is plausible that we can see success with even less data.

Comparison against prior work. We compare against the Neural GPU (Kaiser & Sutskever, 2016) as they represent the state of the art. Their metrics are slightly different from ours in Table 2; they report the fraction of test instances whose every output bit is correct for their single best model. To compare against this, we take one of the programs we learn for each domain and evaluate it at the longest test length measured for the Neural GPU (Table 3). Note that test length is the length of the entire input string, so for an addition problem $a + b$, a and b are both length 2000; accounting for the '+' gives a total length of 4001. It is unclear how many test instances are used when evaluating the Neural GPU. We use 20 test instances to evaluate our model at total input lengths of 4001,

Table 3: Comparison with the Neural GPU.

		Neural GPU		Our Method	
Domain	**Test Length**	Dataset	Success	Dataset	Success
String tasks					
Copy	4001	Unknown	100%	100	100%
Reverse	4001	Unknown	100%	100	100%
Bigram flip	4001	N/A	N/A	100	100%
Arithmetic					
Addition	4001	200k	100%	386	100%
Multiplication	4001	200k	100%	870	100%

demonstrating the same degree of success, although this is guaranteed as we have already proven the correctness of our programs. The difference is that we use over 200 times less data.

6 Discussion & Conclusion

In the previous line of work stemming from Neural Turing Machines (Graves et al., 2014), the idea of a machine and a program are heavily entangled. We show the benefits of separating them out, with the hopes of creating a more reliable and understandable system for algorithmic discovery.

Our method can learn a generalizing, deterministic program from small amounts of data, with the ability to verify correctness of the program returned by the search. Furthermore, we show empirically that by searching over a space of nondeterministic program traces, we can more efficiently find programs that solve algorithmic tasks than if we searched over a deterministic space. Starting in the nondeterministic space gives us more room to explore, allowing for mistakes to be made. As we bring the search toward a more deterministic space, the compression of the program allows us to generalize more easily with respect to length, relying on the signal we gained from exploring and learning earlier in the search.

We focused our effort on ensuring consistency of the method across seeds, which is something that much of the prior work did not heavily emphasize. While we were successful for simpler domains, we still find that for more complex problems like multiplication, generalizable solutions are difficult to find consistently. However, by using the verification pass described in Section 4.2.3, the confidence that a returned program is generalizable is quite high. In the multiplication experiments, only one seed returned a program within a 40 hour search time limit, but that program is proven to generalize over any input length.

Limitations and Future Work. As we increase the complexity of problems, our method struggles to use the base machine effectively to find generalizing programs. Instead, we rely on increasing the capabilities of the machine by injecting our own domain-specific bias so that the search space remains feasibly sized. It would be interesting to see if the composed operations in the advanced machine could be discovered during the search, taking inspiration from prior work on library learning (Ellis et al., 2021).

Furthermore, the machine handles the termination of programs by using a solution verifier, so a program does not need to know when to stop. This makes the problem easier and limits the application of this method to domains where a verifier can be run efficiently. Future work could involve learning to use a halt operation to indicate when a program's output should be evaluated.

Another interesting angle is to look at transfer across problem domains. We would expect that a program for copy would be useful in solving a reverse problem, and similarly, a program for addition would be useful for multiplication. In other words, we could form a curriculum over problem domains in order to transfer knowledge gained in simpler domains for more challenging ones.

Lastly, the choice of a context model as the policy is motivated by the simplicity of the architecture and its ability to use observation representations that are verifiably length generalizable. It would be interesting to see whether a transformer acting as the policy, or perhaps even a full-scale LLM, would achieve similar results, bringing this work in line with more modern approaches.

AI USE STATEMENT

In this work, we used generative AI tools for desigining and providing feedback on research methodology and experiments, implementing methods, orchestrating experiments, consolidating and reporting experiment results, and formulating, proving, and writing mathematical claims and proofs of program correctness. We have not used generative AI tools for proposing and refining hypotheses, interpreting results, and qualitatively and thematically analyzing data. AI for developing theoretical models and conceptual frameworks, generating synthetic datasets, cleaning and reformatting a dataset, and assisting with translation are not applicable to this work. Additionally, we used generative AI tools for creating and modifying scientific figures and tables, suggesting experimental parameters, writing code, brainstorming, searching for information and references, and improving research paper formatting. We have reviewed all AI-assisted work. LLM-generated code was tested for correctness with unit tests. LLM-generated code was inspected manually to understand the implementation. LLM-generated figures and tables were inspected manually to ensure the correct relationships and values were retained. LLM-summarized results were compared regularly with the raw experiment logs to ensure correctness in reporting. LLM-summarized references were checked for accuracy of methods and claims. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide the pseudocode for the main proposed framework in Algorithm 1 and Algorithm 2, along with other techniques used in Section 4.2. We provide details about the machine we designed in Section 3 and in Appendix A. We provide details about the context model policy we used in Section 4.1 and Appendix B. Exact details about problem domains, experiments, and hyperparameters are provided in Appendix C and Appendix G. Combined, this should provide enough detail to implement and use our method for solving similar kinds of problems.

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, 2013. doi: 10.1109/FMCAD.2013.6679385.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=BkbY4psgg.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gfOiAqYm.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pp. 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Adrian de Wynter. Is in-context learning learning? In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 101317–101355, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/a598163f84a7d6d64476ec6762ca25ce-Paper-Conference.pdf.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang (Lorraine) Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 70293–70332. Curran Associates, Inc., 2023. doi: 10.52202/075280-3081. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/deb3c28192f979302c157cb653c15e90-Paper-Conference.pdf.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction, 2016. URL https://arxiv.org/abs/1608.04428.

Angeliki Giannou, Shashank Rajput, Jy-Yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 11398–11442. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/giannou23a.html.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014. URL https://arxiv.org/abs/1410.5401.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476, Oct 2016. ISSN 1476-4687. doi: 10.1038/nature20101. URL https://doi.org/10.1038/nature20101.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory, 2015. URL https://arxiv.org/abs/1506.02516.

Caglar Gulcehre, Sarath Chandar, Kyunghyun Cho, and Yoshua Bengio. Dynamic neural turing machine with continuous and discrete addressing schemes. *Neural Computation*, 30(4):857–884, 04 2018. ISSN 0899-7667. doi: 10.1162/neco_a_01060. URL https://doi.org/10.1162/neco_a_01060.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926423. URL https://doi.org/10.1145/1925844.1926423.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 62–73, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993506. URL https://doi.org/10.1145/1993498.1993506.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 215–224, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587196. doi: 10.1145/1806799.1806833. URL https://doi.org/10.1145/1806799.1806833.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/hash/26657d5ff9020d2abefe558796b99584-Abstract.html.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms, 2016. URL https://arxiv.org/abs/1511.08228.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.06392.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 3843–3857. Curran Associates, Inc., 2022. doi: 10.52202/068431-0278. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/18abbeef8cfe9203fdf9053c9c4fe191-Paper-Conference.pdf.

Chengtao Li, Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, and Nate Kushman. Neural program lattices. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=HJjiFK5gx.

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. Long-context LLMs struggle with long in-context learning. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=Cw2xlg0e46.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=De4FYqjFueZ.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In Smaranda

Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8086–8098, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.556. URL https://aclanthology.org/2022.acl-long.556/.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 7690–7706, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/1f59721c106ea80f613299039112f651-Paper-Conference.pdf.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11048–11064, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.759. URL https://aclanthology.org/2022.emnlp-main.759/.

Iman Mirzadeh, Keivan Alizadeh-Vahid, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 94743–94765, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/ec2e7a896f8250986b3907f57621ce94-Paper-Conference.pdf.

Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.04834.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. URL https://arxiv.org/abs/2112.00114.

Laurent Orseau, Levi Lelis, Tor Lattimore, and Theophane Weber. Single-agent policy tree search with guarantees. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/hash/52c5189391854c93e8a0e1326e56c14f-Abstract.html.

Laurent Orseau, Marcus Hutter, and Levi H. S. Lelis. Levin tree search with context models. In Edith Elkind (ed.), *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pp. 5622–5630. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/624. URL https://doi.org/10.24963/ijcai.2023/624. Main Track.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *SIGPLAN Not.*, 50(6):619–630, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2738007. URL https://doi.org/10.1145/2813885.2738007.

Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rJ0JwFcex.

Eric Price, Wojciech Zaremba, and Ilya Sutskever. Extensions and limitations of the neural gpu, 2016. URL https://arxiv.org/abs/1611.00736.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021. URL http://jmlr.org/papers/v22/20-302.html.

Ruizhong Qiu, Zhe Xu, Wenxuan Bao, and Hanghang Tong. Ask, and it shall be given: On the turing completeness of prompting. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=AS8SPTyBgw.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.06279.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Dale Schuurmans. Memory augmented large language models are computationally universal, 2023. URL https://arxiv.org/abs/2301.04589.

Dale Schuurmans, Hanjun Dai, and Francesco Zanini. Autoregressive large language models are computationally universal. *arXiv preprint arXiv:2410.03170*, 2024. URL https://arxiv.org/abs/2410.03170.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models'sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 25055–25083, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/6c0e99d736da621403018ca7b32b1a4d-Paper-Conference.pdf.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=qhC8mr2LEKq.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News*, 34(5):404–415, October 2006. ISSN 0163-5964. doi: 10.1145/1168919.1168907. URL https://doi.org/10.1145/1168919.1168907.

Daniel Tanneberg, Elmar Rueckert, and Jan Peters. Evolutionary training and abstraction yields algorithmic generalization of neural computers. *Nature Machine Intelligence*, 2(12):753–763, Dec 2020. ISSN 2522-5839. doi: 10.1038/s42256-020-00255-1. URL https://doi.org/10.1038/s42256-020-00255-1.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. URL https://proceedings.neurips.cc/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines - revised, 2016. URL https://arxiv.org/abs/1505.00521.

A MACHINE DETAILS

Table A.1: Machine operations.

Operation	Description
Register	
`READ`	Write the symbol under the input head to the register.
`TAPE_READ`	Write the symbol under the work head to the register.
`WRITE`	Write the symbol in the register to the cell under the work head.
`LOAD_0`	Set the register to the literal 0.
`LOAD_1`	Set the register to the literal 1.
Head Movement	
`IN_LEFT`	Move the input head one position to the left.
`IN_RIGHT`	Move the input head one position to the right.
`TAPE_LEFT`	Move the work head one position to the left.
`TAPE_RIGHT`	Move the work head one position to the right.

Table A.2: Overview of the advanced machine configuration.

Field	Description
Input Tape	Holds the problem instance (read-only).
Work Tape	Holds the output; initially blank (read-write).
Input Head	Current read position on the input tape.
Input Anchor	Saved position on the input tape.
Work Head	Current read-write position on the work tape.
Work Anchor	Saved position on the work tape.
State	Internal state $q \in Q$.

Table A.3: Advanced machine operations.

Operation	Description
Head Movement	
IN_LEFT	Move the input head one position to the left.
IN_RIGHT	Move the input head one position to the right.
TAPE_LEFT	Move the work head one position to the left.
TAPE_RIGHT	Move the work head one position to the right.
Writing	
COPY	Write the symbol under the input head to the work head position.
PUT_0	Write a literal 0 to the position under the work head.
PUT_1	Write a literal 1 to the position under the work head.
Anchors	
ANCHOR_SET	Save the work head position as the work anchor.
ANCHOR_GO	Move the work head to the work anchor position.
IN_ANCHOR_SET	Save the input head position as the input anchor.
IN_ANCHOR_GO	Move the input head to the input anchor position.
Scanning	
IN_SEEK_LEFT	Move the input head to the left until it reaches a blank symbol (_).
Composed Operations	
BOTH_RIGHT	Execute IN_RIGHT and TAPE_RIGHT.
ANCHOR_STEP	Move work anchor one position right, then execute ANCHOR_GO.
IN_ANCHOR_STEP	Move input anchor one position right, then execute IN_ANCHOR_GO.
BOTH_ANCHOR_STEP	Execute ANCHOR_STEP and IN_ANCHOR_STEP.
PUTSTEP_0	Execute PUT_0 followed by BOTH_RIGHT.
PUTSTEP_1	Execute PUT_1 followed by BOTH_RIGHT.
IN_SEEK_HOME	Execute IN_SEEK_LEFT followed by IN_RIGHT.

B Context Model Details

Table B.1: Contexts used by the policy. *Configuration* contexts read the machine at the moment of the decision. *Program* contexts read the partial table built so far; a row is decided differently depending on what the program has already committed to. Here q is the internal state, s_{in} and s_{wk} the symbols under the input and work heads, r the register, $h_{\text{in}}, h_{\text{wk}}$ the head positions, $|Q|$ the number of states and $|\Sigma|$ the number of symbols.

Family	**Context key**	**Sets**
Unary		
State	(q)	1
Input symbol	(s_{in})	1
Work symbol	(s_{wk})	1
Register	(r)	1
Pairs		
All pairwise conjunctions	(q, s_{in}), (q, s_{wk}), (q, r), $(s_{\text{in}}, s_{\text{wk}})$, (s_{in}, r), (s_{wk}, r)	6
Windows ($\rho = 1, 2, 3$)		
Input window	$(\rho,\ s_{h_{\text{in}}-\rho} \dots s_{h_{\text{in}}+\rho})$	3
Work window	$(\rho,\ s_{h_{\text{wk}}-\rho} \dots s_{h_{\text{wk}}+\rho})$	3
Input window $\times$ state	$(\rho,\ q,\ s_{h_{\text{in}}-\rho} \dots s_{h_{\text{in}}+\rho})$	3
Work window $\times$ state	$(\rho,\ q,\ s_{h_{\text{wk}}-\rho} \dots s_{h_{\text{wk}}+\rho})$	3
Cross-tape and history		
Head alignment	$\text{bucket}(h_{\text{in}} - h_{\text{wk}})$	1
Last action	the action that produced this observation	1
Configuration subtotal		*24*
Program (the partial table T at the decision)		
Sibling agreement	per obs. field f in committed rows differing from this one in only f: none / all took the same action / took different actions	4
Committed rows	per (obs. field, symbol): the action committed at the row with that field set to that symbol / none	$\lvert Q\rvert + 3\lvert\Sigma\rvert$
Operation profile	the set of operations used anywhere in T	1
Table size	$\text{bucket}(\lvert T\rvert)$	1
State shape	is q already used; how many states are in play; is q the newest state	3
Last action	the most recently committed action	1
Program subtotal		$\lvert Q\rvert + 3\lvert\Sigma\rvert + 10$
Total		$\lvert Q\rvert + 3\lvert\Sigma\rvert + 34$

C Problem Domain Details

Copy. A problem instance is a binary string written on the input tape as $\mathtt{w}_0\mathtt{w}_1 \dots \mathtt{w}_{n-1}$. The output written to the work tape reproduces it position by position ($\mathtt{w}_0\mathtt{w}_1 \dots \mathtt{w}_{n-1}$).

Example: 1010 → 1010

Reverse. A problem instance is a binary string written on the input tape as $\mathtt{w}_0\mathtt{w}_1 \dots \mathtt{w}_{n-1}$. The output written to the work tape is the same string in reverse order ($\mathtt{w}_{n-1}\mathtt{w}_{n-2} \dots \mathtt{w}_0$).

Example: 1010 → 0101

Bigram Flip. A problem instance is a binary string written on the input tape as $\mathtt{w}_0\mathtt{w}_1 \dots \mathtt{w}_{n-2}\mathtt{w}_{n-1}$. The output written to the work tape swaps the two symbols of each adjacent pair ($\mathtt{w}_1\mathtt{w}_0 \dots \mathtt{w}_{n-1}\mathtt{w}_{n-2}$).

Example: 101000 → 010100

For copy, reverse, and bigram flip, we use $|Q| = 13$ for the number of states; with 9 operations for manipulating the machine, this gives $|\mathcal{A}| = 117$. Training instances are sampled from lengths 6, 7, 10, 11. Testing instances are sampled from lengths 6, 7, 10, 11, 20, 21, 30, 31, 40, 41. 20 instances are sampled from each length to give 200 testing instances.

Addition. A problem instance $a + b$ is written on the input tape as $\mathtt{a}_0\mathtt{a}_1 \ldots \mathtt{a}_{\mathtt{n}_\mathtt{a}-1} + \mathtt{b}_0\mathtt{b}_1 \ldots \mathtt{b}_{\mathtt{n}_\mathtt{b}-1}$, where operands are encoded least-significant bit (LSB) first. The output sum $c = a + b$ written to the work tape also follows the same LSB-first notation ($\mathtt{c}_0\mathtt{c}_1 \ldots \mathtt{c}_{\mathtt{n}_\mathtt{c}-1}$).

Example: $1010 + 1010 \to 0101$

For addition, we use $|Q| = 13$, giving $|\mathcal{A}| = 117$. The curriculum used is given in Table C.1. Training instances are sampled from lengths 6, 11, 14, 19, 21. Testing instances are sampled from lengths 6, 10, 20, 21, 30, 31, 40, 41. 20 instances are sampled from each length to give 160 testing instances.

Table C.1: Addition curriculum for $a + b$.

Level	Restriction	Description
1	$b = 0$	Adding a number to 0
2	$a \wedge b, \quad \lvert a\rvert = \lvert b\rvert$	No carry needed
3	$\lvert a\rvert = \lvert b\rvert$	Carries allowed, but same length only
4	unrestricted	—

Multiplication. A problem instance $a \times b$ is written on the input tape as $\mathtt{a}_0\mathtt{a}_1 \ldots \mathtt{a}_{\mathtt{n}_\mathtt{a}-1} \times \mathtt{b}_0\mathtt{b}_1 \ldots \mathtt{b}_{\mathtt{n}_\mathtt{b}-1}$, where operands are encoded least-significant bit (LSB) first. The output product $c = a \times b$ written to the work tape also follows the same LSB-first notation ($\mathtt{c}_0\mathtt{c}_1 \ldots \mathtt{c}_{\mathtt{n}_\mathtt{c}-1}$).

Example: $1010 \times 1010 \to 10011$

We use $|Q| = 7$, giving $|\mathcal{A}| = 133$. The curriculum used is given in Table C.2, which also gives the training length distribution. A curriculum shape has the form $A \times BpPgG$, where p denotes that P bits in B are 1 and g denotes that consecutive 1-bits in B are at least G apart. Testing instances are sampled from the shapes $2{\times}8p2$, $4{\times}8p2$, $5{\times}8p2$, $8{\times}8p2$, $12{\times}8p2$, $16{\times}8p2$, $16{\times}8$, $6{\times}8p3$, $6{\times}8$, $6{\times}8p6$, $12{\times}12$, $8{\times}16$ and $24{\times}24$. 20 instances are sampled from each shape to give 260 testing instances.

Table C.2: Multiplication curriculum for $A \times BpPgG$.

Level	A	B	P	G
1	2	8	1	—
2	2	8	2	2
3	2	8	2	—
4	3	8	2	—
5	4	8	2	—
6	6	8	2	—
7	8	8	2	—
8	12	8	2	—
9	16	8	2	—
10	16	8	—	—

D Baseline Details

Note that the score in Algorithm D.1 is calculated in a different way from our method:

$$J = \left(M_{\text{solved}}, \sum_{i=1}^{M} \frac{P_i}{T_i} \right) \tag{D.1}$$

It first scores by M_{solved}, and in the case of ties, it scores by the partial progress toward the target answer of length T_i. The partial progress P_i is the length of the longest correct answer prefix written to the work tape.

Algorithm D.1 Beam Hill-Climb over Deterministic Programs (Baseline)

Require: Problem instances $\{1, \ldots, M\}$, observation space $\mathcal{O}_{\text{m}}$, action set $\mathcal{A}$
Require: Beam width W, patience threshold P
Ensure: Solved program table T^*

while time remaining **do** ▷ Random restarts
  $\mathcal{B} \leftarrow W$ tables drawn uniformly from $\mathcal{A}^{\mathcal{O}_{\text{m}}}$ ▷ Each is a complete deterministic program
  $J_{\text{best}} \leftarrow \max_{T \in \mathcal{B}} \textsc{Score}(T)$ ▷ Equation (D.1)
  while J_{best} improved within the last P steps **do**
    if some $T \in \mathcal{B}$ solves all M instances **then**
      return T
    $\mathcal{N} \leftarrow \left\{ T[o \mapsto a] \;:\; T \in \mathcal{B},\ o \in \textsc{Visited}(T),\ a \in A \right\}$ ▷ 1-step neighbour mutations
    $\mathcal{B} \leftarrow$ the W best of $\mathcal{B} \cup \mathcal{N}$ under $\textsc{Score}$
    $J_{\text{curr}} \leftarrow \max_{T \in \mathcal{B}} \textsc{Score}(T)$
    if $J_{\text{curr}} > J_{\text{best}}$ **then**
      $J_{\text{best}} \leftarrow J_{\text{curr}}$
return Failure

E EXAMPLE PROGRAMS

Table E.1: Program table discovered for copy.

Current Observation					**Action**	
State	Input	Work	Register		Operation	Next State
q_0	0	_	_	→	READ	q_8
q_0	1	_	_	→	READ	q_8
q_1	0	0	0	→	READ	q_8
q_1	0	_	0	→	READ	q_8
q_1	1	1	1	→	TAPE_RIGHT	q_3
q_3	0	_	0	→	IN_RIGHT	q_8
q_3	1	_	1	→	IN_RIGHT	q_8
q_8	0	0	0	→	TAPE_RIGHT	q_3
q_8	0	_	0	→	WRITE	q_1
q_8	0	_	1	→	READ	q_1
q_8	1	_	0	→	READ	q_8
q_8	1	_	1	→	WRITE	q_1

Table E.2: Program table discovered for reverse.

Current Observation					**Action**	
State	Input	Work	Register		Operation	Next State
q_0	0	_	0	→	WRITE	q_9
q_0	0	_	_	→	WRITE	q_4
q_0	1	_	_	→	IN_RIGHT	q_4
q_0	_	_	_	→	LOAD_1	q_3
q_1	1	_	1	→	IN_LEFT	q_{12}
q_3	_	_	1	→	TAPE_LEFT	q_9
q_4	0	_	_	→	IN_RIGHT	q_0
q_4	1	_	_	→	TAPE_READ	q_0
q_4	_	_	0	→	IN_LEFT	q_{12}
q_4	_	_	1	→	IN_LEFT	q_{12}
q_4	_	_	_	→	LOAD_1	q_4
q_5	1	_	0	→	LOAD_1	q_{12}
q_8	0	_	0	→	WRITE	q_9
q_8	1	0	0	→	TAPE_RIGHT	q_5
q_9	0	0	0	→	IN_LEFT	q_{12}
q_9	1	1	1	→	TAPE_RIGHT	q_1
q_9	_	_	1	→	LOAD_0	q_4
q_{12}	0	0	0	→	TAPE_RIGHT	q_8
q_{12}	0	_	0	→	TAPE_RIGHT	q_0
q_{12}	0	_	1	→	LOAD_0	q_8
q_{12}	1	0	0	→	WRITE	q_8
q_{12}	1	_	1	→	WRITE	q_9

Table E.3: Program table discovered for bigram flip.

Current Observation					**Action**	
State	Input	Work	Register		Operation	Next State
q_0	0	_	_	→	LOAD_0	q_8
q_0	1	_	1	→	IN_RIGHT	q_{10}
q_0	1	_	_	→	LOAD_1	q_{10}
q_1	0	1	1	→	READ	q_8
q_1	0	_	0	→	TAPE_READ	q_{11}
q_2	0	1	0	→	TAPE_RIGHT	q_2
q_2	0	_	0	→	IN_RIGHT	q_8
q_2	1	0	1	→	TAPE_RIGHT	q_0
q_2	1	_	1	→	IN_RIGHT	q_{11}
q_3	1	0	1	→	WRITE	q_6
q_5	0	0	0	→	IN_RIGHT	q_{10}
q_6	0	0	0	→	TAPE_RIGHT	q_9
q_6	0	1	0	→	WRITE	q_9
q_6	1	1	1	→	TAPE_RIGHT	q_2
q_8	0	1	0	→	READ	q_6
q_8	0	_	0	→	WRITE	q_6
q_8	1	0	1	→	TAPE_LEFT	q_3
q_8	1	_	0	→	READ	q_8
q_8	1	_	1	→	WRITE	q_6
q_9	0	0	0	→	TAPE_RIGHT	q_2
q_9	0	1	1	→	TAPE_LEFT	q_1
q_9	0	_	0	→	WRITE	q_5
q_9	0	_	_	→	IN_RIGHT	q_0
q_9	1	1	1	→	TAPE_RIGHT	q_0
q_{10}	0	0	0	→	TAPE_RIGHT	q_1
q_{10}	0	_	1	→	LOAD_0	q_8
q_{10}	1	0	0	→	READ	q_8
q_{10}	1	_	1	→	WRITE	q_6
q_{11}	0	_	1	→	WRITE	q_9
q_{11}	0	_	_	→	WRITE	q_9
q_{11}	1	_	1	→	WRITE	q_9

Table E.4: Program table discovered for addition on the base machine.

Current Observation					Action	
State	Input	Work	Register		Operation	Next State
q_0	0	0	1	→	WRITE	q_6
q_0	0	_	_	→	LOAD_1	q_3
q_0	1	0	0	→	READ	q_{12}
q_0	1	0	1	→	TAPE_RIGHT	q_6
q_0	1	1	0	→	WRITE	q_6
q_0	1	1	1	→	IN_RIGHT	q_8
q_0	1	_	_	→	LOAD_1	q_3
q_1	0	1	0	→	WRITE	q_6
q_1	1	1	0	→	TAPE_READ	q_4
q_1	_	1	0	→	IN_LEFT	q_6
q_3	0	0	1	→	IN_RIGHT	q_8
q_3	0	1	1	→	IN_RIGHT	q_8
q_3	0	_	1	→	WRITE	q_5
q_3	1	0	1	→	WRITE	q_0
q_3	1	1	1	→	LOAD_0	q_0
q_3	1	_	1	→	WRITE	q_0
q_4	1	1	1	→	TAPE_RIGHT	q_6
q_4	_	1	0	→	TAPE_LEFT	q_1
q_5	+	_	0	→	LOAD_1	q_{12}
q_5	0	0	0	→	LOAD_1	q_{12}
q_5	0	1	0	→	IN_RIGHT	q_1
q_5	0	1	1	→	READ	q_{10}
q_5	0	_	0	→	LOAD_1	q_{12}
q_6	+	0	0	→	TAPE_RIGHT	q_5
q_6	0	0	0	→	TAPE_RIGHT	q_5
q_6	0	1	1	→	TAPE_LEFT	q_8
q_6	1	0	0	→	READ	q_0
q_6	1	0	1	→	IN_RIGHT	q_0
q_6	1	1	1	→	IN_RIGHT	q_7
q_6	1	_	1	→	WRITE	q_0
q_6	_	1	0	→	WRITE	q_{11}
q_7	+	1	0	→	WRITE	q_6
q_7	0	1	0	→	WRITE	q_6
q_7	0	1	1	→	READ	q_1
q_7	1	1	0	→	WRITE	q_0
q_7	1	1	1	→	TAPE_RIGHT	q_6
q_8	+	0	1	→	TAPE_LEFT	q_8
q_8	+	1	1	→	TAPE_LEFT	q_8
q_8	+	_	1	→	IN_RIGHT	q_8
q_8	0	0	1	→	TAPE_RIGHT	q_3
q_8	0	1	1	→	TAPE_RIGHT	q_3
q_8	0	_	1	→	TAPE_RIGHT	q_3
q_8	1	0	1	→	TAPE_RIGHT	q_3
q_8	1	1	1	→	TAPE_RIGHT	q_3
q_8	1	_	1	→	TAPE_RIGHT	q_3
q_8	_	1	1	→	IN_RIGHT	q_{10}
q_{10}	0	1	0	→	IN_RIGHT	q_7
q_{10}	_	1	1	→	LOAD_0	q_4
q_{12}	+	_	1	→	TAPE_LEFT	q_8
q_{12}	0	0	1	→	IN_RIGHT	q_0
q_{12}	0	_	1	→	TAPE_LEFT	q_8
q_{12}	1	0	1	→	LOAD_1	q_0

Table E.5: Program table discovered for addition on the advanced machine.

Current Observation				**Action**	
State	Input	Work		Operation	Next State
q_0	+	1	$\rightarrow$	PUTSTEP_0	q_1
q_0	+	_	$\rightarrow$	PUTSTEP_0	q_3
q_0	0	0	$\rightarrow$	ANCHOR_STEP	q_1
q_0	0	1	$\rightarrow$	ANCHOR_STEP	q_1
q_0	0	_	$\rightarrow$	PUTSTEP_0	q_0
q_0	1	0	$\rightarrow$	COPY	q_6
q_0	1	1	$\rightarrow$	PUT_0	q_3
q_0	1	_	$\rightarrow$	PUTSTEP_1	q_0
q_1	0	0	$\rightarrow$	BOTH_RIGHT	q_0
q_1	0	1	$\rightarrow$	BOTH_RIGHT	q_0
q_1	1	0	$\rightarrow$	PUT_1	q_5
q_1	1	1	$\rightarrow$	IN_LEFT	q_0
q_3	0	_	$\rightarrow$	ANCHOR_GO	q_1
q_3	1	0	$\rightarrow$	TAPE_RIGHT	q_0
q_3	1	_	$\rightarrow$	ANCHOR_GO	q_1
q_4	1	0	$\rightarrow$	BOTH_RIGHT	q_0
q_4	1	1	$\rightarrow$	BOTH_RIGHT	q_0
q_5	1	1	$\rightarrow$	ANCHOR_GO	q_4
q_6	1	1	$\rightarrow$	ANCHOR_STEP	q_4

Table E.6: Program table discovered for multiplication on the advanced machine.

Current Observation				**Action**	
State	Input	Work		Operation	Next State
q_0	0	0	$\rightarrow$	BOTH_ANCHOR_STEP	q_0
q_0	0	1	$\rightarrow$	BOTH_ANCHOR_STEP	q_0
q_0	0	_	$\rightarrow$	PUT_0	q_0
q_0	1	0	$\rightarrow$	IN_RIGHT	q_3
q_0	1	1	$\rightarrow$	IN_RIGHT	q_3
q_0	1	_	$\rightarrow$	PUT_0	q_5
q_1	0	0	$\rightarrow$	PUTSTEP_0	q_1
q_1	0	1	$\rightarrow$	BOTH_RIGHT	q_1
q_1	0	_	$\rightarrow$	COPY	q_1
q_1	1	0	$\rightarrow$	PUTSTEP_1	q_1
q_1	1	1	$\rightarrow$	PUTSTEP_0	q_4
q_1	1	_	$\rightarrow$	PUTSTEP_1	q_1
q_1	_	_	$\rightarrow$	BOTH_ANCHOR_STEP	q_0
q_2	_	_	$\rightarrow$	IN_SEEK_LEFT	q_1
q_3	*	0	$\rightarrow$	IN_RIGHT	q_1
q_3	*	1	$\rightarrow$	IN_RIGHT	q_1
q_3	0	0	$\rightarrow$	IN_RIGHT	q_3
q_3	0	1	$\rightarrow$	IN_RIGHT	q_3
q_3	1	0	$\rightarrow$	IN_RIGHT	q_3
q_3	1	1	$\rightarrow$	IN_RIGHT	q_3
q_4	0	0	$\rightarrow$	PUT_1	q_1
q_4	0	1	$\rightarrow$	PUTSTEP_0	q_4
q_4	0	_	$\rightarrow$	PUT_1	q_1
q_4	1	0	$\rightarrow$	COPY	q_5
q_4	1	1	$\rightarrow$	BOTH_RIGHT	q_4
q_4	1	_	$\rightarrow$	COPY	q_5
q_4	_	_	$\rightarrow$	PUT_0	q_5
q_5	1	0	$\rightarrow$	IN_RIGHT	q_3
q_5	1	1	$\rightarrow$	PUTSTEP_0	q_4
q_5	_	0	$\rightarrow$	PUTSTEP_1	q_2

F Program Correctness Proofs

The supplementary material contains Lean 4 programs, one file per domain (`copy.lean`, `reverse.lean`, `bigram.lean`, `addition.lean`, `multiplication.lean`), which formalise the machine of Section 3 and prove the discovered programs correct.

G Additional Experiment Details

The hyperparameters used in experiments that have not already been mentioned are listed in Table G.1 and Table G.2. The experiments were all run on Intel Xeon Gold 6448Y CPUs using 32 CPU cores for each run. At most 32G of memory was used for each run.

Table G.1: Domain-specific hyperparameters.

	Copy	Reverse	Bigram	Addition	Multiplication
Step cap	$6n{+}20$	$8n{+}20$	$8n{+}20$	$14n{+}20$	$0.75n^2{+}n{+}20$
Decision cap	—	—	—	180	120
Parallel rollouts (K)	1400	1400	1400	15000	1400
Restart patience (in rounds)	24	24	24	24	60

Table G.2: Hyperparameters held constant across every experiment.

Parameter	**Value**	**Description**
Instances		
Training set size M	20	Instances a program must solve simultaneously.
Dataset cap	100	Distinct training instances drawn per curriculum level, sampled with replacement thereafter.
Search		
λ	3.5	Weight of the efficiency term in the program scoring function.
Confirmation sets B	10	Fresh sets of M problem instances a program must pass before final acceptance.
Verification cap	$200\times$	Step cap during verification, as a multiple of the search step cap.
Curriculum		
Level confirmations B_{level}	3	Sets of M problem instances required to advance a level.
Level patience	60	Rounds without improvement before restarting or demoting.
Policy*		
ε	0.05	Uniform mixing weight on the action distribution.
Batch size	4096	Number of (o^π, a) pairs trained on per gradient step.
Gradient steps	20	Optimiser steps per round.
Experts	4	Additive terms per context: identity, operation, state relation, symbol relation.
Baseline		
Beam width W	32	Candidate programs retained per step.
Depth limit	4000	Max single-entry edits before restarting.
Patience	40	Steps without improvement before restarting.
Hardware		
Workers	32	CPU processes for running parallel rollouts.

*We follow Orseau et al. (2023) for the parameterisation and training conventions of the context model.