PRECOCIAL: PRETRAINING WITH CODE CURRICULUMS VIA ITERATIVE SEARCH FOR ACCELERATED LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Fully synthetic pretraining data generated by symbolic programs can bootstrap downstream training on naturalistic tasks, such as language modeling, but the generating programs have so far been designed by hand. We introduce PRECOCIAL, a method for automatically optimizing programs that generate synthetic pretraining data. Starting from an initial generator program, an LLM iteratively rewrites the program, and each rewrite is scored by pretraining a randomly-initialized transformer on its data and fine-tuning it on the downstream task. Because every score requires training, PRECOCIAL keeps a single network for the entire search, continuing its pretraining for each accepted program instead of training a new network per candidate, making it significantly more efficient. On four language and code datasets, at a fixed fine-tuning budget, searched programs reach lower perplexity than hand-designed curricula, published procedural generators, random program mutation and the LLM's own first draft, and reach the same perplexity as the hand-designed curricula with 9–33% fewer fine-tuning steps. We also observe that using a searched program generator from one domain transfers to datasets in other domains. Additionally, searched programs allow for better downstream performance on four different planning tasks.

1 INTRODUCTION

Large Language Models (LLMs) begin as randomly-initialized transformer networks before pretraining on masses of human data. Must training begin as a randomly-initialized blank slate? Biological learning suggests otherwise. From birth, wildebeests can run and spiders can weave webs, drawing only on their genetic endowment: a compact discrete code (DNA) prewires useful behaviors and biases. Analogously, here we set out to discover compact symbolic programs which prime a randomly-initialized transformer with useful inductive biases, which we call *pretraining programs*.

Previous work explores two families of pretraining programs: procedures which directly generate weights and biases (Stanley et al., 2009; Barabási et al., 2023), and programs which generate synthetic data (Zhang et al., 2025; Jiang et al., 2026) for "pre-pretraining." Both families could be fruitful, but synthetic data is presently more practical and better-aligned with modern LLM pipelines, and so we adopt it here (Figure 1). Recent work suggests that such pretraining programs can improve both the sample and computational complexity of downstream learning: first training on cellular automata, formal languages, or logical inferences then accelerates downstream learning in natural language, visual reasoning, program source code, and more (Zhang et al., 2025; Hu et al., 2025; Jiang et al., 2026; Wu et al., 2021; Lee et al., 2026). In all of this work the program is designed by hand. This prior literature naturally suggests our core research question: how could a pretraining program be learned, synthesized, evolved, or automatically improved? This question also has immediate implications for synthetic training data more broadly, as synthetic post-training has become an important way of teaching language models to reason (Trinh et al., 2024; Zhao et al., 2025).

To learn a pretraining program, we assume a collection of downstream datasets, such as natural language corpora or planning problems. These downstream tasks define an objective for pretraining programs, because we can measure how well a program's synthetic data equips the learner to tackle downstream data. Optimizing this objective is challenging for two reasons: the program search space

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.

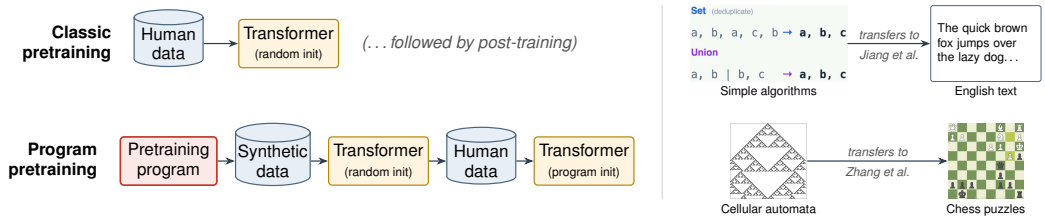


Figure 1: Program pretraining generates synthetic training data to initialize the network prior to exposure to human data. Right panel: representative prior works (Jiang et al., 2026; Zhang et al., 2025).

is discrete, and scoring a single program is expensive, since it requires pretraining a transformer on the program's data and then fine-tuning it on the downstream task. LLM-guided program search such as FunSearch and AlphaEvolve (Romera-Paredes et al., 2024; Novikov et al., 2025) score each candidate independently, which is only affordable when the evaluation is cheap. Here, it would mean pretraining a transformer from scratch for every candidate.

To address these challenges we introduce PRECOCIAL (PREtraining with COde Curriculums via Iterative search for Accelerated Learning), a method for synthesizing and optimizing pretraining programs. An LLM rewrites the program, each rewrite is scored by training on its data, and a rewrite is kept only if it improves the downstream objective. To make each score cheap, PRECOCIAL keeps a *single* network for the whole search: every accepted program continues pretraining the network where the previous one left off, and a rejected program is rolled back (Section 3.2). Our goal is *not* to show that pretraining programs can help downstream tasks, which is already well-established. Instead, our primary goal is to evaluate whether such pretraining programs can be automatically optimized for a dataset of interest, and whether such optimization *transfers* across datasets. This paper makes the following contributions:

- **Algorithm.** PRECOCIAL: LLM-guided search over pretraining programs whose fitness is the downstream performance of a network trained on the program's data (§3).
- **Efficient continual pretraining.** PRECOCIAL maintains a single model that is continuously pretrained across candidate programs. The protocol applies to any program search whose fitness requires training a network: at a fixed fine-tuning budget it is both cheaper and better than training a fresh model per candidate, for FunSearch and AlphaEvolve as well (§4.4).
- **Empirical validation.** On four language and code datasets, searched programs reach lower perplexity than every other source of a curriculum we test (hand-written, procedural, random mutation, the LLM's first draft) at every fine-tuning budget we measure; they match FunSearch and AlphaEvolve and transfer to unseen datasets. At a 10k-step fine-tuning budget, PRECOCIAL wins on all four planning and search puzzles, and a Blocksworld search that never sees the task does as well as one that does (§4).

2 Background and Related Work

Structural synthetic pretraining. A growing body of work shows that pretraining on structured synthetic data teaches neural networks inductive biases that transfer to natural tasks. Cellular automata at the edge of chaos yield models with strong compositional structure (Zhang et al., 2025); neural cellular automata seed persistent memory circuits (Lee et al., 2026); and the phenomenon extends to vision transformers (Shinnick et al., 2025). On the formal language side, Dyck languages and context-free grammars teach bracket tracking and hierarchical composition (Hu et al., 2025; Papadimitriou & Jurafsky, 2020), while LIME's synthetic inference tasks aid compositional generalisation (Wu et al., 2021). A common theme is that *abstract structural patterns*, not semantic content, drive transfer, and the benefit is usually one of efficiency: pre-pretraining on formal languages lets a 1B-parameter model reach the final loss of a model without it with 33% fewer tokens (Hu et al., 2025), and procedural data lets models reach the same loss with 55–86% of the original data (Jiang et al., 2026). This is also mostly what pretraining on real data buys: detectors trained from scratch on COCO match ImageNet-pretrained ones once given a longer schedule (He et al., 2019), and pretraining helps most, relative to the fine-tuning data, when that data is scarce (Hernandez et al., 2021). All of these generators are fixed by their designers. We compare against the

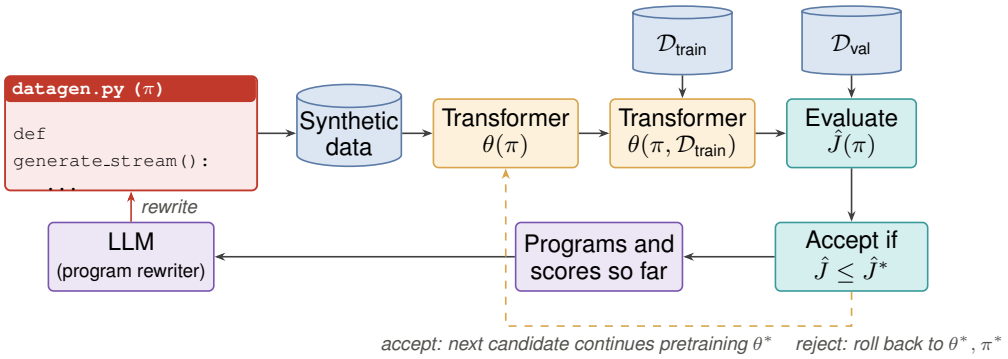


Figure 2: PRECOCIAL. The LLM rewrites the generator program; the rewrite is scored by continuing to pretrain the one network kept for the whole search, fine-tuning a copy on $\mathcal{D}_{\text{train}}$ and evaluating it on $\mathcal{D}_{\text{val}}$. An improvement is accepted; otherwise both the program and the network are rolled back to the last accepted state.

procedural generators of Jiang et al. (2026) directly, and report our gains in the same currency of steps saved.

LLM-guided program search. FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) use LLMs as mutation operators over programs, maintaining populations selected by black-box fitness, and are applied where a candidate is scored by running it. PRECOCIAL uses the same LLM-as-mutator paradigm, but its fitness requires training a network. We therefore search over about ten candidates rather than thousands, keep one continually pretrained network across them, and compare against both methods at the same budget.

Learning to generate training data. Generative Teaching Networks (Such et al., 2020) learn a neural data generator by meta-gradients; Montessori-Instruct (Li et al., 2025) trains a teacher LLM on instruction data using a student's data influence; DataEvolve (Mi et al., 2026) evolves curation strategies for natural pretraining data with trained models as fitness; CADEvolve (Elistratov et al., 2026) evolves data-generating programs selected by validity checks; and Eurekaverse (Liang et al., 2024) has an LLM write training environments scored by a reinforcement learner. Our work is also related to meta-learning (Finn et al., 2017) and learned data augmentation (Ho et al., 2019). We are not aware of prior work that searches over programs that generate pre-pretraining data using a network trained on that data as the fitness.

3 METHODS

Given a downstream dataset $\mathcal{D}_{\text{down}}$, our goal is to arrive upon a data generating program π whose outputs prime a transformer to do well on that dataset, and potentially generalise to unseen downstream tasks as well. Each downstream dataset comprises train/val/test splits and a scoring metric, $(\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}, \mathcal{D}_{\text{test}}, \mathcal{L})$. We view each data generating program π as defining a distribution over token sequences, and also view a downstream training split as a distribution over token sequences. We write $\theta(\mathcal{D}_1, \mathcal{D}_2, \dots)$ for the parameters of a transformer which is first trained on $\mathcal{D}_1$, then $\mathcal{D}_2$, etc. Using this notation, we seek a program π optimising

$$\hat{J}(\pi) = \mathcal{L}(\mathcal{D}_{\text{val}}, \theta(\pi, \mathcal{D}_{\text{train}})), \tag{1}$$

where $\hat{J}$ approximates the true objective (which uses $\mathcal{D}_{\text{test}}$) on the validation data, ensuring that test data is never seen during program search. We represent programs π in Python. Our transformers are standard autoregressive models, except that we use byte-level tokenisation: tokenisers are constructed from massive human data, and a key part of our problem statement is to arrive upon a pretrained state *without* massive human data.

Figure 2 and Algorithm 1 summarise the method.

Initial program. An important choice is the starting point of the search. We construct a hand-written program π_0 by taking the union of existing synthetic pretraining paradigms from the recent literature (Table 1); it serves as a strong baseline. Programs follow a fixed generator contract (Appendix B): each emits streams of abstract ASCII symbols, tagged with the rule that produced them; English words are rejected by a validation step.

Table 1: Elements of the hand-written program π_0 (28 rules).

Tier	Rules	Literature source / rationale
1. Hierarchical	`dyck1/2, dyck_shuf, nest1--3`	Bracket tracking, recursive composition (Hu et al., 2025)
2. Cellular automata	`ca_rule`	Edge-of-chaos rules maximise transfer (Zhang et al., 2025)
3. Sequence ops	`sort, reverse, union, inter, copy, dedup`	Set and sequence primitives (Wu et al., 2021)
4. Reasoning	`deduct, induct, abduct`	Three modes of primitive inference (Wu et al., 2021)
5. SCAN-style	`twice, thrice, after, around`	Compositional commands (Lake & Baroni, 2018)
6. Procedural/CFG	`cfg, arith, mod, swap, pair`	CFG derivations and arithmetic (Jiang et al., 2026)
7. State tracking	`stack, fsm, counter`	Push/pop and FSM; persistent memory (Lee et al., 2026)

Algorithm 1 PRECOCIAL

Require: Downstream data $\mathcal{D}$, LLM, rounds K, steps T_0, T_c
$\pi \leftarrow \text{LLM}(\text{contract}, \mathcal{D}_{\text{sample}})$ ▷ first draft
$\theta \leftarrow \text{Pretrain}(\text{RandomInit}(), \text{Execute}(\pi), T_0)$
$\hat{J}^* \leftarrow \text{Eval}(\text{Finetune}(\theta, \mathcal{D}_{\text{train}}), \mathcal{D}_{\text{val}})$; $(\pi^*, \theta^*) \leftarrow (\pi, \theta)$
for $k = 1, \ldots, K-1$ **do**
  $\pi_k \leftarrow \text{LLM}(\pi^*, \hat{J}^*, \mathcal{D}_{\text{sample}})$ ▷ rewrite the accepted program
  $\theta \leftarrow \text{ContinuePretrain}(\theta^*, \text{Execute}(\pi_k), T_c)$ ▷ resume the one network
  $\hat{J}_k \leftarrow \text{Eval}(\text{Finetune}(\theta, \mathcal{D}_{\text{train}}), \mathcal{D}_{\text{val}})$ ▷ with early stopping
  if $\hat{J}_k \leq \hat{J}^*$ **then** $(\pi^*, \theta^*, \hat{J}^*) \leftarrow (\pi_k, \theta, \hat{J}_k)$ ▷ accept
  end if ▷ otherwise roll back to (π^*, θ^*)
end for
return π^*, θ^*

3.1 LLM-GUIDED SEARCH FOR PRETRAINING PROGRAMS

Following AlphaEvolve and others, PRECOCIAL generates variants of a program via LLM prompting and keeps the best under $\hat{J}$. The search starts from a first program that the LLM writes from the generator contract, which lists the rule families of Table 1, and a sample of downstream data (80–89% of the data in these first drafts comes from the rules of π_0). At each subsequent round the LLM receives the current accepted program, its objective value and the best so far, the validation curves of the last two candidates, 300 bytes of downstream training data and the contract, and returns a complete rewritten program: it may reweight, remove, rewrite or invent rules freely (Appendix C). The rewrite is scored by Eq. 1. It is accepted if $\hat{J}(\pi_k) \leq \hat{J}^*$, the best value so far; otherwise the search reverts to the last accepted program.

3.2 CONTINUAL PRETRAINING ACROSS CANDIDATES

Our objective function is very expensive to evaluate, requiring both pretraining and fine-tuning. This is in stark contrast to most program synthesis problems, where evaluation is so fast that even guess-and-check strategies are often viable (Li et al., 2022; Schkufza et al., 2013). After running this expensive evaluation, a naive approach would throw away hard-won pretrained parameters before generating and evaluating another program, as FunSearch and AlphaEvolve do when each candidate is scored independently. Instead, PRECOCIAL *continues* pretraining the same model from the last accepted checkpoint with each new program: the first program pretrains a randomly initialized network for T_0 steps, and every later candidate continues it for $T_c < T_0$ steps. A rejected candidate's steps are discarded, so the network always reflects the accepted programs only. This can be viewed as a form of continual learning, but also of curriculum learning: the system gradually enriches the pretraining program, with each program resuming where the previous one left off. With T_0=15k, T_c=5k and ten candidates, a search performs 60k pretraining steps, against 150k for a fresh 15k-step network per candidate.

Comparison to FunSearch and AlphaEvolve. Both maintain a population of programs from which parents are sampled; PRECOCIAL is a single chain. We implement both at the same budget of ten evaluations (Appendix A) with a fresh network per candidate or with PRECOCIAL's single continually trained network.

Table 2: Language and code under matched pretraining: test perplexity after a 10k-step fine-tune (lower is better), mean ± SD over 9 runs (3 searches × 3 training seeds). Last column: difference from PRECOCIAL pooled over the four datasets (positive: PRECOCIAL lower; 36 paired runs), p in parentheses; bold: $p < .05$. Rounds of one search: the LLM first draft is round 0, PRECOCIAL the program kept after round 9, and the bottom row the program kept after round 5; each program pretrains a fresh network for 60k steps.

Pretraining	WikiText-103	TinyStories	CodeParrot	CodeSearchNet	Δ vs PRECOCIAL
PRECOCIAL	5.284±.126	2.810±.081	3.207±.092	3.996±.177	–
LLM first draft	5.455±.044	2.875±.018	3.293±.032	4.295±.453	**+0.155** (<.001)
Random mutation	5.471±.041	2.877±.032	3.306±.042	4.205±.117	**+0.140** (<.001)
Procedural	5.688±.074	2.942±.025	3.388±.051	4.766±.581	**+0.371** (<.001)
Hand-written (π_0)	5.466±.033	2.887±.028	3.298±.024	4.495±.546	**+0.212** (<.001)
No pretraining	5.595±.045	2.932±.061	3.204±.015	5.511±.416	**+0.486** (<.001)
PRECOCIAL after round 5	5.296±.106	2.814±.063	3.227±.133	4.077±.131	+0.029 (.069)

4 EXPERIMENTS

4.1 SETUP

Downstream datasets. *Language and code:* TinyStories (Eldan & Li, 2023) (short narrative stories), WikiText-103 (Merity et al., 2017) (encyclopedia articles), CodeParrot (CodeParrot, 2022) (Python source code) and CodeSearchNet (Husain et al., 2019) (Python functions), with 10k training documents each (TinyStories: 7,818 stories of at most 512 characters), byte-encoded as raw text, and test sets from held-out data (Appendix D). *Planning and search puzzles:* Blocksworld with 12 blocks, a standard planning domain for language models (Valmeekam et al., 2023) (input: a PDDL problem; output: a plan, solved if replaying it in a simulator reaches the goal); the 8-puzzle (states 10–20 moves from the goal); Sokoban (7×7, two boxes); and Countdown with four numbers (Gandhi et al., 2024). Each has 5,000 training problems (Blocksworld 10,000) and is scored by an exact checker on 500 test problems (Blocksworld 200), with greedy decoding.

Model training. We use small 0.69M-parameter GPT-2 models (3 layers, 4 heads, 128-dimensional embeddings) with byte-level tokenization and a 512-byte context (1,024 for Blocksworld), optimised with AdamW. A search runs K=10 rounds with T_0=15k and T_c=5k steps on 2,000 synthetic streams per round. Each candidate is fine-tuned for up to 10k steps with early stopping (patience 5, evaluated every 500 steps), and $\hat{J}$ is validation perplexity (on the answer tokens, for the puzzles). The LLM is Claude Sonnet 4 (Anthropic, 2025). The 10k-step budget and the ten rounds were set once and used unchanged for every dataset, task and baseline (Appendix A).

Baselines. All share the architecture and training budget. *Hand-written* trains on π_0 (Table 1). *Procedural* trains on the published generators of Jiang et al. (2026). *LLM first draft* trains on the program every search starts from, never edited, isolating the search loop from the LLM's prior. *Random mutation* runs the same loop with the LLM replaced by random edits of that first draft (reweight all rules, remove one, or duplicate one). *No pretraining* fine-tunes a randomly initialized network. Each arm is represented by one program: the last program its search accepted, or the fixed program itself.

Matched pretraining. Comparing arms at the end of the search loop would be confounded: a rejected candidate is rolled back, so arms accumulate different amounts of pretraining (15k to 55k steps across our runs). We therefore compare curricula under *matched pretraining*: each arm's kept program pretrains a fresh network for exactly 60k steps, then gets the same fine-tuning and test: test perplexity after a 10k-step fine-tune for language, the exact checker after a 10k-step fine-tune for puzzles (Appendix A). Sections 4.3–4.4, which compare search procedures, evaluate the network each search produces. We report mean ± SD over runs and two-sided t-tests on paired differences.

4.2 LANGUAGE AND CODE: SEARCH AGAINST EVERY OTHER SOURCE OF A CURRICULUM

Table 2 compares PRECOCIAL with every other way of obtaining a curriculum. At the 10k-step budget PRECOCIAL has the lowest test perplexity of any curriculum on all four datasets, and it is level with no pretraining only on CodeParrot (3.207 against 3.204). Pooled over the four datasets, it improves on the LLM's first draft by 0.155 (lower in 30 of 36 paired runs), on random mutation of

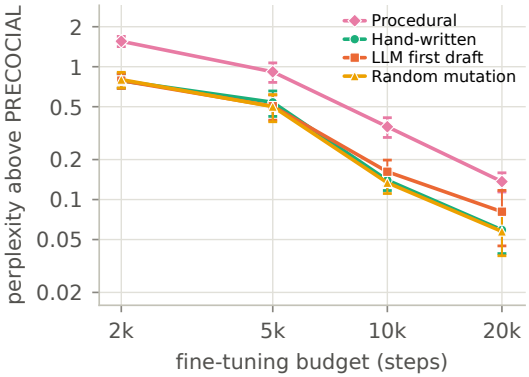


Figure 3: Perplexity above PRECOCIAL at each fine-tuning budget, pooled over the four datasets under matched pretraining (best validation perplexity within the budget; mean ± s.e.m. over 36 paired runs; logarithmic axes). PRECOCIAL is ahead of every other curriculum at every budget shown; the lead is largest at small budgets and narrows as fine-tuning proceeds. Table 9 gives every value with its p-value.

Table 3: Search strategies at the same budget of ten evaluations, each with a single continually trained network. Test perplexity after a 10k-step fine-tune (lower is better), mean ± SD over 3 seeds. Last column: difference from PRECOCIAL pooled over the four datasets (positive: PRECOCIAL lower), p in parentheses. The PRECOCIAL row is an independent repeat of the ten-round loop of Table 11 (Appendix E.2).

Strategy	WikiText-103	TinyStories	CodeParrot	CodeSearchNet	Δ vs PRECOCIAL
PRECOCIAL (chain)	5.151±.201	2.818±.003	3.201±.110	4.167±.149	–
FunSearch	5.419±.033	2.839±.027	3.228±.081	4.016±.083	+0.041 (.505)
AlphaEvolve	5.411±.044	2.824±.011	3.271±.017	4.283±.127	+0.113 (.056)

that draft by 0.140 (32 of 36), on the hand-written program by 0.212 (31 of 36) and on procedural pretraining by 0.371 (36 of 36), all with $p<.001$. Since the first draft and random mutation start from the same program as the search, the gain over them is due to the LLM-guided search loop. Program quality also improves steadily over the rounds: under matched pretraining the program kept after round 5 already beats the first draft (round 0) by 0.126 ($p=.005$; 28 of 36), and the last four rewrites add a further 0.029 ($p=.07$). On every dataset round 9 beats round 5, which beats round 0. Without matched pretraining, at the end of the ten-round loop, PRECOCIAL is again lowest on every dataset (Appendix E.2). Appendix G plots the search itself.

Across fine-tuning budgets (Figure 3), PRECOCIAL is ahead of all four alternative curricula at every budget from 2k to 20k steps, significantly so against every arm at every budget (Table 9). The lead is largest early, 0.78–1.55 perplexity at 2k steps, and narrows to 0.06–0.14 by 20k steps. Searching therefore buys downstream learning speed. Measured in fine-tuning steps, as Hu et al. (2025) and Jiang et al. (2026) do, PRECOCIAL reaches its 2k-step perplexity with 57%, 67% and 80% of the steps that no pretraining, procedural pretraining and the hand-written program need, and its 10k-step perplexity with 83%, 80% and 89% (medians over 36 runs; Table 10).

4.3 Search strategy: PRECOCIAL, FunSearch and AlphaEvolve

Table 3 compares the three strategies at the same budget of ten network evaluations (9–11 LLM calls), all using a single continually trained network. PRECOCIAL and the two population methods are not significantly different: PRECOCIAL's chain is ahead of FunSearch by 0.041 perplexity ($p=.51$) and of AlphaEvolve by 0.113 ($p=.056$) at the 10k-step budget. The simple chain does as well as either population method, and any of them can use the single-network protocol of Section 3.2. One caveat: FunSearch and AlphaEvolve normally keep the program and its network paired, training each candidate on a network descended from its own selected ancestor. Here only the program selection follows that lineage since the network is one continuous line that advances every round regardless of which ancestor was picked. The tie in Table 3 reflects how well each method picks and edits programs, not the full population method as published.

4.4 One network across candidates is cheaper and better

Every candidate must be scored on a pretrained network, and Section 3.2 proposed continuing one network across the whole search instead of pretraining a fresh one for each candidate. We test this with all three search strategies on WikiText-103 and CodeSearchNet, running each strategy both ways; everything else, including the GPU type, is identical (Table 4). Table 4 shows the result.

Table 4: One network continued across candidates ("one") versus a fresh network per candidate ("fresh"), for each search strategy. Test perplexity after a 10k-step fine-tune, mean ± SD over 3 seeds; Δ = fresh − one (positive: one network better), p in parentheses. Last column: GPU-hours of one search (its ten evaluations), one / fresh, on the same GPU type (RTX A6000).

	WikiText-103			CodeSearchNet			GPU-h
Strategy	one	fresh	Δ	one	fresh	Δ	one / fresh
PRECOCIAL (chain)	5.151±.201	5.507±.128	+0.356 (.053)	4.167±.149	4.299±.399	+0.133 (.514)	0.71 / 0.85
FunSearch	5.419±.033	5.553±.041	+0.134 (.075)	4.016±.083	4.227±.360	+0.212 (.324)	0.62 / 0.88
AlphaEvolve	5.411±.044	5.496±.127	+0.085 (.249)	4.283±.127	4.414±.167	+0.131 (.496)	0.67 / 0.85
Pooled (9 runs each)			+**0.191** (.006)			+0.158 (.092)	0.67 / 0.86

Table 5: Planning and search puzzles from scratch, matched pretraining (each curriculum pretrains a fresh network for 60k steps; every arm then gets the same 10k-step fine-tune). Test solve rate by an exact checker, mean ± SD over 9 runs (Blocksworld) or 6 runs (others). Bottom rows: seed-paired difference, p in parentheses; bold: $p < .05$. No pretraining: 6 training seeds, not paired.

Pretraining	Blocksworld (12 blocks)	8-puzzle	Sokoban	Countdown
PRECOCIAL	0.216±.254	0.246±.028	0.086±.030	0.161±.012
LLM first draft	0.053±.158	0.156±.077	0.013±.006	0.156±.014
Hand-written (π_0)	0.019±.040	0.172±.035	0.019±.013	0.136±.019
No pretraining	0.000±.000	0.014±.011	0.007±.003	0.056±.039
PRECOCIAL − hand-written	+**0.196** (.033)	+**0.073** (.005)	+**0.067** (.002)	+**0.025** (.025)
PRECOCIAL − first draft	+0.163 (.060)	+**0.090** (.036)	+**0.073** (.003)	+0.005 (.104)

Training one network is *better* at the 10k-step fine-tuning budget: pooled over the 18 paired runs, a fresh network per candidate ends 0.175 perplexity worse (p=.002; worse in 14 of 18), significantly so on WikiText-103 alone (p=.006) and in the same direction on CodeSearchNet (p=.09). Training one network is also *cheaper* for every strategy: fresh networks need 2.5× the pretraining steps and 29% more GPU time per search on average. Part of the gain is that the continued network accumulates more pretraining (up to 60k steps, against 15k), a benefit obtained at lower cost.

Cost against the published protocols. Used as published, FunSearch and AlphaEvolve give each candidate its own network, pretrained from random initialisation for T_0=15k steps, so ten candidates cost 10×15k=150k pretraining steps; PRECOCIAL pretrains one network for T_0=15k steps and continues it by T_c=5k for each later candidate, 60k in total. Both fine-tune all ten candidates identically, so the difference is entirely pretraining: 2.5× fewer pretraining steps and 19.5% and 17.1% less GPU time per search, while ending 0.231 and 0.296 perplexity lower at the 10k-step budget (averaged over the two datasets), for about the same LLM bill. Fine-tuning the candidates is the largest part of every search, so the saving is modest here; it grows with the number of candidates and with model size (Appendix I).

4.5 Planning and search puzzles from scratch

Do searched programs help a small model learn tasks that require multi-step planning? Each task has three searched programs (one per search seed), each trained with several training seeds: three for Blocksworld (9 runs per arm) and two for the others (6 runs per arm).

Searched programs beat the hand-written curriculum on all four tasks at the 10k-step fine-tuning budget (Table 5): Sokoban +0.067 (p=.002), the 8-puzzle +0.073 (p=.005), Countdown +0.025 (p=.025) and Blocksworld +0.196 (p=.033), ahead in every paired run on the first three. Blocksworld is all-or-nothing: across the 33 runs of Table 5 every model that reaches a test perplexity below 1.04 solves some problems and every model above it solves none (Appendix H), and 6 of 9 searched runs land below that line, against 1 of 9 for the first draft and 2 of 9 for the hand-written program (Figure 8 in Appendix H). At this budget a network with no synthetic pretraining learns almost nothing on the three planning tasks (solve rates 0.000 on Blocksworld, 0.014 on the 8-puzzle and 0.007 on Sokoban).

The search loop matters, not only the LLM's prior. Every search starts from the LLM's first draft, so comparing against it isolates what the rounds of search add. The searched programs beat the first draft significantly on Sokoban (+0.073, p=.003) and the 8-puzzle (+0.090, p=.036), and lead it

Table 6: Transfer: networks pretrained on programs obtained for WikiText-103, fine-tuned for 10k steps on a different dataset with no further search (matched pretraining). Test perplexity, mean ± SD over 9 runs (3 searches × 3 training seeds); last column pooled over the three targets (27 paired runs; positive: PRECOCIAL lower). Bottom row: the program PRECOCIAL found when searching on the target itself (Table 2).

WikiText-103 program	TinyStories	CodeParrot	CodeSearchNet	Δ vs PRECOCIAL
PRECOCIAL	2.804±.070	3.215±.067	4.279±.508	–
LLM first draft	2.897±.031	3.322±.024	4.363±.227	+0.095 (.058)
Random mutation	2.888±.038	3.320±.039	4.655±.665	+0.188 (.120)
Procedural	2.944±.014	3.406±.022	4.620±.424	**+0.224** (.011)
Hand-written (π_0)	2.868±.016	3.309±.017	4.228±.032	+0.035 (.550)
No pretraining	2.915±.058	3.204±.017	5.632±.327	**+0.484** (.002)
Searched on the target	2.810±.081	3.207±.092	3.996±.177	

Task	π_0	First draft	All cand.	Kept
Blocksworld	0.0	0.0±0.0	4.3±6.7	3.3±4.9
8-puzzle	0.0	8.4±14.5	18.1±15.1	22.5±10.5
Sokoban	0.0	0.0±0.0	9.4±7.3	12.3±9.2
Countdown	3.6	3.4±2.2	20.3±10.8	22.8±6.7
CodeParrot	0.0	0.0±0.0	8.0±12.6	20.1±20.1
CodeSearchNet	0.0	0.0±0.0	9.4±12.3	17.2±18.5

Table 7: Share of synthetic data (%) from rules of the task-matching type (Figure 4), mean ± SD over three searches: in π_0, the first draft, all candidates of rounds 0–9, and the kept program.

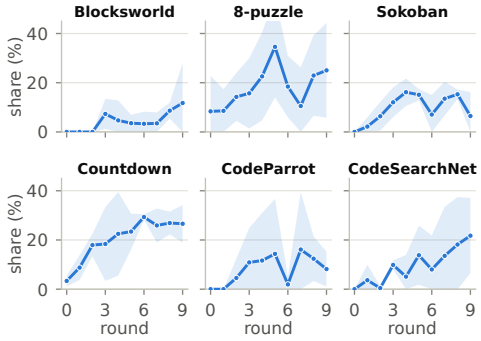


Figure 4: Task-matching share by search round, mean ± SD over three searches: planning rules for Blocksworld, the 8-puzzle and Sokoban, arithmetic for Countdown, code syntax for code.

on Blocksworld (+0.163, p=.06). On Sokoban the first draft is no better than the hand-written program, so the whole gain there comes from search. Countdown is the exception: its first draft is already close to the searched program (+0.005, p=.10). The search adds most where the LLM's first guess at a curriculum is poor, and little where that guess, like π_0, already covers the task.

4.6 Transfer to unseen datasets

Does a program searched for one dataset help others? We take the network pretrained on each arm's WikiText-103 program (matched pretraining) and fine-tune it on TinyStories, CodeParrot and CodeSearchNet for 10k steps, with no further search (Table 6).

The program searched for WikiText-103 transfers well to TinyStories and CodeParrot, where it is the best curriculum and within 0.008 perplexity of a program searched on the target itself, and less well to CodeSearchNet. Pooled over the three targets it beats procedural pretraining (+0.224, p=.011; 25 of 27 paired runs) and no pretraining (+0.484, p=.002), is ahead of the first draft (+0.095, p=.058) and random mutation (+0.188, p=.12), and is level with the hand-written program (+0.035, p=.55). Transfer even crosses domains (Appendix E.3). The programs searched on Blocksworld and Sokoban do as well on the four language datasets as the programs searched on those datasets themselves (within 0.017 and 0.003 perplexity, $p \geq .69$), and in the other direction the WikiText-103 program solves more problems than the hand-written program on all four puzzles. Planning and language have nothing in common at the task level, so this only makes sense if what transfers is the same structural skill (nesting, nesting, state tracking) rather than anything task-specific.

4.7 What the LLM writes

Every program is a weighted set of named rules, each generating one kind of sequence. We classify rules by name into content types (Appendix J) and measure, for each task, the share of synthetic data coming from rules of the type that matches the task.

The LLM writes task-specific rules that π_0 does not contain. In its own names: `precond` and `state_trans` for Blocksworld, `grid_move` and `sokoban_push` for the move puzzles, `arith_precedence` for Countdown, `func_def` and `try_catch` for code. The first drafts contain almost none of this (0.0–8.4% of the data, Table 7); the share grows over the rounds of search (Figure 4) to 3.3–22.8% of the kept programs.

But task-specific content is not what the search selects for. Comparing, within each task and round, the candidates the search accepted with those it rejected, accepted candidates carry no more task-matching data (+2.2 points, stratified permutation test p=.35, 162 candidates). A direct test agrees. We re-ran the Blocksworld search with the downstream data hidden from the LLM, which then sees only its programs and their scores: it wrote no planning rules at all (0 of 298 added rules), yet under matched pretraining its programs solve as many problems as those of the search that sees the task (0.196 against 0.216, p=.82, 9 runs each). Much of what the search keeps is general structure: the 28 generic rules of π_0 (nesting, sequence operations and state tracking, among others), which any byte-level sequence model can use, still produce 43–74% of the kept programs' data, down from 80–89% of the first drafts. This may explain why programs transfer across domains (Section 4.6): the puzzle programs that transfer best to language, Blocksworld and Sokoban, carry the least task-specific data (3.3% and 12.3% of their kept programs, against 22.5% and 22.8% for the 8-puzzle and Countdown), though four programs are too few to be sure.

5 Discussion

We set out to ask whether a pretraining program can be optimized automatically, and at a fixed fine-tuning budget our answer is yes. An LLM that rewrites a generator program, scored by training a small network on the program's data, finds programs that beat the hand-written curriculum, published procedural generators, random mutation and its own first draft on four language and code datasets, and that beat the hand-written curriculum on four planning puzzles. The programs improve steadily over the rounds of search, and they transfer to datasets they were not searched on, in part even across domains.

What makes the search work. Our results point to the trained model rather than the LLM's knowledge of the task. On Blocksworld a search that never sees the task does as well as one that does, and across tasks the programs the search accepts carry no more task-specific content than those it rejects. The LLM is still not interchangeable with random edits: random mutation, which cannot write new rules, falls short of it on language. Our reading is that the LLM's job is to propose varied, valid programs, including kinds of data the current program lacks, and that scoring each proposal on a trained network is what decides which of them help. The same view accounts for transfer: much of what survives is general structure that any sequence model can use.

Evaluating candidates when fitness requires training. Because every score requires training, how candidates are evaluated matters as much as how they are proposed. Continuing one network across candidates is cheaper and, at a fixed budget, better for all three strategies we tried.

When is this useful? Like other synthetic pretraining papers, the benefit we measure is efficiency: the searched programs reach a given perplexity in fewer fine-tuning steps (Table 10), and their lead is largest with the least downstream data (Appendix F). PRECOCIAL is therefore aimed at settings where downstream data is scarce and no more human data is available, since synthetic data costs nothing to sample. And because programs transfer, a search can be run once and its program reused on related datasets.

Limitations and future work. Our models are small (0.15M–3.36M parameters, Appendix F): across that range the head start grows with the model but is spent sooner, so the lead at a fixed budget shrinks with size. Larger models, larger downstream datasets and longer searches were beyond our compute budget. Each task used three searches of ten evaluations with a single LLM (Claude Sonnet 4), so smaller effects, such as the differences between search strategies, remain unresolved, and the task-hidden test was run on Blocksworld only. Mixing different datasets together, and mixing the discovered pretraining data into fine-tuning, may improve results further, especially for already pretrained models such as Llama or Qwen, where replaying synthetic data could keep the model from forgetting what it has learned.

AI USE STATEMENT

In this work, the method itself uses a generative AI model: Claude Sonnet 4 writes and rewrites the data generating programs during search (Section 3). We also used generative AI tools to implement experiment code, run and analyse experiments, and assist in writing and editing the paper. All AI-assisted code, analyses and text were reviewed by the authors; every number in the paper is computed by a script from the raw run logs. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work trains small models on public datasets (TinyStories, WikiText-103, CodeParrot and CodeSearchNet) and on procedurally generated puzzles; it involves no human subjects or personal data, and the synthetic pretraining data consists only of abstract symbols (Appendix B).

REPRODUCIBILITY STATEMENT

The model, optimiser, search schedule and evaluation protocol are specified in Section 4.1 and Appendix A; the generator contract and prompt in Appendices B and C; dataset sources and splits in Appendix D. Code, all searched programs and the script that computes every table from the run logs will be released.

REFERENCES

Anthropic. System card: Claude Opus 4 & Claude Sonnet 4. `https://www.anthropic.com/claude-4-system-card`, 2025.

Dániel L. Barabási, Taliesin Beynon, Ádám Katona, and Nicolas Perez-Nieves. Complex computation from developmental priors. *Nature Communications*, 14(1):2226, 2023.

CodeParrot. CodeParrot clean python dataset. `https://huggingface.co/datasets/codeparrot/codeparrot-clean-train`, 2022.

Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent English? *arXiv preprint arXiv:2305.07759*, 2023.

Maksim Elistratov, Marina Barannikov, Gregory Ivanov, Valentin Khrulkov, Anton Konushin, Andrey Kuznetsov, and Dmitrii Zhemchuzhnikov. CADEvolve: Creating realistic CAD via program evolution. *arXiv preprint arXiv:2602.16317*, 2026.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning (ICML)*, 2017.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (SoS): Learning to search in language. *arXiv preprint arXiv:2404.03683*, 2024.

Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking ImageNet pre-training. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. Scaling laws for transfer. *arXiv preprint arXiv:2102.01293*, 2021.

Daniel Ho, Eric Liang, Ion Stoica, Pieter Abbeel, and Xi Chen. Population based augmentation: Efficient learning of augmentation policy schedules. In *International Conference on Machine Learning (ICML)*, 2019.

Michael Y. Hu, Jackson Petty, Chuan Shi, William Merrill, and Tal Linzen. Between circuits and Chomsky: Pre-pretraining on formal languages imparts linguistic biases. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Liangze Jiang, Zachary Shinnick, Anton van den Hengel, Hemanth Saratchandran, and Damien Teney. Procedural pretraining: Warming up language models with abstract data. *arXiv preprint arXiv:2601.21725*, 2026.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning (ICML)*, 2018.

Dan Lee, Seungwook Han, Akarsh Kumar, and Pulkit Agrawal. Training language models via neural cellular automata. *arXiv preprint arXiv:2603.10055*, 2026.

Xiaochuan Li, Zichun Yu, and Chenyan Xiong. Montessori-instruct: Generate influential training data tailored for student learning. In *International Conference on Learning Representations (ICLR)*, 2025.

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

William Liang, Sam Wang, Hung-Ju Wang, Osbert Bastani, Dinesh Jayaraman, and Yecheng Jason Ma. Eurekaverse: Environment curriculum generation via large language models. In *Conference on Robot Learning (CoRL)*, 2024.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017.

Tiantian Mi, Dongming Shan, Zhen Huang, Yiwei Qin, Muhang Xie, Yuxuan Qiao, Yixiu Liu, Chenyang Zhou, and Pengfei Liu. Data Darwinism part II: DataEvolve – AI can autonomously evolve pretraining data curation. *arXiv preprint arXiv:2603.14420*, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Isabel Papadimitriou and Dan Jurafsky. Learning music helps you read: Using transfer to study linguistic structure in language models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

Zachary Shinnick, Liangze Jiang, Hemanth Saratchandran, Damien Teney, and Anton van den Hengel. Can you learn to see without images? Procedural warm-up for vision transformers. *arXiv preprint arXiv:2511.13945*, 2025.

Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009.

Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data. In *International Conference on Machine Learning (ICML)*, 2020.

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

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

Yuhuai Wu, Markus N. Rabe, Wenda Li, Jimmy Ba, Roger B. Grosse, and Christian Szegedy. LIME: Learning inductive bias for primitives of mathematical reasoning. In *International Conference on Machine Learning (ICML)*, 2021.

Shiyang Zhang, Aakash Patel, Syed A. Rizvi, Nianchen Liu, Sizhuang He, Amin Karbasi, Emanuele Zappala, and David van Dijk. Intelligence at the edge of chaos. In *International Conference on Learning Representations (ICLR)*, 2025.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

A Further experimental setup details

Model. Vocabulary 260 (256 bytes + PAD/BOS/EOS/SEP), 3 layers, 4 attention heads, 128-dimensional embedding, 512-position context (1,024 for Blocksworld), ≈0.69M parameters. All weights are randomly initialised; embeddings are not reinitialised between pretraining and fine-tuning.

Pretraining. 2,000 streams per round, 50 sentences per stream, truncated to the context length. Round 0: 15,000 steps from scratch; later rounds: 5,000 continued steps from the last accepted checkpoint. A ten-round search therefore performs 60,000 pretraining steps, and its kept network carries between 15,000 and 60,000 of them, depending on how many rounds were accepted. Batch size 8, AdamW, learning rate 3×10^{-4}, gradient clipping at 1.0.

Fine-tuning. Up to 10,000 steps, learning rate 3×10^{-4}, batch size 8, early stopping with patience 5 evaluations every 500 steps; the checkpoint with the lowest validation perplexity is tested. A second, longer fine-tune of the kept pretrained network (early stopping with patience 10; every run continued well past 20k steps) supplies the validation curve from which we read the best perplexity reached within each fine-tuning budget from 2k to 20k steps (Figures 3 and 5, Appendix E). For the puzzles the loss and the perplexity are computed on the answer tokens only.

Choice of budgets. A search evaluates ten candidates, each with its own pretraining and fine-tuning, so each evaluation must be short. We fine-tune for 10k steps (about eight passes over a 10k-document training set at batch size 8), which keeps a ten-candidate search under one GPU-hour on the 0.69M model (Appendix I). It is a budget, not convergence: validation perplexity is still falling at 10k steps in 96% of the fine-tunes run during our searches. Because the search's fitness is measured at this budget, results at 10k steps evaluate exactly what the search optimizes; Figure 3 and Table 9 report 2k to 20k steps as well. Ten rounds keep the LLM cost near $1 per search, and the program kept after round 5 already captures most of the gain (Table 2), so more rounds would add little. Both values were set once, before the experiments reported here, and used unchanged for every dataset, task, baseline and model size.

Search. Ten rounds; a candidate is accepted if its validation perplexity is no worse than the best so far (tolerance 0), and otherwise both the program and the pretrained checkpoint are rolled back. The LLM (Claude Sonnet 4 via the OpenRouter API) samples the first draft at temperature 0.9 and rewrites at temperature 0.7. FunSearch uses 4 islands and prompts with the best programs of one island; AlphaEvolve uses one database, fitness-proportional parent sampling and 2 inspiration

programs per prompt. All strategies make one LLM call per evaluation, plus a retry whenever a returned program fails validation.

Matched pretraining. For every comparison of curricula (Tables 2, 5 and 6), every arm's kept program pretrains a fresh network for exactly 60,000 steps on 2,000 streams, followed by the fine-tuning above. For language, each of the three kept programs per arm (one per search seed) is trained with three training seeds, the search seed and two others, giving 9 runs per cell of Tables 2 and 6; no pretraining uses the same nine training seeds, and runs are paired by training seed. Blocksworld is fine-tuned for the full 10,000 steps without early stopping: its validation perplexity plateaus early, and patience-5 stopping ends fine-tuning at 1,000 to 3,500 steps, above the perplexity at which plans start to succeed (Appendix H).

B GENERATOR PROGRAM SPECIFICATION

The generator contract requires programs to define:

```
def generate_stream(num_sentences: int) -> List[Tuple[str, str]]:
    """Returns list of (rule_name, content_string) pairs.
    content_string must use only abstract ASCII symbols."""
```

All content strings are drawn from the following vocabulary: uppercase letters (A–Z), digits (0–9), brackets (`()[]<>{}`), operators (`+-*/=^%~`), separators (`|;,.!@#$ `), and arrows (`->`, `=>`). English words and multi-character token names are forbidden by a validation step that checks for common English words before accepting any generated program. The rule name returned with each content string tags the stream and is used for the rule analysis (Section 4.7).

C PROMPT STRUCTURE

First draft. The LLM is asked to design synthetic pretraining data for a small GPT-2 model whose transformer layers (not its token embeddings' meaning) will transfer to the downstream task, and receives the generator contract, which lists the rule families of Table 1 with their literature justifications, the task type, and 500 bytes of downstream training data.

Each later round. The prompt contains, in order: (1) the round number and a note that the network is continued from the previous round rather than retrained; (2) the current validation perplexity and the best so far; (3) the validation perplexity along the fine-tuning curves of the current and previous candidates; (4) 300 bytes of downstream training data; (5) the current program in a Python code fence; (6) the generator contract; and (7) the instruction: add any number of new rule types, remove any rules that are unhelpful, and modify existing generators and their weights, with no cap on the size of the change (the prompt notes, as information only, that large distribution shifts can cause forgetting in a continually trained network). The LLM returns the complete program. For the task-hidden ablation (Section 4.7) the downstream sample is replaced by a notice that it is withheld.

D DATASETS

Table 8: Dataset splits. Language and code datasets are byte-encoded as raw text with no filtering at training time beyond truncation to the context. Puzzles are generated with fixed seeds, with a reference answer that is optimal for the move puzzles; any answer the checker accepts counts as solved. †The first 200 of 1,000 test problems, decoded greedily.

Dataset	Train	Val	Test	Source / checker
TinyStories	7,818	868	155	HF train / validation, $\leq$512 characters
WikiText-103	10,000	1,111	1,241	HF train / HF test (raw)
CodeParrot	10,000	2,499	2,499	codeparrot-clean-train, held-out tail
CodeSearchNet	10,000	1,000	1,000	Python functions, held out
Blocksworld (12 blocks)	10,000	1,000	200	PDDL; plan replayed in a simulator†
8-puzzle	5,000	200	500	10–20 moves from the goal; reach it
Sokoban (7×7, 2 boxes)	5,000	200	500	push both boxes onto targets
Countdown (4 numbers)	5,000	200	500	each number once; equals the target

E ADDITIONAL RESULTS

E.1 EVERY FINE-TUNING BUDGET

Table 9: Matched pretraining: perplexity above PRECOCIAL within each fine-tuning budget, pooled over the four datasets, 36 paired runs; p in parentheses; bold: $p < .05$. Each entry is the best validation perplexity reached within the budget along a single long fine-tuning run. These are the values plotted in Figure 3; they differ slightly from the 10k-step column of Table 2, which is the test perplexity of a separate 10k-step fine-tune.

Arm	2k	5k	10k	20k
LLM first draft	+**0.786** (<.001)	+**0.504** (<.001)	+**0.162** (<.001)	+**0.081** (.032)
Random mutation	+**0.800** (<.001)	+**0.501** (<.001)	+**0.134** (<.001)	+**0.058** (.006)
Procedural	+**1.554** (<.001)	+**0.915** (<.001)	+**0.353** (<.001)	+**0.137** (<.001)
Hand-written (π_0)	+**0.784** (<.001)	+**0.540** (<.001)	+**0.140** (<.001)	+**0.059** (.005)
No pretraining	+**1.685** (<.001)	+**0.900** (<.001)	+**0.476** (<.001)	+**0.307** (.001)

Table 10: Fine-tuning steps saved, matched pretraining. For each budget k, the share of an arm's fine-tuning steps that PRECOCIAL needs to reach the same validation perplexity: k divided by the steps the arm needs to reach the best perplexity PRECOCIAL attains within k steps, median over the 36 runs (4 datasets × 9). Lower means a larger saving. Curves are evaluated every 500 steps.

Arm	2k	5k	10k	20k
LLM first draft	80%	83%	91%	93%
Random mutation	80%	91%	91%	94%
Procedural	67%	74%	80%	82%
Hand-written (π_0)	80%	83%	89%	91%
No pretraining	57%	77%	83%	83%

E.2 THE TEN-ROUND LOOP WITHOUT MATCHED PRETRAINING

The tables below evaluate each arm with the network its own ten-round loop produced, instead of retraining its kept program on a fresh network. Arms then differ in accumulated pretraining: the kept checkpoints carry 42,500 pretraining steps on average for PRECOCIAL against 37,100–38,800 for the other arms. PRECOCIAL is lowest on every dataset, and every pooled difference is significant.

Table 11: End of the ten-round loop: test perplexity after a 10k-step fine-tune, mean ± SD over 3 seeds; last column pooled over the four datasets (positive: PRECOCIAL lower).

Pretraining	WikiText-103	TinyStories	CodeParrot	CodeSearchNet	Δ vs PRECOCIAL
PRECOCIAL	5.312±.146	2.825±.040	3.176±.086	4.035±.108	–
LLM first draft	5.493±.045	2.885±.013	3.291±.025	4.278±.114	+**0.150** (<.001)
Random mutation	5.479±.027	2.904±.022	3.332±.051	4.418±.414	+**0.196** (.005)
Procedural	5.813±.080	2.964±.032	3.493±.130	4.509±.077	+**0.358** (<.001)
Hand-written (π_0)	5.508±.082	2.891±.021	3.311±.015	4.288±.084	+**0.162** (<.001)
No pretraining	5.585±.055	2.939±.077	3.205±.004	5.676±.149	+**0.514** (.026)

Table 12: Transfer with the networks produced by the WikiText-103 loops (no matched pretraining): test perplexity after a 10k-step fine-tune on the target, mean ± SD over 3 seeds; last column pooled over the three targets.

WikiText-103 program	TinyStories	CodeParrot	CodeSearchNet	Δ vs PRECOCIAL
PRECOCIAL	2.840±.067	3.206±.082	4.161±.218	–
LLM first draft	2.917±.020	3.350±.060	5.565±.065	+**0.542** (.041)
Random mutation	2.915±.048	3.344±.039	5.093±.723	+0.382 (.072)
Procedural	3.010±.037	3.453±.015	4.591±.194	+**0.283** (.006)
Hand-written (π_0)	2.922±.056	3.357±.054	4.452±.171	+**0.175** (.002)
No pretraining	2.908±.047	3.200±.016	5.704±.099	+0.535 (.070)
Searched on the target	2.825±.040	3.176±.086	4.035±.108	

E.3 Transfer across domains

Here the programs cross between language and puzzles, with matched pretraining and no further search. Table 13 pretrains on the program searched for each puzzle (Section 4.5) and fine-tunes on the four language datasets; Table 14 pretrains on the program searched for WikiText-103 and fine-tunes on the four puzzles, with the same training seeds as the planning arms. Programs searched on Blocksworld and Sokoban are level with programs searched on the language datasets themselves and ahead of the hand-written program in 9 and 12 of 12 runs; the 8-puzzle and Countdown programs transfer less well. In the other direction the WikiText-103 program solves more problems than the hand-written program on every puzzle, but no single difference is significant, and it stays slightly below the program searched on each task.

Table 13: Puzzle programs on language: test perplexity after a 10k-step fine-tune, mean ± SD over 3 seeds (matched pretraining). Last two columns: pooled difference over the four datasets (12 seed-paired runs; positive: the puzzle program lower), p in parentheses. First row: the program searched on each language dataset itself, with the same three seeds (a subset of the nine runs of Table 2).

Pretraining	WikiText-103	TinyStories	CodeParrot	CodeSearchNet	vs hand-written	vs searched on target
Searched on the target	5.281±.162	2.795±.111	3.199±.099	3.976±.243		
Blocksworld program	5.259±.346	2.797±.152	3.151±.218	3.976±.287	+0.280 (.052)	+0.017 (.687)
8-puzzle program	5.434±.105	2.866±.062	3.253±.109	4.406±.451	+0.086 (.561)	−0.177 (.107)
Sokoban program	5.270±.151	2.782±.060	3.200±.051	4.011±.163	+0.260 (.082)	−0.003 (.960)
Countdown program	5.390±.126	2.862±.068	3.246±.033	4.553±.878	+0.063 (.728)	−0.200 (.224)
Hand-written (π_0)	5.458±.036	2.874±.030	3.300±.023	4.671±.770		
No pretraining	5.585±.055	2.939±.077	3.205±.004	5.676±.149		

Table 14: The WikiText-103 program on the puzzles: test solve rate after a 10k-step fine-tune, mean ± SD over 9 runs (Blocksworld) or 6 (others), matched pretraining; the other rows are those of Table 5. Bottom rows: differences paired by training seed (positive: the WikiText-103 program solves more), p in parentheses.

Pretraining	Blocksworld (12 blocks)	8-puzzle	Sokoban	Countdown
Searched on the task	0.216±.254	0.246±.028	0.086±.030	0.161±.012
Searched on WikiText-103	0.158±.237	0.207±.136	0.070±.046	0.147±.014
Hand-written (π_0)	0.019±.040	0.172±.035	0.019±.013	0.136±.019
No pretraining	0.000±.000	0.014±.011	0.007±.003	0.056±.039
WikiText-103 program − hand-written	+0.139 (.101)	+0.035 (.590)	+0.050 (.052)	+0.012 (.371)
WikiText-103 program − searched on the task	−0.057 (.312)	−0.038 (.532)	−0.016 (.598)	−0.013 (.233)

F MODEL SIZE AND DOWNSTREAM DATA SIZE

Both sweeps use WikiText-103 with raw text and evaluate the network each arm's ten-round loop produced (as in Appendix E.2, not matched pretraining). Every model size runs the same search schedule on 2,000 synthetic streams per round.

Model size. Table 15 repeats the WikiText-103 comparison at 0.15M, 0.69M and 3.36M parameters, and Figure 5 follows the lead over no pretraining across fine-tuning budgets. The head start is larger for larger models but comes earlier: at its peak a network with no synthetic pretraining trails PRECOCIAL by 1.44 perplexity at 0.15M (after 5k fine-tuning steps), 2.83 at 0.69M (after 2k) and 3.56 at 3.36M (after 1k). A larger model also closes the gap sooner, so at the 10k-step budget of Table 15 the lead shrinks with size: over the hand-written curriculum it falls from 1.445 to 0.196 and 0.098 perplexity, and over no pretraining from 1.248 to 0.273 and 0.195. With three seeds none of these differences is significant on its own (p between .076 and .351). The synthetic budget is the same for every size, so larger models receive less synthetic data per parameter; whether a curriculum scaled with the model keeps the lead is untested.

Table 15: Model size sweep, N=10k WikiText-103 documents: test perplexity after a 10k-step fine-tune, mean ± SD over 3 seeds. Last two columns: seed-paired differences (positive: PRECOCIAL lower), p in parentheses.

Parameters	PRECOCIAL	Hand-written (π_0)	No pretraining	PRECOCIAL − hand	PRECOCIAL − none
0.15M	8.254±.746	9.699±.885	9.502±.069	+1.445 (.093)	+1.248 (.110)
0.69M	5.312±.146	5.508±.082	5.585±.055	+0.196 (.112)	+0.273 (.076)
3.36M	4.063±.126	4.161±.097	4.258±.038	+0.098 (.351)	+0.195 (.110)

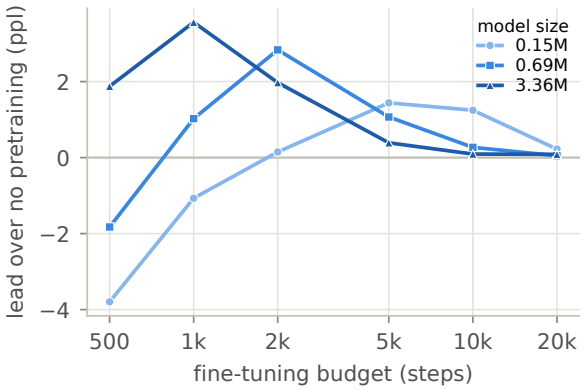


Figure 5: Head start by model size on WikiText-103: perplexity of a network with no synthetic pretraining minus PRECOCIAL's, at each fine-tuning budget (best validation perplexity reached within the budget, mean over 3 seeds; above 0: PRECOCIAL is ahead). Larger models reach a larger lead, earlier, and close the gap sooner. The two smallest pretrained networks start behind (at 500 steps, and also at 1k for 0.15M).

Downstream data. Table 16 varies the number N of fine-tuning documents for the 0.69M model. The lead is largest with the least data: at N=2k, PRECOCIAL is ahead of the hand-written curriculum by 0.213 perplexity (p=.040) and of no pretraining by 0.410 (p=.024), the only significant differences in the sweep. At larger N the lead is smaller and does not change monotonically with N (0.041, 0.196 and 0.053 over the hand-written curriculum at 5k, 10k and 20k), so we claim no trend beyond the smallest N.

Table 16: Downstream data sweep on WikiText-103, 0.69M model: test perplexity after a 10k-step fine-tune, mean ± SD over 3 seeds. N is the number of fine-tuning documents; N=10k is the main experiment (Table 11). Last two columns: seed-paired differences (positive: PRECOCIAL lower), p in parentheses; bold: $p < .05$.

N	PRECOCIAL	Hand-written (π_0)	No pretraining	PRECOCIAL − hand	PRECOCIAL − none
2,000	5.454±.058	5.667±.033	5.863±.066	+**0.213** (.040)	+**0.410** (.024)
5,000	5.531±.056	5.572±.080	5.647±.016	+0.041 (.650)	+0.116 (.102)
10,000	5.312±.146	5.508±.082	5.585±.055	+0.196 (.112)	+0.273 (.076)
20,000	5.450±.085	5.503±.040	5.587±.022	+0.053 (.460)	+0.137 (.157)

G SEARCH DYNAMICS AND PRETRAINING LOSS

Figure 6 shows what the search does round by round, and Figure 7 what the network sees while it happens.

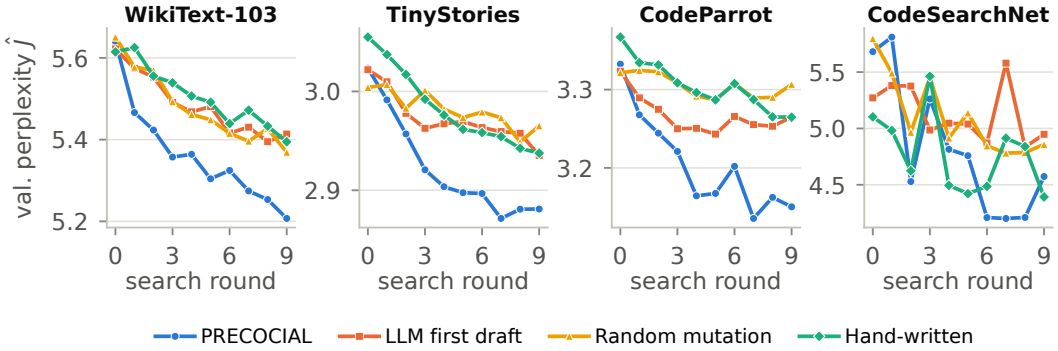


Figure 6: The search objective over rounds: validation perplexity $\hat{J}$ of each round's candidate, mean over 3 seeds. PRECOCIAL improves over the rounds and separates from the baselines on three of the four datasets; on CodeSearchNet every arm's trajectory is noisy, and the kept program is the best round rather than the last.

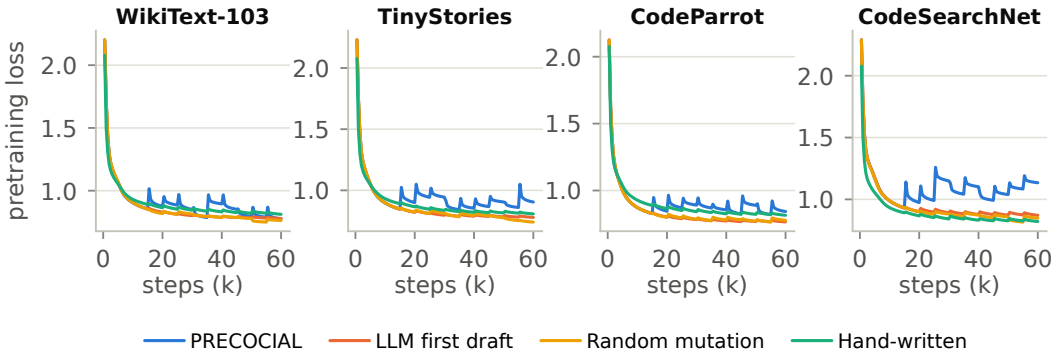


Figure 7: Pretraining loss during the search, concatenated over rounds (mean over 3 seeds); the x axis is thousands of pretraining steps. From round 1 on, PRECOCIAL sits *above* the first-draft and random-mutation arms (in round 0 all three train on the same first draft) and spikes at each round boundary, where the next candidate changes the data, yet reaches the best downstream perplexity (Table 11).

In Figure 7, from round 1 on PRECOCIAL's pretraining loss is higher than that of the first-draft and random-mutation arms on all four datasets, and in the last round it is also higher than the hand-written program's on three of them (on WikiText-103 the hand-written program ends with the highest pretraining loss of any arm), while PRECOCIAL's downstream perplexity is the lowest everywhere. Part of the gap is mechanical: the first draft and the hand-written program show the network the same distribution in every round, and random mutation only reweights, drops or duplicates the first draft's rules, whereas an LLM rewrite can add new kinds of data. This is also what produces the sawtooth: PRECOCIAL's loss rises at 32 of its 36 round boundaries. We therefore report the pattern without claiming that the searched data is intrinsically harder to fit; separating "harder data" from "newer data" would need a run that replays the searched programs in a fixed order.

H PLANNING DETAILS

Table 17: Planning, matched pretraining: final test perplexity on the answer tokens, mean ± SD (same runs as Table 5). Bottom rows: seed-paired difference in perplexity, positive when PRECOCIAL is lower.

Pretraining	Blocksworld	8-puzzle	Sokoban	Countdown
PRECOCIAL	1.059±.075	1.165±.009	1.440±.041	1.635±.016
LLM first draft	1.124±.056	1.193±.030	1.625±.038	1.638±.014
Hand-written (π_0)	1.119±.089	1.185±.013	1.566±.048	1.640±.016
No pretraining	1.265±.075	1.285±.019	1.628±.023	1.843±.119
PRECOCIAL − hand-written (ppl, + = PRECOCIAL lower)	+0.061 (.100)	**+0.019** (.009)	**+0.127** (.003)	+0.005 (.597)
PRECOCIAL − first draft (ppl, + = PRECOCIAL lower)	+0.065 (.071)	+0.027 (.083)	**+0.185** (.001)	+0.003 (.733)

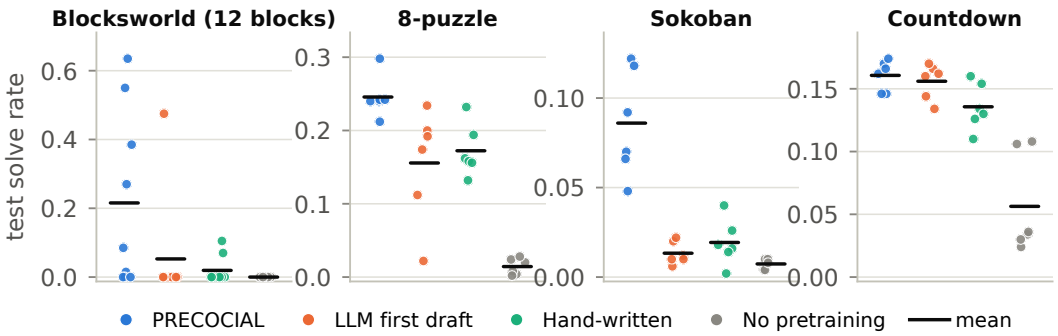


Figure 8: Every matched-pretraining run of Table 5 (dots; bars: means). On Blocksworld the solve rate is nearly all-or-nothing: it is a step function of the test perplexity the fine-tuned model reaches (Table 18), so single runs sit at zero or well above it.

Blocksworld is nearly all-or-nothing. A 12-block plan is about 30 actions and every action must be legal and correct, so a model either produces whole correct plans often or almost never. Across the 33 Blocksworld runs of Table 5 the solve rate is a step function of the test perplexity the fine-tuned model reaches (Table 18; Spearman $\rho = -0.78$ over 33 runs, $p = 6.5 \times 10^{-8}$): all 9 runs below 1.04 solve some problems (0.015 to 0.635) and all 24 runs above it solve none, with no run in between. This is why single Blocksworld runs sit either at zero or far above it (Figure 8), and why we use 9 runs per arm there. Table 17 gives the perplexities behind those solve rates.

Table 18: Blocksworld: mean solve rate of the 33 runs of Table 5, grouped by final test perplexity.

Test perplexity	<1.01	1.01–1.03	1.03–1.05	≥1.05
Runs	5	2	2	24
Mean solve rate	0.463	0.095	0.043	0.000

I COST

Table 19: Cost of one search with ten candidates on the 0.69M model: WikiText-103 and CodeSearchNet, 3 seeds each (6 searches per row), all on RTX A6000 GPUs. The five time columns are GPU-hours of one search and add up to the total (up to rounding). How to read each column is explained below the table.

	Pretraining	GPU-hours per search					LLM	API	Test
Search	steps	Pretrain	Fine-tune	LLM	Other	Total	calls	cost	ppl
PRECOCIAL (one network)	60,000	0.14	0.37	0.17	0.02	0.71±.12	10.8	$1.20	4.659
FunSearch (network per candidate)	150,000	0.36	0.37	0.12	0.02	0.88±.17	9.0	$1.10	4.890
AlphaEvolve (network per candidate)	150,000	0.35	0.36	0.12	0.02	0.85±.16	9.3	$1.08	4.955

Reading Table 19. Each row is one way of running a search that evaluates ten candidate programs. *Pretraining steps* is the number of synthetic pretraining steps summed over the whole search, not a fine-tuning budget: PRECOCIAL pretrains one network for 15k steps for the first candidate and

continues it for 5k steps for each of the nine later ones ($15\text{k} + 9\times5\text{k} = 60\text{k}$), whereas FunSearch and AlphaEvolve as published pretrain a fresh network for 15k steps for each of the ten candidates ($10\times15\text{k} = 150\text{k}$). The five time columns split one search into synthetic pretraining; fine-tuning the ten candidates for up to 10k steps each, which is identical under both protocols; waiting for the LLM; everything else (generating the synthetic data, evaluation and bookkeeping); and their total, mean $\pm$ SD over the six searches. *LLM calls* counts every call, including retries of programs that failed validation, and *API cost* prices them at Claude Sonnet 4 rates. *Test ppl* is the test perplexity of the program each search kept after a 10k-step fine-tune, averaged over the six searches (three on each dataset). One network cuts pretraining time by 59–61%, but fine-tuning is unchanged and takes 42–53% of a search, so the total falls by 17.1–19.5%.

Longer searches and larger models (projection). From the WikiText-103 searches of Appendix F (3 per model size, RTX A6000), one candidate costs 0.039 GPU-h of fine-tuning at 0.69M parameters and 0.067 at 3.36M, one LLM call 0.015, and everything else 0.003, while synthetic pretraining costs 2.54 GPU-h (0.69M) and 5.07 (3.36M) per million steps. With one network a search of n candidates pretrains $15\text{k} + 5\text{k}\,(n-1)$ steps; with a fresh network per candidate, $15\text{k}\,n$. Table 20 multiplies these rates out. These are projections, not runs. At ten candidates they reproduce the measured one-network searches (0.73 against 0.71 GPU-h at 0.69M, 1.15 against 1.18 at 3.36M), but they predict a 24% saving at 0.69M against the 17–20% measured in Table 19, so they overstate the saving somewhat. The saving grows with the number of candidates and with model size, because pretraining is a larger share of each candidate's cost for a larger model.

Table 20: Projected GPU-hours of one search, with one network continued across candidates ("one") and with a fresh network per candidate ("fresh"), from the per-candidate rates above. Last column: synthetic pretraining steps summed over the search, one vs fresh.

	0.69M model			3.36M model			
Candidates	one	fresh	saving	one	fresh	saving	Pretraining steps (one vs fresh)
10	0.73	0.95	24%	1.15	1.61	28%	60,000 vs 150,000
100	7.03	9.54	26%	11.1	16.1	31%	510,000 vs 1,500,000
1,000	70.0	95.4	27%	110	161	31%	5,010,000 vs 15,000,000

J RULE CLASSIFICATION

Rules are classified by keyword matching on their names, which are the LLM's own and usually say what the rule does: *planning* (e.g. `pddl`, `plan`, `action`, `state`, `precond`, `grid`, `move`, `puzzle`, `sokoban`), *arithmetic* (`arith`, `expr`, `math`, `fraction`, `make24`), *code syntax* (`func`, `class_def`, `import`, `method`, `param`, `try_catch`), list and string operations, logic, cellular automata, and formal structure (`nest`, `dyck`, `tree`, `cfg`, `stack`); a name takes the first matching type in a fixed order. The share of a type is its fraction of the synthetic data generated in a round, by volume. The classifier is a keyword map, so single percentages are approximate; the script that assigns every name is released with the code.