

Towards Training Transformers as a Universal Computer

Ruize Xu¹ Chenxiao Yang² Yanhong Li³ David McAllester²

¹The University of Chicago ²Toyota Technological Institute at Chicago

³Allen Institute for AI

richardxu@ttic.edu

{chenxiao,mcallester}@ttic.edu yanhongl@allenai.org

Abstract

We demonstrate that a small transformer can learn to execute programs in MicroPy, a simplified yet computationally universal programming language. Given procedure definitions together with an expression to evaluate, the transformer predicts small-step execution using PENCIL scaffolding for space-efficient execution within a bounded context window. After training on randomly generated, semantically arbitrary MicroPy programs, the learned transformer generalizes to held-out human-written programs including bit copying and flipping, binary addition and multiplication, and SAT verification and solving. The trained model exhibits out-of-distribution generalization: it correctly evaluates programs drawn from a substantially different distribution than the one used to generate its training data. Since MicroPy can express any computation, our results provide empirical evidence that a standard transformer can be trained to act as a universal computer when paired with PENCIL scaffolding. Project page: micropy.org.

1 Introduction

Transformer-based language models have achieved remarkable empirical success across a wide range of tasks. Beyond empirical performance, recent theoretical work shows that transformers augmented with Chain of Thought (CoT) (Wei et al., 2022; Nye et al., 2021; Kojima et al., 2022) are computationally universal: given sufficiently many intermediate steps, a transformer of constant size and depth can simulate Turing machines (Merrill and Sabharwal, 2023; Yang et al., 2025). In other words, in principle, any computable problem can be solved by a transformer with CoT.

However, there is a significant gap between what transformers can represent in principle and what they can learn to do through training. In particular, it is unclear whether gradient-based optimization with next-token prediction can produce a trans-

former that actually exhibits universal computational behavior. This is a non-trivial challenge: even on much narrower tasks, such as length generalization in addition (Zhou et al., 2023, 2024) and constraint-satisfaction problems like Sudoku and 3-SAT (Seely et al., 2025; Hazra et al., 2024), current language models with CoT still fall short. The question is also interesting on its own terms: a positive answer would mean there exists a programming language and a task representation under which expressiveness translates into learnability for transformers, and that such a transformer-native combination of language and intermediate-step structure is worth searching for. This motivates a central question: **can we train a transformer to be a universal computer?** Recent work on Neural Computers asks a related question of whether learned systems can acquire computer-like primitives from I/O traces (Zhuge et al., 2026). The closest prior work falls short of answering our question. Reed and de Freitas (2016) train a neural programmer-interpreter using a seq2seq model, but only for specific, pre-designed functions. Giannou et al. (2023) show that a looped transformer can simulate a universal computer (a random access machine), but this is still an expressiveness result.

In this paper, we provide evidence that the answer is yes. We design MicroPy, a simplified yet Turing-complete programming language, and train a small transformer to execute MicroPy programs step by step, using PENCIL scaffolding (Yang et al., 2025) for space-efficient execution within a bounded context window. Since MicroPy can express any computation, a model that learns to interpret it acts as a universal computer. In particular:

MicroPy. MicroPy supports procedure calls, conditionals, recursion, and mutable state over objects with attributes. The input to the model is a set of procedure definitions together with an expression to evaluate; the model then generates, token by to-

ken, a step-by-step execution trace. Each step in the trace is either a *retrieval* (looking up a procedure definition, variable binding, or attribute value from the context) or a local *rewrite* (reducing a primitive such as IF, SEQ, or LookupAttr to a simpler expression). A key design principle is that MicroPy’s semantics consists of a small, finite set of such retrieval and rewrite rules, so a small subset of training programs is sufficient to cover every rule type at least once.

Beyond language design, a challenge is that CoT’s universality comes with an important caveat: intermediate computations accumulate indefinitely, so a task with time complexity T requires context length $\mathcal{O}(T)$ (Merrill and Sabharwal, 2023). This couples task difficulty directly to context length, making both training and generalization difficult: to handle harder tasks, the model must operate correctly in contexts far longer than those seen during training, i.e. a length generalization problem; moreover, as context grows, the model’s ability to retrieve relevant information degrades (Liu et al., 2024). To address this, we adopt PENCIL scaffolding (Yang et al., 2025).

PENCIL Scaffolding. PENCIL introduces a reduction rule that reclaims completed intermediate computations from the context, so that the context length is bounded by $\mathcal{O}(S)$, where S is the space complexity of the simulated computation, rather than its time complexity T . Specifically, when the context has the form

$$\alpha \text{ [call] } \beta \rightarrow \gamma \text{ [ret]} \quad (1)$$

where [call], $\rightarrow$, and [ret] are special PENCIL tokens, PENCIL replaces the entire sequence with $\alpha \gamma$, erasing the intermediate reasoning β and keeping only the conclusion γ . Intuitively, $\alpha \text{ [call] } \beta$ can be viewed as a stack where β is the top frame; γ is the value returned when that frame is popped. This implements a form of stack discipline analogous to function calls in programming languages: once a sub-computation completes, its intermediate steps are discarded and only the result is kept. In practice, PENCIL enables the model to execute much longer programs within a fixed context window. Figure 2 gives a space-efficiency analysis on the held-out tasks, comparing PENCIL against append-only traces under the same 4096-token budget.

Experiments. We trained a 59.5M-parameter decoder-only transformer with RoPE (Su et al.,

2024) on execution traces generated from random MicroPy programs, capped at 128 trace lines per program. We evaluated on held-out, human-written MicroPy programs that are substantially longer and semantically meaningful: bit copying and flipping, binary addition and multiplication (with traces up to 7,552 lines), and SAT verification and solving. The model achieves 100% token-level accuracy on all tasks within the tested context window. The near-zero overlap between the training-program distribution and the held-out evaluation suite (Figure 4a) provides direct evidence of both compositional and length generalization.

To summarize, our main contributions are:

1. We design MicroPy, a computationally universal language.
2. We provide the first empirical evidence that a standard transformer can be trained to simulate a universal computer by running a Turing-complete language under PENCIL scaffolding, up to the model’s context window.
3. We demonstrate strong generalization: 100% token-level accuracy on held-out human-written programs with traces up to $60\times$ longer than those seen during training, under PENCIL scaffolding.

Broader Implication. It is well known that generalization requires inductive bias (the no free lunch theorem; Wolpert 1996), and not all universal priors are equally efficient. Our results suggest that transformers with suitable scaffolding like PENCIL may possess an inductive bias well-suited to phrase structure and compositional semantics, the core structures shared by all high-level programming languages, each of which is computationally universal. Since natural language exhibits analogous compositional structure, this may shed light on why transformers, with context management scaffoldings, are so effective at natural language tasks: the same implicit bias that enables learning a programming language interpreter may also facilitate learning the compositional semantics of natural language.

2 MicroPy and Program Execution

In this section, we describe the MicroPy language and how programs are executed. We first present the two execution primitives, retrieval and rewriting (Section 2.1), then describe the evaluation of procedure calls and tail recursion (Section 2.2), and

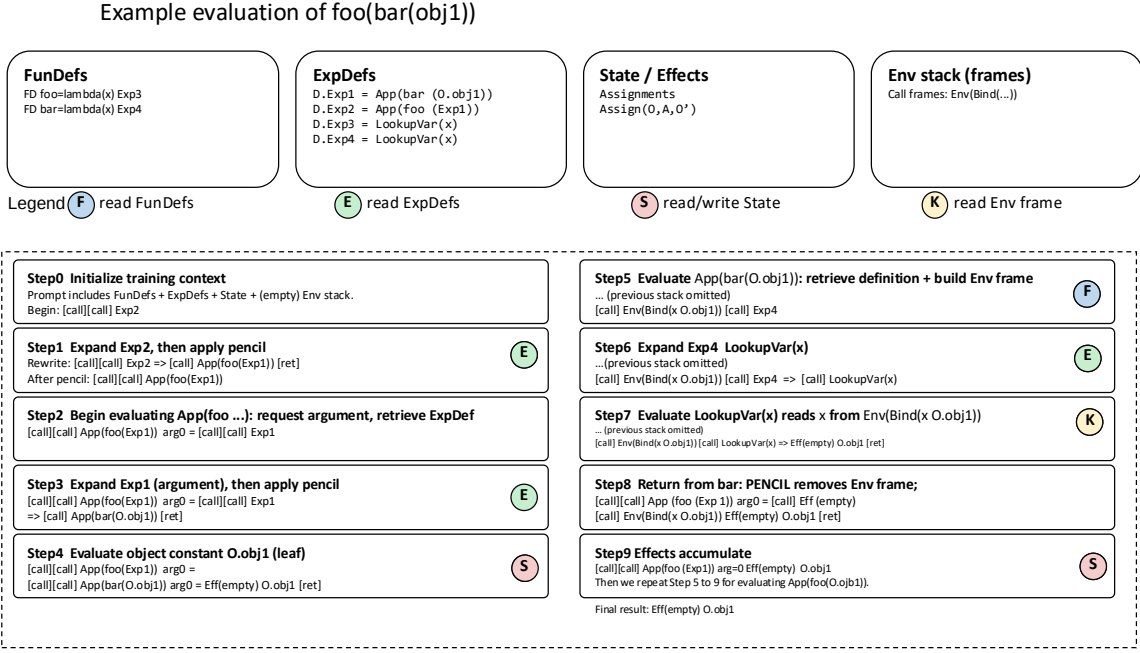


Figure 1: Evaluation of a MicroPy expression. We define foo and bar to both be the identity function and evaluate foo(bar(O)) where O is an object symbol. The general rules for evaluation are described in the text. More details about what each step is doing and the exact definitions on this table can be found in Section C.

finally give the formal definition of the MicroPy language (Section 2.3).

2.1 Retrieval and Rewriting

The evaluator relies on two conceptual primitives: retrieval and rewriting.

We first focus on retrieval. Our evaluator uses five kinds of symbols: function symbols, expression symbols, variable symbols, object symbols and attribute symbols. The context contains definitions for function symbols, expression symbols, and variable symbols. For example a function definition is (some version of) $f(x_1, \dots, x_n) = e$ where f is a function symbol, each x_i is a variable symbol, and e is an expression symbol. An expression definition has a form such as $e_0 = \text{IF}(e_1, e_2, e_3)$ where e_0, e_1, e_2 and e_3 are expression symbols. Each expression definition specifies a primitive of the language and expression symbol arguments for that primitive. The primitives include looking up the value of a variable. Variable values are held in stack frames. The value of a variable must be retrieved from the most recent stack frame. The context also contains a state consisting of triples $\langle O_1, A, O_2 \rangle$ stating that the value of attribute A of object O_1 is the object O_2 . The language supports effects which can change the value of an attribute. As with variables, we need the most

recent assignment of a value to a given attribute of a given object. The transformer must be capable of retrieving the values of function symbols, expression symbols, variable symbols and attribute values. Each retrieval does a replacement. For example $\text{LookupVar}(x)$ at the end of the context gets replaced by the value of x in the current stack frame (using a PENCIL rewrite).

In addition to the required retrievals, we need to be able to compute within an active stack frame. This involves computing the value of expressions under the variable values in the stack frame.¹ We can think of computation within a given stack frame as a rewriting process. For example $\text{IF}(e_1, e_2, e_3)$ gets rewritten to e_2 if the value of e_1 is the special object true and rewritten to e_3 if the value of e_1 is not true.

2.2 Procedure Calls and Tail Recursion

Figure 1 shows an example of evaluating an expression using the transformer context for the required memory. Here we outline the general steps of evaluation. A call to a procedure is a context of the

¹Variable symbols are analogous to registers and a definition of an expression symbol is analogous to a machine instruction in a RAM model of computation. Here there are no primitive arithmetic operations. However, arithmetic can be implemented as computation on bit strings.

form

$$\alpha [\text{call}] e_1 [\text{call}] \text{Apply}(f, E_1, \dots, E_n) \quad (2)$$

where e_1 is a “state” specifying the values of attributes of objects, f is a function symbol and $E_1, \dots, E_n$ are expression symbols. This is eventually rewritten to

$$\alpha [\text{call}] e_2 [\text{call}] \text{TailApply}(f, O_1, \dots, O_n) \quad (3)$$

where $O_1, \dots, O_n$ are object symbols and e_2 is a new attribute-value state that combines e_1 with effects performed during the evaluation of the argument expressions.

In the normal case (where there is no tail recursion optimization), and for f defined by $f(x_1, \dots, x_n) = E$, we have that (3) gets rewritten to

$$\alpha [\text{call}] e_2 [\text{call}] \text{BIND}((x_1, O_1), \dots, (x_n, O_n)) [\text{call}] E \quad (4)$$

Here $\text{BIND}((x_1, O_1), \dots, (x_n, O_n))$ is a stack frame holding the values for the “local variables” $x_1, \dots, x_n$. It is possible that (4) gets rewritten to

$$\alpha [\text{call}] e_2 [\text{call}] \text{BIND}((x_1, O_1), \dots, (x_n, O_n)) e_3 O \quad (5)$$

where O is an object symbol. In this case (3) gets rewritten to

$$\alpha e_4 O \quad (6)$$

where e_4 is a combination of the effects e_2 and e_3 .

However, an alternative possibility is that (3) gets rewritten to

$$\alpha [\text{call}] e_2 [\text{call}] \text{BIND}((x_1, O_1), \dots, (x_n, O_n)) e_3 c \quad (7)$$

where $c = \text{TailApply}(g, O_1, \dots, O_m)$. In this case (3) gets rewritten to

$$\alpha [\text{call}] e_4 [\text{call}] \text{TailApply}(g, O_1, \dots, O_m) \quad (8)$$

Note that (8) has the same form as (3). If $g = f$ we get a form of iteration.

2.3 The MicroPy Language

As described above, MicroPy is a Turing-complete language whose state is built from symbols; it can be used as an object-oriented programming language by treating these symbols as objects with attributes. A state of the system is represented by a sequence of triples $\langle O_1, A, O_2 \rangle$ stating the value of the attribute A of object O_1 is the object O_2 . A MicroPy procedure definition has the form

$$f(x_1, \dots, x_n) = E$$

where $x_1, \dots, x_n$ are variable symbols and E is a MicroPy expression symbol. An expression definition has one of the forms $E_0 = \text{App}(f, E_1, \dots, E_n)$ or $\text{LookupVar}(x)$, or $E_0 = \text{op}(E_1, \dots, E_n)$ where op is a primitive operation and each E_i is either an expression symbol or an attribute symbol. The primitive operations are the following.

- $\text{IF}(E_1, E_2, E_3)$: conditional expression.
- $\text{EQUAL}(E_1, E_2)$: equality test.
- $\text{HasAttr}(E, A)$: returns true if the value of E has a value assigned for attribute A .
- $\text{Assert}(E_1, A, E_2)$: updates the value of attribute A of the value of E_1 to the value of E_2 .
- $\text{LookupAttr}(E, A)$: returns the value of attribute A of the value of E .
- $\text{SEQ}(E_1, E_2)$: evaluates E_1 for effect, then evaluates and returns E_2 .
- $\text{Try}(E_1, E_2)$: evaluates E_1 ; if the result is the special object fail, all effects of E_1 are rolled back and E_2 is evaluated instead. Otherwise, the result and effects of E_1 are kept.

In all test data the state is initialized to contain linked lists defined by the attribute "next". For example a list of "bit objects" would be represented by a state with attribute values such as $\text{Assign}(\text{BIT5}, \text{NEXT}, \text{BIT6})$. Bit strings can be represented by assigning a value to each bit in a list of bit objects. For example $\text{Assign}(\text{BIT7}, \text{VALUE}, \text{TRUE})$. Bit strings can be interpreted as binary numbers and arithmetic operations such as addition and multiplication can be computed on bit strings.

It is also possible to initialize the state to a long linked list of objects representing a heap memory. We can have an object "freepit" whose value is one of the objects in the heap. This allows one to implement heap allocation of data structures. We can then implement list operations such as appending two lists.

Although not done in our current experiments, it is possible to replace the assert primitive with a pairing primitive $\text{Pair}(e_1, e_2)$. The pairing primitive would create a new object with "first" and "second" attributes that are assigned the values of e_1 and e_2 respectively. With a pairing primitive one could write purely functional (effect-free) programs. We see no reason to believe that our experimental re-

sults would be any different in a purely functional language.

Model input and prediction target. Figure 1 breaks the evaluation of `foo(bar(obj1))` into local transitions. For the first transition in that example, the model receives the following flat token sequence containing the procedure dictionary (FunDefs), expression dictionary (ExpDefs), accumulated pending state updates (Eff), active variable-binding frames (Env), and execution stack. Tokens through [EOP] form the input, and the continuation after [EOP] is the prediction target (line breaks are added for readability):

```
MODEL INPUT:
FunDefs (
  FD foo = lambda ( x ) Exp3 ;
  FD bar = lambda ( x ) Exp4 ; )
***** ExpDefs (
  D . Exp1 = App ( bar ( 0 . obj1 ) )
  D . Exp2 = App ( foo ( Exp1 ) )
  D . Exp3 = LookupVar ( x )
  D . Exp4 = LookupVar ( x ) )
***** Eff ( empty )
***** Env ( ) [call] [call] Exp2 [EOP]
PREDICTION TARGET:
=> [call] App ( foo ( Exp1 ) ) [ret]
```

PENCIL applies the predicted transition to construct the next model input, and evaluation proceeds one local transition at a time. The complete walkthrough appears in Section C.

3 Training and Experiments

This section describes how we generate training data (Section 3.1), the model and training setup (Section 3.2), and the experimental results (Section 3.3).

3.1 Data Generation

Every PENCIL step in a MicroPy execution (Equation (1)) is one of two things: a *retrieval* that reads a definition or binding from one of the context blocks (FunDefs, ExpDefs, Env, Eff/State), or a *rewrite* that reduces a primitive (If, Seq, LookupAttr, and so on) using values and effects already on the stack (Figure 1). There are finitely many of each, so a model that learns this fixed set of moves can execute any MicroPy program up to the context window. Training data should therefore cover these moves directly, not particular programs.

Online trace generation. Training data is generated online: at each step we sample a MicroPy expression, run it, and turn one of its PENCIL trace lines into a (prompt, completion) pair. We call each such pair, which implements one local retrieval or rewrite before PENCIL reduction, a *trace step*. We

use two samplers. The first samples whole MicroPy programs directly. The second samples a *plan*, a target stack configuration, then compiles it into an expression that hits that configuration and uses only the matching trace line.

Program sampler. The first sampler $\mathcal{G}_{\text{prog}}$ is a probabilistic context-free grammar (PCFG) over MicroPy procedure definitions and expressions, capped by global depth/size/effect/binding budgets. We sample a program $e \sim \mathcal{G}_{\text{prog}}$, execute it in the MicroPy interpreter, and turn each PENCIL rewrite in the trace into a training example. This gives wide syntactic coverage but no direct handle on rare runtime configurations.

Plan sampler. The second sampler targets a specific shape directly. We first sample a *plan* π that says what one trace step should look like at the deepest call-stack tail. A plan consists of a stack skeleton together with binding and effect requirements that the realized program must satisfy. The skeleton is an ordered sequence of frame labels printed after [call] along the deepest call-stack tail; it ends in one leaf retrieval (LOOKUPVAR, LOOKUPATTR, or TAILAPP) wrapped by some number of continuation frames (SEQ, IF, TRY, ENV, or EFF). The plan sampler $\mathcal{G}_{\text{plan}}$ draws the skeleton recursively: start from a leaf and, with each step, either stop or wrap the current skeleton with one more continuation frame, until a depth budget is reached. The binding and effect requirements are then turned into concrete `Env(. . .)` bindings and `Eff(. . .) / State/Effects` assertions during compilation. When the skeleton is compiled into an expression, we route the recursive descent through the first child of SEQ, the condition of IF, or the second child of TRY, so that the sampled labels land contiguously in the deepest tail at runtime. Figure 3 shows an example plan with a matched trace step.

From plan to training example. Compiling a plan produces an executable expression e_π . Running e_π generates a trace whose deepest call-stack tail matches the planned skeleton at exactly one step, and we use that step’s prompt and completion as the training example; other steps from the same execution are discarded. To vary what surrounds the planned step (the outer stack prefix, unrelated bindings, distractor effects), we optionally drop e_π into a larger expression sampled by $\mathcal{G}_{\text{ctx}}$ and execute that instead. The remaining training budget is

filled with examples from $\mathcal{G}_{\text{prog}}$.

Program sampler parameterization. The program-level sampler $\mathcal{G}_{\text{prog}}$ is implemented via two coupled samplers: (i) a *surrounding-code sampler* $\mathcal{G}_{\text{ctx}}$ that generates enclosing expression templates with a designated insertion point, and (ii) an *expression sampler* $\mathcal{G}_{\text{expr}}$ that generates the side expressions used by those templates. Rule probabilities are renormalized under depth/size/effect/binding budgets.

Surrounding-code sampler (general stack section). Templates are drawn from a separate sampler $\mathcal{G}_{\text{ctx}}$: starting from a hole $\square$ where the planned expression will be inserted, $\mathcal{G}_{\text{ctx}}$ recursively wraps the hole with a compositional MicroPy operator (SEQ, IF, TRY, APP, LET, LOOKUPATTR), filling the non-hole argument slots with side expressions drawn from $\mathcal{G}_{\text{expr}}$. IF/TRY place the hole on *both* branches so the planned expression is reached under either sampled branch outcome. Wrapping continues under depth/size budgets until the hole becomes a complete enclosing context, at which point the planned expression is substituted in. As above, we retain only executions whose realized trace contains the planned configuration in the intended stack position.

Expression sampler (side expressions). Side expressions are drawn from a sampler $\mathcal{G}_{\text{expr}}$ defined recursively: given the planned bindings and effect requirements, at each step $\mathcal{G}_{\text{expr}}$ either places a compositional MicroPy operator (SEQ, IF, TRY, APP, LET, LOOKUPATTR) and recurses into its argument slots, or stops at a leaf that reads a planned binding (LOOKUPVAR) or emits a Bool/object constant. Probabilities over the operator and leaf categories give us per-op control over which MicroPy primitives the sampled expression exercises.

3.2 Model and Training Setup

Vocabulary and tokenizer. We use a whitespace tokenizer with a fixed vocabulary containing MICROPY-specific syntax and formatting tokens (e.g. “If ” and the delimiter “[call] ”). A fixed vocabulary is appropriate here because the language emits a small, finite set of syntactic forms; we trade flexibility for unambiguous token boundaries during step-wise prediction. The implications of this choice are discussed in the Limitations section.

Model and optimization. We train a 59.5M-parameter decoder-only transformer (depth 10)

with rotary position embeddings (Su et al., 2024) on traces from the two samplers of Section 3.1. Following the NanoChat recipe (Karpathy, 2025), we use AdamW for token embedding and unembedding (Loshchilov and Hutter, 2019) and the Muon optimizer for transformer-block matrices (Jordan et al., 2024), with learning rates 0.2/0.004 (Adam) and 0.02 (Muon). We train for 250,000 iterations at maximum sequence length 4096 and global batch size 1,966,080 tokens, with 5% linear warmup, final-LR decay to 5% of peak, and no weight decay. All reported numbers use greedy next-token decoding under a single training seed; hyperparameters were not selected by sweep but follow the NanoChat defaults. Training runs on a single 8-GPU node of a university cluster in bfloat16 (about one node-week of wall-clock time). The training-data generator, trained checkpoint, evaluation programs, and program-execution code will be released under a permissive open-source license through the project page upon publication. Full compute, data-volume, and dataset-statistics details are reported in Section A.

Evaluation. We evaluate on actual human-written MicroPy functions: COPY, FLIP, ADDITION, MULTIPLICATION, SAT-VERIFY, and SAT-SOLVER. The function definitions are in Section D and per-task statistics in Table 1 and Table 2. We achieve 100% token-level accuracy on each task within the tested context lengths.

3.3 Results

Compositional and length generalization. Figure 4a compares the program-length distribution of generator-sampled training programs with the held-out evaluation suite. In theory we do not have to train on any concrete program. In practice, we cap generator-sampled training programs at 128 trace lines. The held-out evaluation suite (Table 1 and Table 2) contains much longer total executions, with per-program traces reaching 7,552 lines (about $60\times$ longer than the training cap on total trace length). PENCIL keeps the per-step input bounded at 4096 tokens regardless of total trace length, so the $60\times$ figure refers to trace length, not to the model’s per-step input. The near-zero overlap between the training-generator distribution and the test suite provides a direct visualization of length generalization at the trace level.

Coverage of MicroPy primitives. Our plan-based generator is designed to cover all proce-

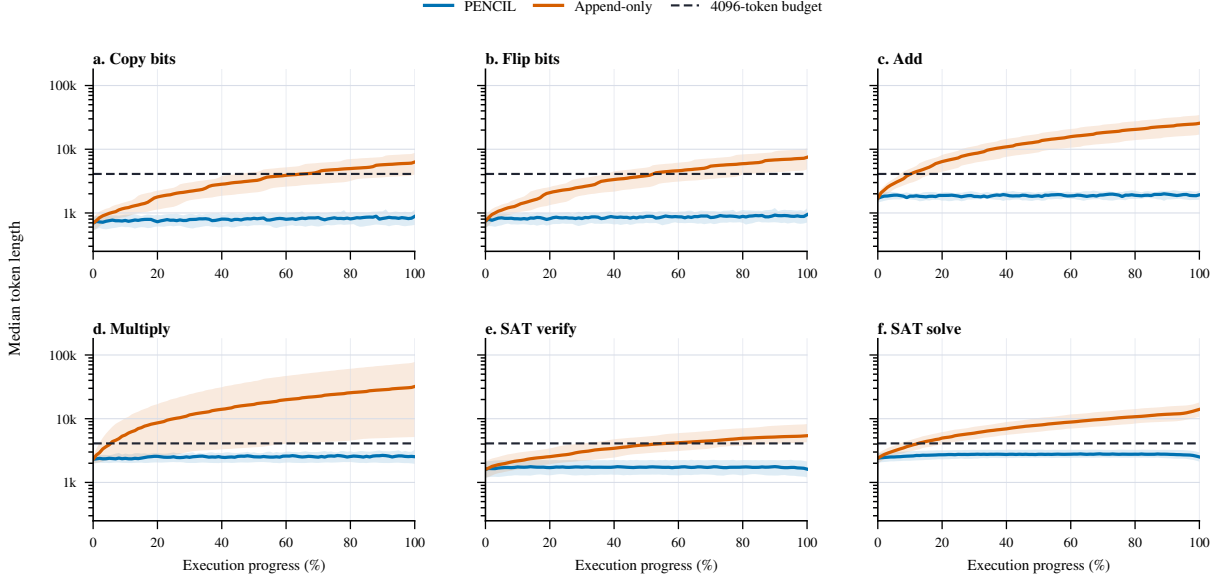


Figure 2: PENCIL bounds the live token length during long MicroPy executions. For each held-out human-written task, we compare the median token length over normalized execution progress under PENCIL scaffolding and under an append-only transcript that retains all previous completions. Shaded regions show the interquartile range across programs, and the dashed line marks the 4096-token budget used by the model. Without PENCIL, many simple programs require very long live contexts: append-only traces grow with execution progress and can exceed the 4096-token budget before completion, substantially increasing the difficulty of training on longer executions. This analysis isolates the space-efficiency role of PENCIL: both curves use the same evaluated programs and next-step completions, differing only in prior-text retention.

Task	bit-len	ctx _{min}	ctx _{max}	lines _{min}	lines _{max}	#programs	#trace lines	avg lines/program	accuracy
copy_bits	2–10	215	725	78	422	9	2,250	250.0	100%
flip_bits	2–10	253	774	108	572	9	3,060	340.0	100%
RPC_add	2–10	722	1514	469	2365	9	12,753	1,417.0	100%
RPC_mult	2–10	971	2197	80	7552	9	24,379	2,708.8	100%
Total (bit-length tasks)						36	42,442	1,178.9	100%

Table 1: Test dataset statistics for bit-length tasks. Context length is tokens per step; lines_{min} / lines_{max} are per-program execution lengths; #trace lines is the total number of step-wise prompt/completion pairs. Accuracy is the token-level success rate.

ture combinations: for any retrieval/rewrite pattern permitted by MicroPy’s small-step semantics, specified as a target call-stack skeleton with concrete Env/Eff binding shapes and retrieval channels, there exists a plan whose execution realizes that runtime configuration. The pooled operation-profile comparison and per-task breakdowns in Figure 5 show that the generator covers every operation type used by the test tasks. Without a budget and subject only to the length constraint, this implies a non-zero sampling probability for any finite MicroPy program. The remaining gap to larger programs is therefore an engineering constraint of the context window, which we discuss next.

4 Related Work

Our work connects to several lines of research. **CoT and bounded context:** Transformers with CoT are theoretically universal but the context grows linearly with computation time (Merrill and Sabharwal, 2023; Li et al., 2024); PENCIL reclaims intermediate steps to bound context by space rather than time (Yang et al., 2025). **Length generalization:** Prior work improves extrapolation via task-specific representations such as position coupling (Cho et al., 2024; McLeish et al., 2024) or Turing-machine-style decomposition (Hou et al., 2024; Abbe et al., 2024); MicroPy applies the decomposition principle at the level of a programming language, training one model for all tasks.

Task	#vars	#clauses	ctx _{min}	ctx _{max}	lines _{min}	lines _{max}	#programs	#trace lines	avg lines/program	accuracy
sat_solve	2–4	1–6	941	2091	188	5020	180	172,251	957.0	100%
sat_verify	2–4	1–6	476	1398	69	628	36	10,015	278.2	100%
Total (SAT tasks)							216	182,266	843.8	100%

Table 2: Test dataset statistics for SAT tasks. Context length is tokens per step; lines_{min} / max are per-program execution lengths; #trace lines is the total number of step-wise prompt/completion pairs. Accuracy is the token-level accuracy.

```

PLAN :
  Eff -> Env -> Seq -> Eff -> Env -> Eff -> Eff -> If ->
  ↳ Try -> Seq -> Eff -> TailApp

MATCHED STEP (general stack):
***** Env ( )
[call] Eff ( Assertion ( 0 . env0 Att . value False )
↳ Assertion ( 0 . env1 Att . value False )
      Assertion ( 0 . noise0 Att . k 0 . env1 )
      ↳ Assertion ( 0 . noise1 Att . k 0 . env1
↳ ) )
[call] Env ( Bind ( p 0 . plan0 ) Bind ( b 0 . lj05H0 )
↳ Bind ( x0 0 . env0 ) Bind ( x1 0 . env1 ) ... )
[call] Seq ( Exp22 , False ) x =
[call] Eff ( empty )
[call] Env ( ... Bind ( y0 0 . env0 ) Bind ( y1 0 . env2
↳ ) )
[call] Eff ( Assertion ( 0 . env0 Att . value True ) )
[call] Eff ( Assertion ( 0 . env2 Att . value False ) )
[call] If ( Exp38 False False ) test =
[call] [call] Try ( Exp34 Exp37 ) second =
[call] [call] Seq ( Exp36 , Exp32 ) x =
[call] Eff ( empty )
[call] TailApp ( _tailapp_id Args ( 0 . env0 ) )

result:
=> [call] let ( z ) be ( 0 . env0 ) in Exp10 [ret]

```

Figure 3: Sampled plan and matched trace step (general stack). Each [call] line is followed by the frame-label sequence specified by the plan. Env frames contain active variable bindings, and Eff frames contain accumulated pending state updates. Some binding lists are omitted for readability.

Neural interpreters and computers: NPI (Reed and de Freitas, 2016; Cai et al., 2017) and learned execution (Zaremba and Sutskever, 2014; Nye et al., 2021; Liu et al., 2023) explore learned program execution but rely on specialized architectures or task-specific training; we use a standard transformer with next-token prediction. Grau-Moya et al. (2024) meta-train sequence models on outputs of random BrainPhoque universal-machine programs to approximate universal next-symbol prediction. Their predictor observes emitted symbols but not the generating program or internal machine configuration; our model instead receives explicit MicroPy definitions and execution state and is supervised on the next local interpreter transition. Zhuge et al. (2026) propose Neural Computers that learn interface and control primitives from I/O traces in CLI and GUI settings; in con-

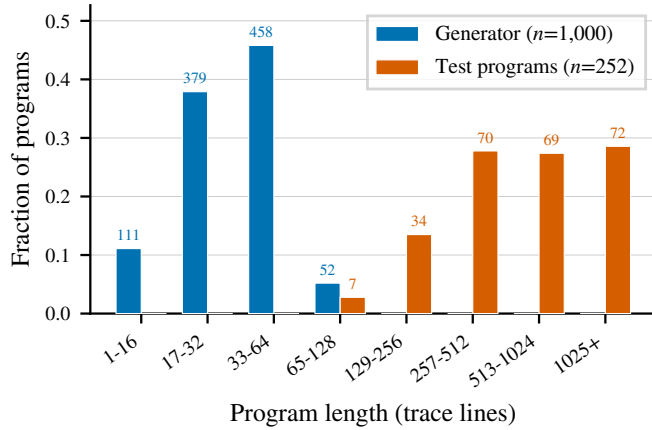
trast, we train a transformer to execute a formal, Turing-complete programming language with explicit operational semantics. **Compositional generalization:** Classical benchmarks such as SCAN, CFQ, and COGS test novel compositions of familiar primitives (Lake and Baroni, 2018; Keysers et al., 2020; Kim and Linzen, 2020); in our setting the composition rule is an operational semantics (Drozdov et al., 2023; Dziri et al., 2023). **External memory:** Retrieval augmentation and long-term memory modules (Lewis et al., 2020; Wang et al., 2023) address bounded context by storing information externally; our focus is on which execution state must stay in context for the next step to be well-defined. See Section B for a detailed discussion.

5 Conclusion

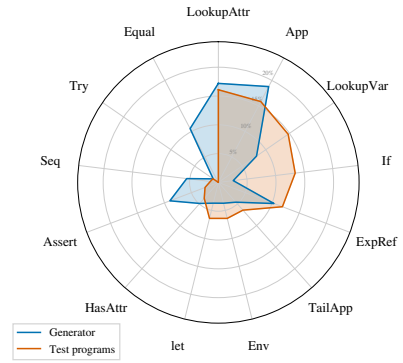
We showed that a standard decoder-only transformer, trained with next-token prediction on randomly generated MicroPy traces, can learn to act as a neural interpreter that generalizes to held-out human-written programs with 100% token-level accuracy. These results provide empirical evidence that, within a finite context window, a Transformer with PENCIL can learn the operational semantics of a computationally universal language. We view computational semantics as a useful lens for language-model agents and long-horizon-task systems, which similarly rely on transformers to maintain structured state and execute step by step over thousands of tokens. Limitations include restriction to the MicroPy language and a fixed context window; extending to richer languages, longer contexts, and reasoning about program behavior without execution are promising directions for future work.

Limitations

Finite execution regime. Our experiments use a fixed 4096-token context window and evaluate a finite suite of held-out programs. PENCIL reduces



(a) Program length distribution for generator-sampled training programs and the held-out evaluation suite (1,000 sampled vs. 252 human-written programs). The near-zero overlap beyond the 65–128 bin shows that the model is trained on short programs but evaluated on traces that are often much longer.



(b) Pooled operation-profile comparison: training generator (blue) vs. held-out suite (orange). The generator covers every operation type; the held-out programs place more mass on If, LookupVar, ExpRef, and TailApp.

Figure 4: Experiment summary plots: (a) length gap between training and held-out programs; (b) operation-profile gap.

live context from computation time to computation space, but programs whose live state exceeds the context window remain outside the evaluated regime. The reported results are empirical and do not establish arbitrary-length correctness.

Training and representation. The model is trained on synthetic traces from one procedural generator under a single training seed. Its behavior on execution shapes not represented by that generator remains unmeasured; Section F gives examples from ablated training distributions. The model also assumes the fixed prompt and trace format used in training. Its whitespace tokenizer has a finite vocabulary, including fixed pools of object and attribute symbols, which bounds the number of distinct names in one serialized program. We leave alternative tokenizations and broader training distributions to future work.

Language scope. MicroPy is a small language designed to expose retrieval, rewriting, mutable state, and stack management in a controlled setting. Our experiments do not test transfer to production languages or compare alternative computationally universal languages. Extending the study to established languages and additional execution interfaces is future work.

Ethical Considerations

Risks. Our contribution is a controlled study of program execution by a small transformer trained

on synthetic traces of a designed language. The trained model emits step-wise execution rewrites for MicroPy programs only and has no capability to access external systems, generate natural language, or execute code in production languages such as Python or C. We are not aware of dual-use risks specific to this artifact beyond those general to small language models trained on synthetic data. Insofar as our results support the broader claim that transformers can learn compositional, universal computation under suitable scaffolding, the same observation also applies to more capable systems, where the standard concerns about misuse of learned program executors (e.g., automated reasoning over untrusted inputs) apply.

Broader impact and intended use. The MicroPy interpreter is intended as a research vehicle for studying compositional and length generalization in transformer language models. We do not advocate using it as a substitute for verified interpreters in safety-critical settings. The released artifacts (training-data generator, model checkpoints, and evaluation suite) are intended for academic reproduction and follow-up research, and we encourage downstream users to preserve this scope.

Data and artifacts. All training data is synthetically generated by a procedural sampler we describe in Section 3.1 and contains no human-subject data, no personally identifying information, and no scraped web text. The held-out evaluation

programs (COPY, FLIP, ADDITION, MULTIPLICATION, SAT-VERIFY, SAT-SOLVER) are author-written MicroPy programs. We use the NanoChat training stack of [Karpathy \(2025\)](#) (MIT-licensed open source), the Muon optimizer of [Jordan et al. \(2024\)](#), and the PENCIL scaffolding rule of [Yang et al. \(2025\)](#); all are used consistent with their published intended use. Our generated dataset, our trained checkpoint, and the evaluation programs will be released under a permissive license upon publication.

Acknowledgments

We thank Tianchong Jiang, Anxin Guo, Xiaodan Du, and Zhiyuan Li for helpful discussions, and the staff at TTIC for maintaining the compute cluster used in our experiments. We also thank the anonymous reviewers for their constructive feedback.

References

- Emmanuel Abbe, Samy Bengio, Aryo Lotfi, Colin Sandon, and Omid Saremi. 2024. [How far can transformers reason? the globality barrier and inductive scratchpad](#). *Preprint*, arXiv:2406.06467. NeurIPS 2024.
- Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2023. [What learning algorithm is in-context learning? Investigations with linear models](#). In *International Conference on Learning Representations*.
- Jonathon Cai, Richard Shin, and Dawn Song. 2017. [Making neural programming architectures generalize via recursion](#). *Preprint*, arXiv:1704.06611. ICLR 2017.
- Hanseul Cho, Jaeyoung Cha, Pranjal Awasthi, Srinadh Bhojanapalli, Anupam Gupta, and Chulhee Yun. 2024. [Position coupling: Improving length generalization of arithmetic transformers using task structure](#). *Preprint*, arXiv:2405.20671. NeurIPS 2024.
- Jonathan Cook, Silvia Saporita, Arash Ahmadian, Akbir Khan, Tim Rocktäschel, Jakob Foerster, and Laura Ruis. 2025. [Programming by backprop: An instruction is worth 100 examples when finetuning LLMs](#). *arXiv preprint arXiv:2506.18777*.
- Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. 2022. [The neural data router: Adaptive control flow in transformers improves systematic generalization](#). In *International Conference on Learning Representations*.
- Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. 2023. Compositional semantic parsing with large language models. In *The Eleventh International Conference on Learning Representations*.
- 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. 2023. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems*. ArXiv:2305.18654.
- Dan Friedman, Alexander Wettig, and Danqi Chen. 2023. Learning transformer programs. *Advances in Neural Information Processing Systems*, 36.
- Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. 2022. [What can transformers learn in-context? A case study of simple function classes](#). In *Advances in Neural Information Processing Systems*.
- Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. 2023. [Looped transformers as programmable computers](#). *Preprint*, arXiv:2301.13196.
- Jordi Grau-Moya, Tim Genewein, Marcus Hutter, Laurent Orseau, Grégoire Delétang, Elliot Catt, Anian Ruoss, Li Kevin Wenliang, Christopher Mattern, Matthew Aitchison, and Joel Veness. 2024. [Learning universal predictors](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16178–16205. PMLR.
- Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. [Neural Turing Machines](#). *arXiv preprint arXiv:1410.5401*.
- Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. [Hybrid computing using a neural network with dynamic external memory](#). *Nature*, 538(7626):471–476.
- Rishi Hazra, Gabriele Venturato, Pedro Zuidberg Dos Martires, and Luc De Raedt. 2024. [Can large language models reason? a characterization via 3-SAT](#). *Preprint*, arXiv:2408.07215.
- Kaiying Hou, David Brandfonbrener, Sham Kakade, Samy Jelassi, and Eran Malach. 2024. [Universal length generalization with Turing Programs](#). *Preprint*, arXiv:2407.03310.
- Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyrillos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Veličković. 2022. [A generalist neural algorithmic learner](#). In *Learning on Graphs Conference*.

- Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. 2024. Muon: An optimizer for hidden layers in neural networks. <https://kellerjordan.github.io/posts/muon/>.
- Andrej Karpathy. 2025. nanochat: The best ChatGPT that \$100 can buy. <https://github.com/karpathy/nanochat>.
- 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. 2020. Measuring compositional generalization: A comprehensive method on realistic data. *arXiv preprint arXiv:1912.09713*.
- Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. *arXiv preprint arXiv:2010.05465*.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.
- Brenden M. Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. *arXiv preprint arXiv:1711.00350*.
- Guillaume Lample and François Charton. 2020. [Deep learning for symbolic mathematics](#). In *International Conference on Learning Representations*.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. *arXiv preprint arXiv:2005.11401*.
- Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. 2024. Chain of thought empowers transformers to solve inherently serial problems. *arXiv preprint arXiv:2402.12875*.
- David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. 2023. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36.
- Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. [Code execution with pre-trained language models](#). *Preprint*, arXiv:2305.05383. Findings of ACL 2023.
- Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.
- Ilya Loshchilov and Frank Hutter. 2019. [Decoupled weight decay regularization](#). In *International Conference on Learning Representations*.
- Sean McLeish, Arpit Bansal, Alex Stein, Neel Jain, John Kirchenbauer, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, Jonas Geiping, Avi Schwarzschild, and Tom Goldstein. 2024. [Transformers can do arithmetic with the right embeddings](#). *Preprint*, arXiv:2405.17399.
- William Merrill and Ashish Sabharwal. 2023. The expressive power of transformers with chain of thought. *arXiv preprint arXiv:2310.07923*.
- Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.
- Thomas Pierrot, Guillaume Ligner, Scott Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. 2019. [Learning compositional neural programs with recursive tree search and planning](#). *Preprint*, arXiv:1905.12941. NeurIPS 2019.
- Scott Reed and Nando de Freitas. 2016. [Neural programmer-interpreters](#). *Preprint*, arXiv:1511.06279. ICLR 2016.
- David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. [Analysing mathematical reasoning abilities of neural models](#). In *International Conference on Learning Representations*.
- Jeffrey Seely, Yuki Imajuku, Tianyu Zhao, Edoardo Cetin, and Llion Jones. 2025. [Sudoku-Bench: Evaluating creative reasoning with Sudoku variants](#). *Preprint*, arXiv:2505.16135.
- Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.
- A Vaswani, N Shazeer, N Parmar, J Uszkoreit, L Jones, A Gomez, L Kaiser, and I Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.
- Petar Veličković and Charles Blundell. 2021. [Neural algorithmic reasoning](#). *Patterns*, 2(7):100273.
- Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. 2023. [Transformers learn in-context by gradient descent](#). In *International Conference on Machine Learning*.

- Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. 2023. Augmenting language models with long-term memory. *Advances in Neural Information Processing Systems*, 36.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers. In *International Conference on Machine Learning*, pages 11080–11090. PMLR.
- David H. Wolpert. 1996. [The lack of a priori distinctions between learning algorithms](#). *Neural Computation*, 8(7):1341–1390.
- Chenxiao Yang, Nathan Srebro, David McAllester, and Zhiyuan Li. 2025. PENCIL: Long thoughts with short memory. *arXiv preprint arXiv:2503.14337*.
- Wojciech Zaremba and Ilya Sutskever. 2014. [Learning to execute](#). *Preprint*, arXiv:1410.4615.
- Dylan Zhang, Curt Tigges, Zory Zhang, Stella Biderman, Maxim Raginsky, and Talia Ringer. 2024. Transformer-based models are not yet perfect at learning to emulate structural recursion. *arXiv preprint arXiv:2401.12947*.
- Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. 2023. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*.
- Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. 2024. [Transformers can achieve length generalization but not robustly](#). *Preprint*, arXiv:2402.09371.
- Mingchen Zhuge, Changsheng Zhao, Haozhe Liu, Zijian Zhou, Shuming Liu, Wenyi Wang, Ernie Chang, Gael Le Lan, Junjie Fei, Wenxuan Zhang, Yasheng Sun, Zhipeng Cai, Zechun Liu, Yunyang Xiong, Ying Yang, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. 2026. Neural computers. *arXiv preprint arXiv:2604.06425*.

A Reproducibility, Compute, and Data Statistics

Compute. Training was run on a single 8-GPU node of an internal university cluster, using bfloat16 autocast and distributed data parallelism via torchrun. The cluster contains a mixture of NVIDIA RTX 6000 Ada, L40S, and H100 GPUs; the reported training run used whichever 8-GPU node was available at submission time. The 250,000-iteration run takes on the order of 300 GPU-hours of wall-clock time (about one node-week). Inference and evaluation use a single GPU.

Hyperparameters and search. We did not perform a hyperparameter search. All values listed in Section 3.2 are taken from the NanoChat defaults for transformers of comparable depth; the only project-specific choices are the maximum sequence length (4096), the global batch size (1,966,080 tokens), and the number of iterations (250,000). The trace generator’s own knobs (line cap, plan-operation range, noise mixture) are listed in Section 3.1.

Training-data volume. Training data is procedurally generated on the fly by the two samplers of Section 3.1, so no fixed training file is stored. Over 250,000 optimization steps the model observes on the order of 5×10^{11} training tokens drawn from this distribution.

Validation and evaluation splits. The validation set used for monitoring during training is a disjoint seed range of the same generator and contains 1,024 programs; it is regenerated deterministically from a fixed RNG state and is never used for model selection (we report the final-iteration checkpoint). The held-out evaluation suite consists of 252 author-written MicroPy programs pooled across six tasks (Table 1, Table 2); none of these programs appears in the generator’s support up to a probability lower bound of 10^{-6} .

Statistics reporting. The headline accuracy numbers report token-level greedy-decoding accuracy on a single training seed; we do not aggregate across seeds because all reported numbers are 100% and have zero variance under deterministic decoding. Figure 2 reports per-task max trace lengths and is computed over the full 252-program suite.

Software. We use PyTorch with the NanoChat training stack of Karpathy (2025) (MIT-licensed

open source), which in turn uses the Muon optimizer of Jordan et al. (2024). The PENCIL scaffolding rule is implemented in our trace generator following Yang et al. (2025). Tokenization uses a custom fixed-vocabulary whitespace tokenizer (described in Section 3.2). All third-party packages are used consistent with their published license and intended use.

B Extended Related Work

Transformers, intermediate generation, and bounded context. Transformers augmented with intermediate generation can in principle carry out inherently serial computation, but the generated scratchpad itself becomes a memory bottleneck because it must remain accessible in context (Vaswani et al., 2017; Wei et al., 2022; Merrill and Sabharwal, 2023; Li et al., 2024; Liu et al., 2024). PENCIL is the closest scaffold to our setting: it reclaims completed reasoning steps so longer computations can proceed under a bounded context budget (Yang et al., 2025). MicroPy builds on this idea, using PENCIL to keep only the execution state needed for the next small-step rewrite rather than an ever-growing free-form scratchpad.

Length generalization by decomposition. A central difficulty in algorithmic reasoning is length generalization: models that fit short traces or short arithmetic instances often fail to extrapolate systematically to longer ones (Zhou et al., 2023, 2024). Structural recursion is another persistent failure mode without explicit stack discipline (Zhang et al., 2024). Recent work shows that progress is possible when the task is reformulated so that each predicted step is more local: inductive scratchpads reduce global dependencies (Abbe et al., 2024), while arithmetic-specific positional schemes such as position coupling and abacus-style embeddings improve extrapolation by aligning digit structure (Cho et al., 2024; McLeish et al., 2024). MicroPy shares this decomposition principle, but applies it at the level of a programming language: instead of designing a better representation for one task such as addition, we train a general interpreter whose primitive predictions are reusable retrieval and rewrite steps.

Turing Programs is especially close in spirit: it shows that transformers can achieve robust length generalization when an algorithm is decomposed into Turing-machine-style updates that mostly copy the previous state with small local changes (Hou et al., 2024). The common principle is that out-of-distribution computations become sequences of simple in-distribution steps. The difference is that Turing Programs trains one model per task, whereas MicroPy trains one model to read prompt-defined procedures and execute unseen programs compositionally under a fixed language semantics.

Program-like analyses of transformers. A complementary line writes, compiles, or recovers computations in transformer-oriented program form. RASP-style analyses study how transformer computations can implement symbolic algorithms, Tracr compiles such programs into transformer weights, and the Learning Transformer Programs work learns modified transformers that can be converted into discrete human-readable programs, including algorithmic tasks (Weiss et al., 2021; Lindner et al., 2023; Friedman et al., 2023). These works make useful comparisons because they expose program structure inside or as task-specific transformer solutions, but they differ from MicroPy in the direction of construction: they compile, analyze, or recover transformer programs, whereas MicroPy trains a standard decoder-only transformer from random MicroPy execution traces and asks whether it learns reusable small-step interpreter rules for prompt-defined procedures under a fixed operational semantics.

Neural Programmer-Interpreters and learned execution. Neural Programmer-Interpreters (NPI) introduced learned execution with a recurrent controller, a learned program memory, and domain-specific interfaces, and later work showed gains from recursion and search in this framework (Reed and de Freitas, 2016; Cai et al., 2017; Pierrot et al., 2019). Architectural precursors with explicit external memory (Graves et al., 2014, 2016) likewise targeted learnable universal computation, but routed control through differentiable memory rather than a discrete trace. Another line trains models more directly on execution traces or intermediate computations, from simple learned execution tasks to scratchpad supervision and Python trace prediction (Zaremba and Sutskever, 2014; Nye et al., 2021; Liu et al., 2023). Our setting differs in that the model reads explicit MicroPy definitions in context, represents bindings, mutable state, and stack frames directly in the prompt, and is supervised on small-step retrieval and rewrite transitions compressed with PENCIL. The goal is not unrestricted program understanding, but a controlled test of whether a standard transformer can learn reusable interpreter rules for a small language.

Universal prediction from latent programs. Grau-Moya et al. (2024) study a different use of universal computation: BrainPhoque universal-machine programs generate output sequences, and

sequence models are meta-trained to predict the next emitted symbol as a finite approximation to universal prediction. The source program and internal machine configuration remain latent to their predictor. In MicroPy, the procedure definitions, bindings, mutable state, and active execution frames are explicit model inputs, and the target is the next interpreter transition rather than the next symbol emitted by an unknown program. PENCIL then repeatedly invokes the model and deterministically reclaims completed context; it does not choose the MicroPy semantic transition.

Algorithmic reasoning and in-context-learned algorithms. Several lines of work train neural models to execute structured procedures step by step. Symbolic-mathematics datasets train sequence models to perform multi-step arithmetic or symbolic computation (Saxton et al., 2019; Lample and Charton, 2020). The Neural Algorithmic Reasoning programme trains a generalist model to imitate a suite of classical algorithms in the CLRS-30 benchmark (Veličković and Blundell, 2021; Ibarz et al., 2022). A complementary thread reverse-engineers what algorithms are being learned in context (Akyürek et al., 2023; Garg et al., 2022; von Oswald et al., 2023). Architectural work on adaptive control flow inside the transformer (Csordás et al., 2022) addresses similar systematic-generalization concerns. MicroPy differs from this line on two axes: (i) we train one transformer on random execution traces of a Turing-complete language and evaluate on held-out human-written *programs* rather than held-out instances of fixed algorithms, and (ii) we supervise on small-step retrieval/rewrite transitions rather than on input-output pairs. A complementary recent direction is to internalise procedural rules into the model weights from declarative instructions rather than execution examples (Cook et al., 2025); that approach trades context-trace verbosity for finetuning-time compute, while MicroPy keeps the entire interpreter behaviour in-context and in-weights jointly.

Compositional generalization. MicroPy is also a test of compositional generalization: can a model execute novel combinations of familiar primitives and control-flow patterns rather than merely extend in-distribution traces? This theme is classically studied with SCAN, CFQ, and COGS-style splits that hold out compositions while keeping atoms familiar (Lake and Baroni, 2018; Keysers et al., 2020; Kim and Linzen, 2020). Our setting differs in that

the composition rule is an operational semantics: generalization means recombining learned evaluation rules into correct executions on unseen programs, intersecting with broader limits of autoregressive transformers on multi-step compositional problems (Drozdo et al., 2023; Dziri et al., 2023).

Retrieval augmentation, memory, and context. External-memory approaches such as retrieval augmentation and long-term memory modules address bounded context by storing information outside the immediate prompt (Lewis et al., 2020; Wang et al., 2023). These approaches are complementary to MicroPy: our focus is not how to retrieve external knowledge, but which execution state must stay in context so that the next interpreter step is well-defined.

C Example Function Walkthrough

The training data generator is fairly complex. Rather than giving a full specification for all the language features we give an example of evaluation and leave it to the reader to imagine a generalization to the full language. Consider two definitions of the identity function.

```
@MicroPy
def foo(x): return x
```

```
@MicroPy
def bar(x): return x
```

We will generate training data by evaluating the expression `foo(bar(obj1))`. This generates a training context that includes the definitions of `foo` and `bar` as

```
FD foo = lambda(x) Exp3;
FD bar = lambda(x) Exp4;
```

The training sequence also includes the following expression definitions.

```
D.Exp1 = App(bar (0.obj1))
D.Exp2 = App(foo (Exp1))
D.Exp3 = LookupVar(x)
D.Exp4 = LookupVar(x)
```

The first training sequence starts with a call to an expression

```
[call] [call] Exp2
```

As will become clear, the two occurrences of `[call]` are used to combine effects. In this example there are no effects but the evaluation rules handle the general case.

This initial context is followed by token generation resulting in a PENCIL rewrite of `Exp2` to its definition.²

```
[call] [call] Exp2
=> [call] App(foo (Exp1)) [ret]
```

PENCIL rewriting then gives:

```
[call] [call] App(foo (Exp1))
```

Continuing with token generation we get.

```
[call] [call] App(foo (Exp1))
arg0= [call] [call] Exp1
```

Token generation then yields a rewriting of `Exp1` to its definition.

²The examples of token sequences are formatted here to improve reading. In the actual context there are no newlines or tabs.

```
[call] [call] App(foo (Exp1))
arg0= [call] [call] Exp1
=> [call] App(bar (0.obj1)) [ret]
```

Which yields:

```
[call] [call] App(foo (Exp1))
arg0= [call] [call] App(bar (0.obj1))
```

Continued token generation yields:

```
[call] [call] App(foo (Exp1))
arg0= [call] [call] App(bar (0.obj1))
arg0 = [call] 0.obj1
```

Only single `[call]` occurs when evaluating an object constant (no effect is involved). The innermost call is then replaced by an effect and a value.

```
[call] [call] App(foo (Exp1))
arg0= [call] [call] App(bar (0.obj1))
arg0 = Eff(empty) 0.obj1
```

The next step is to retrieve the definition of the procedure `bar` and replace the call to `bar` with a stack frame in which to evaluate the body expression. This is done in three steps: the procedure definition is retrieved; the effect of the argument evaluation is separated from the evaluation of the procedure body; and a stack frame is constructed for the evaluation of the body. Here we show the context after the construction of the stack frame.

```
[call] [call] App(foo (Exp1))
arg0= [call] Eff(empty)
[call] Env(Bind(x 0.obj1))
[call] Exp4
```

From here `Exp4` is rewritten to its definition.

```
[call] [call] App(foo (Exp1))
arg0= [call] Eff(empty)
[call] Env(Bind(x 0.obj1))
[call] LookupVar(x)
```

Then the `Lookup` is evaluated.

```
[call] [call] App(foo (Exp1))
arg0= [call] Eff(empty)
[call] Env(Bind(x 0.obj1))
Eff(empty) 0.obj1
```

Next, the innermost call (the stack frame) is replaced by its effect and value.

```
[call] [call] App(foo (Exp1))
arg0= [call] Eff(empty)
Eff(empty)
0.obj1
```

Next the two effects are combined.

```
[call] [call] App(foo (Exp1))  
  arg0= Eff(empty) 0.obj1
```

From here the computation proceeds similarly
to the computation of the application of bar.

D MicroPy Function Definitions

```

@MicroPy
def copy_bits(b1,b2):
    return Seq(Assert(b2,Attr("value"),LookupAttr(b1,Attr("va
    ↪ ue"))),
                If(HasAttr(b1,Attr("next")),
                    copy_bits(LookupAttr(b1,Attr("next")),
                               LookupAttr(b2,Attr("next"))),
                    unit_))

@MicroPy
def not_(p):
    return If(p,false_,true_)

@MicroPy
def flip_bits(b1,b2):
    return Seq(Assert(b2,Attr("value"),not_(LookupAttr(b1,Att
    ↪ r("value")))),
                If(HasAttr(b1,Attr("next")),
                    flip_bits(LookupAttr(b1,Attr("next")),
                               LookupAttr(b2,Attr("next"))),
                    unit_))

@MicroPy
def exor_(p,q):
    return If(p,If(q,false_,true_),If(q,true_,false_))

@MicroPy
def and_(p,q):
    return If(p,q,false_)

@MicroPy
def or_(p,q):
    return If(p,true_,q)

@MicroPy
def rpc_sum_(a1,a2,c):
    return exor_(a1,exor_(a2,c))

@MicroPy
def rpc_carry_(a1,a2,c):
    return or_(and_(a1,a2),
                and_(c,or_(a1,a2)))

@MicroPy
def RPC_add(b1,b2,b3,c):
    return RPC_aux(b1,b2,b3,c,
                    rpc_carry_(LookupAttr(b1,Attr("value")),
                                LookupAttr(b2,Attr("value")),
                                c))

@MicroPy
def RPC_aux(b1,b2,b3,c,nc):
    return Seq(Assert(b3,Attr("value"),
                        rpc_sum_(LookupAttr(b1,Attr("value")),
                                LookupAttr(b2,Attr("value")),
                                c)),
                If(HasAttr(b1,Attr("next")),
                    RPC_add(LookupAttr(b1,Attr("next")),
                             LookupAttr(b2,Attr("next")),
                             LookupAttr(b3,Attr("next")),
                             nc),
                    Assert(LookupAttr(b3,Attr("next")),
                           Attr("value"),
                           nc)))

@MicroPy
def RPC_mult(b1,b2,b3):
    return Seq(If(LookupAttr(b1,Attr("value")),
                    RPC_add(b2,b3,b3,false_),
                    unit_),
                If(HasAttr(b1,Attr("next")),
                    RPC_mult(LookupAttr(b1,Attr("next")),
                              b2,
                              LookupAttr(b3,Attr("next"))),
                    unit_))

@MicroPy
def sat_assign(var, cnf_head_, out_):
    return If(
        HasAttr(var, Attr("next")),
        Try(
            Seq(Assert(var, Attr("value"), true_),
                sat_assign(LookupAttr(var, Attr("next")),
                           cnf_head_,
                           out_)),
            Seq(Assert(var, Attr("value"), false_),
                sat_assign(LookupAttr(var, Attr("next")),
                           cnf_head_,

```

```

                                out_)),
        Try(
            Seq(
                Assert(var, Attr("value"), true_),
                If(eval_cnf(cnf_head_),
                    Assert(out_, Attr("value"), true_),
                    fail_),
            ),
            Seq(
                Assert(var, Attr("value"), false_),
                If(eval_cnf(cnf_head_),
                    Assert(out_, Attr("value"), true_),
                    fail_),
            ),
        ),
    ),
)

@MicroPy
def sat_solve(var_head_, cnf_head_, out_):
    return Try(sat_assign(var_head_, cnf_head_, out_),
                Assert(out_, Attr("value"), false_))

```

E Frontier-LLM Probe on MicroPy Traces

We gave Claude Opus 4.7 read access to the MicroPy source code and asked it to predict step-by-step completions on a balanced sample of 996 held-out trace steps (166 per task). The model was free to use tools, and in this setup chose to do so: it read the operational semantics from the source, wrote its own Python re-implementation of the MicroPy rewrite rules, and invoked that re-implementation as a tool to produce predictions. The probe therefore measures what a tool-using frontier LLM achieves when it tries to *simulate* MicroPy’s trace generator from the source documentation, rather than what an LLM does by next-token reasoning alone. Gold completions are withheld from the model’s input.

Task	n	Exact-match
RPC_add	166	0.777
RPC_mult	166	0.747
flip_bits	166	0.729
sat_solve	166	0.741
sat_verify	166	0.693
copy_bits	166	0.669
Overall	996	0.726

Table 3: Claude Opus 4.7 with source-code access and self-authored tool use on MicroPy step-prediction ($n = 996$, balanced across the six held-out tasks). The remaining error stems from places where the model’s hand-written Python simulator diverges from MicroPy’s actual semantics, particularly on accumulated in-trace effects.

We read this number two ways. First, MicroPy step-prediction is not a string-pattern task a pre-trained LLM incidentally solves: even with full language documentation and a self-written Python re-implementation invoked as a tool, a frontier LLM does not match the trained interpreter’s 100%. Second, this is *not* a task-performance comparison. A tool-using LLM agent could trivially answer the underlying bit-arithmetic and SAT questions by invoking Python or Z3, and (as observed here) it can even author the Python re-implementation used in this probe. The relevant comparison is therefore not "LLM with tools" against our trained interpreter but "LLM *without* external solvers, forced to perform extended arithmetic or SAT directly"; existing length-generalization work (Zhou et al., 2023, 2024; Seely et al., 2025; Hazra et al., 2024) reports that pretrained LLMs do not generalize in

this regime. Our contribution is specifically about whether a transformer can be trained to internalize a Turing-complete operational semantics from execution traces, which is orthogonal to whether a different system, given different affordances, could solve the underlying tasks.

F Distribution-Shape Failure Examples

As MicroPy designers, we anticipated two axes along which a held-out test program can exercise a structural shape that a sparser training distribution would miss: (i) long single-step writes inside an effect-accumulation step, and (ii) zero-argument call boundaries. The training generator used for the model reported in Section 3.3 explicitly samples both axes. To illustrate *what an OOD failure on these axes looks like*, we describe below two examples produced by an ablated training distribution that under-samples each axis. These examples are not the model that produces the headline numbers; they are case studies of the failure mode an under-covered distribution induces.

Failure 1: Long Pre-TailApp Eff write exceeding training write length

Setup. We constructed a held-out program whose execution requires a tail-call [ret] step that closes two adjacent Eff frames into a single combined ~52-entry Eff immediately preceding a TailApp, after extended linked-list traversal accumulates many Assertion(o, attr0, bool) writes. At the token level this is a long copy-with-merge: the model must reproduce dozens of Assertion(. . .) entries in order, drawn from the two upstream Eff frames in the prompt tail, then re-emit the TailApp payload unchanged.

The write length here exceeds the longest single-step write the model saw during training. The immediate prompt tail is:

```
general stack (prompt tail):
[call] Env ( )
[call] Eff ( Assertion(obj2 attr0 False)
             Assertion(obj51 attr0 False)
             ... 37 more Assertions ...
             Assertion(obj34 attr0 False) )
Eff ( Assertion(obj45 attr0 False)
     Assertion(obj44 attr0 True)
     ... 11 more Assertions ...
     Assertion(obj33 attr0 True) )
TailApp(fun7 Args(obj6 obj1 obj44)) [EOP]
```

Gold vs. prediction. The gold completion merges the two upstream Eff frames into one combined 52-entry Eff and forwards the TailApp payload to the parent step:

```
Gold: => [call] Eff (
             Assertion(obj2 attr0 False)
             ... position 108: obj47 attr0 True ...
             ... 51 more Assertions ...
             Assertion(obj33 attr0 True) )
TailApp(fun7 Args(obj6 obj1 obj44)) [ret]
```

```
Model: => [call] Eff (
```

```
Assertion(obj2 attr0 False)
... position 108: obj182 attr0 True ...
... 51 more Assertions ...
Assertion(obj33 attr0 True) )
TailApp(fun7 Args(obj6 obj1 obj44)) [ret]
```

The first 107 tokens match exactly; at token position 108 (inside the 18th Assertion slot of the combined Eff) the gold token is obj47 but the model emits obj182. The remaining ~200 tokens of the long completion are regenerated correctly. The error is silent: structurally the output is a well-formed Eff(. . .) TailApp(. . .) [ret] step, but one object reference inside the accumulator has been swapped to a different valid-looking object name. obj47 appears explicitly in the immediate prompt tail (the gold token is a direct copy), yet obj47 is not in the top-10 predictions at this position (gold_in_topk = False).

Interpretation. A 52-entry combined Eff write is ~300 tokens of structural copy, supervised only by the end-to-end completion loss. When this length exceeds the longest combined-Eff write present in training data, per-slot error rate compounds across the length: the model gets the structural skeleton, attribute names, and bool literals right, but the obj-name slot, which has tens of equally-typed candidates in the immediate context, becomes weakly identified.

Curriculum coverage in the reported model.

Because long pre-TailApp Eff-merge writes are a foreseeable failure axis given MicroPy’s stack discipline, the training generator used for the model in Section 3.3 explicitly samples such long writes with distractor object names sharing the same attribute. On the same example the reported model produces the gold token obj47 at position 108.

Failure 2: Zero-argument function calls (dummy-arg coverage gap)

Setup. Several natural held-out programs include helper procedures that, semantically, take no arguments. For example, a top-level entry such as

```
@MicroPy
def helper():
    ...
```

compiles into MicroPy traces whose call sites are App(funX ()) and TailApp(funX Args()): function tokens with an immediately closing empty argument list.

The training distribution did not exercise the zero-argument shape: every procedure in the sampled programs had at least one variable bound at

call time. The empty-argument-list token sequence (Args () adjacent in the trace) was therefore essentially out-of-distribution at the structural level.

Observed failure. On a held-out program that uses such a zero-argument helper, the original definition produced 7 token-level misses across 3544 trace rows at our reference checkpoint, all clustered around the App(funX ()) / TailApp(funX Args()) call boundaries. The whole-program success rate was 0/1.

Curriculum coverage in the reported model.

The training generator used for the model in Section 3.3 treats the zero-argument call boundary as a first-class shape and samples App(funX ()) / TailApp(funX Args()) explicitly. On the same program, the reported model produces no misses at the zero-argument call boundaries.

Combined takeaway. Both examples are consistent with the same pattern. When the trained model is presented with a structural shape its training distribution did not cover, the model degrades gracefully *into the relevant prior* rather than failing structurally. The reported model in Section 3.3 covers both axes by design, but we include these examples to make the under-coverage failure mode concrete.

G Per-Task Evaluation Operation Profiles

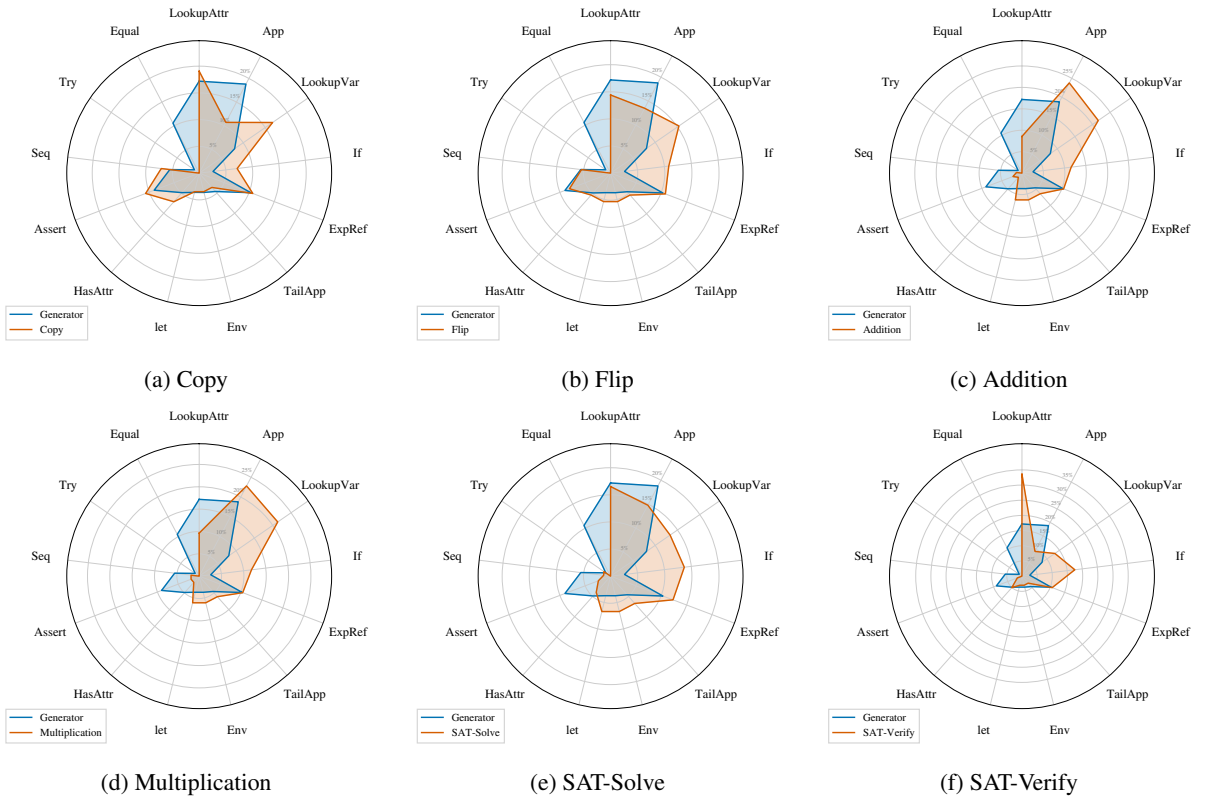


Figure 5: Per-task operation profiles for the held-out evaluation suite. Each subplot compares the training generator’s last-op distribution (blue) against a single evaluation task (orange). Proportions are within-corpus. Copy and Flip are dominated by App/LookupAttr/ExpRef patterns from linked-list traversal; Addition and Multiplication rely more heavily on If/LookupVar; and the SAT tasks show the expected Try-heavy profile from backtracking search.