# NEURAL LANGUAGE PROGRAMMING CAN GENERALIZE OUT OF COMPOSITION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Artificial intelligence systems are showing promise on difficult tasks such as world modeling and abstract reasoning. A central challenge, however, is whether they can reuse what is learned *compositionally*, i.e., combine familiar operations in novel ways to address previously unseen tasks. We refer to this capability as *out-of-composition* (OOC) generalization, where a learner must recover reusable computational structure from input-output observations without access to the generating programs or a predefined executable language. Neural networks can learn directly from input-output observations, but typically represent the underlying computation implicitly and monolithically. Symbolic program synthesis is explicitly compositional, but assumes a predefined language. *Neural Language Programming* (NeuLP) bridges this gap by learning (1) a language consisting of a codebook of vectors that represent abstract operations, (2) an encoder that constructs a sequence out of that codebook based on input-output pairs from the task at hand, and (3) an interpreter that applies the selected sequence to a novel input. The discrete language is amenable to composition as well as search. Learning such a language requires discovering reusable operations without knowing the complexity of the programs that generated the training data or which learned operations can be trusted. NeuLP addresses the former with a program-space self-curriculum that progresses from shorter to longer programs, and the latter with a Bayesian execution-evidence prior that uses successful executions as evidence of operation reliability. Empirically, across four domains spanning list processing, ARC-AGI-style abstract reasoning, image editing, and cellular automata, we show that NeuLP generalizes to unseen recombinations and compositional depth, outperforming neural program synthesis baselines.

## 1 INTRODUCTION

In many problems, the learner observes a data-generating process with compositional structure only through input-output pairs. A world model of a physical or dynamical system is a natural example: the same local interactions or transition rules are applied iteratively to the system state, giving the data-generating process a program-like structure (Ha & Schmidhuber, 2018; Brunton et al., 2016).

**Problem setting.** The focus of this paper is on what we term *out-of-composition* (OOC) generalization: in a programming-by-example (PBE) setting, the learner sees only input-output examples without access to the generating programs or their domain-specific language (DSL). Training data include examples of primitive operations and some compositions, while OOC generalization is tested on examples borne out of novel compositions that were either not represented in the training set (Recombination OOC) or deeper in length (Length OOC).

Formally, each task consists of a query input $x_q$ along with $M$ input-output demonstrations,

$$\mathcal{U} = \{(x_m, y_m)\}_{m=1}^{M}.$$

The learner must predict $y_q$ for $x_q$ from $\mathcal{U}$, without access to the generating program, its length, or intermediate states. To do so, a training set of examples based on short programs (e.g., lengths 1–3) is available. Figure 1 illustrates this setting for an abstract visual reasoning domain.

**Prior work.** Monolithic neural networks can be fit to the observed input-output pairs without explicitly uncovering the underlying program structure. In fact, even high-capacity Transformers and



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.

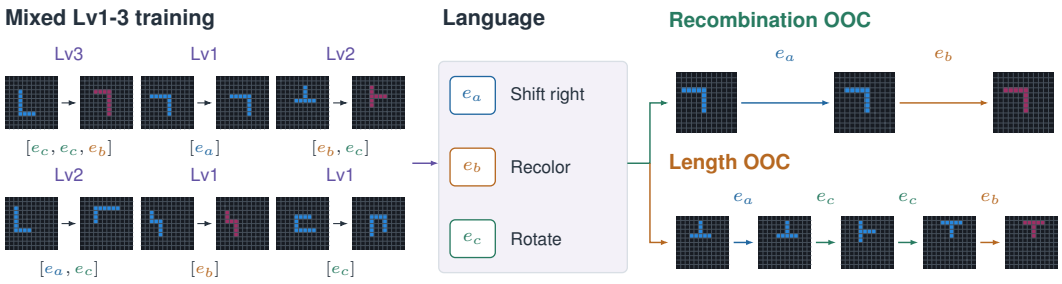



Figure 1: Out-of-composition generalization in CompARC, an ARC-AGI-style grid domain. Training includes the level-1 (Lv1) primitive operations $e_a$, $e_b$, and $e_c$, as well as level-2 (Lv2) and level-3 (Lv3) compositions such as $[e_b, e_c]$ and $[e_c, e_c, e_b]$. An in-distribution test sample applies one of these programs to a new input (not pictured); Recombination OOC evaluates an unseen composition such as $[e_a, e_b]$; Length OOC evaluates a longer composition such as $[e_a, e_c, e_c, e_b]$. Operation names, programs, and levels are shown only for illustration.

language models remain unreliable at OOC generalization (Lake & Baroni, 2018; Keysers et al., 2020; Dziri et al., 2023). On the other hand, symbolic program synthesis supports explicit composition, but assumes that a suitable executable DSL is given (Gulwani et al., 2017). Modular neural and neuro-symbolic systems similarly rely on predefined languages, types, or composition rules (Andreas et al., 2016; Valkov et al., 2018). More recently, LPN was introduced as a search over learned latent programs (Macfarlane & Bonnet, 2025). NLI extends LPN by jointly learning a discrete neural language and a differentiable interpreter from input-output data (Macfarlane et al., 2026). While promising, LPN/NLI exhibit subpar performance in many cases.

**This paper.** We hypothesize that answering the following two research questions can enable reliable OOC generalization: how do we learn a high-quality executable language, and how do we use that language to search in the composition space of learned operations?

We introduce *NeuLP* (Neural Language Programming) to address both questions. NeuLP learns an executable language from input-output data and uses it for composition and search. The proposed approach has three components: (1) *a language* consisting of a codebook of vectors that represent abstract operations; (2) *a programmer* consisting of an encoder that constructs a sequence out of that codebook based on input-output pairs $\mathcal{U}$ from the task at hand; and (3) *an interpreter* that applies the selected sequence to a query input $x_q$. At the core of NeuLP are two key technical contributions that enable OOC generalization empirically.

Learning a useful neural language from input-output data is itself difficult: the learner must discover reusable operations while fitting well to the data. Deeper programs are harder to predict, naturally invoking the use of curriculum learning which typically orders or selects examples according to *known* difficulty (Bengio et al., 2009; Kumar et al., 2010; Graves et al., 2017). Here, however, the generating programs and their depths are hidden. NeuLP addresses this by forming curricula in program rather than data space: this so-called *self-curriculum* initially restricts the programmer to outputting short programs and progressively makes them longer, all on the same set of length-unknown training examples.

Learning the language also requires a notion of program language quality. The *minimum description length* (MDL) principle provides a natural view in symbolic program synthesis: reusable operations lead to shorter descriptions of observed programs (Grünwald, 2007). Existing MDL-style program priors, however, are typically defined over a fixed symbolic DSL whose operations already have known semantics (Liang et al., 2010; Lake et al., 2015; Ellis et al., 2021). NeuLP must instead learn both the operations and their semantics from data. Inspired by this view, we introduce a *Bayesian execution-evidence prior* (BEEP), which uses programs whose execution reproduces the observed outputs as evidence for a posterior over learned productions. Operations that recur across many successful compositions receive stronger support, while the induced description cost favors shorter programs. BEEP is therefore not only a regularizer for language acquisition but also provides a Bayesian belief over the learned language that can be used to guide composition, search, and synthetic program generation ("dreaming").

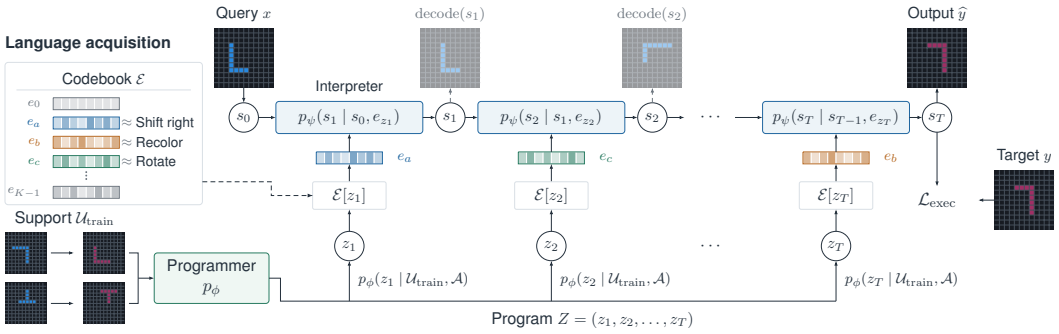



Figure 2: **Stage 1: language acquisition.** A domain-dependent encoder maps the support set $\mathcal{U}_{\text{train}}$ to a non-autoregressive Transformer-style programmer, which predicts all program slots jointly. Each $z_t$ indexes an embedding $e_{z_t}$ in the shared codebook, which conditions the interpreter step $p_\psi(s_t \mid s_{t-1}, e_{z_t})$. Execution loss jointly trains the encoder, programmer, codebook, and interpreter through Gumbel–Softmax token selection. Faded grids visualize intermediate states if decoded; they receive no intermediate target supervision. Not every embedding acquires stable, interpretable semantics; in this example, only the three colored entries are post-hoc validated as level-1 operations.

We introduce Comp-II, CompARC, CompCIFAR, and CompLife, a controlled evaluation suite spanning list transformations, grid reasoning, visual transformations, and cellular dynamics. CompARC and CompLife test composition over discrete state transitions, while CompCIFAR uses continuous visual states. The benchmarks separately test held-out inputs, unseen recombinations, and greater composition depth. NeuLP substantially outperforms neural and learned-program baselines such as LPN/NLI in the hardest OOC settings, and can acquire a new operation and reuse it in unseen compositions.

## 2 NeuLP: Neural Language Programming

**Neural language programs.** NeuLP is a system that combines problem-solving (i.e., proposing a neural solution to an input problem through a few input-output examples), reusable knowledge, and execution. Given input-output demonstrations $\mathcal{U}$, a "programmer" predicts an ordered token sequence $Z$ in a learned neural language $\mathcal{A}$, and a differentiable interpreter executes $Z$ on a new input. Training proceeds in three stages. Stage 1 learns the language, programmer, and interpreter from input-output data; Stage 2 freezes them and searches for high-quality discrete programs that can further improve the programmer; Stage 3 trains the programmer on the programs discovered in Stage-2 while keeping the language and interpreter fixed.

A neural language $\mathcal{A} = (\mathcal{E}, \mathcal{G})$ consists of a finite codebook $\mathcal{E} = \{e_0, \ldots, e_{K-1}\}$ and a sequential grammar $\mathcal{G}$, where each $e_k \in \mathbb{R}^d$ is a learned token embedding with no predefined symbolic meaning. Token 0 is SKIP. The same codebook is shared across tasks and program positions. A program is an ordered sequence $Z = (z_1, \ldots, z_T)$. The programmer is non-autoregressive and predicts all $T$ slots jointly, while the interpreter factorizes over execution steps:

$$\underbrace{p_\phi(Z \mid \mathcal{U}, \mathcal{A})}_{\text{programmer}} = \prod_{t=1}^{T} p_\phi(z_t \mid \mathcal{U}, \mathcal{A}), \tag{1}$$

$$\underbrace{p_\psi(y, s_{0:T} \mid x, Z, \mathcal{A})}_{\text{execution}} = p_\psi(s_0 \mid x) \left[ \prod_{t=1}^{T} p_\psi(s_t \mid s_{t-1}, e_{z_t}) \right] p_\psi(y \mid s_T). \tag{2}$$

Given $Z$, the interpreter applies $p_\psi(s_t \mid s_{t-1}, e_{z_t})$ sequentially. We write $y \sim F_\psi(x, Z, \mathcal{A})$ for the output of the complete rollout. Figure 2 illustrates how the programmer, codebook, and interpreter are trained jointly; domain-specific architectures are in Appendix C.3. Figure 3 then shows how the same program interface is used across the three stages.

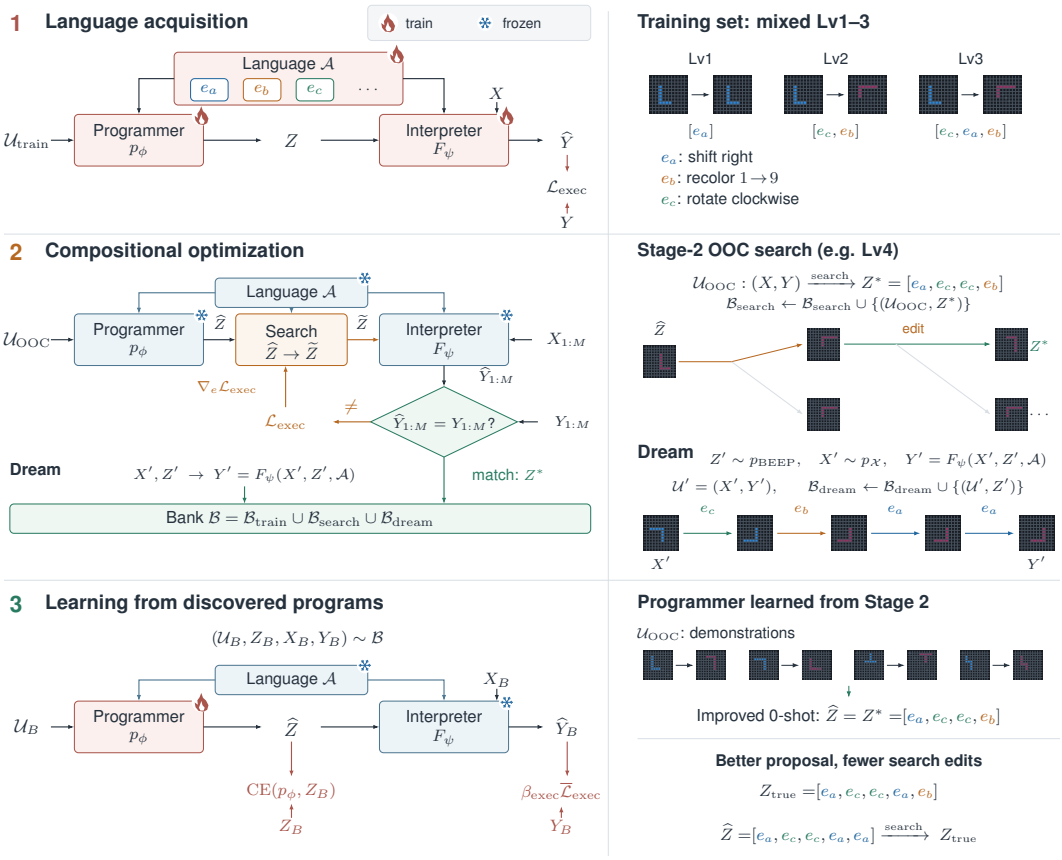



Figure 3: NeuLP uses one explicit neural program interface across three stages. **(1) Language acquisition** learns the programmer, language $\mathcal{A}$, and interpreter from input-output data. **(2) Compositional optimization** freezes these modules and searches over discrete programs using execution gradients. Dreaming samples programs from the learned language prior, samples inputs, and generates outputs with the frozen interpreter. Successful search traces and dreamed programs are stored in a replay bank. **(3) Learning from discovered programs** trains the programmer to reproduce banked programs while keeping the language and interpreter frozen. Right: CompARC examples show search, dreaming, and improved Stage 3 proposals. Icons mark trainable and frozen modules.

### 2.1 Stage 1: Language Acquisition

What structure should a learned program language have? Program synthesis typically assumes a finite DSL, a grammar for composing its elements, and a prior over programs. NeuLP uses a simple sequential grammar and learns the neural vocabulary and an execution-informed prior over that vocabulary. Stage 1 therefore jointly trains the programmer $\phi$, interpreter $\psi$, and codebook $\mathcal{E}$, while keeping the sequential grammar $\mathcal{G}$ fixed. For a single training sample $(\mathcal{U}_{\text{train}}, x_q, y_q)$, the loss is:

$$\mathcal{L}_{\text{S1}}(\phi, \psi, \mathcal{E}) = \mathbb{E}_{Z \sim p_\phi(\cdot \mid \mathcal{U}_{\text{train}}, \mathcal{A})} \left[ \mathcal{L}_{\text{exec}}\big(y_q, F_\psi(x_q, Z, \mathcal{A})\big) + \lambda_{\text{BEEP}} \mathcal{C}_{\text{BEEP}}(Z) \right] + \mathcal{L}_{\text{aux}}(\phi, \psi, \mathcal{E}). \tag{3}$$

Here $\mathcal{L}_{\text{exec}}$ favors a sampled program that reproduces the query output, $\mathcal{C}_{\text{BEEP}}$ is the execution-informed program cost defined below, and $\mathcal{L}_{\text{aux}}(\phi, \psi, \mathcal{E})$ collects the stabilization terms detailed in Appendix C.3. Gumbel–Softmax provides gradients through token selection; argmax decoding gives the hard programs used in stages 2 and 3.

**Finite neural DSL.** Without a discrete vocabulary, the programmer can emit a free continuous vector at each program position to condition the interpreter. This is the less constrained representation and would seem easier to fit. NeuLP instead selects each position from a shared finite codebook $\mathcal{E}$ of $K$ vectors, encouraging reusable components similar to non-neural functions in a domain-specific language. In practice, this added language structure substantially improves performance over the continuous baseline (Table 2).

**Sequential grammar and self-curriculum.** Compositional learning often uses a curriculum of simple to difficult (long) programs. In our PBE setting, however, the generating program and its depth are unknown to the learner, so the training examples cannot be ordered this way. NeuLP instead uses the program space itself as the curriculum. The same mixed training data are used throughout while the available program length grows from one active slot to all $T$ slots. We call this a program-space *self-curriculum*. We will show in Table 2 that gradually allowing for longer programs outperforms exposing all slots from the start; schedule details are given in Appendix C.3.

**Bayesian execution-evidence prior (BEEP).** The Minimum Description Length (MDL) principle provides a natural way to measure language quality: a better language gives shorter descriptions of the observed programs through reusable productions (Vitányi & Li, 2000; Gulwani et al., 2017). This assumes that the DSL operations already have fixed semantics. In NeuLP, their semantics are learned. Some codebook entries become stable discrete operations, while others may remain weakly supported or depend on soft execution (Figure 2).

BEEP uses execution itself to determine which learned tokens are reliably supported. During Stage 1, we periodically execute hard argmax programs and retain only those whose executions reproduce the observed outputs—exactly for categorical domains and according to the continuous acceptance rule for CompCIFAR (Appendix C.2). Let $n_k$ denote the accumulated execution evidence for codebook entry $e_k$. Each accepted program contributes its usage of $e_k$, normalized by the number of non-SKIP positions in that program. We place a symmetric Dirichlet prior with concentration $\alpha$ on the usage probabilities of the non-SKIP codebook entries, giving posterior mean $\theta_k$ and normalized token cost $\bar{c}_k$:

$$\theta_k = \frac{n_k + \alpha}{\sum_{j>0}(n_j + \alpha)}, \qquad \bar{c}_k = \frac{-\log \theta_k}{\frac{1}{K-1}\sum_{j=1}^{K-1} -\log \theta_j}, \qquad \bar{c}_0 = 0. \tag{4}$$

Codebook entries that recur across more accepted programs receive higher posterior probability and lower production cost. Following the short-program bias of the Solomonoff–Levin universal prior, we place a geometric prior on the number of active productions (Solomonoff, 1964; Levin, 1974):

$$\mathcal{C}_{\mathrm{BEEP}}(Z) = \textstyle\sum_{t:z_t \neq 0} \bar{c}_{z_t} + L(Z) \log \frac{1}{1-p_{\mathrm{stop}}}, \tag{5}$$

where $L(Z)$ is the number of non-SKIP tokens and $p_{\mathrm{stop}}$ is the fixed probability that the sequential grammar terminates after an active production. BEEP therefore favors both execution-supported productions and shorter programs. Together, these terms form a Bayesian belief over the learned language that regularizes Stage 1 and guides Stage 2 search; the learned production posterior also guides dreaming. Compared with previous learned-program objectives, NLI penalizes the number of distinct tokens used, while NEO MDL corresponds to a uniform production cost in the common coding view. BEEP instead combines token-specific production costs learned from execution evidence with an explicit termination cost. Table 3 shows the corresponding comparison. The full derivation, counting rule, and coefficient controls are given in Appendix C.2.

### 2.2 Stage 2: compositional optimization

Stage 2 freezes the Stage-1 components and searches over programs in the learned language. It begins with *language calibration*: NeuLP runs the Stage-1 programmer over the training stream and executes each hard program proposal. BEEP counts are recomputed from zero, and $\mathcal{V}_{\mathrm{exec}} = \{k > 0 : n_k > 0\}$ collects tokens appearing in at least one admitted complete trace.

**Execution-guided discrete search.** Existing learned-program methods use either continuous gradient refinement, as in NLI, or sampling of complete discrete programs, as in NEO-S (Macfarlane et al., 2026; Baek et al., 2026). NeuLP keeps the program discrete while using execution gradients to decide which legal token replacement to try. For a support PBE $\mathcal{U} = \{(x_m, y_m)\}_{m=1}^{M}$, the programmer proposes an initial hard program $Z^{(0)}$ and search optimizes only $Z$:

$$\mathcal{E}_{\mathrm{search}}(Z;\mathcal{U}) = \frac{1}{M}\sum_{m=1}^{M} \mathcal{L}_{\mathrm{exec}}(y_m, F_\psi(x_m, Z, \mathcal{A})) + \lambda_{\mathrm{search}}\mathcal{C}_{\mathrm{BEEP}}(Z), \quad Z \in (\mathcal{V}_{\mathrm{exec}} \cup \{0\})^T, \tag{6}$$

During search, BEEP carries both Stage 1 execution evidence and the short-program bias into program construction. Inspired by first-order token replacement in HotFlip (Ebrahimi et al., 2018) , we

use a *gradient-based coordinate update* (GCU). For the current token $j = z_t$ at position $t$, we score a legal replacement $k$ by

$$g_t = \nabla_{\tilde{z}_t} \mathcal{E}_{\text{search}}, \qquad \Delta_{t,k} = g_{t,j} - g_{t,k}. \tag{7}$$

The directional score ranks single-token edits. Because gradient scales can differ across recurrent positions (Pascanu et al., 2013), we rank edits within each slot and keep the same number of top-ranked edits from every slot. At each round, these candidates are executed as hard programs, and the best improving candidate becomes the next program for that search trajectory. Table 4 compares GCU with unguided and random editing; full search details are given in Appendix C.4.

**Task-free dreaming.** Search discovers programs only when a target PBE is supplied. The learned language itself, together with the interpreter, can also generate executable programs. NeuLP samples token identities from the BEEP production posterior, samples inputs from the domain input distribution $p_{\mathcal{X}}$, and executes the resulting program:

$$\begin{aligned} &L' \sim \text{Unif}\{1, \ldots, T\}, \qquad z'_t \sim \boldsymbol{\theta}(\cdot \mid \mathcal{V}_{\text{exec}})\ (t \le L'), \quad z'_t = 0\ (t > L'), \\ &x'_m \sim p_{\mathcal{X}}, \qquad y'_m = F_\psi(x'_m, Z', \mathcal{A}). \end{aligned} \tag{8}$$

We reject dreams that leave all inputs unchanged or reproduce the output signature of a previously retained dream. Accepted program-I/O pairs are added to the replay bank alongside programs found by search. See Appendix C.4 for admission and sampling details.

### 2.3 Stage 3: Learning from discovered programs

Search and dreaming produce useful programs; can these be used to improve the programmer beyond its Stage-1 training? To this end, NeuLP stores the programs of Stage 2 in a replay bank, $\mathcal{B} = \mathcal{B}_{\text{train}} \cup \mathcal{B}_{\text{search}} \cup \mathcal{B}_{\text{dream}}$, and uses them for additional supervision. Each bank record stores $(\mathcal{U}_i, Z_i^*)$, together with query data $(x_i, y_i)$ when available. A stored program is only meaningful under the codebook and interpreter that validated it. Continuing to update either can change the execution of the same token sequence. Stage 3 therefore freezes codebook and interpreter alike and updates only the programmer and its input encoder.

**Learning from discovered programs.** A discovered trace provides a direct target for the programmer, i.e., the sequence $Z_i^*$ which it must try to produce from $\mathcal{U}_i$. However, one would also prefer a slightly different program that is semantically similar to $Z_i^*$, i.e., that produces the correct output. Executing the predicted program on input $x_i$ using the frozen interpreter produces a prediction that can be compared to ground-truth output $y_i$. The loss function combines cross-entropy w.r.t. $Z_i^*$ and output quality w.r.t. $y_i$:

$$\mathcal{L}_{\text{bank}}(\phi; \mathcal{U}_i, Z_i^*, x_i, y_i) = \text{CE}(p_\phi(Z \mid \mathcal{U}_i), Z_i^*) + \beta_{\text{exec}}\, \mathcal{L}_{\text{exec}}\left(p_\phi, F_{\psi_{\text{S1}}}; \mathcal{U}_i, x_i, y_i\right). \tag{9}$$

Training only on stored support–program pairs can overfit the programmer to the bank and weaken its Stage 1 ability to infer the same program from held-out supports $\mathcal{U}_{\text{test}}$ of a train-seen task. For retention, we take a validated training record $(\mathcal{U}_{\text{train}}, Z_{\text{train}}) \in \mathcal{B}_{\text{train}}$, sample new inputs, and use the frozen Stage 1 model to construct $\widetilde{\mathcal{U}}_{\text{train}}$ and distill the Stage 1 programmer:

$$\begin{aligned} \widetilde{\mathcal{U}}_{\text{train}} &= \{(\tilde{x}_m, F_{\psi_{\text{S1}}}(\tilde{x}_m, Z_{\text{train}}, \mathcal{A}))\}_{m=1}^{M}, \\ \mathcal{L}_{\text{train}}(\phi; \widetilde{\mathcal{U}}_{\text{train}}) &= \text{KL}\Big(p_{\phi_{\text{S1}}}(Z \mid \widetilde{\mathcal{U}}_{\text{train}}) \,\|\, p_\phi(Z \mid \widetilde{\mathcal{U}}_{\text{train}})\Big). \end{aligned} \tag{10}$$

Stage 3 alternates replay-bank updates using $\mathcal{L}_{\text{bank}}$ with training-retention updates using $\mathcal{L}_{\text{train}}$. Only the current programmer $\phi$, including its input encoder, is trained; the Stage 1 teacher $\phi_{\text{S1}}$, language, and interpreter remain frozen. Bank construction details are given in Appendix C.5.

**Inference at test time.** At test time, the Stage-3 programmer provides the initial program and the same Stage-2 discrete search is applied when search is enabled. Search essentially uses the gradients of the objective (6) w.r.t. the codebook to modify the initial program and improve the quality of its outputs on the observed OOC support $\mathcal{U}_{\text{OOC}}$. The final program is then executed on the query input.

**Continual extension of the language.** The three-stage cycle above assumes a fixed learned language. We also consider when new input-output data require a previously unseen operation and compositions using it. Running Stage 1 again can learn the new behavior, but the old token assignments may drift. Freezing the validated codebook rows preserves their addresses, but not necessarily their meaning: the shared interpreter can still change how the same frozen embedding executes.

Table 1: Categorical-domain OOC generalization (Section 1). Test I/O is soft zero-shot accuracy on training tasks with held-out support/query; other rows are recombination/length OOC. Exact accuracy (%), mean ± s.d. over five seeds. NeuLP hard Test I/O is in Appendix C.5.5. Bold/underline mark best/second-best means. †: test-time search on OOC; Ep. TF: Episode Transformer.

| | | **NeuLP**† | NLI† | LPN† | Ep. TF | HRM | TRM |
|---|---|---|---|---|---|---|---|
| **Comp-II** | Test I/O | **100.0 ± 0.0** | 99.4 ± 0.6 | 66.9 ± 46.4 | <u>99.8 ± 0.3</u> | 0.0 ± 0.0 | 97.4 ± 2.1 |
| | Recomb. | **100.0 ± 0.0** | 12.7 ± 22.0 | 9.5 ± 9.5 | <u>31.8 ± 27.1</u> | 0.0 ± 0.0 | 11.1 ± 15.3 |
| | Length | **99.9 ± 0.2** | 1.6 ± 1.9 | 2.6 ± 3.6 | 10.7 ± 9.3 | 0.0 ± 0.0 | <u>12.7 ± 17.0</u> |
| **CompARC** | Test I/O | <u>97.3 ± 1.1</u> | 92.6 ± 1.1 | **99.6 ± 0.2** | 90.9 ± 3.0 | 82.7 ± 4.5 | 84.1 ± 9.1 |
| | Recomb. | **80.8 ± 4.1** | 45.5 ± 12.0 | <u>75.0 ± 1.8</u> | 67.3 ± 3.8 | 18.2 ± 11.1 | 44.5 ± 21.4 |
| | Length | **64.9 ± 2.1** | 20.8 ± 3.4 | 23.6 ± 1.4 | 21.7 ± 10.2 | 10.9 ± 7.7 | <u>29.9 ± 15.5</u> |
| **CompLife** | Test I/O | <u>95.2 ± 1.3</u> | 92.5 ± 6.4 | 90.3 ± 1.6 | 1.2 ± 0.6 | **96.1 ± 4.8** | 90.0 ± 8.3 |
| | Recomb. | **88.2 ± 9.3** | 60.5 ± 18.2 | <u>71.1 ± 9.5</u> | 2.6 ± 0.0 | 42.6 ± 10.6 | 49.0 ± 17.2 |
| | Length | **74.0 ± 7.1** | <u>3.2 ± 1.3</u> | 0.0 ± 0.0 | 0.0 ± 0.0 | 0.1 ± 0.2 | 0.1 ± 0.2 |

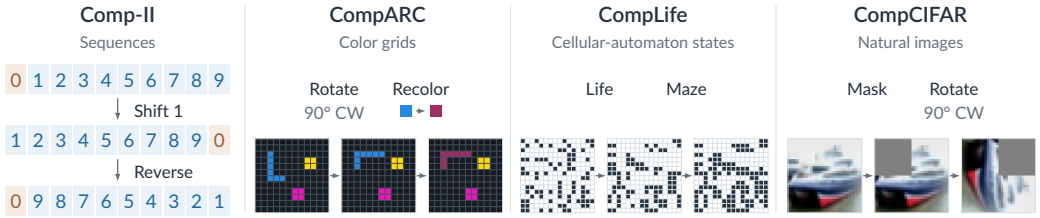



Figure 4: **Four benchmark domains.** Examples show two-operation compositions; models observe only the input-output pairs, not the operations or intermediate states.

We therefore protect non-SKIP codebook rows that occur in the validated bank, leave the remaining rows trainable, and replay stored $(\mathcal{U}, Z, X, Y)$ records while learning from the new data. Replay preserves old behavior while unprotected codebook capacity learns the new operation. The expanded language can then compose the new operation with the retained ones. Table 6 evaluates retention, acquisition of the new operation, and unseen old–new compositions. Freeze-only controls and semantic drift diagnostics are given in Appendix C.5.

## 3 EXPERIMENTS

We evaluate NeuLP through four questions: **RQ1**, whether it generalizes to unseen recombinations and greater compositional depth; **RQ2**, which language-learning and search choices drive this generalization; **RQ3**, whether programs discovered by search can be absorbed into the programmer to improve its search proposals; and **RQ4**, whether the learned language can acquire a new operation while retaining and composing existing ones.

**Protocol.** All four Comp benchmarks use the splits defined in Section 1. We report three evaluation settings: **Test-I/O**, using held-out support $\mathcal{U}_{\text{test}}$ and query I/O for functions seen during training; **recombination OOC**, using unseen compositions within the training depth; and **length OOC**, using compositions deeper than those seen during training. Comp-II, CompARC, and CompLife report exact query accuracy; CompCIFAR reports normalized $\ell_1$ (lower is better). For the final OOC comparison, NeuLP uses 128 program proposals from the Stage-3 programmer as search starts and runs up to 100 search rounds from each; controlled ablations use the budgets stated with each experiment. All main-text results use five seeds except the codebook and search ablations, which use three.

**Baselines.** Episode Transformer is a Transformer trained from scratch over demonstration–query episodes, following the meta-learning setup for compositional generalization and its extension to spatial reasoning. LPN, NLI, and NEO are the closest latent-program baselines (Macfarlane & Bonnet, 2025; Macfarlane et al., 2026; Baek et al., 2026). LPN and NLI use their original $8192 \times 100$ gradient-search budget; NEO-S evaluates 1024 sampled programs without editing, while NeuLP uses $128 \times 100$ discrete search. HRM and TRM provide recurrent-reasoning baselines without an explicit reusable program language (Wang et al., 2025a; Jolicoeur-Martineau, 2025). All models are trained from scratch with approximately matched parameter counts. Details are in Appendix C.6.

Table 2: Structure.

| Choice | Hard Test I/O |
|---|---|
| **Codebook** | $\mathbf{99.73 \pm 0.31}$ |
| Continuous | $33.33 \pm 57.74$ |
| **Self-curr.** | $\mathbf{99.86 \pm 0.19}$ |
| All slots | $69.45 \pm 25.90$ |

Table 3: Neural language prior.

| Objective | Hard Test I/O | CL. | Len.OOC |
|---|---|---|---|
| NLI reuse | $99.9 \pm 0.2$ | 78.5 | $23.8 \pm 14.1$ |
| NEO MDL | $60.3 \pm 44.9$ | 98.1 | $27.8 \pm 7.6$ |
| **BEEP** | $\mathbf{100.0 \pm 0.0}$ | **99.7** | $\mathbf{34.2 \pm 6.4}$ |

Table 4: Search.

| Rule | Solved |
|---|---|
| **GCU** | $\mathbf{86.3 \pm 7.4}$ |
| Rnd. | $72.5 \pm 20.9$ |
| Raw | $43.1 \pm 9.8$ |

Table 5: Learning from discovered programs.

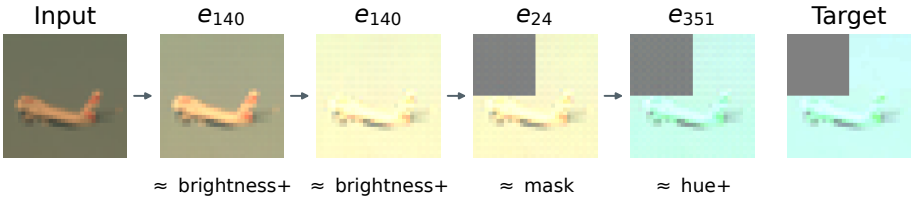
| Inference | Split | Stage 1 | Stage 3 |
|---|---|---|---|
| Zero-shot | Recomb. | $13.64 \pm 6.43$ | $\mathbf{75.00 \pm 3.21}$ |
| | Length | $0.83 \pm 0.24$ | $\mathbf{48.33 \pm 1.89}$ |
| Proposal ($\times 128$) | Recomb. | $20.45 \pm 9.64$ | $\mathbf{81.82 \pm 6.43}$ |
| | Length | $9.38 \pm 4.42$ | $\mathbf{62.50 \pm 1.47}$ |

Table 6: Continual language growth.

| Evaluation | Accuracy (%) |
|---|---|
| Original Test I/O | $97.08 \pm 0.59$ |
| Test I/O (+rev.) | $98.62 \pm 0.98$ |
| Recomb. OOC (+rev.) | $94.64 \pm 2.53$ |
| Length OOC (+rev.) | $88.17 \pm 1.65$ |

Table 7: **CompCIFAR.** Normalized $\ell_1$ (lower is better), mean $\pm$ s.d. Below, a level-4 length-OOC example. $Z^{(0)}$ is the programmer proposal and $Z^*$ the program after search. The chain shows decoded intermediate states after each neural language token; post-hoc token interpretations approximately match the ground-truth operations.

| Method | Test I/O | Recomb. | Length |
|---|---|---|---|
| **NeuLP**$^\dagger$ | $\mathbf{0.012 \pm 0.001}$ | $\mathbf{0.009 \pm 0.001}$ | $\mathbf{0.012 \pm 0.001}$ |
| NEO | $0.138 \pm 0.002$ | $0.113 \pm 0.004$ | $0.163 \pm 0.009$ |
| NEO-S | – | $0.113 \pm 0.004$ | $0.162 \pm 0.010$ |




$$Z^{(0)} = [e_{140}, e_{140}, e_{24}] \xrightarrow{\text{search } +e_{351}} Z^* = [e_{140}, e_{140}, e_{24}, e_{351}], \qquad \ell_1(Z^*) = 0.01542.$$

**RQ1: Out-of-composition generalization.** Table 1 shows that NeuLP is nearly exact on Comp-II and gives the best OOC results among the compared methods on both CompARC and CompLife, with especially large gains at greater compositional depth. On CompLife length OOC, NeuLP reaches $74.0 \pm 7.1\%$, compared with $3.2 \pm 1.3\%$ for NLI, the strongest baseline on this split. On CompCIFAR, NeuLP achieves more than an order-of-magnitude lower normalized $\ell_1$ than NEO-S on both OOC splits (Table 7).

**RQ2: Evidence for the design choices.** Tables 2–4 report controlled Comp-II ablations. The finite codebook (Section 2.1) substantially outperforms **Continuous**, which replaces token selection with a free continuous vector at each program position. The program-space self-curriculum (Section 2.1) also outperforms **All slots**, which exposes the full program horizon from the start. Both comparisons use Hard Test I/O: exact accuracy after taking the argmax token at each program slot. For the language prior, BEEP outperforms NLI reuse (Macfarlane et al., 2026) and NEO MDL (Baek et al., 2026), while maintaining near-perfect Hard Test I/O, the highest language closure (CL.), and the best length-OOC result under a $32 \times 30$ search budget. Language closure is the fraction of training Lv1–3 functions that are correctly reproduced by composing stable learned tokens within max. program length. Finally, directional GCU (Section 2.2) outperforms two search baselines: **Rnd.** chooses the edited slot at random, while **Raw** chooses the slot with the largest gradient norm; both then enumerate legal token replacements at that slot. Full protocols are in Appendices C.3 and C.4.

**RQ3: Learning from discovered programs.** On the original CompARC Stage-2 supports, Stage 3 improves zero-shot accuracy on both OOC splits (Table 5), including length OOC from $0.83 \pm 0.24\%$ to $48.33 \pm 1.89\%$. In the 128-start proposal setting, length accuracy increases from $9.38 \pm 4.42\%$ to $62.50 \pm 1.47\%$, giving search a better starting point on these same supports.

**RQ4: Continual language growth.** We first learn a seven-operation Comp-II language through a complete Stage 1–3 cycle. In the next cycle, the data stream mixes the original data with examples

requiring an eighth operation, `reverse`. During Stage 1 of this second cycle, protected codebook rows from the preceding validated bank are fixed, unprotected rows remain trainable, and old bank records are replayed during acquisition (above). Table 6 shows that NeuLP retains the original language while reaching high accuracy on both Test I/O and OOC programs containing `reverse`. Details are given in Appendix C.5.

## 4 Related Work

**Program synthesis and neural modules.** Classical inductive synthesis searches programs in a specified DSL (Gulwani et al., 2017). Neural module networks and HOUDINI learn reusable neural components, but compose them through supplied layouts, languages, or typed operators (Andreas et al., 2016; Valkov et al., 2018). Differentiable interpreters such as TERPRET propagate gradients through relaxed program execution (Gaunt et al., 2016), while DreamCoder and LILO learn symbolic library abstractions over a seed language (Ellis et al., 2021; Grand et al., 2024). NeuLP instead learns both the discrete vocabulary and its neural execution semantics from input-output data.

**Compositional and latent-program baselines.** Episode Transformer uses episodic meta-learning for compositional generalization (Lake & Baroni, 2023; Mondorf et al., 2026). LPN searches continuous latent programs (Macfarlane & Bonnet, 2025); NLI learns a discrete neural language and differentiable interpreter but searches a continuous relaxation (Macfarlane et al., 2026); NEO learns recurrent executable traces, while NEO-S selects sampled proposals (Baek et al., 2026). NeuLP instead keeps a finite-token program interface throughout learning and search, separating programmer and interpreter.

**Recurrent reasoning.** HRM and TRM obtain additional computation through recurrent continuous states (Wang et al., 2025a; Jolicoeur-Martineau, 2025). NeuLP instead exposes the intermediate computation as a finite sequence of reusable token choices. This separation makes programs directly editable, storable, and replayable. Extended comparisons to language priors, curricula, latent reasoning, and continual learning are given in Appendix A.

## 5 Limitations and Conclusion

**Limitations.** NeuLP currently represents programs as bounded straight-line sequences of token applications, without explicit memory or control flow. A natural extension is to add memory for storing and reusing intermediate values, enabling more general dataflow. Learned control flow could further support branching, iteration, and recursion.

Secondly, input-output supervision only identifies executable behavior, but not a unique human-defined decomposition into primitives. Learned tokens may therefore represent primitives, macros, or behaviorally equivalent aliases. At the same time, the explicit discrete language provides a useful interface for semantic auditing and intervention: token executions and compositions can be inspected, and the vocabulary, grammar, or language prior can be constrained when undesirable behaviors are identified. Reliable semantic control remains an important direction.

**Conclusion.** We study OOC generalization when both the program and the language used to express it must be learned from input-output data. NeuLP separates three roles that are usually entangled in a neural model: the programmer proposes programs, the neural language stores reusable operations, and the differentiable interpreter executes them. NeuLP supports both gradient-based language learning and discrete search, validation, storage, and reuse.

Across four domains with distinct modalities, NeuLP generalizes to unseen recombinations and greater compositional depth that remain difficult for the compared neural and learned-program baselines. The same representation also supports learning beyond a single prediction: programs discovered by search can become new supervision for the programmer, while the language itself can be extended with a new operation without discarding previously learned ones. Together, these results support a view of compositional knowledge as an executable learned language, with programming, representation, and execution handled by distinct components.

## AI USE STATEMENT

Large language models were used to assist with iterative editing, phrasing, and polishing of author-written manuscript text, and AI tools assisted with baseline code development, data analysis, plotting, and figure preparation. All generated text and visual outputs were reviewed and edited by the authors.

## REPRODUCIBILITY STATEMENT

An anonymized implementation and scripts for reproducing the reported experiments are prepared. We will release them publicly upon acceptance and can provide them to reviewers upon request during the review period.

## REFERENCES

Ekin Akyürek, Afra Feyza Akyürek, and Jacob Andreas. Learning to recombine and resample data for compositional generalization. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=PS3IMnScugk.

Jacob Andreas. Good-enough compositional data augmentation. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7556–7566, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.676. URL https://aclanthology.org/2020.acl-main.676/.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 39–48, 2016.

Doojin Baek, Gyubin Lee, Junyeob Baek, Hosung Lee, and Sungjin Ahn. Learning to theorize the world from observation. In *International Conference on Machine Learning*, 2026.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 41–48. ACM, 2009. doi: 10.1145/1553374.1553380. URL https://ronan.collobert.com/pub/2009_curriculum_icml.pdf.

Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016. doi: 10.1073/pnas.1517384113.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019. URL https://arxiv.org/abs/1911.01547.

Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. *arXiv preprint arXiv:2305.18654*, 2023. URL https://arxiv.org/abs/2305.18654.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 31–36, 2018.

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*, pp. 835–850, 2021.

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. TerpreT: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 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 *International Conference on Machine Learning*, 2023.

Gabriel Grand, Lio Wong, Maddy Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. In *International Conference on Learning Representations*, volume 2024, pp. 30399–30446, 2024.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1311–1320. PMLR, 2017. URL https://proceedings.mlr.press/v70/graves17a.html.

Peter D Grünwald. *The minimum description length principle*. The MIT press, 2007.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2):1–119, 2017.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason E Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Itxz7S4Ip3.

Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. In *Advances in Neural Information Processing Systems*, volume 19, pp. 641–648. MIT Press, 2006. URL https://proceedings.neurips.cc/paper_files/paper/2006/hash/62f91ce9b820a491ee78c108636db089-Abstract.html.

Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. Bayesian inference for PCFGs via Markov Chain Monte Carlo. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pp. 139–146. Association for Computational Linguistics, 2007. URL https://aclanthology.org/N07-1018/.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *International Conference on Learning Representations*, 2020. URL https://arxiv.org/abs/1912.09713.

Ruhma Khan, Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ashish Tiwari, and Gust Verbruggen. Llm-guided compositional program synthesis, 2025. URL https://arxiv.org/abs/2503.15540.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

M. Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems*, volume 23, pp. 1189–1197, 2010.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2873–2882, 2018.

Brenden M Lake and Marco Baroni. Human-like systematic generalization through a meta-learning neural network. *Nature*, 623(7985):115–121, 2023.

Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. doi: 10.1126/science.aab3050.

Leonid Anatolevich Levin. Laws of information conservation (nongrowth) and aspects of the foundation of probability theory. *Problemy Peredachi Informatsii*, 10(3):30–35, 1974.

Wenhao Li, Yudong Xu, Elias Boutros Khalil, and Scott Sanner. Structure-aware graph hypernetworks for neural program synthesis. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=x7zOzUwtR7.

Zhizhong Li and Derek Hoiem. Learning without forgetting. In *Computer Vision – ECCV 2016*, pp. 614–629. Springer, 2016. URL https://arxiv.org/abs/1606.09282.

Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 639–646, 2010.

LifeWiki contributors. Rulestring and list of life-like rules. https://conwaylife.com/wiki/Rulestring; https://conwaylife.com/wiki/List_of_Life-like_cellular_automata, n.d. Accessed September 10, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Matthew V. Macfarlane and Clément Bonnet. Searching latent program spaces. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3457.

Matthew V. Macfarlane, Clément Bonnet, Herke van Hoof, and Levi H. S. Lelis. Gradient-based program synthesis with neurally interpreted languages. In *International Conference on Learning Representations*, 2026.

Arun Mallya and Svetlana Lazebnik. PackNet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7765–7773, 2018. URL https://arxiv.org/abs/1711.05769.

Philipp Mondorf, Shijia Zhou, Monica Riedler, and Barbara Plank. Compositional-arc: Assessing systematic generalization in abstract spatial reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 108708–108744, 2026.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 1310–1318, 2013.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In *Advances in Neural Information Processing Systems*, 2019.

Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1): 1–22, 1964.

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. *Advances in neural information processing systems*, 31, 2018.

Paul MB Vitányi and Ming Li. Minimum description length induction, bayesianism, and kolmogorov complexity. *IEEE Transactions on information theory*, 46(2):446–464, 2000.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025a.

Zixuan Wang, Eshaan Nichani, Alberto Bietti, Alex Damian, Daniel Hsu, Jason D. Lee, and Denny Wu. Learning compositional functions with transformers from easy-to-hard data. In *Proceedings of the Thirty Eighth Conference on Learning Theory*, volume 291 of *Proceedings of Machine Learning Research*, pp. 5632–5711. PMLR, 2025b. URL https://proceedings.mlr.press/v291/wang25a.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E Weston. Self-rewarding language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 57905–57923. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/yuan24d.html.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

# A Extended Related Work

**Program synthesis and neural modules.** Classical program synthesis assumes an executable DSL with specified syntax and semantics (Gulwani et al., 2017). Neural module networks and HOUDINI learn neural functions while retaining supplied layouts, types, or composition rules (Andreas et al., 2016; Valkov et al., 2018). TERPRET differentiates through symbolic program choices (Gaunt et al., 2016); DreamCoder and LILO learn reusable abstractions over an existing symbolic language (Ellis et al., 2021; Grand et al., 2024). NeuLP instead learns a discrete neural language and its execution semantics from input-output data.

A general-purpose language such as Python supplies syntax, composition, and executable libraries, but not the reusable operations of a learned dynamical or world model (Ha & Schmidhuber, 2018; Brunton et al., 2016; Cranmer et al., 2020; Li et al., 2026). NeuLP learns those operations from input-output data, including continuous visual states in CompCIFAR.

**Latent programs and learned executable languages.** LPN optimizes continuous latent programs through execution gradients (Macfarlane & Bonnet, 2025). NLI learns a discrete neural language and differentiable interpreter, then searches a continuous relaxation of the program at inference (Macfarlane et al., 2026). NEO learns recurrent state-conditioned executable traces, and NEO-S samples complete programs for execution-based selection (Baek et al., 2026). NeuLP builds on the learned neural-language setting but keeps the searched program discrete. Execution gradients rank legal token edits, and validated programs can later supervise the programmer.

**Program-space self-curriculum.** Curriculum learning usually changes the examples or tasks presented during training (Bengio et al., 2009; Graves et al., 2017). Compositional methods also recombine or resample training examples (Andreas, 2020; Akyürek et al., 2021). In LLM-based PBE, Khan et al. (2025) decompose a failed program into simpler synthesis subproblems. NeuLP instead keeps the training stream fixed and expands the available program length from one active slot to all $T$ slots. Easier compositions can also help when easy and hard examples are mixed (Wang et al., 2025b).

**Program priors.** Probabilistic grammar induction assigns probabilities to productions in a fixed symbolic language (Johnson et al., 2007); adaptor grammars and DreamCoder additionally model reuse of fragments or library elements (Johnson et al., 2006; Ellis et al., 2021). NLI's reuse objective and NEO's prefix MDL provide the closest learned-program comparisons. BEEP learns token-specific production costs from execution evidence while the token semantics are being learned, and combines these costs with the sequential grammar's termination cost. Appendix C.2 derives the connection to NLI and NEO.

**Language models and latent reasoning.** Language-model agents can generate programs, call Python or other tools, and use execution feedback (Yao et al., 2023; Zhang et al., 2025). They typically act through a supplied language or tool interface, and tool execution lies outside the model. Credit therefore comes from search, sampling, or reinforcement learning rather than gradients through the executed program.

When an exact verifier is available, execution can be checked directly. Otherwise, training may use human preferences, process labels, or learned reward models (Ouyang et al., 2022; Lightman et al., 2024); self-rewarding methods instead generate reward signals with the language model (Yuan et al., 2024). NeuLP uses the PBE output itself: it compares the predicted output $\hat{y}$ with the observed output $y$ and differentiates the execution loss through the interpreter, program choices, and learned language.

NeuLP also acts directly on the domain state, including continuous images in CompCIFAR, rather than requiring observations to be expressed through a general-purpose programming interface. Continuous latent reasoning provides a different form of iterative computation without an explicit program interface (Hao et al., 2025).

**Recurrent reasoning.** Universal and looped Transformers, HRM, and TRM increase computation through recurrent hidden states (Dehghani et al., 2019; Giannou et al., 2023; Wang et al., 2025a; Jolicoeur-Martineau, 2025). NeuLP also uses recurrent execution, but exposes the computation as a finite program between the programmer and interpreter. The program can therefore be searched, validated, stored, and reused independently of the recurrent execution state.

Table 8: Semantic task identities used in the benchmark. Training contains D1–D3 functions; recombination OOC holds out functions at trained depths; length OOC evaluates D4–D6 programs.

| | Train | | | Recombination OOC | | Length OOC | | |
|---|---|---|---|---|---|---|---|---|
| Domain | D1 | D2 | D3 | D2 | D3 | D4 | D5 | D6 |
| Comp-II | 8 | 33 | 137 | 4 | 17 | 500 | 500 | 500 |
| CompARC | 8 | 31 | 64 | 6 | 16 | 100 | 100 | 100 |
| CompCIFAR | 8 | 24 | 32 | 6 | 12 | 60 | 60 | 60 |
| CompLife | 6 | 32 | 136 | 4 | 34 | 100 | 100 | 100 |

**Continual learning and language growth.** Replay, distillation, and selective freezing are standard mechanisms for retaining previously learned behavior (Rolnick et al., 2019; Li & Hoiem, 2016; Mallya & Lazebnik, 2018). HOUDINI also grows a neural library while retaining previously learned functions (Valkov et al., 2018). In NeuLP, the codebook and interpreter jointly determine what a token executes. During language growth, validated codebook rows are protected, replay preserves their execution, and unused rows remain available for the new operation. This is the setting evaluated in RQ4.

# B Benchmark Construction and Dataset Details

## B.1 Shared programming-by-example protocol

Each benchmark task is defined by a hidden straight-line program

$$Z = (z_1, \ldots, z_L), \qquad f_Z(x) = f_{z_L} \circ \cdots \circ f_{z_1}(x), \qquad 1 \le L \le 6, \tag{11}$$

where each $z_t$ selects one primitive from a finite domain-specific DSL. We refer to a length-$L$ program as D$L$. The state space of each domain is closed under its primitives, so the output of one step is a valid input to the next.

A model-facing episode contains four support examples and one query,

$$\mathcal{U} = \{(x_i, y_i)\}_{i=1}^{4}, \qquad x_q \mapsto y_q = f_Z(x_q). \tag{12}$$

The hidden program, its length, and all intermediate execution states are used only to construct and audit the benchmark. They are never provided to the model.

**Semantic task identity.** Splits are defined over executed functions rather than token strings. Programs that implement the same transformation are treated as one task. Functional equivalence is exact for Comp-II. For CompARC, CompCIFAR, and CompLife, programs are merged when they agree on a fixed deterministic probe set; task identity in these domains is therefore defined on that probe set.

**Train, Test I/O, and OOC splits.** Training contains D1–D3 tasks. Train-validation and Test I/O episodes use the same hidden functions with new, disjoint inputs. Recombination OOC contains held-out D2 and D3 functions at training depths, while length OOC contains D4–D6 functions and therefore extends the execution horizon beyond anything seen in training.

For the three grid/image domains, the recombination split also constrains the program trace. A held-out D2 pair does not occur as a contiguous pair in a selected training D3 trace. For a held-out D3 trace $(a, b, c)$, both adjacent D2 chunks $(a, b)$ and $(b, c)$ are train-seen, while the complete D3 function is held out. Thus the D3 split tests a new composition of locally familiar subcompositions rather than the introduction of a new primitive. Comp-II is partitioned directly by exact function identity, as described in Appendix B.2.

Table 8 gives the fixed allocation of semantic task identities. All D1 primitives appear in training in every domain. After semantic filtering, the D1–D3 selections in CompARC, CompCIFAR, and CompLife are balanced over primitive identity and token position.

**Episode generation.** CompARC, CompCIFAR, and CompLife render 500 training episodes per identity. Comp-II instead fixes a batch-aligned corpus of 499,712 episodes sampled uniformly over

Table 9: Number of model-facing PBE episodes. Every episode contains four support pairs and one query.

| Domain | Train | Train val. | Test I/O | Recomb. OOC | Length OOC |
|---|---|---|---|---|---|
| Comp-II | 499,712 | 1,780 | 1,780 | 2,100 | 1,500 |
| CompARC | 51,500 | 1,030 | 1,030 | 2,200 | 30,000 |
| CompCIFAR | 32,000 | 640 | 640 | 1,800 | 18,000 |
| CompLife | 87,000 | 1,740 | 1,740 | 3,800 | 30,000 |

its 178 training identities. Train-validation and Test I/O contain ten fresh episodes per training identity, and recombination contains 100 episodes per held-out identity. The three visual domains also use 100 episodes per length-OOC identity. Comp-II samples 500 exact semantic identities at each of D4–D6 and renders one episode per identity. Table 9 summarizes the resulting model-facing banks.

Within an episode, all five inputs are distinct. Candidate examples are executed through the complete hidden trace and filtered by the domain-specific validity criteria below. The four support examples must collectively expose every hidden step: at each program position, at least one support execution must exceed the domain's activity threshold. The hidden trace is used only for this dataset-construction check.

We verify that semantic identities do not cross splits, model-facing records contain no trace information, and training inputs do not reappear in the train-validation or Test I/O views. Split construction and rendering are deterministic.

**Evaluation aggregation.** Metrics are first averaged over episodes belonging to the same hidden task and then over the declared task identities, giving each semantic identity equal weight within a split. Zero-shot decoding and test-time search are always reported separately.

### B.2 COMP-II: PERMUTATIONS

**State space and DSL.** An input is a permutation of the integers $0, \ldots, 9$. Each primitive is a fixed permutation of positions, so every intermediate state remains in the same space. Table 10 lists the eight primitives. The one-step examples use the identity permutation as input so that the positional action of each primitive is explicit.

**Exact task identities.** For this DSL, applying a program to the identity permutation $[0, 1, \ldots, 9]$ exactly identifies the induced position permutation. We therefore enumerate all token strings through D6, remove the identity transformation, merge syntactic aliases, and assign each function its minimum program depth. This gives 8, 41, 171, 708, 2,896, and 11,460 distinct functions at minimum depths D1 through D6, respectively.

Training contains all D1 functions and approximately $80\%$ of the D2 and D3 functions. Recombination uses fixed subsets of the remaining D2 and D3 functions, and length OOC samples 500 exact minimum-depth functions at each of D4, D5, and D6. Because task identity is exact here, the Comp-II split does not impose the additional trace-chunk constraint used by the other domains.

For the 17 held-out D3 functions, a chosen minimum trace has both adjacent D2 chunks train-seen for 12 tasks, one train-seen chunk for three tasks, and neither for two tasks. Although one nondegenerate I/O pair is sufficient to identify a position permutation, we retain four supports to keep the PBE interface identical across domains.

**Input generation.** Inputs are sampled uniformly from the $10!$ possible permutations. Training, train-validation, and Test I/O use deterministic disjoint partitions of this input space; the two OOC splits use independent deterministic samples. Since every input contains distinct values, any non-identity position permutation changes the input, and all intermediate states are valid permutations.

**Metric.** The primary metric is exact query-permutation accuracy. Position-wise accuracy is retained only as a diagnostic.

Table 10: Comp-II primitive language. The same identity permutation is used as the input in every one-step example.

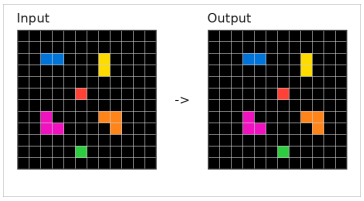
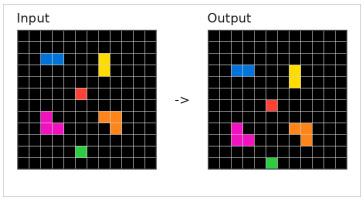
| Token | Transformation | One-step I/O |
|---|---|---|
| `shift_1` | Circular left shift by one position. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [1,2,3,4,5,6,7,8,9,0]` |
| `shift_2` | Circular left shift by two positions. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [2,3,4,5,6,7,8,9,0,1]` |
| `shift_3` | Circular left shift by three positions. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [3,4,5,6,7,8,9,0,1,2]` |
| `reverse` | Reverse all ten positions. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [9,8,7,6,5,4,3,2,1,0]` |
| `interleave` | Split into two length-five halves and alternate their elements. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [0,5,1,6,2,7,3,8,4,9]` |
| `deinterleave` | Collect even and odd positions into two halves, inverting `interleave`. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [0,2,4,6,8,1,3,5,7,9]` |
| `swap_adjacent` | Swap positions $(0,1),(2,3),\ldots,(8,9)$. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [1,0,3,2,5,4,7,6,9,8]` |
| `swap_halves` | Exchange the first and second length-five halves. | `in: [0,1,2,3,4,5,6,7,8,9]`<br>`out: [5,6,7,8,9,0,1,2,3,4]` |

### B.3 CompARC: categorical grid editing

**State space and DSL.** CompARC is a programmatically generated grid-editing domain using $12 \times 12$ categorical grids and the standard ten-color ARC palette (Chollet, 2019); it is not a subset of the original ARC tasks. Color 0 is background, connected components use four-neighbor connectivity, and all primitives map a grid back to the same state space.

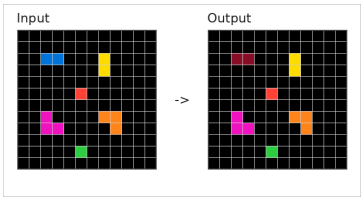
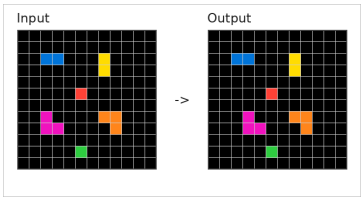
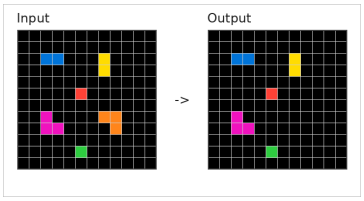
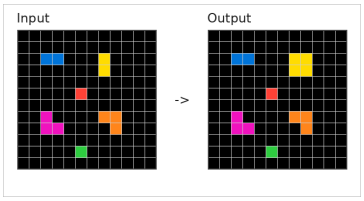
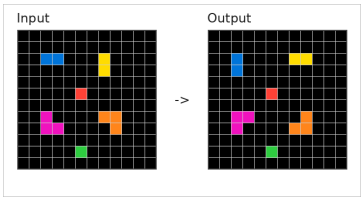
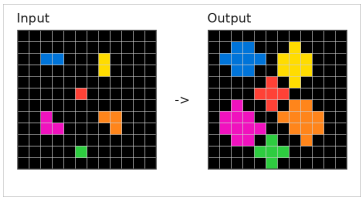
Geometric operations use strict execution. A candidate scene is rejected if translation, copying, rotation, or dilation would leave the grid or collide with a different object. This avoids introducing clipping or overwrite rules that are not represented by the program token itself.

Table 11: CompARC primitive language. Each row shows the primitive applied once to a shared multi-object witness scene.

| Token | Transformation | One-step I/O |
|---|---|---|
| `tx+1` | Translate all foreground cells one column to the right. |  |
| `ty+1` | Translate all foreground cells one row downward. |  |

*Continued on next page.*

*CompARC primitive language (continued).*

| Token | Transformation | One-step I/O |
|---|---|---|
| `recolor1to9` | Replace every color-1 cell by color 9. |   |
| `move6_+1_+0` | Move only the color-6 object one column to the right. |   |
| `delete7` | Replace every color-7 cell by background. |   |
| `copy4_+1_+0` | Copy the color-4 object one column to the right while retaining the source. |   |
| `rotate_cw` | Rotate each connected component $90^\circ$ clockwise about its bounding-box origin. |   |
| `dilate1` | Apply one-cell Manhattan dilation while preserving component color. |   |

**Task construction.** Program identity is defined by execution on 48 deterministic multi-object probe grids. We enumerate all D1–D3 token strings and retain a candidate only if at least $20\%$ of probes admit strict execution, each hidden step changes at least $0.5\%$ of cells on at least $10\%$ of valid probes, and the mean final changed-cell fraction lies in $[0.5\%, 75\%]$. Programs with identical probe outputs are merged. After filtering, the D1–D3 semantic pool contains 8, 37, and 129 functions, respectively.

The train/recombination allocation is chosen subject to the trace constraints in Appendix B.1: held-out D2 pairs do not appear as adjacent chunks of selected training D3 traces, whereas both D2 chunks of each held-out D3 trace are train-seen. D4–D6 candidates pass the same activity gates and

must have a fingerprint not already present in the selected bank. For D4–D6, depth is defined by the selected probe bank rather than by a global minimum-depth proof.

**Input generation.** Each base scene contains four separated objects with colors 1, 4, 6, and 7. Their small shapes are independently rotated and placed with a one-cell empty margin so that both global and color-specific operations can be observed. Two isolated distractor cells are added with colors sampled from $\{2, 3, 5, 8\}$.

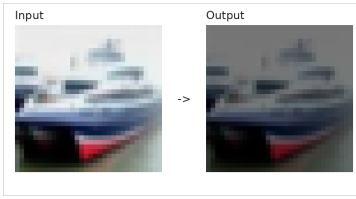
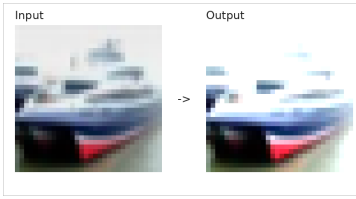
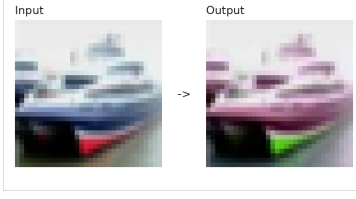
A candidate example is rejected if any strict operation fails, if the final grid is all background or all foreground, or if the final changed-cell fraction falls outside $[0.5\%, 75\%]$. Support examples must also collectively witness every program step. Train, train-validation, and Test I/O use disjoint rendered inputs.

**Metric.** Exact query-grid accuracy is primary; cell accuracy is retained only as a diagnostic.

### B.4 CompCIFAR: natural-image editing

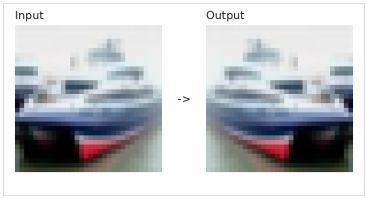
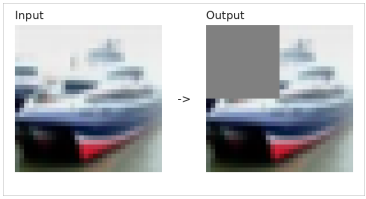
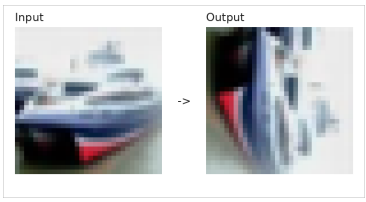
**State space and DSL.** CompCIFAR uses $32 \times 32$ CIFAR-10 RGB images (Krizhevsky, 2009) stored as lossless uint8 arrays. CIFAR labels are never used. The primitive edits follow the compositional image transformations used by NEO (Baek et al., 2026). Program order is part of the task: for example, masking before a rotation is generally different from masking after it.

Table 12: CompCIFAR primitive language. Each primitive is applied once to the same CIFAR-10 image.

| Token | Transformation | One-step I/O |
|---|---|---|
| `brightness_minus` | Multiply RGB intensities by $0.5$ and round to uint8. |   |
| `brightness_plus` | Multiply RGB intensities by $1.5$, clip to $[0, 255]$, and round to uint8. |   |
| `hue_minus` | Convert to HSV, rotate hue by $-0.3$ turns, and convert back to uint8 RGB. |   |
| `hue_plus` | Convert to HSV, rotate hue by $+0.3$ turns, and convert back to uint8 RGB. |   |

*Continued on next page.*

*CompCIFAR primitive language (continued).*

| Token | Transformation | One-step I/O |
|---|---|---|
| `horizontal_flip` | Reverse the image columns. |   |
| `mask` | Replace the top-left quadrant by gray $(128, 128, 128)$. |   |
| `rotation_cw` | Rotate the image $90^\circ$ clockwise. |   |
| `vertical_flip` | Reverse the image rows. |   |

**Task construction.** Semantic fingerprints are computed on 24 deterministic synthetic images, separately from the CIFAR images used to render PBE episodes. Each probe is constructed from a random $8 \times 8$ RGB block image, expanded to $32 \times 32$ by repeating each block four times in each direction, with fixed row and column color ramps added to make geometric and photometric edits distinguishable.

We enumerate D1–D3 programs. A candidate is retained when every hidden step changes the image by normalized $\ell_1$ distance at least $0.005$ on at least half of the probes and the mean final normalized distance from the input is at least $0.01$. Probe-equivalent traces are merged. Recombination uses the same D2/D3 trace constraints as CompARC. D4–D6 candidates pass the same activity gates and must have a new fingerprint within the selected bank. Their depth is therefore defined on the probe bank.

**Image partitions and episode generation.** The rendered views use disjoint CIFAR-10 source partitions:

| View | CIFAR-10 source indices |
|---|---|
| Train | training images 0:40,000 |
| Train val. | training images 40,000:50,000 |
| Test I/O | test images 0:4,000 |
| Recombination OOC | test images 4,000:7,000 |
| Length OOC | test images 7,000:10,000 |

Duplicate source images are not allowed within an episode. A rendered pair is accepted only if the complete composition changes the image by normalized $\ell_1$ distance at least $0.01$. Across the four

supports, every hidden edit must produce normalized change at least $0.005$ on at least one image. Pixel values remain uint8 throughout dataset serialization.

**Metrics.** For RGB images scaled to $[0, 1]$, normalized pixel $\ell_1$ is

$$d_1(\widehat{y}, y) = \frac{1}{3HW} \sum_{h,w,r} |\mathrm{clip}(\widehat{y}_{hwr}, 0, 1) - y_{hwr}|, \qquad H = W = 32. \tag{13}$$

Thus the metric averages absolute error over all $3HW = 3{,}072$ RGB values. For 8-bit predictions and targets, the equivalent value is the summed absolute difference divided by $255 \cdot 3HW$; it is not normalized by the input-copy error. An $\ell_1$ value of $0.01$ therefore corresponds to a mean absolute error of $2.55$ intensity levels. Exact uint8 match is retained as a strict diagnostic.

### B.5 CompLife: compositions of cellular-automaton rules

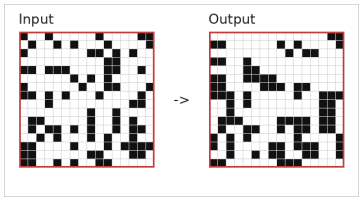
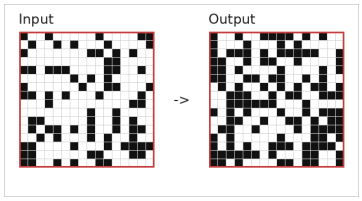
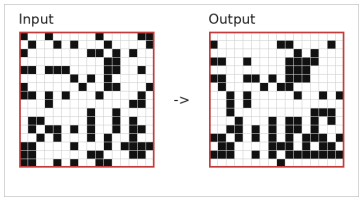
**State space and boundary condition.** A CompLife state is a $16 \times 16$ binary grid, and initial live cells may occupy the full viewport. Every primitive updates the complete grid synchronously, treating cells outside the stored window as dead, and crops the result back to $16 \times 16$ after each transition. There is no toroidal wraparound. Activity that leaves the viewport cannot influence a later step; the benchmark therefore models a clipped finite view rather than an exact crop of an infinite world.

**Primitive rules.** Let $x_{r,c} \in \{0, 1\}$ and let $n_{r,c}$ denote the number of live cells in its eight-cell Moore neighborhood. For a birth/survival rule $\mathrm{B}B/\mathrm{S}S$,

$$x'_{r,c} = \begin{cases} 1, & x_{r,c} = 0 \text{ and } n_{r,c} \in B, \\ 1, & x_{r,c} = 1 \text{ and } n_{r,c} \in S, \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

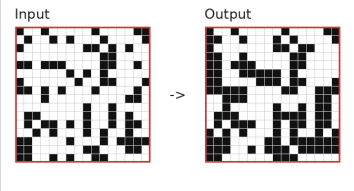
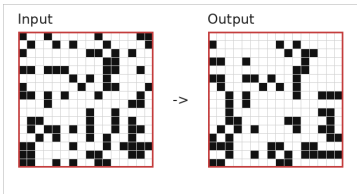
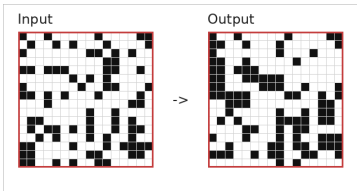
One program token applies one complete synchronous update. We use six established Life-like rules using standard rulestring notation (LifeWiki contributors, n.d.).

Table 13: CompLife primitive language. Each row applies one synchronous cellular-automaton update to the same witness state.

| Token | Birth | Survival | One-step I/O |
|---|---|---|---|
| `life` `B3/S23` | Exactly 3 live neighbors. | Exactly 2 or 3 live neighbors. |   |
| `replicator` `B1357/S1357` | 1, 3, 5, or 7 live neighbors. | 1, 3, 5, or 7 live neighbors. |   |
| `assimilation` `B345/S4567` | 3, 4, or 5 live neighbors. | 4, 5, 6, or 7 live neighbors. |   |

*Continued on next page.*

*CompLife primitive language (continued).*

| Token | Birth | Survival | One-step I/O |
|---|---|---|---|
| `maze`<br>`B3/S12345` | Exactly 3 live neighbors. | Between 1 and 5 live neighbors. |  |
| `daynight`<br>`B3678/S34678` | 3, 6, 7, or 8 live neighbors. | 3, 4, 6, 7, or 8 live neighbors. |  |
| `2x2`<br>`B36/S125` | 3 or 6 live neighbors. | 1, 2, or 5 live neighbors. |  |

The six-rule vocabulary was fixed before model training. We selected it from 19 established Life-like rules using probe-based distinguishability, noncommutativity, and nondegeneracy through D6. Model performance was not used for this selection.

**Vocabulary-size audit.** Table 14 reports the pretraining screen over four vocabulary sizes. Pairwise distinguishability is the fraction of probes on which two primitive rules produce different outputs. Pairwise noncommutativity is the fraction on which their two execution orders differ. D6 admissibility is the fraction of sampled six-step programs that retain visible intermediate activity, avoid degenerate terminal states, and remain behaviorally distinct from evaluated shallower programs. The six-rule set is the elbow: it preserves strong worst-pair separation and order sensitivity while providing a substantially larger compositional space than the four-rule set. Adding the seventh through twelfth rules weakens the worst pair and makes valid D6 programs less common.

Table 14: CompLife vocabulary-size audit over five deterministic probe-bank seeds. Distinguishability and noncommutativity report mean/minimum pairwise rates. D6 columns report mean $\pm$ population standard deviation over seeds.

| Rules | Distinguishability | Noncommutativity | D6 admissible | D6 unique |
|---|---|---|---|---|
| 4 | $0.967/0.907$ | $0.939/0.852$ | $0.981 \pm 0.008$ | $1.000 \pm 0.000$ |
| 6 | $0.944/0.804$ | $0.913/0.798$ | $0.961 \pm 0.010$ | $1.000 \pm 0.000$ |
| 8 | $0.918/0.579$ | $0.877/0.623$ | $0.846 \pm 0.023$ | $1.000 \pm 0.000$ |
| 12 | $0.893/0.435$ | $0.854/0.491$ | $0.810 \pm 0.026$ | $1.000 \pm 0.000$ |

The uniqueness column measures collisions among 2,048 sampled D6 fingerprints on 48 probes. The six-rule choice is based on this finite benchmark-design screen.

**Task construction.** Programs are fingerprinted on 48 deterministic probes spanning random live-cell densities and structured motifs. D1–D3 strings are enumerated and probe-equivalent programs are merged before the training and recombination allocation.

For D4–D6, a candidate must pass the same activity checks and differ on the probe bank from every evaluated shallower program. Minimum depth is therefore defined on the probe bank.

**Input generation.** Initial activity may occupy the full $16 \times 16$ viewport. Three quarters of candidate states are Bernoulli grids with density sampled from

$$\{0.06, 0.10, 0.16, 0.24, 0.34, 0.46, 0.60, 0.76\}.$$

The remaining candidates are constructed from one to six small motifs, including line segments, a $2 \times 2$ block, a glider-like pattern, and denser $3 \times 3$ patterns. Six additional randomly placed live cells break exact repetitions, and a subset of structured states is complemented within the viewport to cover high-density behavior.

A rendered pair is accepted when the final changed-cell fraction lies in $[1\%, 90\%]$ and the output is neither all dead nor all live. Across the four supports, every hidden step must change at least $0.5\%$ of one support grid and have mean support change at least $0.5\%$. These criteria remove silent rules and degenerate trajectories without conditioning on the query output.

**Four-example identifiability check.** We separately checked whether four support pairs determine the query behavior within the benchmark program class. For an episode $\mathcal{U}$ and candidate set $\mathcal{Z}$, define

$$\mathcal{C}(\mathcal{U}) = \{Z' \in \mathcal{Z} : f_{Z'}(x_i) = y_i \text{ for every } (x_i, y_i) \in \mathcal{U}\}. \tag{15}$$

An episode is query-ambiguous if some $Z' \in \mathcal{C}(\mathcal{U})$ disagrees with the hidden program on $x_q$.

Across the 512 selected CompLife task identities, no audited episode admitted a second selected program that matched all four supports but disagreed on the query. A second audit of 34 tasks against all 55,986 D1–D6 strings in the six-rule language found the same result.

**Metric.** Exact query-grid accuracy is primary. Cell accuracy is retained only as a per-depth diagnostic.

# C Methodological Details and Reproducibility

This appendix gives the implementation details for the reported NeuLP experiments. Stage 1 learns the language, programmer, and interpreter from PBE episodes. Stage 2 freezes them and collects execution-validated programs by search and dreaming. Stage 3 keeps the language and interpreter fixed and trains the programmer on the collected programs while retaining its Stage 1 predictions. Unless stated otherwise, programs have maximum length $T = 6$, token 0 is SKIP, and the other $K - 1 = 511$ token IDs index learned operations.

## C.1 Evaluation and data use

Hidden programs, generating lengths, and symbolic primitive names are used only for dataset construction and post-hoc analysis. Stage 1 uses the training episodes for optimization and the held-out train-validation split for checkpoint selection. Stage 2 search is conditioned only on the observed support PBE; final query targets never rank or accept search edits.

Stage 2 uses different support pools across domains. Comp-II uses acquisition PBEs that are separate from the final reporting episodes. CompARC and CompLife may reuse final-evaluation supports, but not their held-out query targets. CompCIFAR also stores support-only search records. Stage 3 may use replay-bank statistics for checkpoint selection; final query targets are not used for optimization or search.

Unless a table explicitly reports pooled counts, results are the mean and sample standard deviation over independent model seeds. Accuracy is macro-averaged over semantic task identities within each seed, and seeds are weighted equally.

## C.2 BEEP, NLI reuse, and NEO MDL

The main text compares NLI reuse, NEO MDL, and BEEP through three aspects of a program prior: which tokens are used, how individual token uses are weighted, and how program length is penalized. The methods make different choices for these three terms. We first give the BEEP objective and counting rule, then relate NLI and NEO to the same coding view.

**BEEP.** BEEP keeps the full finite codebook available during Stage 1 and uses execution-validated hard programs to estimate how often each non-SKIP token is reused. With a symmetric Dirichlet prior,

$$\theta_k = \frac{n_k + \alpha}{\sum_{j>0}(n_j + \alpha)}, \qquad \bar{c}_k = \frac{-\log \theta_k}{\frac{1}{K-1}\sum_{j=1}^{K-1} -\log \theta_j}, \qquad \bar{c}_0 = 0, \qquad \alpha = 0.5. \tag{16}$$

Here $n_k$ is the accumulated execution evidence for token $k$, and $\bar{c}_k$ is the normalized production cost used in the main text.

The counts use only admitted hard programs. If admitted program $i$ contains token $k$ with multiplicity $m_{ik}$ and has $L_i$ active tokens, we update

$$n_k \leftarrow n_k + \frac{m_{ik}}{L_i}. \tag{17}$$

Thus every admitted program contributes total count one, independent of its length. The counts and resulting costs are detached from the gradient.

For the categorical domains, a hard program is admitted only when it exactly reproduces the support outputs and training query. CompCIFAR instead uses the normalized $\ell_1$ improvement defined in Appendix B.4. For a hard program $\widehat{Z}$, let

$$g(x, y; \widehat{Z}) = d_1(x, y) - d_1\Big(F_\psi(x, \widehat{Z}, \mathcal{A}), y\Big). \tag{18}$$

The program is admitted when

$$a(\widehat{Z}) = \mathbf{1}\left[\frac{1}{M}\sum_{m=1}^{M} g(x_m, y_m; \widehat{Z}) \geq 10^{-3}\right] \mathbf{1}\Big[g(x_q, y_q; \widehat{Z}) \geq 10^{-3}\Big]. \tag{19}$$

Both the mean support improvement and the training-query improvement must pass the threshold. No evaluation-query label is used.

The sequential grammar terminates after each active production with probability $p_{\text{stop}}$, giving

$$p(L = \ell) \propto p_{\text{stop}}(1 - p_{\text{stop}})^{\ell}, \qquad 0 \leq \ell \leq T. \tag{20}$$

Hence the BEEP cost from the main text is

$$\mathcal{C}_{\text{BEEP}}(Z) = \sum_{t: z_t \neq 0} \bar{c}_{z_t} + L(Z) \log \frac{1}{1 - p_{\text{stop}}}. \tag{21}$$

Stage 1 minimizes the expectation of this cost under the programmer distribution. With $p_{\text{stop}} = 1/2$, each active production contributes an additional $\log 2$ length cost.

**NLI reuse.** NLI penalizes the number of distinct tokens used by a minibatch (Macfarlane et al., 2026). Let $a_k = 1$ when token $k$ appears at least once. With an independent Bernoulli prior on token inclusion,

$$a_k \sim \text{Bernoulli}(\pi) \quad \Longrightarrow \quad -\log p(\mathcal{V}) = \gamma \sum_{k=1}^{K-1} a_k + \text{const}. \tag{22}$$

For the factorized soft programmer, NLI uses the noisy-OR estimate

$$\widehat{a}_k(\mathcal{B}) = 1 - \prod_{b \in \mathcal{B}} \prod_{t=1}^{T} (1 - p_{btk}), \qquad \mathcal{L}_{\text{NLI}} = \mathcal{L}_{\text{exec}} + \lambda_{\text{reuse}} \sum_{k \in \mathcal{K}_{\text{NLI}}} \widehat{a}_k(\mathcal{B}). \tag{23}$$

Thus NLI charges for vocabulary support: once a token appears in the minibatch, additional uses of that token add no further cost. The estimate is local to the minibatch. The original implementation includes SKIP in $\mathcal{K}_{\text{NLI}}$.

**NEO MDL.** NEO assigns the same production cost to every token identity (Baek et al., 2026). For a fixed vocabulary,

$$-\log p(Z_{1:\ell} \mid L = \ell, \mathcal{V}) = \ell \log |\mathcal{V}|. \tag{24}$$

NEO selects a prefix using

$$\widehat{\ell} = \arg\min_{\ell} \lambda_{\mathrm{NEO}}^{\ell} \mathcal{L}_{\mathrm{exec}}(Z_{1:\ell}). \tag{25}$$

For positive execution loss, this is equivalent to minimizing

$$\log \mathcal{L}_{\mathrm{exec}}(Z_{1:\ell}) + \ell \log \lambda_{\mathrm{NEO}}. \tag{26}$$

NEO therefore applies the same additional cost at every prefix step, whereas BEEP assigns token-specific production costs from execution evidence.

Table 15: Comparison of the program priors used by NLI, NEO, and NeuLP.

| Method | Distinct-token cost | Production cost | Length cost | Program selection |
|---|---|---|---|---|
| NLI reuse | Minibatch distinct-token penalty | None | None | Soft programmer distribution |
| NEO MDL | None | Uniform across tokens | Prefix-length penalty | Best recurrent prefix |
| BEEP | None | Token-specific, from execution evidence | Geometric termination cost | Stage-1 programmer; Stage-2 discrete search |

NLI therefore regularizes how many distinct tokens are used, NEO assigns a uniform cost to each prefix step, and BEEP assigns token-specific production costs from execution evidence together with a geometric termination cost. Table 18 compares these objectives under the matched Comp-II setup; Table 19 gives the BEEP coefficient controls.

### C.3 Stage 1: language acquisition

#### C.3.1 Objective and estimators

Stage 1 optimizes

$$\mathcal{L}_{\mathrm{S1}} = \mathcal{L}_{\mathrm{exec}} + \lambda_{\mathrm{BEEP}} \mathcal{L}_{\mathrm{BEEP}} + \lambda_{\mathrm{state}} \mathcal{L}_{\mathrm{state}} + \lambda_{\mathrm{iso}} \mathcal{L}_{\mathrm{iso}}. \tag{27}$$

The reported model uses no library-size or soft–hard alignment term. The programmer receives four support pairs and does not see the query input or output. After the program is proposed, the executor receives the query input; the query output provides the training target.

For categorical domains, $\mathcal{L}_{\mathrm{exec}}$ is mean categorical negative log-likelihood. For CompCIFAR it combines the mean Charbonnier loss

$$\mathcal{L}_{\mathrm{charb}} = \frac{1}{D} \sum_{d=1}^{D} \sqrt{(y_d - \widehat{y}_d)^2 + 10^{-6}}. \tag{28}$$

with terminal patch-state matching,

$$\mathcal{L}_{\mathrm{exec}}^{\mathrm{CIFAR}} = \mathcal{L}_{\mathrm{charb}} + 0.25 \ \mathrm{MSE}\big(h_T, \mathrm{patch}_{2\times 2}(Y)\big). \tag{29}$$

PatchViT-2 represents RGB values by a lossless patch rearrangement, so the terminal patch target requires no learned codec. In the categorical domains, state grounding decodes and re-encodes each recursive latent state, stops the gradient through the projection, and penalizes its squared distance from the original state. The isotropy term regularizes the centered, normalized codebook covariance. Neither term uses primitive names, hidden programs, or hidden intermediate states.

If $p_{btk}$ is the programmer probability of token $k$ at slot $t$, the active-length statistic is

$$\mathcal{L}_{\mathrm{active}} = \frac{1}{B} \sum_{b=1}^{B} \sum_{t=1}^{T} (1 - p_{bt0}). \tag{30}$$

Following NEO's state-grounding principle (Baek et al., 2026), for a categorical transition state $s_{bt}$ let $D$ be the observable-state decoder, $R$ its embedding map, and $a_{bt} = 1 - p_{bt0}$ the probability that the slot is active. The selected grounding term is

$$\mathcal{L}_{\mathrm{state}} = \frac{1}{C} \sum_{t=1}^{T} \frac{\sum_b a_{bt} \, \| s_{bt} - \mathrm{sg}[R(D(s_{bt}))] \|_2^2 / D_s}{\sum_b a_{bt} + 10^{-8}}, \tag{31}$$

where sg stops gradients and $D_s$ is the number of state elements. This keeps applied transitions close to the observable-state manifold without using hidden intermediate targets.

For isotropy, normalize each non-SKIP code vector $u_k = e_k/\|e_k\|_2$, center the rows, and form $\Sigma = (K-2)^{-1}\sum_{k>0}(u_k-\bar{u})(u_k-\bar{u})^\top$. The implemented penalty is

$$\mathcal{L}_{\text{iso}} = \left\| \frac{\Sigma}{\operatorname{tr}\Sigma} - \frac{I}{d} \right\|_F^2 . \tag{32}$$

The active-length statistic, grounding loss, and isotropy loss are computed without task-depth labels or generating DSL traces.

At every online BEEP update, the current hard programmer trace is evaluated by the domain admission rule. Categorical domains require exact execution on the support set and training query; CompCIFAR uses Eq. 19. Accepted traces update the detached counts using Eq. 17. Updates begin at optimizer step 10,000 and run every 100 steps on at most 128 episodes for Comp-II and categorical 2D, and at most 32 episodes for CompCIFAR. The final Stage-2 calibration pass discards these path-dependent online counts and recomputes them from the selected checkpoint.

### C.3.2 ARCHITECTURE AND OPTIMIZATION

Table 16 summarizes the Stage 1 configuration. The programmer is non-autoregressive: it predicts all six slot distributions from the four demonstrations before execution of the query. The same interpreter $p_\psi(s_t \mid s_{t-1}, e_{z_t})$ is reused at every occupied slot.

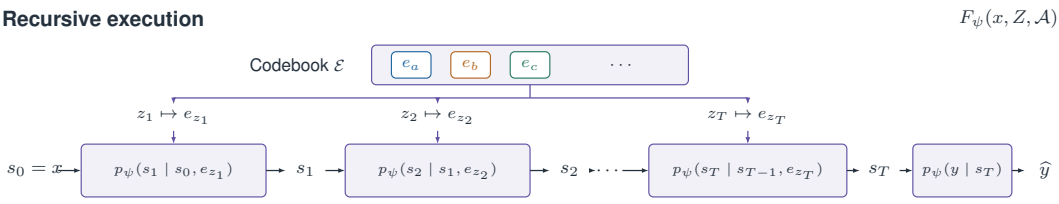



Figure 5: Recursive execution inside $F_\psi$. Each token $z_t$ selects $e_{z_t}$ from the codebook and conditions the shared transition $p_\psi(s_t \mid s_{t-1}, e_{z_t})$. The final $p_\psi(y \mid s_T)$ produces the output prediction. Thus $F_\psi(x, Z, \mathcal{A})$ depends on both the codebook entries and the shared interpreter.

Table 16: Stage-1 architecture, optimization, curriculum, and regularization. Parameter counts include the programmer, codebook, executor, and I/O representations.

| Setting | Comp-II | CompARC | CompLife | CompCIFAR |
|---|---|---|---|---|
| *Program interface and architecture* | | | | |
| Codebook / max. length | 512 / 6 | 512 / 6 | 512 / 6 | 512 / 6 |
| Model width / FFN / heads | 128 / 512 / 8 | 128 / 512 / 8 | 128 / 512 / 8 | 128 / 512 / 8 |
| Programmer | 2-layer NAR Transformer | 2-layer spatial pair encoder; pooled spatial/support summary; 2-layer slot Transformer | Same as CompARC | Exact $2 \times 2$ patches; 2-layer slot-to-spatial cross-attention |
| Executor | 2-layer sequence Transformer | 2-layer ViT transition | 2-layer ViT transition | 3-layer position-aware ViT transition |
| State / positions | Categorical sequence; fixed 1D sinusoidal positions | $12 \times 12$ categorical grid; fixed row/column sinusoids | Clipped $16 \times 16$ categorical grid; fixed row/column sinusoids | Continuous $16 \times 16$ patch grid; fixed row/column sinusoids |
| Recursive state | Soft categorical train; hard argmax evaluation | Soft categorical train; hard argmax evaluation | Soft categorical train; hard argmax evaluation | Continuous patch state |
| Parameters | 929,546 | 1,375,754 | 1,372,674 | 1,724,300 |
| *Optimization* | | | | |
| Updates | 150k | 1M | 1.5M | 220k |
| Batch / microbatch | 2,048 / 2,048 | 128 / 16 | 128 / 16 | 32 / 2 |
| Optimizer | AdamW | AdamW | AdamW | AdamW |
| Peak learning rate | $8 \times 10^{-4}$ | $8 \times 10^{-4}$ | $8 \times 10^{-4}$ | $5 \times 10^{-4}$ |
| Adam $(\beta_1, \beta_2)$ / $\epsilon$ | $(0.9, 0.999)$ / $10^{-8}$ | same | same | same |

Continued on next page

Table 16 (continued)

| Setting | Comp-II | CompARC | CompLife | CompCIFAR |
|---|---|---|---|---|
| Weight decay | $10^{-2}$ | 0 | 0 | $10^{-2}$ |
| Learning-rate schedule | Cosine to zero | Cosine to zero | Cosine to zero | 10% warmup; cosine to 0.1 peak |
| Gradient-norm clip | 2 | 2 | 2 | 1 |
| Precision | FP32 | BF16 autocast; FP32 master state | BF16 autocast; FP32 master state | BF16 autocast; FP32 master state |
| Checkpoint metric | Hard train-val. exact | Hard train-val. grid exact | Hard train-val. grid exact | Train-val. normalized $\ell_1$ |
| *Program-length curriculum and relaxation* | | | | |
| Slot release | Fixed 25k updates per horizon | Validation plateau | Validation plateau | Validation plateau |
| Plateau rule | – | Evaluate every 10k; $\geq$ 15k per horizon; patience 2; gain 0.002 | Same as CompARC | Same, with 40k maximum per horizon and $\ell_1$ gain 0.002 |
| Program temperature | $8 \to 0.5$ | $8 \to 0.5$ | $8 \to 0.5$ | $2 \to 0.5$ |
| State temperature | $2 \to 0.5$ | $2 \to 0.5$ | $2 \to 0.5$ | – |
| *Stage-1 loss* | | | | |
| Execution loss | Categorical NLL | Categorical NLL | Categorical NLL | Charbonnier $+0.25$ terminal patch MSE |
| BEEP $\lambda$ / Dirichlet $\alpha$ / $p_{\text{stop}}$ | $10^{-3}$ / 0.5 / 0.5 | same | same | same |
| $\lambda_{\text{state}}$ | 0.1 | 0.1 | 0.1 | 0 |
| $\lambda_{\text{iso}}$ | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ | $10^{-3}$ |

All Transformer blocks use pre-normalization, GELU feed-forward layers, and zero dropout. CompARC and CompLife encode $(x, y, y-x)$ at every cell before pooling spatial locations and supports. CompCIFAR instead preserves all $4 \times 16 \times 16$ support patches as cross-attention memory. Its patchification is lossless and its all-SKIP path is exactly the input image.

Every temperature schedule uses exponential interpolation,

$$\tau(s) = \tau_0 \exp\left[\frac{\min(s, S)}{S} \log \frac{\tau_S}{\tau_0}\right], \tag{33}$$

and remains at $\tau_S$ after step $S$. Comp-II uses the fixed release schedule; the longer 2D tasks use validation-driven global release because their hard learning onset varies substantially across seeds.

All horizons use the same D1–D3 training mixture; only the number of active program slots changes. Hidden program length and task identity are not used. For the 2D categorical domains, after 15k updates at a horizon, the train-validation hard metric is checked every 10k updates. An improvement above 0.002 resets the counter; two evaluations without improvement release the next slot. CompCIFAR uses the same rule with normalized $\ell_1$ and a maximum of 40k updates at one horizon.

### C.3.3 SOFT FIT AND HARD-PROGRAM LANGUAGE QUALITY

Soft-program accuracy evaluates the relaxed token distribution used during training. Hard-program accuracy takes the argmax token at each slot and executes the resulting discrete trace. Both use the same Test I/O examples. Table 1 reports soft-program Test I/O for NeuLP and NLI, whereas the Stage 1 design ablations use hard execution.

Hard execution is the direct test of whether the programmer and learned language yield executable discrete programs. A high soft score alone can rely on token mixtures that do not survive token selection. Hard execution does not, by itself, establish atomic primitive recovery; the closure and standalone-token audits below probe reusable discrete semantics separately.

### C.3.4 DISCRETE PROGRAM INTERFACE

The interface control uses three matched Comp-II runs from Table 2. At the final training snapshot, the discrete codebook reaches $99.73 \pm 0.31\%$ hard Test I/O, compared with $33.33 \pm 57.74\%$ for the continuous interface.

### C.3.5 SELF-CURRICULUM

The self-curriculum changes the available program length while keeping the training data fixed. The model always contains six slots, but only the first $T_{\text{active}}(s)$ are active at update $s$. Every arm uses the same mixed Level-1–3 Comp-II stream; generating depth, task identity, and target program length are not provided.

The sweep contains seven five-seed schedules. The full control activates all six slots from the start. One-to-all activates one slot and then releases the remaining five, either after 50k updates or at a soft/hard validation plateau. Gradual schedules follow $1 \to 2 \to \cdots \to 6$, using either fixed 25k-step releases or the same plateau rule. All arms train for 200k updates with batch 2,048 and otherwise matched settings.

Table 17: Complete five-seed Comp-II self-curriculum design sweep. Values are best held-out mixed-task hard exact match.

| Path | Horizons | Trigger | Hard Test I/O (%) | $> 95\%$ |
|---|---|---|---|---|
| Full from start | 6 | none | $69.45 \pm 25.90$ | 0/5 |
| One-to-all | $1 \to 6$ | fixed at 50k | $46.63 \pm 38.50$ | 1/5 |
| One-to-all | $1 \to 6$ | soft plateau | $64.28 \pm 30.08$ | 1/5 |
| One-to-all | $1 \to 6$ | hard plateau | $64.28 \pm 30.08$ | 1/5 |
| **Gradual** | $\mathbf{1 \to \cdots \to 6}$ | **every 25k** | $\mathbf{99.86 \pm 0.19}$ | **5/5** |
| Gradual | $1 \to \cdots \to 6$ | soft plateau | $99.28 \pm 0.74$ | 5/5 |
| Gradual | $1 \to \cdots \to 6$ | hard plateau | $99.55 \pm 0.65$ | 5/5 |

Fixed gradual release reaches $99.86 \pm 0.19\%$ hard Test I/O and exceeds 95% in all five seeds. The soft- and hard-plateau gradual schedules reach $99.28 \pm 0.74\%$ and $99.55 \pm 0.65\%$. Full-horizon training reaches $69.45 \pm 25.90\%$, and all one-to-all schedules remain below 65% on average. For the weaker schedules, soft accuracy is often much higher than hard accuracy, indicating poor discretization.

### C.3.6 LANGUAGE PRIOR AND LENGTH MODEL

Table 18 compares the three language objectives on Comp-II. Stage 1 contains all 220 D1–D3 functions, and OOC search uses 90 unseen D4–D6 tasks per seed. Each row contains five seeds.

Language closure asks whether stable learned tokens can be composed to recover the training functions within six slots. Let $\mathcal{F}_{\text{train}}$ be the audited training functions and $\mathcal{V}_{\text{stable}}$ the non-SKIP tokens with validated standalone semantics. A stable token may represent a primitive or a macro. For each $f \in \mathcal{F}_{\text{train}}$, we construct one shortest composition $Z_f$ from $\mathcal{V}_{\text{stable}}$, when one exists, and execute it on fresh audit pairs $\mathcal{D}_f^{\text{fresh}}$:

$$\widehat{\text{CL}}_6 = \frac{1}{|\mathcal{F}_{\text{train}}|} \sum_{f \in \mathcal{F}_{\text{train}}} \mathbf{1}\left[ Z_f \text{ exists } \wedge \prod_{(x,y) \in \mathcal{D}_f^{\text{fresh}}} \mathbf{1}[F_\psi(x, Z_f) = y] = 1 \right]. \tag{34}$$

A function counts as closed only when this composition succeeds on every audit pair. We use this single shortest witness rather than searching alternative traces after a failure. The NLI reuse weight is $10^{-3}$, and NEO uses $\lambda_{\text{NEO}} = 1.05$. All other architecture, curriculum, regularization, optimization, and checkpoint settings match Table 16.

Table 18: Five-seed Comp-II objective comparison. Stage 1 contains all 220 D1–D3 identities; OOC search is restricted to D4–D6 and uses 32 starts with 30 stratified directional search rounds per start.

| Objective | Test I/O (%) | language closure (%) | OOC +search (%) |
|---|---|---|---|
| NLI reuse | $99.9 \pm 0.2$ | 78.5 | $23.8 \pm 14.1$ |
| NEO prefix MDL | $60.3 \pm 44.9$ | 98.1 | $27.8 \pm 7.6$ |
| **BEEP** $10^{-3}$ | $100.0 \pm 0.0$ | **99.7** | $34.2 \pm 6.4$ |

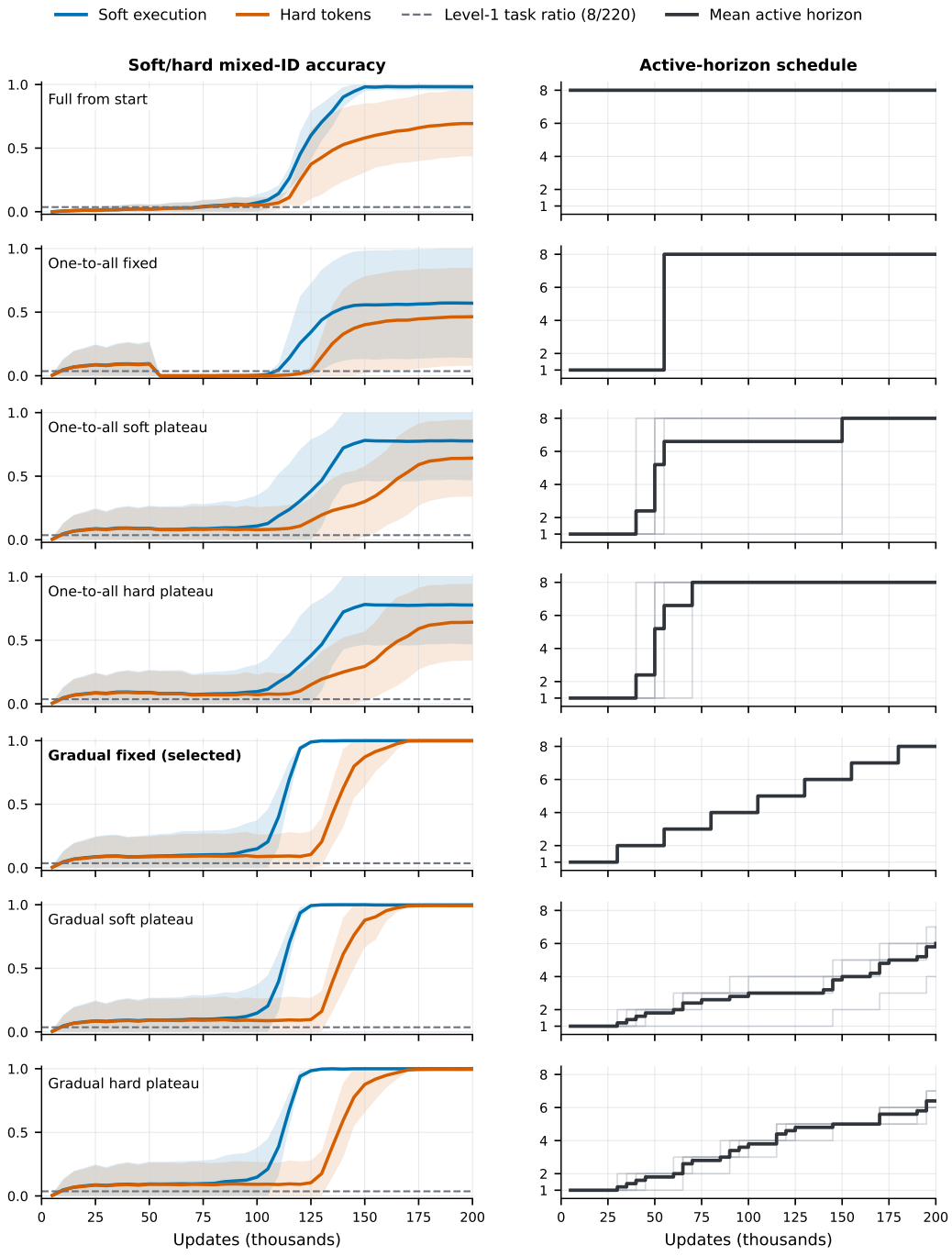



Figure 6: Five-seed Comp-II self-curriculum sweep. Each row is one design from Table 17. Left: mean soft-execution and hard-token exact match on a fixed mixed Level-1–3 evaluator; shading is one standard deviation. The dashed Level-1 task-ratio line is $8/220$, the mixed-set fraction obtained by solving every Level-1 identity and no deeper identity. Right: active horizon; light curves are individual seeds and the dark curve is their mean.

The search result also depends on the programmer initialization; NEO MDL can produce more diverse initial programs. BEEP nevertheless gives complete training-task induction, complete closure in all five seeds, and the highest matched five-seed OOC search result.

Table 19: BEEP component controls. Blind search uses 15 fixed tasks and 65,536 programmer-independent uniform-codebook proposals per task. Rows retain their available four or five seeds.

| Treatment | seeds | Test I/O exact | full closure | blind solved | OOC +search (%) |
|---|---|---|---|---|---|
| Production only $10^{-3}$ | 4 | 880/880 | 3/4 seeds | 15/60 | $20.6 \pm 16.0$ |
| Production only $10^{-2}$ | 4 | 876/880 | 4/4 seeds | 16/60 | $26.1 \pm 7.1$ |
| Production only $10^{-1}$ | 5 | 1,100/1,100 | 5/5 seeds | 15/75 | $32.0 \pm 7.0$ |
| **BEEP** $10^{-3}$ | 5 | 1,100/1,100 | 5/5 seeds | 19/75 | $34.2 \pm 6.4$ |

On matched seeds 1, 2, and 5, including BEEP's termination term gives $28.1 \pm 7.1\%$, $30.0 \pm 2.2\%$, and $33.0 \pm 7.1\%$ searched OOC at production coefficients $10^{-2}$, $10^{-1}$, and $10^{-3}$, respectively. Stronger production penalties therefore do not replace BEEP's termination component.

### C.3.7 Repeated-token failure

In seed 5, production-only BEEP solves all 220 Test I/O functions but has zero stable standalone closure. Its selected programs contain 14 two-token and 206 three-token traces. For example, `shift1` is realized only by

$$[96, 96, \textsc{skip}, \textsc{skip}, \textsc{skip}, 96]. \tag{35}$$

The repeated trace, rather than a standalone token, therefore accounts for the high Test I/O score. Full BEEP removes this failure in all five seeds: every selected trace uses one active token and every seed has complete stable closure. Semantic aliases remain in two seeds, so the learned language need not contain a unique token for each primitive.

### C.3.8 Auxiliary stabilization terms

Stage 1 uses state grounding and codebook isotropy as auxiliary stabilization terms. State grounding keeps recursive categorical states near the observable-state manifold without using hidden intermediate targets (Eq. 31). Codebook isotropy regularizes the learned token directions (Eq. 32).

Table 20 gives a five-seed Comp-II ablation of these two terms around the full NeuLP objective. These runs use 100k training updates, batch size 512, and 32 search starts with 30 rounds per start. Search is evaluated on an early held-out Lv4–5 subset, not the final length-OOC evaluation set, and the training hyperparameters differ from those used in the main comparison.

Table 20: Five-seed Comp-II auxiliary-regularization ablation under the 100k early-search protocol described above.

| Treatment | Hard Test I/O (%) | Lv4–5 + search (%) |
|---|---|---|
| No state grounding | $100.00 \pm 0.00$ | $27.41 \pm 10.08$ |
| No isotropy | $89.97 \pm 15.74$ | $28.83 \pm 5.76$ |
| **Both terms** | $\mathbf{100.00 \pm 0.00}$ | $\mathbf{30.37 \pm 4.49}$ |

Removing isotropy lowers hard Test I/O, while removing either term lowers the mean Lv4–5 search accuracy. We use this experiment as evidence for the auxiliary stabilization terms rather than as part of the final OOC comparison.

## C.4 Stage 2: calibration, search, and dreaming

Stage 2 freezes the programmer, codebook, and interpreter. Gradients are taken only with respect to the current program representation and are used to rank legal discrete edits. Token counts, the execution-supported vocabulary, and the replay bank may change; model parameters do not.

Each search state is represented by one-hot token vectors with gradients enabled. The executor runs without Gumbel sampling: categorical intermediate states remain probability vectors, while the candidate program is discrete. Every proposed replacement is then executed as a hard program. Search uses

$$\mathcal{E}_{\text{search}}(Z) = \mathcal{L}_{\text{support}}(Z) + \lambda_{\text{search}}\, \mathcal{C}_{\text{BEEP}}(Z), \qquad \bar{c}_k = \frac{-\log \theta_k}{\frac{1}{K-1}\sum_{j=1}^{K-1} -\log \theta_j}. \tag{36}$$

All reported searches use $\lambda_{\text{search}} = 0.1$. Legal edits are restricted to execution-supported tokens, while the BEEP cost remains normalized over all non-SKIP codebook rows as in Stage 1. The query output is not used in the search score.

Table 21: Stage-2 execution and admission configuration. Parameter counts are the frozen Stage-1 models; Stage 2 has zero trainable parameters. "Exact" requires every listed example to match after hard final decoding.

| Domain | frozen params. | $\lambda_{\text{search}}$ | support score | admission gate |
|---|---|---|---|---|
| Comp-II | 929,546 | 0.1 | Summed categorical NLL over support examples and sequence positions | Exact support during search; exact support and reserved acquisition query before replay admission |
| CompARC | 1,375,754 | 0.1 | Mean categorical NLL over support cells | Exact support during search; exact support and query during calibration |
| CompLife | 1,372,674 | 0.1 | Mean categorical NLL over support cells | Exact support during search; exact support and query during calibration |
| CompCIFAR | 1,724,300 | 0.1 | Mean RGB Charbonnier loss, $\epsilon = 10^{-3}$ | Normalized-$\ell_1$ gain over input copy of at least $10^{-3}$ on support during search, and separately on support and query during calibration |

### C.4.1 LABEL-BLIND CALIBRATION

Calibration scans the frozen training population with the selected Stage 1 checkpoint. Comp-II uses 499,712 deterministically generated episodes from its 178 training identities. The objective ablation in Table 18 instead uses all 220 stated training identities. Counts are merged additively using Eq. 17, and token support is merged by set union. Calibration does not use task identity, task depth, or the hidden program.

Admission is based on hard execution. Comp-II, CompARC, and CompLife require exact execution on all four support pairs and the reserved query. CompCIFAR uses the $10^{-3}$ improvement rule in Eq. 19, evaluated on rounded 8-bit predictions. A token is *execution-supported* once it appears in an admitted complete program.

Calibration resets the counts and defines

$$\mathcal{V}_{\text{exec}} = \{k > 0 : n_k > 0\}. \tag{37}$$

Noisy starts, replacement edits, and dreams use only $\mathcal{V}_{\text{exec}} \cup \{\text{SKIP}\}$. The unmodified programmer proposal is kept as trajectory zero. If it passes admission, any new token it contains is added to $\mathcal{V}_{\text{exec}}$; edited programs cannot add unsupported tokens. A uniform reservoir of at most 256 admitted training episodes forms $\mathcal{B}_{\text{train}}$. All admitted episodes update the counts and execution-supported vocabulary before reservoir sampling.

### C.4.2 STRATIFIED DIRECTIONAL GCU

Directional GCU scores the one-hot edit from the current token $j = z_t$ to a legal replacement $k$ using Eq. 7. The raw-gradient baseline instead selects the slot with the largest gradient norm and performs worse (Table 22). Because gradient scales vary across recurrent positions, we rank

replacements within each slot, keep the same number from each slot, and choose among the resulting hard programs by execution.

The full multi-start procedure is:

1. Run the programmer once. Keep its hard trace as trajectory zero and initialize the other $S - 1$ trajectories by Gaussian perturbation followed by a nearest-token snap in $\mathcal{V}_{\text{exec}} \cup \{\text{SKIP}\}$.
2. For each trajectory and round, execute its complete trace on support, evaluate the search energy, and differentiate that energy with respect to its relaxed slot representations.
3. Rank legal replacements separately within each of the six slots by directional score and reserve equal shortlist capacity for every slot. Replace 20% of the resulting shortlist with uniform legal token–slot edits.
4. Execute every shortlisted hard trace. Comp-II accepts the best strict $\mathcal{E}_{\text{search}}$ improvement. Categorical 2D search accepts a support-exact candidate when available and otherwise the best strict behavior-loss improvement. For CompCIFAR, gradients use the smooth Charbonnier loss, while accepted candidates are ranked by the actual quantized-output normalized $\ell_1$ on support pairs.
5. For categorical tasks, mark a trajectory solved when its endpoint passes the exact support rule. Otherwise, after two consecutive non-improving rounds, replace it with a fresh noisy snap and continue from the next round.
6. Stop after $R$ rounds or, for categorical tasks, when the task-level exact stopping rule is met. CompCIFAR always runs the fixed budget and retains the best support-scoring program seen, including the initial proposal: improvement over copying the input permits replay admission, but does not stop optimization. Apply any reserved acquisition-query gate only to the final support-solved endpoint, before replay admission; never use it to rank or accept edits.

The Gaussian snap scale is 7.5. Snapped and edited trajectories are restricted to the calibrated vocabulary. The raw programmer proposal is the only program that may contain a previously unsupported token; if admitted, that token expands the vocabulary for later tasks.

Table 22: Matched Comp-II search-score control over three seeds and 68 tasks per seed. Exact accuracy is the mean ± sample standard deviation across seeds; pooled counts are summed over all 204 tasks.

| Method | mean exact (%) | pooled solved / 204 |
|---|---|---|
| Raw gradient norm | $43.1 \pm 9.8$ | 88 |
| Global directional score | $77.9 \pm 0.0$ | 159 |
| **Stratified directional GCU** | $\mathbf{86.3 \pm 7.4}$ | **176** |
| Random coordinate (Rnd.) | $72.5 \pm 20.9$ | 148 |
| Random edit | $82.4 \pm 14.9$ | 168 |

### C.4.3 COLLECTION AND DREAMING

Table 23: Selected Stage-2 collection budget per seed. "64/level" means at most 64 fixed OOC tasks from each available depth stratum; smaller strata are exhausted. The replacement cap is per trajectory and round.

| Domain | calib. batch | search tasks | starts | rounds | replacement cap | dreams | dream cap |
|---|---|---|---|---|---|---|---|
| Comp-II | 128 | 64/level | 32 | 30 | $\lvert\mathcal{V}_{\text{exec}} \cup \{\text{SKIP}\}\rvert$ | 1,024 | 256 |
| CompARC | 16 | 64/level | 32 | 30 | 32 | 1,024 | 256 |
| CompLife | 16 | 64/level | 32 | 30 | 32 | 1,024 | 256 |
| CompCIFAR | 16 | 64/level | 32 | 30 | 32 | 1,024 | 256 |

For Comp-II, the adaptation pool contains 4 D2 identities, 17 D3 identities, and at least 64 identities at each of D4–D6. We therefore search all D2/D3 identities and 64 identities from each of D4–D6, for 213 search tasks per seed. These tasks are disjoint from the 321-task reporting panel.

The dream cap in Table 23 applies only to admitted dream records. Calibration uses a separate 256-record reservoir (Appendix C.4.1); all admitted calibration programs still update the counts and execution-supported vocabulary.

Dreaming uses no benchmark I/O. It samples length uniformly from $\{1, \ldots, 6\}$, samples active tokens from the recalibrated distribution over $\mathcal{V}_{\text{exec}}$, samples an input, and executes the program. The uniform dream-length proposal is separate from the geometric Stage 1 prior. Admitted search programs and dreams update the posterior counts with the same trace-normalized rule. Comp-II dreams must be non-identity, stable on audit inputs, and semantically distinct. Categorical 2D dreams reject identity and duplicate output signatures; CompCIFAR uses the normalized-$\ell_1$ gain rule. Hidden DSL matching is used only for post-hoc reporting.

### C.5 Stage 3: learning from discovered programs

#### C.5.1 Replay-bank contents

Each replay record stores the observed support $\mathcal{U}$, the complete six-slot hard program $Z^*$, and its source $\{\text{train}, \text{search}, \text{dream}\}$. A reserved or generated query pair is stored when available.

Query handling follows the source of the record. Comp-II search requires exact execution on both support and a reserved acquisition query before admission. Categorical 2D search uses support only; in Stage 3, its execution loss draws one of the four stored support pairs. Dream queries are generated by the frozen interpreter and can be used directly. Comp-II uses newly generated acquisition PBEs, CompARC and CompLife may reuse final supports, and CompCIFAR stores support-only search records.

Table 24: Validated replay schema and use in the selected Stage-3 objective.

| Field | Meaning | Selected Stage-3 use |
|---|---|---|
| $\mathcal{U} = (X_{1:M}, Y_{1:M})$ | Four observed PBE support pairs | Programmer input; source of execution supervision for 2D search records |
| $X_q, Y_q$ (optional) | Reserved acquisition or generated query | Execution supervision when available; otherwise not used by Stage 3 |
| $Z^*$ | Complete admitted hard trace, including SKIP | Six-slot trace cross-entropy target |
| source | train, search, or dream | Sampling/accounting and the query-label rule above |

#### C.5.2 Selected objective

Stage 3 freezes the codebook, interpreter, executor-side state representation, and any frozen codec. All six program slots remain active, and only the programmer is updated for 150k steps. In the categorical models, the programmer's I/O embedding remains trainable.

Stage 3 alternates replay-bank and training-retention updates. For a replay record,

$$\mathcal{L}_{\text{bank}} = \text{CE}\big(p_\phi(Z \mid \mathcal{U}), Z^*\big) + 0.1\, \overline{\mathcal{L}}_{\text{exec}}\big(p_\phi, F_{\psi_{\text{S1}}}; \mathcal{U}, X, Y\big), \tag{38}$$

where CE covers all six slots, including SKIP, and the execution NLL is averaged over output elements before weighting. For Comp-II, CompARC, and CompLife this is mean categorical NLL. For CompCIFAR it is mean RGB Charbonnier loss with $\epsilon = 10^{-3}$; the Stage-1 terminal patch-state term is not applied during Stage-3 replay.

For the retention branch, take a validated training record $(\mathcal{U}_{\text{train}}, Z_{\text{train}}) \in \mathcal{B}_{\text{train}}$, sample new inputs, and execute $Z_{\text{train}}$ with the frozen Stage 1 interpreter to form

$$\begin{aligned} \widetilde{\mathcal{U}}_{\text{train}} &= \{(\tilde{x}_m, F_{\psi_{\text{S1}}}(\tilde{x}_m, Z_{\text{train}}, \mathcal{A}))\}_{m=1}^{M}, \\ \mathcal{L}_{\text{train}}(\phi; \widetilde{\mathcal{U}}_{\text{train}}) &= \text{KL}\Big(p_{\phi_{\text{S1}}}(Z \mid \widetilde{\mathcal{U}}_{\text{train}}) \,\|\, p_\phi(Z \mid \widetilde{\mathcal{U}}_{\text{train}})\Big). \end{aligned} \tag{39}$$

The two update types alternate 1:1, with distillation temperature 1.0. Comp-II samples retention inputs outside the held-out Test I/O rows. The 2D domains sample them from the frozen training split.

Table 25: Stage-3 modules and losses. Replay-bank and training-retention examples use separate update branches.

| Branch | Data and target | Loss and parameter update |
|---|---|---|
| Bank | Execution-validated replay record; complete hard trace $Z^*$ | Trace CE $+0.1$ normalized execution loss; update programmer |
| Training retention | Model-generated $\widetilde{\mathcal{U}}_{\text{train}}$; frozen Stage-1 programmer distribution | KL distillation at temperature 1; update programmer |
| Shared | Alternate branches 1:1; all six slots active | Freeze codebook, executor state representation, interpreter, and codec |
| Inactive | – | No BEEP, length, isotropy, state grounding, or terminal patch-state loss |

Table 26: Stage-3 optimization by domain. All runs use 150k updates, all six slots, AdamW at learning rate $2 \times 10^{-4}$ with cosine decay, weight decay $10^{-2}$, and gradient-norm clipping at 2.

| Setting | Comp-II | CompARC | CompLife | CompCIFAR |
|---|---|---|---|---|
| Total parameters | 929,546 | 1,375,754 | 1,372,674 | 1,724,300 |
| Trainable parameters | 464,896 | 910,848 | 909,824 | 1,044,224 |
| Batch size | 512 | 32 | 32 | 2 |
| Precision | FP32 | BF16 | BF16 | BF16 |
| Bank execution loss | cat. NLL | cat. NLL | cat. NLL | Charbonnier |
| Program temperature | | $2 \to 0.5$ | | |
| State temperature | $1 \to 0.5$ | $1 \to 0.5$ | $1 \to 0.5$ | – |

Validation runs every 5,000 updates and considers only full-horizon checkpoints. Eligible checkpoints must remain within 0.02 of the Stage 1 train-validation metric (two exact-accuracy points, or $+0.02$ normalized $\ell_1$ for CompCIFAR). Selection then prioritizes search/dream records not solved by Stage 1, followed by overall bank execution, trace recovery, and train-validation. Task depth and hidden DSL labels are not used.

### C.5.3 Stage-3 ablation

The Comp-II ablation tests the two Stage 3 choices used in the final model: freezing the execution pathway and retaining the Stage 1 programmer. All runs start from matched Stage 1 sources, use enlarged Stage 2 banks, and train for 150k updates. The search column contains 144 evaluations using 32 starts and 30 rounds per start.

Table 27: Comp-II Stage-3 ablation. Bank uptake counts records unsolved by Stage 1 and solved after Stage 3.

| Treatment | Supervision | Test I/O (%) | bank uptake | search |
|---|---|---|---|---|
| Stage 1 source | – | 100.00 | – | 114/144 |
| Joint update | Trace + execution; training execution | 96.63 | 10/1,126 | 83/144 |
| Frozen execution pathway | Trace + execution; training execution | 99.81 | 46/1,126 | 119/144 |
| Frozen, bank trace only | Trace CE | 13.67 | 1,125/1,126 | 110/144 |
| **Frozen + branched train distill.** | Trace CE + train KL | 99.81 | 1,125/1,126 | 116/144 |

Joint adaptation lowers both Test I/O and search accuracy, while freezing the execution pathway preserves them. With the pathway frozen, bank-only trace CE recovers almost every stored trace but loses Test I/O accuracy. Stage 1 distillation restores Test I/O while preserving the same bank uptake.

### C.5.4 RQ3 FIXED-SUPPORT PROPOSAL EXPERIMENT

Table 5 compares Stage 1 and Stage 3 on the same CompARC Stage 2 support and query examples; only the programmer checkpoint changes. Zero-shot evaluation executes each checkpoint's hard program prediction directly.

The 128-start proposal setting uses only the search initialization and performs no coordinate edit. The first start is the greedy programmer prediction. When the population is expanded, the remaining 127 starts independently perturb each continuous slot output and snap it to the nearest execution-supported token:

$$z_c^{(b,0)} = \underset{k \in \mathcal{V}_{\text{exec}} \cup \{0\}}{\arg\min} \|h_c(\mathcal{U}) + \epsilon_{b,c} - e_k\|_2^2, \qquad \epsilon_{b,c} \sim \mathcal{N}(0, 7.5^2 I). \tag{40}$$

Duplicate programs are allowed. If the greedy proposal already solves the support set, CompARC and CompLife stop before constructing the remaining starts. Evaluation ends before any GCU update, so this experiment measures the initial search proposals only. The support and query examples are identical for Stage 1 and Stage 3.

### C.5.5 STAGE-1 TO STAGE-3 DECOMPOSITION

Table 28 separates the contribution of Stage 3 from the final search. The Stage 1 and Stage 3 columns execute greedy hard programs without search. The final column reproduces the current headline OOC endpoints from Tables 1 and 7; final search uses 128 initial programs and up to 100 rounds per start.

The zero-shot columns report hard-program diagnostics. Their Test I/O values therefore differ from the soft-program Test I/O values reported for NeuLP in the main comparison, although both are evaluated over the same five seeds. CompCIFAR Stage-1/3 zero-shot diagnostics score continuous decoded images, whereas calibration and final search score rounded 8-bit outputs.

Table 28: NeuLP from Stage 1 to Stage 3 and final search. Categorical entries are exact accuracy (%); CompCIFAR reports normalized $\ell_1$ (lower is better). Stage 1 and Stage 3 use greedy hard programs.

| Domain | Split | Stage 1 zero-shot | Stage 3 zero-shot | Stage 3 +search | Budget |
|---|---|---|---|---|---|
| Comp-II | Hard Test I/O | $100.00 \pm 0.00$ | $100.00 \pm 0.00$ | – | – |
| Comp-II | Recomb. OOC | $0.00 \pm 0.00$ | $0.00 \pm 0.00$ | $100.00 \pm 0.00$ | $128 \times 100$ |
| Comp-II | Length OOC | $0.00 \pm 0.00$ | $0.00 \pm 0.00$ | $99.87 \pm 0.18$ | $128 \times 100$ |
| CompARC | Hard Test I/O | $69.90 \pm 4.12$ | $81.55 \pm 7.16$ | – | – |
| CompARC | Recomb. OOC | $13.64 \pm 6.43$ | $75.00 \pm 3.21$ | $80.80 \pm 4.07$ | $128 \times 100$ |
| CompARC | Length OOC | $0.83 \pm 0.24$ | $48.33 \pm 1.89$ | $64.93 \pm 2.06$ | $128 \times 100$ |
| CompCIFAR | Hard Test I/O | $0.013 \pm 0.001$ | $0.012 \pm 0.002$ | – | – |
| CompCIFAR | Recomb. OOC | $0.084 \pm 0.009$ | $0.011 \pm 0.002$ | $0.009 \pm 0.001$ | $128 \times 100$ |
| CompCIFAR | Length OOC | $0.120 \pm 0.005$ | $0.014 \pm 0.002$ | $0.012 \pm 0.001$ | $128 \times 100$ |
| CompLife | Hard Test I/O | $88.51 \pm 7.25$ | $91.98 \pm 2.14$ | – | – |
| CompLife | Recomb. OOC | $4.21 \pm 2.35$ | $48.95 \pm 28.07$ | $88.16 \pm 9.30$ | $128 \times 100$ |
| CompLife | Length OOC | $0.07 \pm 0.15$ | $13.60 \pm 12.56$ | $73.96 \pm 7.12$ | $128 \times 100$ |

### C.5.6 LANGUAGE GROWTH

Language growth runs a second NeuLP cycle:

$$(\text{S1} \to \text{S2} \to \text{S3})_{\text{base}} \to (\text{S1}_{\text{growth}} \to \text{S2} \to \text{S3}) \to \text{final search}. \tag{41}$$

Growth starts from the Stage 3 checkpoint and validated bank of the seven-operation base language. The second cycle adds `reverse` while keeping the codebook size fixed at $K = 512$.

**Growth Stage 1.** The programmer, interpreter, both I/O embedding pathways, and all unprotected codebook rows are trainable. A row is protected when it is a non-SKIP codebook row occurring in the

preceding validated bank; protection does not depend on a ground-truth primitive label. Protected rows are snapshotted and restored after every AdamW step so that weight decay cannot move them. Unprotected non-SKIP rows remain trainable and retain their learned initialization. The protected set is defined by bank usage and can therefore include aliases or composed behaviors, not only the seven base primitives. SKIP retains its architectural identity behavior.

The growth stream contains 132 retained base identities (7 Lv1, 25 Lv2, 100 Lv3) and 29 reverse-bearing identities (1 Lv1, 5 Lv2, 23 Lv3). Each identity contributes 500 PBE episodes with four demonstrations and one query, for 66,000 old and 14,500 new episodes. The 80,500 episodes are shuffled without rebalancing.

Each 512-example update uses 256 growth episodes and 256 replay records whose programs fit the current active horizon. Replay records are sampled with replacement from the preceding bank. If none fit the current horizon, the update uses only growth episodes.

Let $M$ denote the mixed raw/replay minibatch and $B_h$ its replay records compatible with the active horizon. The implemented acquisition objective is

$$\begin{aligned}\mathcal{L}_{\text{growth}} = \mathcal{L}_{\text{IO}}(M) + 10^{-3}\mathcal{L}_{\text{prod}}(M) + (10^{-3}\log 2)\mathcal{L}_{\text{active}}(M) \\ + 10^{-3}\mathcal{L}_{\text{iso}}(M) + 0.1\mathcal{L}_{\text{state}}(M) + 0.5\mathcal{L}_{\text{trace}}(B_h).\end{aligned} \tag{42}$$

Here $\mathcal{L}_{\text{prod}}$ is the BEEP production term, $\mathcal{L}_{\text{active}}$ is the geometric length term, and $\mathcal{L}_{\text{trace}}$ is cross-entropy over the complete trace, including SKIP. Growth and replay examples are both executed through the current programmer and interpreter; replay records also provide the trace target.

Growth Stage 1 runs for 150,000 updates with AdamW, initial learning rate $2\times10^{-4}$, cosine decay to zero, betas $(0.9, 0.999)$, $\epsilon = 10^{-8}$, weight decay $10^{-2}$, and gradient-norm clipping at 2. The active horizon follows $1 \to 2 \to \cdots \to 6$, releasing one slot every 25,000 updates. Program temperature decays exponentially from 2 to 0.5 and state temperature from 1 to 0.5. Evaluation runs every 5,000 updates. Growth Stage 1 selects the best hard checkpoint by depth-macro train-validation rather than final OOC search performance.

**Growth Stages 2–3.** Stage 2 searches the expanded training identities with 32 starts and 30 rounds per start and performs dreaming with 1,024 proposals, admitting at most 256. New records are merged with the preceding bank. Stage 3 then runs for 150,000 updates with batch size 512 and AdamW at initial learning rate $2 \times 10^{-4}$, while freezing the complete execution pathway. It uses the selected Stage-3 objective: alternating bank trace CE plus $0.1$ mean execution loss with Stage-1-programmer KL supervision. Full-horizon checkpoints are selected by balanced bank-amortization criteria subject to a train-validation retention floor of source minus 0.02; if none satisfies the floor, selection falls back to the best train-validation checkpoint.

In a separate freeze-only control, fixing validated token rows while training the shared interpreter leaves forced-token execution at $25.0\%$. Replaying their I/O raises forced execution to $99.55\%$; adding trace supervision also restores programmer use of the validated tokens.

**Growth evaluation.** Growth evaluation measures original-task hard Test I/O on 1,320 episodes (132 identities $\times$ 10) and reverse-bearing Test I/O on 290 episodes (29 identities $\times$ 10). Final OOC search uses one frozen episode for each of 28 recombination identities and 300 length identities, with 100 length identities at each of Lv4–6. Final search uses 128 initial programs and up to 100 rounds per start, with $\epsilon = 0.20$.

### C.5.7 FINAL SEARCH

Final search uses the Stage 2 procedure with a larger budget. The Stage 3 programmer supplies 128 initial programs; each runs for up to 100 search rounds. Edits remain restricted to execution-supported tokens and are accepted by hard execution on the support set. The query target is not used.

## C.6 BASELINE IMPLEMENTATION CONTROLS

All baselines use the same PBE demonstrations, query inputs, task splits, and output metrics as NeuLP. HRM and TRM retain their native task identifiers; all examples from the same task share one identifier. Episode Transformer, HRM, and TRM use zero-shot inference. LPN uses 8,192 latent

samples with 100 gradient steps per sample. NLI uses its published relaxed search with 8,192 starts and 100 updates, including the 2D experiments. NeuLP search is applied only after Stage 3.

Table 29: Model sizes, including learned task embeddings and learned components frozen in subsequent stages. Search variants reuse the same model.

| Method | Comp-II | CompARC | CompLife | CompCIFAR |
|---|---|---|---|---|
| NeuLP | 929,546 | 1,375,754 | 1,372,674 | 1,724,300 |
| NLI | 928,266 | 1,374,474 | 1,372,418 | – |
| LPN | 911,808 | 1,424,040 | 1,428,840 | – |
| Episode Transformer | 929,034 | 1,258,577 | 1,258,577 | – |
| HRM | 1,378,306 | 2,496,258 | 2,507,394 | – |
| TRM | 1,551,874 | 3,508,290 | 3,527,778 | – |
| NEO / NEO-S | – | – | – | 1,218,639 |

HRM/TRM parameter counts include their learned task embeddings: 64,000/112,000 parameters (HRM/TRM) for Comp-II, 54,528/95,424 for CompARC, and 65,664/114,912 for CompLife. Comp-II uses the compact numeric-I/O variant; CompARC and CompLife use the larger structural-vocabulary models. NEO-S changes only inference and shares NEO's parameters.

Table 30: Baseline architectures and inference procedures.

| Method | Representation and model | Task conditioning | Inference |
|---|---|---|---|
| Original NLI | Six-slot NAR programmer and learned interpreter; matched NeuLP width/depth; shared I/O embeddings | Four demonstrations | Relaxed latent gradient search |
| Episode Transformer | Two-layer encoder–decoder Transformer; width 128 in Comp-II and 144 in 2D | Serialized demonstrations and query | Greedy autoregressive decoding |
| HRM | Width 128; four high- and four low-level layers; two cycles per level; rotary positions | Native task identifier shared by demonstrations and query | Recursive ACT decoding, at most 16 steps |
| TRM | Width 224; two recursive layers; three outer and six inner cycles; rotary positions | Native task identifier shared by demonstrations and query | Recursive decoding, at most 16 steps |
| LPN | Four-layer encoder and decoder; latent dimension 64; width 96 in Comp-II and 120 in 2D | Four demonstrations | Mean latent for Test I/O; optimized latent for OOC |
| NEO | Frozen convolutional VAE; 16-action codebook; two-block FiLM policy and transition | Four demonstrations | Greedy decode; NEO-S samples 1,024 programs and selects by support loss |

Table 31: Baseline optimization budgets. HRM and TRM use matched optimizer-update budgets in the reported comparisons; other rows follow the stated update or inference conventions.

| Method | Training budget | Batch | Learning rate / regularization | OOC inference budget |
|---|---|---|---|---|
| Original NLI | 200k Comp-II; 1M CompARC; 1.5M CompLife | 512 Comp-II; 128 2D | $8 \times 10^{-4}$; reuse $10^{-3}$ | $8192 \times 100$ in all domains |
| Episode Transformer | 125k updates | 32 | $3 \times 10^{-4}$; WD $10^{-2}$; dropout 0.1 | Greedy decode |
| HRM | Update-matched; 150k in Comp-II | 32 | $10^{-4}$; 2k warmup; WD 0.1 | Native ACT |
| TRM | Update-matched; 150k in Comp-II | 32 | $10^{-4}$; 2k warmup; WD 0.1 | Native recursion |
| LPN | 200k updates | 64, accumulation 2 | $4 \times 10^{-4}$; prior KL $10^{-4}$ | $8192 \times 100$ latent search at LR 0.1 |
| NEO (CompCIFAR) | 62.5k codec + 25k closed-loop updates | 512 codec; 64 loop | $5 \times 10^{-4}$ codec; $10^{-3}$ loop | Greedy or 1,024 sampled programs |

CompLife episodes serialize to 5,836 tokens, so Episode Transformer uses 8,192 fixed sinusoidal positions and gradient accumulation. Held-out query targets are excluded from baseline adaptation.

The five-seed CompCIFAR NEO reproduction has 1.22M parameters, compared with 1.72M for PatchViT-2. Its convolutional VAE is pretrained for 62,500 updates with batch 512, AdamW learning rate $5 \times 10^{-4}$, latent-prior weight $10^{-4}$, and cosine decay, then frozen. Closed-loop training uses batch 64, microbatch 4, training/evaluation lengths 3/6, gradient clipping 1, 5% linear warmup, and cosine decay to 0.1 of the peak learning rate. The Gumbel temperature is fixed at 1.0; VQ and state-grounding weights are 0.5 and 0.1, and the prefix-MDL factor decays from 1.01 to 1.0. Train-validation selects the 15k checkpoint. NEO-S changes only inference. Where the published image-editing setup leaves a choice unspecified, we use a two-block residual FiLM model and the frozen CompCIFAR PBE records.

Dataset manifests, selected checkpoints, and per-task predictions are archived for every reported run.

### C.7 Software, hardware, and compute accounting

NeuLP and NLI use Python 3.10.13 and PyTorch 2.6.0. Episode Transformer, HRM, and TRM use PyTorch; LPN uses JAX. Each training run uses one accelerator, although multiple runs may share a node. Comp-II uses full H100 GPUs; the reported CompARC and CompCIFAR measurements use H100 2g.20gb MIG instances. Final search is CPU-parallel and is reported in CPU core-hours.

Table 32: Representative NeuLP compute measurements. H denotes full-H100 hours; M denotes H100 2g.20gb MIG instance-hours. Final search uses 128 starts and up to 100 rounds per start and is reported in CPU core-hours.

| Domain | Stage 1 | Stages 2–3 | Final search |
|---|---|---|---|
| Comp-II | $\sim$ 2.63 H | $\sim$ 2.30 H | 1,727 CPU core-h |
| CompARC | 69.72–71.98 M | 2.84–3.03 M | 1,094 CPU core-h |
| CompCIFAR | 2.48–2.51 M | 604 CPU core-h + 0.77 M | 5,747 CPU core-h |
| CompLife (early seed 4) | 78.71 M | 363 CPU core-h + 1.02 H | 2,545 CPU core-h |

The final-search measurements cover 321 Comp-II, 322 CompARC, 198 CompCIFAR, and 338 CompLife tasks. These are representative measured runs rather than all-seed averages.

### C.8 POST-HOC SEMANTIC AUDIT

Token names are assigned only after training. Each execution-supported token is run alone on held-out D1 episodes. In the categorical domains, a token is given a DSL name only when it executes that primitive exactly on every audit pair; all exact aliases are retained. CompCIFAR uses the held-out normalized $\ell_1$ improvement of one-token executions to assign approximate edit interpretations. Token IDs are checkpoint-specific, so every displayed program uses the mapping from its own learned codebook.

Table 33: Post-hoc token-to-D1 audit for the selected checkpoints. Categorical rows list exact standalone semantic aliases. CompCIFAR rows give one-token interpretations on the same checkpoint/codebook used for the main qualitative example; because image outputs are continuous, these are approximate semantic matches rather than exact symbolic equivalences.

| Domain | D1 operation | Learned token ID(s) |
|---|---|---|
| Comp-II | `deinterleave` | 110 |
| Comp-II | `interleave` | 292 |
| Comp-II | `reverse` | 21 |
| Comp-II | `shift_1` | 354 |
| Comp-II | `shift_2` | 100 |
| Comp-II | `shift_3` | 106 |
| Comp-II | `swap_adjacent` | 197 |
| Comp-II | `swap_halves` | 425 |
| CompARC | `tx+1` | 115, 396 |
| CompARC | `ty+1` | 334, 507 |
| CompARC | `recolor1to9` | 266, 392 |
| CompARC | `move6_+1_+0` | 92, 362 |
| CompARC | `delete7` | 63, 262 |
| CompARC | `copy4_+1_+0` | 221 |
| CompARC | `rotate_cw` | 354 |
| CompARC | `dilate1` | 25 |
| CompLife | `life` | 85, 295 |
| CompLife | `replicator` | 446 |
| CompLife | `assimilation` | 116 |
| CompLife | `maze` | 153, 408 |
| CompLife | `daynight` | 266 |
| CompLife | `2x2` | 389, 485 |
| CompCIFAR | `brightness_minus` | 364 |
| CompCIFAR | `brightness_plus` | 140 |
| CompCIFAR | `horizontal_flip` | 342 |
| CompCIFAR | `hue_minus` | 316 |
| CompCIFAR | `hue_plus` | 167, 351 |
| CompCIFAR | `mask` | 24 |
| CompCIFAR | `rotation_cw` | 195 |
| CompCIFAR | `vertical_flip` | 497 |

Primitive names are used only for this post-hoc audit. The audited Comp-II BEEP checkpoints cover all eight D1 primitives with stable standalone tokens. The CompARC and CompLife rows show one selected checkpoint per domain; each reported model is calibrated on its own codebook. In CompCIFAR, every listed token improves over the input-copy baseline for its assigned one-step edit. Tokens 167 and 351 both map to Hue-+, with token 167 giving the lower standalone error.

Aliases in Table 33 are distinct codebook rows with the same verified one-step behavior; a macro token can represent a multi-primitive transformation. Both are compatible with input-output supervision, so the learned language need not recover a unique symbolic DSL.

### C.9 QUALITATIVE REPORTING PROTOCOL

Qualitative examples are selected after fixing the reported checkpoints and search budgets. The appendix shows six examples per 2D split. For categorical domains with exact rate $r$, we select $\mathrm{round}(6r)$ successes and fill the remaining rows with failures, using the other stratum if one quota cannot be filled. CompCIFAR also uses six examples per split, chosen to cover interpretable learned tokens and several composition lengths. Each OOC panel includes three examples whose non-SKIP program changes under search and whose query $\ell_1$ improves. The main-text example is excluded. Table 34 summarizes the selection rule.

Table 34: Qualitative reporting setup. For categorical domains, the six-row panels use the rounded success/failure ratio from the full quantitative result.

| Panel | Population and endpoint | Selection | Contents |
|---|---|---|---|
| Main categorical | Frozen test split; final reported endpoint | Exact-success examples only; no frequency claim | Input, target, predicted output, and typeset ground-truth/predicted programs |
| Appendix categorical 2D Test I/O | Full Test I/O population; Stage-3 zero-shot | Six rows with $\mathrm{round}(6r)$ exact successes | Stage-1 soft and hard, Stage-3 hard, and typeset programs |
| Appendix categorical 2D OOC | Full recombination or length test population; final searched endpoint | Same six-row rounded-ratio rule | Stage-1 soft and hard, Stage-3 hard, searched output, and typeset programs |
| Appendix CompCIFAR | Full split; Stage-3 zero-shot for Test I/O and searched endpoint for OOC | Six qualitative examples; explained tokens prioritized; three changed, query-improved programs per OOC panel; main-text task excluded | Input, target, Stage-1/Stage-3/final output, edit family, and typeset programs |

OOC panels use the final searched endpoint, matching the main tables; Test I/O uses the Stage 3 zero-shot endpoint. Selection is deterministic. Categorical rows show the input, target, Stage 1 soft and hard outputs, Stage 3 hard output, and the final searched output for OOC. The corresponding ground-truth, Stage 1, Stage 3, and searched programs are typeset below each figure. Comp-II remains in text form rather than rendering integer sequences as images.

For a Stage-1 failure repaired by Stage 3, we report

$$\begin{aligned} Z_{\mathrm{S1}} : [k_1, \ldots, k_T] \quad &\mapsto \quad [g(k_1), \ldots, g(k_T)], \\ Z_{\mathrm{S3}} : [k'_1, \ldots, k'_T] \quad &\mapsto \quad [g(k'_1), \ldots, g(k'_T)], \end{aligned}$$

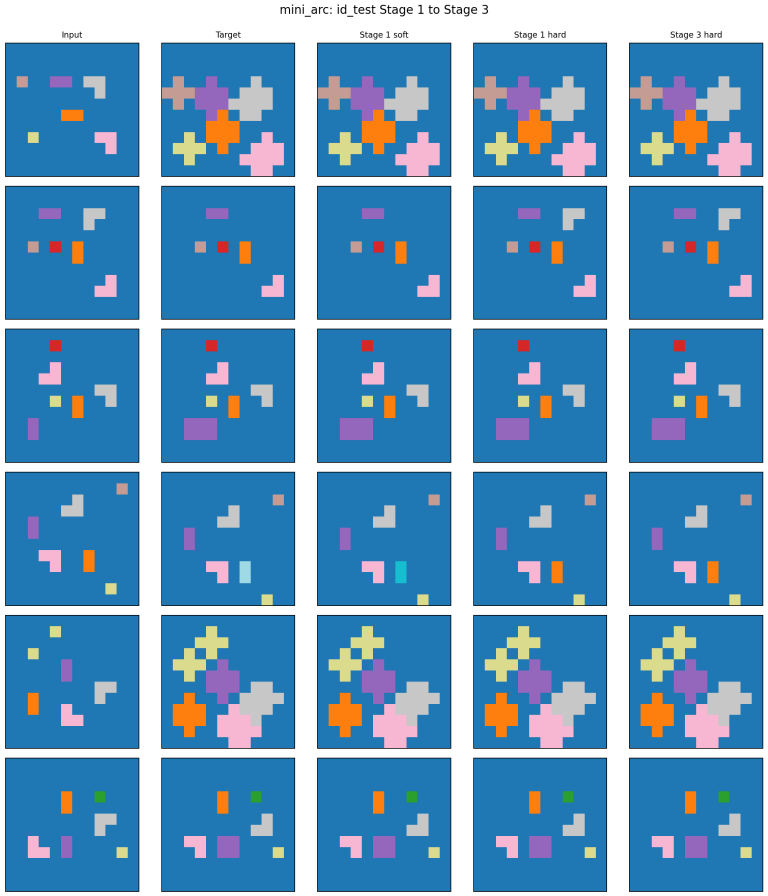
where $g$ is the calibrated mapping in Table 33. Unmapped tokens are printed as $\texttt{TOKEN-}k$. Ground-truth DSL traces are shown only for comparison. If Stage 3 fails but final search succeeds, we also show $Z_{\mathrm{search}}$. Among successful examples, selection first prefers Stage 1 failures repaired by Stage 3, then Stage 3 failures repaired by search. The requested success/failure ratio is unchanged.

### C.10 COMPARC GALLERY

**CompARC final gallery.** Each split contains six deterministic examples with the rounded final success/failure ratio. OOC programs come from the completed search with 128 initial programs and up to 100 rounds per start; Test I/O uses Stage 3 zero-shot programs. Token names come from the same checkpoint's standalone execution audit.

| D1 operation | audited standalone token ID(s) | held-out criterion |
|---|---|---|
| copy4_+1_+0 | 231 | exact on every held-out audit pair |
| delete7 | 89, 410 | exact on every held-out audit pair |
| dilate1 | 374 | exact on every held-out audit pair |
| move6_+1_+0 | 472 | exact on every held-out audit pair |
| recolor1to9 | 71, 368 | exact on every held-out audit pair |
| rotate_cw | 168 | exact on every held-out audit pair |
| tx+1 | 21 | exact on every held-out audit pair |
| ty+1 | 426 | exact on every held-out audit pair |

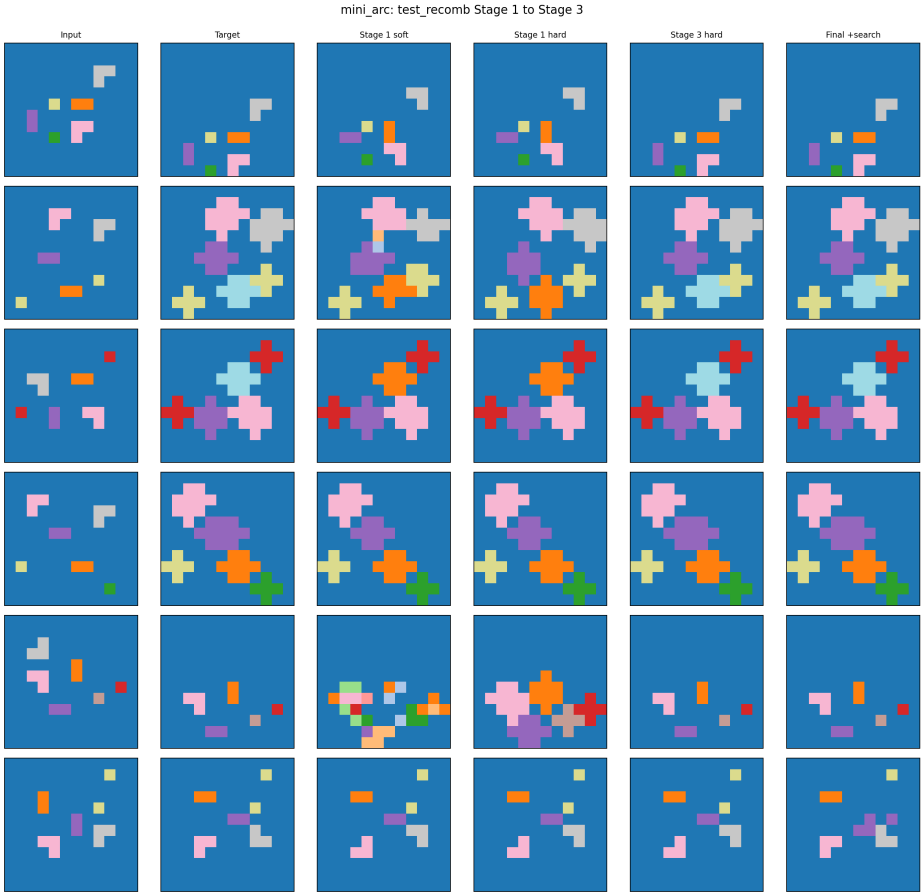
**CompARC: Test I/O.**




| Row | Ground-truth DSL | Stage 1 prediction | Stage 3 prediction | Final program | Result |
|---|---|---|---|---|---|
| 1 | `ty+1 → rotate_cw → dilate1` | `TOKEN-367` | `TOKEN-367` | `TOKEN-367` | S3 exact |
| 2 | `tx+1 → delete7` | `tx+1` | `tx+1` | `tx+1` | failure |
| 3 | `move6_+1_+0 → copy4_+1_+0 → copy4_+1_+0` | `TOKEN-193 → TOKEN-91` | `TOKEN-193 → TOKEN-91` | `TOKEN-193 → TOKEN-91` | S3 exact |
| 4 | `ty+1 → recolor1to9 → move6_+1_+0` | `TOKEN-147` | `TOKEN-147` | `TOKEN-147` | failure |
| 5 | `ty+1 → dilate1 → move6_+1_+0` | `TOKEN-409` | `TOKEN-409` | `TOKEN-409` | S3 exact |
| 6 | `copy4_+1_+0 → rotate_cw → rotate_cw` | `TOKEN-16` | `TOKEN-16` | `TOKEN-16` | S3 exact |

*Split:* `id_test`.

**CompARC: recombination OOC.**




| Row | Ground-truth DSL | Stage 1 prediction | Stage 3 prediction | Final program | Result |
|---|---|---|---|---|---|
| 1 | `ty+1` → `ty+1` → `ty+1` | `TOKEN-442` → `ty+1` | `TOKEN-442` → `TOKEN-402` → `TOKEN-442` | `TOKEN-442` → `TOKEN-402` → `TOKEN-442` | S1 fail → S3 exact |
| 2 | `tx+1` → `recolor1to9` → `dilate1` | `TOKEN-470` | `tx+1` → `TOKEN-485` → `dilate1` → `TOKEN-196` | `tx+1` → `TOKEN-485` → `dilate1` → `TOKEN-196` | S1 fail → S3 exact |
| 3 | `recolor1to9` → `delete7` → `dilate1` | `TOKEN-370` | `TOKEN-370` → `TOKEN-5` | `TOKEN-370` → `TOKEN-5` | S1 fail → S3 exact |
| 4 | `delete7` → `copy4_+1_+0` → `dilate1` | `TOKEN-370` | `delete7` → `TOKEN-94` | `delete7` → `TOKEN-94` | S1 fail → S3 exact |
| 5 | `ty+1` → `ty+1` → `delete7` | `ty+1` → `TOKEN-22` | `TOKEN-402` → `TOKEN-448` → `TOKEN-396` → `TOKEN-471` → `delete7` | `TOKEN-402` → `TOKEN-448` → `TOKEN-396` → `TOKEN-471` → `delete7` | S1 fail → S3 exact |
| 6 | `rotate_cw` → `rotate_cw` → `rotate_cw` | `rotate_cw` | `rotate_cw` | `rotate_cw` → `TOKEN-123` | failure |

*Split:* `test_recomb`.

**CompARC: length OOC.**



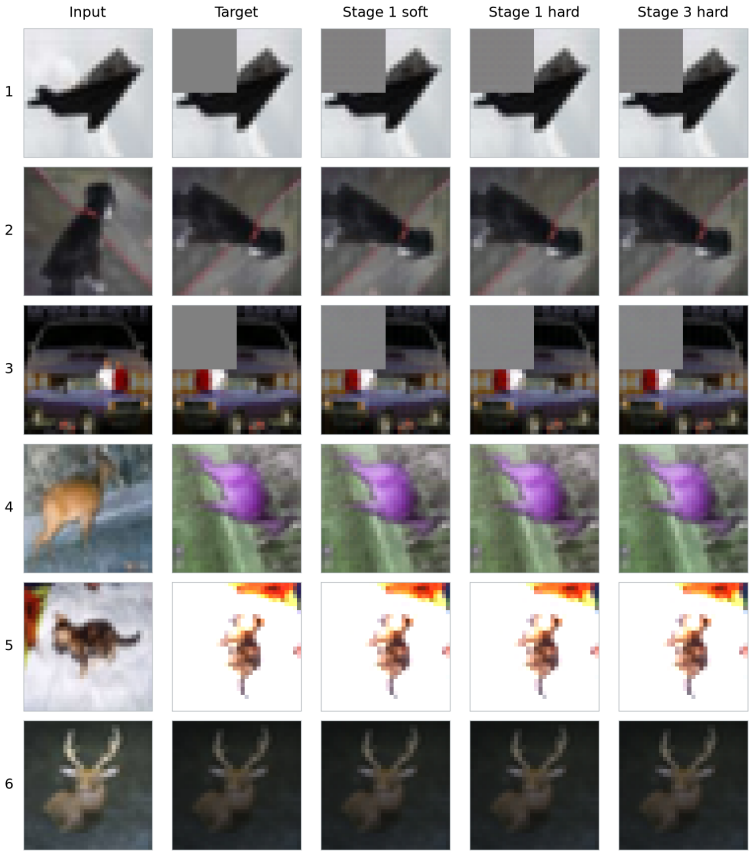

| Row | Ground-truth DSL | Stage 1 prediction | Stage 3 prediction | Final program | Result |
|---|---|---|---|---|---|
| 1 | copy4_+1_+0 → delete7 → dilate1 → move6_+1_+0 → move6_+1_+0 | TOKEN-174 | delete7 → TOKEN-193 → TOKEN-255 → TOKEN-370 | delete7 → TOKEN-193 → TOKEN-255 → TOKEN-370 | S1 fail → S3 exact |
| 2 | dilate1 → delete7 → tx+1 → ty+1 → recolor1to9 → copy4_+1_+0 | TOKEN-490 → TOKEN-490 | TOKEN-448 → TOKEN-448 → TOKEN-114 → delete7 → TOKEN-196 | TOKEN-448 → TOKEN-448 → TOKEN-114 → delete7 → TOKEN-196 | S1 fail → S3 exact |
| 3 | ty+1 → delete7 → dilate1 → move6_+1_+0 → ty+1 | TOKEN-409 | TOKEN-448 → TOKEN-402 → TOKEN-409 → TOKEN-370 → dilate1 | TOKEN-448 → TOKEN-402 → TOKEN-409 → TOKEN-370 → dilate1 | S1 fail → S3 exact |
| 4 | copy4_+1_+0 → delete7 → dilate1 → rotate_cw → ty+1 | TOKEN-490 → TOKEN-490 | ty+1 → TOKEN-402 → TOKEN-409 → TOKEN-409 | TOKEN-51 → dilate1 → TOKEN-133 → TOKEN-223 → TOKEN-30 → TOKEN-370 | failure |
| 5 | ty+1 → move6_+1_+0 → dilate1 → copy4_+1_+0 | TOKEN-409 | TOKEN-147 → TOKEN-196 → TOKEN-94 | TOKEN-147 → TOKEN-196 → TOKEN-94 | S1 fail → S3 exact |
| 6 | delete7 → move6_+1_+0 → ty+1 → tx+1 → move6_+1_+0 → rotate_cw | TOKEN-415 → tx+1 | delete7 → TOKEN-359 → TOKEN-359 → tx+1 | ty+1 → delete7 → TOKEN-81 → delete7 → TOKEN-510 | failure |

*Split:* `test_length`.

### C.11 CompCIFAR gallery

**CompCIFAR: programs and execution.** Six examples are shown for each split from one selected PatchViT-2 checkpoint. Only non-SKIP tokens are printed; unmapped tokens remain unlabeled. B/H denote brightness/hue, HF/VF horizontal/vertical flip, M mask, and R clockwise rotation. OOC panels include three examples where search changes the program and improves query $\ell_1$; these programs are bolded. The main-text example is excluded.

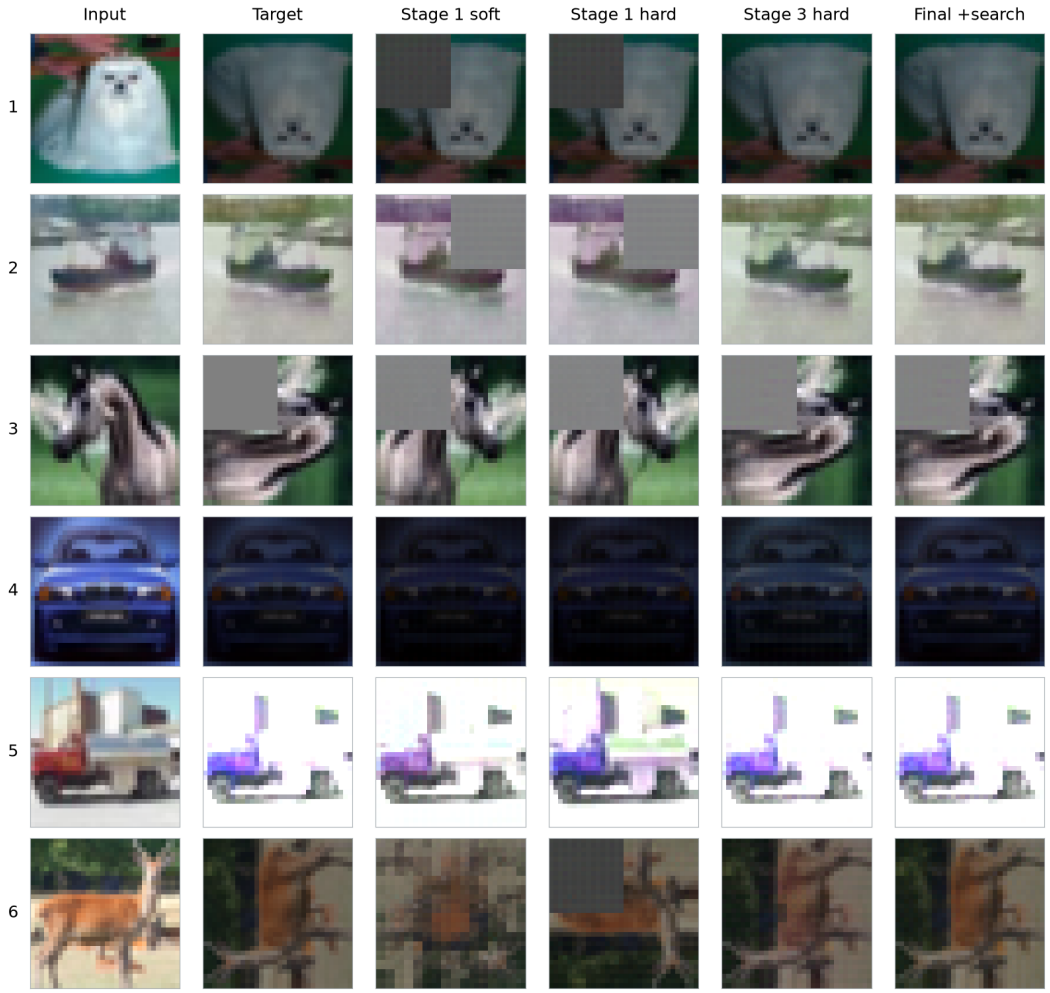
**Test I/O**




| Row | Target DSL | Stage 1 hard | Stage 3 hard | Final +search |
|---|---|---|---|---|
| 1 | M | [24] | [24] | – |
| | lv1 | $\ell_1$: 0.00345 | $\ell_1$: 0.00345 | – |
| 2 | R | [195] | [189] | – |
| | lv1 | $\ell_1$: 0.00444 | $\ell_1$: 0.00585 | – |
| 3 | HF → M | [173] | [173] | – |
| | lv2 | $\ell_1$: 0.01014 | $\ell_1$: 0.01014 | – |
| 4 | H− → R | [383] | [226] | – |
| | lv2 | $\ell_1$: 0.00737 | $\ell_1$: 0.00809 | – |
| 5 | B+ → B+ → R | [292] | [292] | – |
| | lv3 | $\ell_1$: 0.00361 | $\ell_1$: 0.00361 | – |
| 6 | H− → B− → H+ | [364] | [364] | – |
| | lv3 | $\ell_1$: 0.00232 | $\ell_1$: 0.00232 | – |

Post-hoc one-token correspondences for this checkpoint: $e_{364} \approx$ B−; $e_{140} \approx$ B+; $e_{342} \approx$ HF; $e_{316} \approx$ H−; $e_{167} \approx$ H+; $e_{24} \approx$ M; $e_{195} \approx$ R; $e_{497} \approx$ VF. Test I/O uses the Stage-3 program without search.

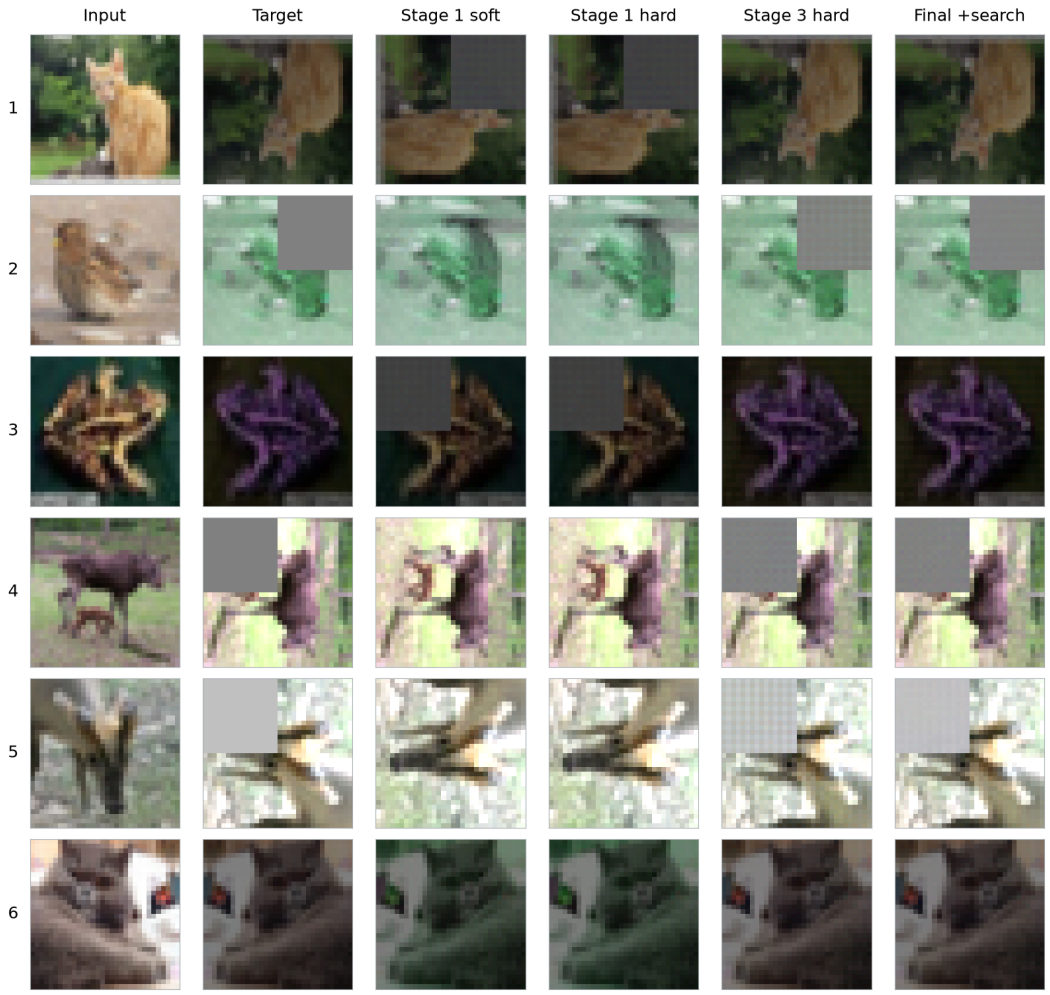
### Recombination OOC




| Row | Target DSL | Stage 1 hard | Stage 3 hard | Final +search |
|---|---|---|---|---|
| 1 | B− → VF<br>lv2 | [166]<br>$\ell_1$: 0.02627 | [497, 364]<br>$\ell_1$: 0.00692 | [497, 364]<br>$\ell_1$: 0.00692 |
| 2 | H− → HF<br>lv2 | [142]<br>$\ell_1$: 0.06062 | [400, 226]<br>$\ell_1$: 0.01076 | **[316, 342]**<br>$\ell_1$: **0.00588** |
| 3 | R → M<br>lv2 | [173]<br>$\ell_1$: 0.20490 | [195, 24]<br>$\ell_1$: 0.00645 | [195, 24]<br>$\ell_1$: 0.00645 |
| 4 | B− → B− → B+<br>lv3 | [19]<br>$\ell_1$: 0.04340 | [182, 157, 470, 364]<br>$\ell_1$: 0.01285 | **[380, 364]**<br>$\ell_1$: **0.00595** |
| 5 | B+ → B+ → H−<br>lv3 | [179]<br>$\ell_1$: 0.09175 | [140, 140, 316]<br>$\ell_1$: 0.00616 | [140, 140, 316]<br>$\ell_1$: 0.00616 |
| 6 | B− → R → HF<br>lv3 | [166]<br>$\ell_1$: 0.11749 | [342, 212, 12, 316, 430]<br>$\ell_1$: 0.01351 | **[195, 189, 342, 414]**<br>$\ell_1$: **0.00801** |

Search uses support I/O only, with 128 starts, up to 100 rounds per start, and $\epsilon = 0.20$. Across the 18-task panel, query $\ell_1$ changes from 0.00893 before search to 0.00798 after search.

## Length OOC




| Row | Target DSL | Stage 1 hard | Stage 3 hard | Final +search |
|---|---|---|---|---|
| 1 | VF → H− → B− → H+<br>lv4 | [511]<br>$\ell_1$: 0.10388 | [497, 364]<br>$\ell_1$: 0.00579 | [497, 364]<br>$\ell_1$: 0.00579 |
| 2 | H+ → R → M → R<br>lv4 | [311]<br>$\ell_1$: 0.03664 | [195, 24, 212, 95]<br>$\ell_1$: 0.00944 | **[497, 24, 142]**<br>$\ell_1$: **0.00651** |
| 3 | HF → B− → H− → HF → HF<br>lv5 | [356]<br>$\ell_1$: 0.05627 | [316, 342, 364]<br>$\ell_1$: 0.00882 | [316, 342, 364]<br>$\ell_1$: 0.00882 |
| 4 | VF → R → HF → B+ → M<br>lv5 | [92]<br>$\ell_1$: 0.08525 | [140, 189, 485, 326]<br>$\ell_1$: 0.01366 | **[140, 195, 24]**<br>$\ell_1$: **0.01224** |
| 5 | HF → VF → M → B+ → VF → R<br>lv6 | [92]<br>$\ell_1$: 0.21283 | [322, 497, 273, 376, 189]<br>$\ell_1$: 0.01835 | **[195, 326, 140]**<br>$\ell_1$: **0.00895** |
| 6 | H+ → HF → H− → VF → VF → B−<br>lv6 | [290]<br>$\ell_1$: 0.02749 | [342, 364]<br>$\ell_1$: 0.00439 | [342, 364]<br>$\ell_1$: 0.00439 |

Search uses support I/O only, with 128 starts, up to 100 rounds per start, and $\epsilon = 0.20$. Across the 180-task panel, query $\ell_1$ changes from 0.01212 before search to 0.01121 after search.

### C.12 COMPLIFE GALLERY

**CompLife: learned operations and final-search examples.**

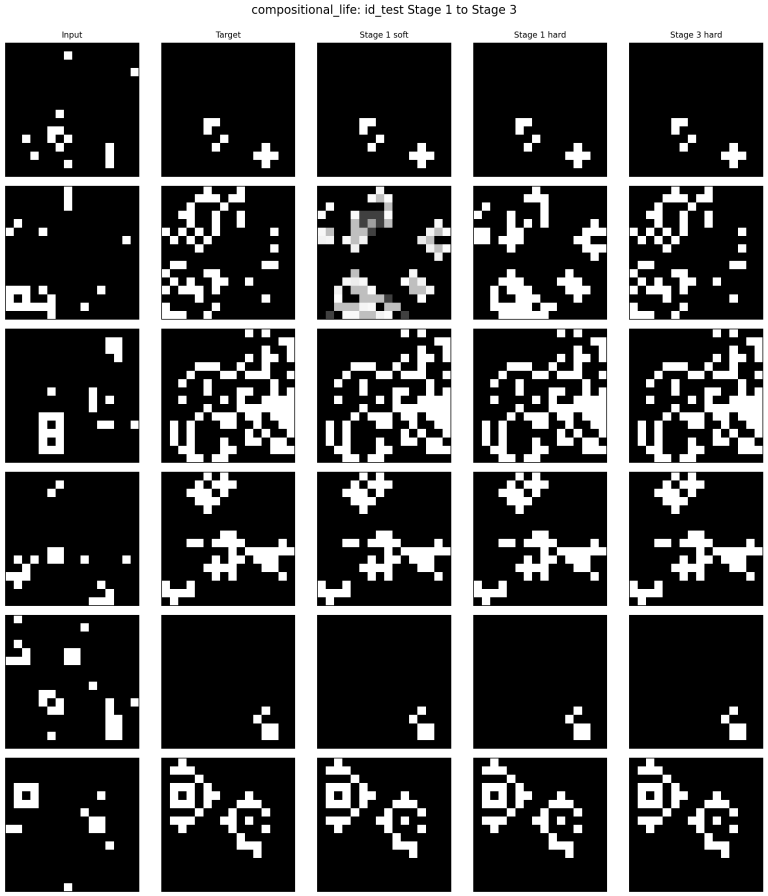
We show one selected model on clipped $16 \times 16$ grids. Stage 1 and Stage 3 share the same execution pathway and differ only in the programmer. Token names are assigned post hoc by exact standalone execution on held-out audit pairs; unmapped tokens remain `TOKEN-`$k$.

| D1 operation | audited standalone token ID(s) | held-out criterion |
|---|---|---|
| `2x2` | `69`, `203` | exact on every held-out audit pair |
| `assimilation` | `210`, `346` | exact on every held-out audit pair |
| `daynight` | `57`, `140`, `386`, `442` | exact on every held-out audit pair |
| `life` | `25`, `145`, `158` | exact on every held-out audit pair |
| `maze` | `67`, `283`, `329`, `347`, `350`, `382`, `492` | exact on every held-out audit pair |
| `replicator` | `108`, `291` | exact on every held-out audit pair |

Each split shows six deterministic examples with the success/failure ratio rounded from the main-table accuracy. The Test I/O, recombination, and length panels therefore contain $6/6$, $5/6$, and $4/6$ exact endpoints. The shown checkpoint solves $36/38$ recombination tasks and $237/300$ length tasks after search. Among successful examples, selection prioritizes cases repaired by Stage 3.

Test I/O uses the hard Stage 3 program, while the main table reports soft-program Test I/O accuracy. The gallery uses one fixed episode per training identity. OOC panels use the final support-selected search programs with 128 starts, up to 100 rounds per start, and $\epsilon = 0.20$. Query targets are used only for evaluation; the supports may overlap the Stage 2 bank.

**CompLife: Test I/O.**

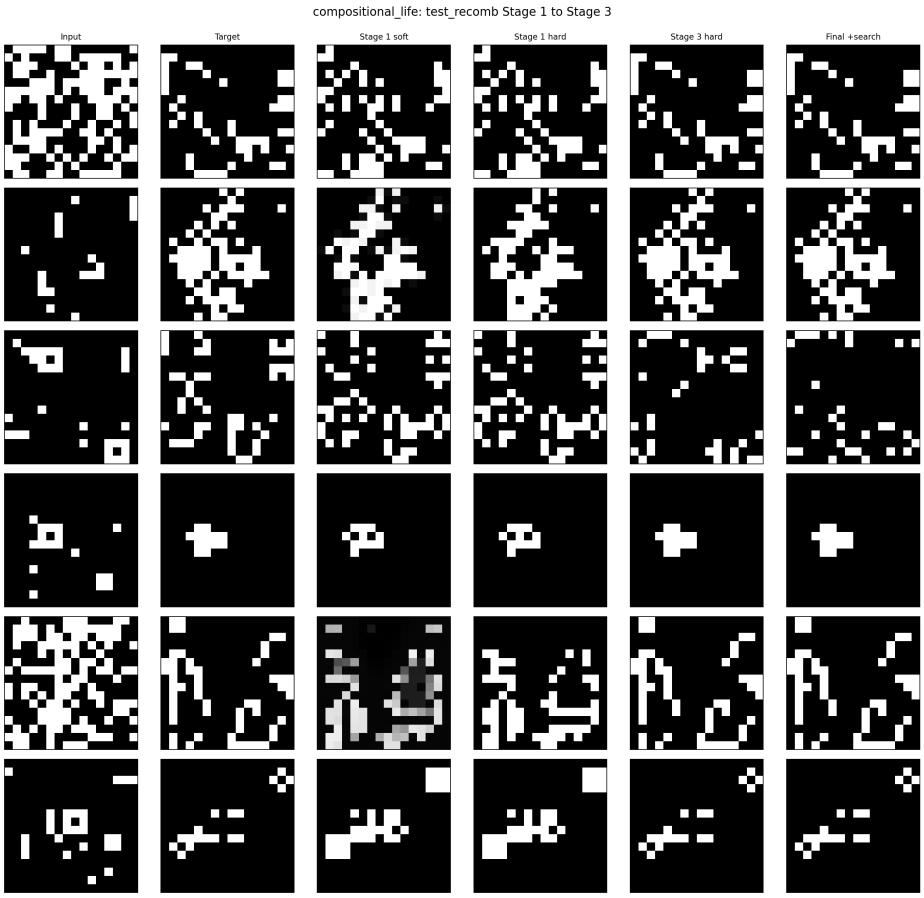



Six exact hard-program endpoints; the corresponding programs are listed below.

| Row | Ground-truth DSL | Stage 1 prediction | Stage 3 prediction | Final program | Result |
|---|---|---|---|---|---|
| 1 | `2x2` | `2x2` | `2x2` | `2x2` | S3 exact |
| 2 | `replicator → replicator → 2x2` | `replicator → life → TOKEN-58` | `replicator → replicator → TOKEN-58` | `replicator → replicator → TOKEN-58` | S1 fail → S3 exact |
| 3 | `replicator → 2x2 → replicator` | `replicator → 2x2 → replicator` | `replicator → 2x2 → replicator` | `replicator → 2x2 → replicator` | S3 exact |
| 4 | `replicator → life → maze` | `replicator → life → maze` | `replicator → life → maze` | `replicator → life → maze` | S3 exact |
| 5 | `assimilation → assimilation → daynight` | `assimilation → assimilation → daynight` | `assimilation → assimilation → daynight` | `assimilation → assimilation → daynight` | S3 exact |
| 6 | `life → replicator → life` | `life → replicator → life` | `life → replicator → life` | `life → replicator → life` | S3 exact |

*Split:* `id_test`.

**CompLife: recombination OOC.**

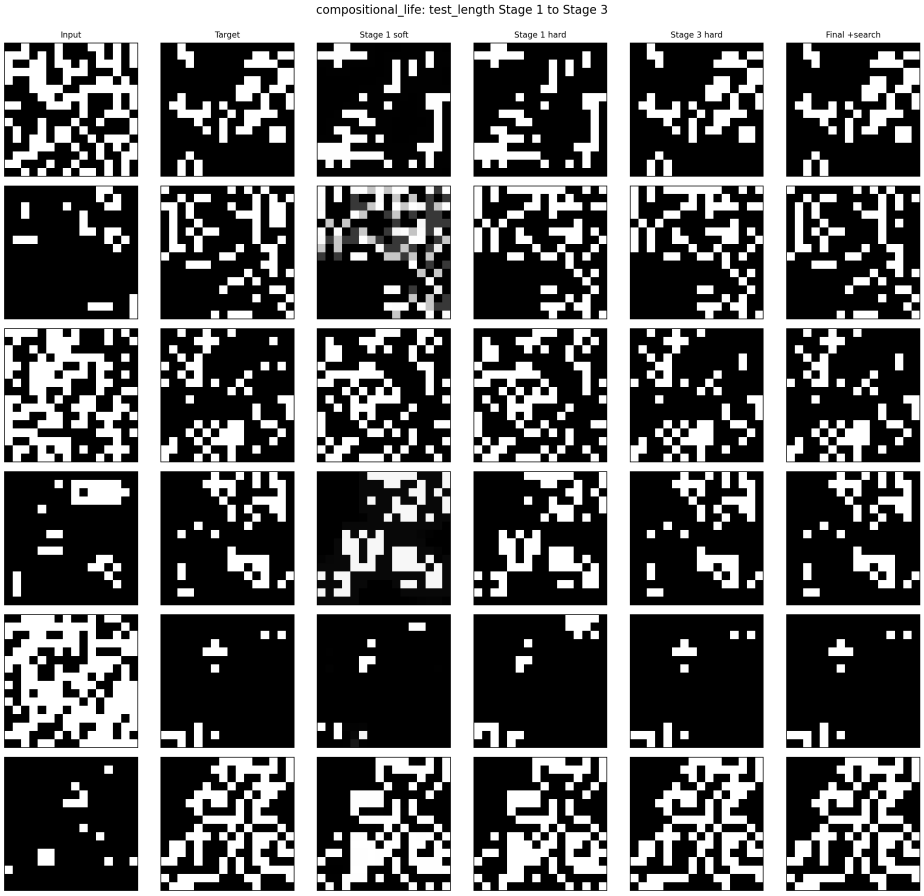



Five exact final endpoints and one failure (row 3). The final column reports the search endpoint, including unchanged proposals.

| Row | Ground-truth DSL | Stage 1 prediction | Stage 3 prediction | Final program | Result |
|---|---|---|---|---|---|
| 1 | daynight → life → life | daynight → life → 2x2 | daynight → life → life | daynight → life → life | S1 fail → S3 exact |
| 2 | replicator → assimilation → assimilation | replicator → assimilation → daynight | replicator → assimilation → TOKEN-349 | replicator → assimilation → TOKEN-349 | S1 fail → S3 exact |
| 3 | replicator → assimilation → life | replicator → life | assimilation → replicator → life | TOKEN-148 → TOKEN-481 → TOKEN-481 → TOKEN-481 → TOKEN-148 → TOKEN-58 | failure |
| 4 | assimilation → assimilation → assimilation | assimilation → assimilation → daynight | assimilation → assimilation → assimilation | assimilation → assimilation → assimilation | S1 fail → S3 exact |
| 5 | life → life → life | life → assimilation → TOKEN-111 | life → life → TOKEN-313 | life → life → TOKEN-313 | S1 fail → S3 exact |
| 6 | 2x2 → assimilation → life | 2x2 → assimilation → maze | 2x2 → assimilation → life | 2x2 → assimilation → life | S1 fail → S3 exact |

*Split:* `test_recomb`.

**CompLife: length OOC.**




Four exact final endpoints and two failures (rows 2–3) under exact grid accuracy.

| Row | Ground-truth DSL | Stage 1 prediction | Stage 3 prediction | Final program | Result |
|---|---|---|---|---|---|
| 1 | 2x2 → daynight → life → life → daynight | life → life → assimilation | 2x2 → daynight → life → life → TOKEN-432 | 2x2 → daynight → life → life → TOKEN-432 | S1 fail → S3 exact |
| 2 | replicator → life → replicator → maze → 2x2 | replicator → replicator → TOKEN-58 | replicator → replicator → TOKEN-58 | replicator → life → TOKEN-148 → maze → TOKEN-58 | failure |
| 3 | maze → life → maze → 2x2 | daynight → replicator → 2x2 | maze → TOKEN-313 → maze → TOKEN-58 | maze → TOKEN-313 → maze → TOKEN-58 | failure |
| 4 | daynight → replicator → life → 2x2 | assimilation → replicator | daynight → replicator → life → TOKEN-58 | daynight → replicator → life → TOKEN-58 | S1 fail → S3 exact |
| 5 | life → life → life → assimilation → assimilation | life → life → assimilation | life → TOKEN-313 → life → TOKEN-349 → TOKEN-349 | life → TOKEN-313 → life → TOKEN-349 → TOKEN-349 | S1 fail → S3 exact |
| 6 | replicator → 2x2 → replicator → maze | replicator → 2x2 → replicator | replicator → 2x2 → replicator → maze | replicator → 2x2 → replicator → maze | S1 fail → S3 exact |

*Split:* `test_length`.