000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053

CFLBENCH: BENCHMARKING NOVEL CONTROL-FLOW LANGUAGE LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

CFLBench evaluates whether tool-augmented language models can acquire the execution semantics of *previously unseen programming languages* from a handful of examples and black-box execution feedback. We construct 19 small deterministic languages that share a compact, assembly-like surface vocabulary but vary the underlying control-flow mechanism. Each language provides four tasks: Tasks 1–2 emphasize direct generalization from examples and are defined as **passive**, while Tasks 3–4 are designed to benefit from active experimentation via probe programs and iterative hypothesis refinement and are defined as **active**. Across a range of state-of-the-art models, we observe a consistent gap between passive and active tasks. The strongest model we evaluate, GPT 5.2, achieves 94.7% on passive tasks but drops to 60.5% on active tasks. Comparatively, other models such as Claude Opus 4.5 (94.7%→42.1%), Gemini-3-Pro (84.2%→31.6%) degrade more sharply. This pattern suggests that learning new execution semantics through interaction is a distinct bottleneck, beyond simply learning semantic patterns from examples. We release code: `https://anonymous.4open.science/r/CFLBench-C01E/`. We release the tasks and interpreters to support fine-grained behavioral analysis of inductive strategies and failure modes.

1 INTRODUCTION

Large language models (LLMs) have shown remarkable performance across many static benchmarks. However, an important question is how well they support *fluid intelligence*, acquiring new rules from sparse evidence and applying them reliably in out-of-distribution settings (Chollet, 2019). A consequential component of fluid intelligence is *interactive induction*: the ability to perform experiments, interpret observations, and update hypotheses before committing to the next action.

In many real-world situations, the so-called "experiment" is not a physical measurement but a black-box system that can be queried: a simulator, a tool API, a legacy workflow engine, or a language-like interface without documentation. Recent benchmarks have begun to evaluate this kind of interaction in scientific domains, where agents probe environments to uncover hidden scientific laws (Chen et al., 2025; Shojaee et al., 2025; Zheng et al., 2025). However, most existing interactive benchmarks still entangle this loop with substantial domain priors such as physics, whereas we want a controlled setting where such prior knowledge does *not* augment the interaction. The agent must learn the rules *only* by probing a black-box system and updating hypotheses from observed executions.

Most inductive-reasoning benchmarks emphasize passive pattern completion (e.g., inferring a mapping from I/O examples), where the agent cannot actively choose what evidence to collect. In the domain of code synthesis, programming-by-example and inductive synthesis frameworks have historically focused on learning a program in a *known* DSL from demonstrations (Gulwani; Polozov & Gulwani, 2015; Devlin et al., 2017). In contrast, CFLBench makes *execution semantics itself* the inductive target: the agent must infer how an unfamiliar instruction system executes via black-box probing, rather than fitting a mapping under fixed semantics.

More recent LLM-era evaluations similarly test whether models can generalize from examples, often with a fixed oracle and held out tests (Li & Ellis, 2024; Wei et al., 2025b). Interactive coding benchmarks provide execution feedback, but typically in familiar languages and toolchains, where the challenge is debugging within known semantics rather than discovering the rules of the underlying system (Yang et al., 2023; Jimenez et al., 2024; Yang et al., 2024). Prior research in esoteric language

054
055
056
057
058
059
060
061
062
063
064
065
066
067
068
069
070
071
072
073
074
075
076
077
078
079
080
081
082
083
084
085
086
087
088
089
090
091
092
093
094
095
096
097
098
099
100
101
102
103
104
105
106
107

ID	Key idea	ID	Key idea
L1	Three instruction lanes (A/B/C) with separate pointers; `UDA` switches execution between lanes in fixed cycle (A→B→C).	L11	Two-pass execution; second pass reorders marked lines (`^`/`v`) by swapping adjacent `v`-blocks followed by `^`-blocks.
L2	Labels execute in ascending order; `XIH a b` declares `a` depends on `b`'s execution.	L12	`UWR` records execution ribbons from next line; `NHO` replays recorded ribbon up to next `UWR` in source order.
L3	Labeled instructions are active/dormant; scheduler picks next greater active label; `HEX` executes and switches state.	L13	Labeled lines with execution quotas; cycles through labels in order until all quotas exhausted; `SIZ` adds quota.
L4	Labeled lines sorted by index; `RCS` sets stride register controlling index increment (no wraparound on out-of-bounds).	L14	Labeled lines executing once in order; `AYZ` pushes return labels; `STOP` returns to most recent pinned label (LIFO).
L5	Labeled lines with optional chorus markers (`*`); `NBA` k replays chorus batch k in ascending label order.	L15	Consumptive labeled lines; `FWN` jumps to label (pushing return); `KIE` returns to next unexecuted after call-site.
L6	Labeled lines with 1-step cooldown after execution; `CGH` toggles between executing min/max active labels.	L16	Labeled lines with global direction (forward/backward); `!` toggles direction; traverses remaining labels with wrapping.
L7	Labeled lines with optional echo delays (`~` k); echoes execute immediately before the k-th subsequent line in label order.	L17	Bracket "tournament" of adjacent labels: bye advances without execution; `AMO` toggles winner condition for next match.
L8	Labeled lines with `QTA`-set cursor; next label minimizes $\lvert\texttt{label} - (\texttt{current} + \texttt{cursor})\rvert$, ties to smaller.	L18	Lines execute only if relative guards hold over prior executed labels; star-offset guards.
L9	Labeled lines with penalty scores; `RGW` increments penalties; scheduler picks minimum penalty (ties to smallest label).	L19	Labels execute in ascending order, some having timers decrementing each step; reaching 0 causes immediate execution; `XYZ`/`ABC` toggle timers.
L10	Two rails (L/R) with alternating execution; sticky markers (`~`) prevent switching; `START` markers set initial rail.		

Table 1: CFLBench language inventory at a high level. Full descriptions are in Appendix Section C (Table Table 3); per-language mechanism examples are in Appendix **??**.

settings often provides documentation and emphasizes reading and applying a specification, rather than black-box discovery of the language itself (Amjith et al.).

We introduce **CFLBench**, an *interactive inductive reasoning benchmark* designed to test an agent's ability to (i) write informative programs, (ii) interpret black-box execution feedback, and (iii) use that evidence to output a solution adhering to given constraints. CFLBench uses short instruction sequences as the *medium* for experimentation and is not intended to test software-engineering or algorithmic programming skill, but to provide a compact action space where the agent can control the experiments it runs.

Benchmark design. CFLBench comprises 19 small deterministic instruction languages with a compact vocabulary (e.g., `SET`, `OUT`) but *unknown* execution semantics through novel control-flow. Each language includes four tasks: Tasks 1–2 are *passive* and are often solvable by generalizing directly from provided examples, while Tasks 3–4 are *active* and are designed to reward deliberate probing and hypothesis refinement. For each task, the agent receives a small set of example programs plus a `README` specifying a target output and structural constraints; the agent may then write and execute additional probe programs against a black-box interpreter under a strict run budget. A submission is counted as correct **if and only if** the agent produces the exact target output and satisfies all given constraints.

Research question. Our central question is: *To what extent can LLM agents carry out the probe–observe–update–act loop to acquire unseen execution semantics under a limited interaction budget, and then produce a final solution that satisfies constraints?*

Contributions and headline results. Overall, the contributions of this work are as follows:

- We introduce **CFLBench**, an interactive inductive reasoning benchmark where the inductive goal is to infer *execution semantics* from examples and black-box executions, rather than learning a mapping under fixed semantics.
- We propose a benchmark design that treats short instruction sequences as an *experimental interface* and separates *passive* generalization tasks (T1–T2) from *active* induction tasks (T3–T4) where performance benefit from informative probe selection under a strict run budget.
- Extensive experiments on state-of-the-art models and agent loops reveal a strong gap between passive and active performance: GPT 5.2 reaches 94.7% on passive tasks but drops to 60.5% on active tasks, while other models degrade more sharply (e.g., Claude Opus 4.5: 94.7%→42.1%; Gemini-3-Pro: 84.2%→31.6%).

2 Related Works

CFLBench targets *interactive semantic induction*: an agent must infer the execution semantics of a novel language from a small set of example programs, using black-box execution feedback, and then synthesize a constrained program that achieves a specified output. This setting relates to work on program synthesis from examples, interactive coding benchmarks with execution feedback, evaluation in unfamiliar languages, and interactive experimentation benchmarks.

Inductive Program Synthesis

Programming-by-example and inductive program synthesis study how to synthesize programs that are consistent with demonstrations through input–output examples. Recent work asks whether LLMs can solve programming-by-example at scale, characterizing when models succeed or fail to generalize beyond the observed examples (Li & Ellis, 2024). CodeARC benchmarks inductive iteration and emphasizes reasoning-driven generalization from small example sets via an oracle (Wei et al., 2025b). In contrast, CFLBench provides examples in a novel language whose control flow must itself be inferred; agents can design and execute probe programs to disambiguate semantics, and must satisfy explicit structural constraints in addition to matching the target output.

Our setting also connects to the broader program-synthesis literature on learning programs from examples and partial specifications, including systems for string transformations and inductive synthesis from I/O pairs (Gulwani; Polozov & Gulwani, 2015; Devlin et al., 2017; Osera & Zdancewic), learning-guided search for synthesis (Balog et al., 2017), and sketch-based synthesis (Solar-Lezama et al.). Neural program induction and program execution tasks likewise study how models can acquire algorithmic execution behavior from supervision (Reed & de Freitas, 2016; Zaremba & Sutskever, 2015).

Additionally, we draw inspiration from work on *oracle-guided* and *interactive* inductive synthesis, where an algorithm selects informative queries to disambiguate candidate programs or invariants (Garg et al., 2014; Ji et al., 2023). Recent work also explores program synthesis and code generation with LLMs (Austin et al., 2021; Li et al., 2023) and evaluates semantic reasoning using equivalence-based checks (Wei et al., 2025a). CFLBench complements these directions by emphasizing black-box acquisition of *execution semantics* in novel control-flow languages.

Interactive Coding and Software Engineering Agents

A complementary line of work evaluates models in interactive coding settings where execution feedback is available. InterCode standardizes and benchmarks interactive coding with execution feedback (Yang et al., 2023). SWE-bench evaluates models on real-world GitHub issues that require editing repositories and validating fixes via tests (Jimenez et al., 2024), and SWE-agent shows that improved agent–computer interfaces can substantially increase performance on such tasks (Yang et al., 2024). These benchmarks largely assume familiar programming languages and toolchains while CFLBench isolates a different capability: inferring the semantics of unseen languages and control-flow mechanisms from examples and feedback from black-box executions.

162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215

In addition to agentic evaluations, a large body of work develops benchmarks and datasets for code generation and code understanding across tasks and languages, including multilingual or cross-language test suites (Cassano et al., 2022; Zai, 2025; Diera et al., 2023) and studies of correctness and failure modes of LLM-generated code (Liu et al., 2023). These resources are complementary: they stress breadth across natural programming languages, whereas CFLBench stresses semantic acquisition under deliberate extraordinary execution rules.

Reasoning in Esoteric and Novel Languages

Prior work has examined LLM generalization to esoteric programming languages, typically in settings where documentation and examples are available (Amjith et al.). CFLBench complements this direction by withholding documentation and focusing on semantic induction through interaction, with tasks designed to discourage trivial hardcoding via constraint checks.

Low-Resource and Domain-Specific Programming Languages

Our motivation also intersects with work on low-resource and domain-specific programming languages, where models must generalize beyond high-frequency pretraining distributions. Recent benchmarks and studies consider code generation and transfer to low-resource or domain-specific languages (Cassano et al., 2024; Mora et al., 2024; Zhao & Fard, 2025; Joel et al., 2025). While low-resource language coverage is an important axis, CFLBench focuses on a different source of distribution shift: *novel execution semantics* presented only through examples and black-box execution.

Interactive Experimentation Benchmarks

Interactive scientific discovery benchmarks study whether LLM agents can propose hypotheses and design experiments to identify underlying rules of an environment (Chen et al., 2025; Shojaee et al., 2025; Zheng et al., 2025). While these benchmarks focus on physical laws or equations, CFLBench instantiates an analogous experimental loop in the domain of programming languages: agents run probe programs against a black-box interpreter to uncover latent control-flow semantics, then synthesize a constrained program consistent with the inferred execution model.

Generalization and Abstract Reasoning Benchmarks

Finally, CFLBench is broadly aligned with benchmark efforts that emphasize systematic generalization and abstraction (outside of programming), such as ARC and ConceptARC (Chollet, 2019; Moskvichev et al., 2023). Our contribution is to instantiate a similarly generalization-focused setup for code: the agent must infer latent rules from limited evidence and then act under constraints to satisfy a target behavior.

3 Method

Task Creation

We generate a suite of deterministic languages via human ideation and create four tasks and four to five example programs per language. Eleven languages have four examples while eight have five. At least one example for each language shows the explicit syntactic cue for that language. Each language follows a schema where each line has a labeled prefix. We do this for uniformity across languages and constraints, although not all languages require the use of labels. Across languages, programs use a compact vocabulary over integers and variables with no strings. Concretely, the core instruction vocabulary is:

- `SET v n`: assign the integer `n` to variable `v`.
- `SET v v + n`: addition (and similarly `SET v v - n` for subtraction).
- `OUT v`: print the value of `v`.
- `STOP`: stop the program.

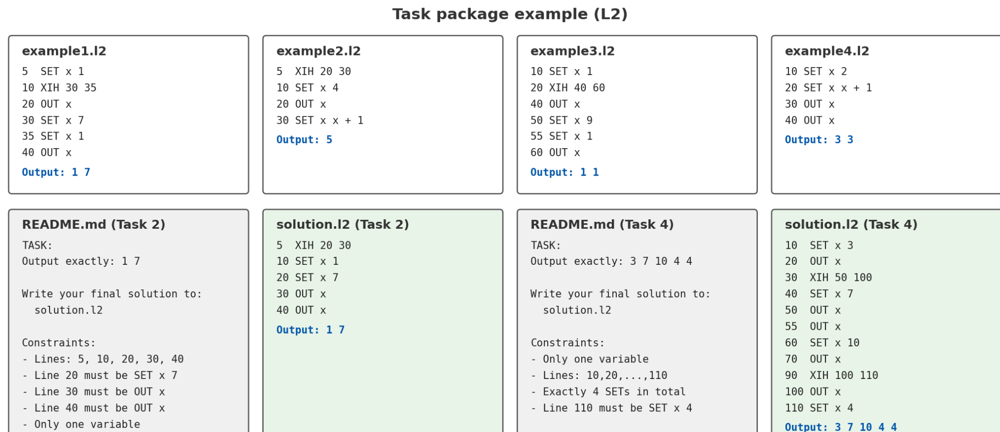


Figure 1: An example L2 task package: a README (target output + constraints), four example programs, and a proposed submission. The agent may run additional probes, but must ultimately output a single `solution.<ext>` that satisfies constraints and produces the target output.

Each language alters control flow via an explicit syntactic cue (illustrated in Figures 1 and 2) that is intentionally non-semantic: the letters or symbols themselves are chosen to avoid suggesting the underlying rule. This helps isolate whether LLMs can discover execution semantics from black-box observations, rather than relying on the meaning of the cue.

We choose explicit cues rather than purely implicit control-flow patterns so that agents are bottlenecked by experiment design and interpretation, rather than by noticing a subtle formatting pattern in the examples. Table 3 summarizes all 19 language instances and their ground-truth execution rules. This design keeps the majority of the syntax familiar and ensures that the main challenge arises from inferring semantics rather than syntax.

Each language provides four tasks that share the same examples. For each task, the model is given (i) the example files and (ii) a `README.md` which contains a target output and task-specific constraints.

Tasks 1–2 emphasize direct generalization from the examples. Tasks 3–4 are structured to benefit from interactive probing and often involve behavior that is weakly constrained by the provided examples. Without interaction, GPT 5.2 achieves 71.05% (Pass@2) accuracy on passive tasks, compared to 94.7% with interaction. On active tasks, GPT 5.2 achieves 60.5% with interaction, compared to 28.95% (Pass@2) with zero interaction. Overall, interaction substantially improves performance on both passive and active tasks but moreso on active tasks.

Constraints are used so models cannot trivially solve the program by just using `SET` and `OUT` to print the expected output. For example, languages that have re-executing control flow have outputs that require printing m numbers while constraints only allow the usage of n `OUT`s where $m > n$. Constraints are used so the solution requires the usage of the language specific control flow mechanism. GPT 5.2 achieves a 100% accuracy on passive tasks and 97.37% accuracy on active tasks without the use of constraints, illustrating that constraints necessitate the usage of unique control flow mechanisms to produce correct solutions.

We keep the instruction set intentionally minimal so that constraints are *hard to circumvent*: without alternate arithmetic expressions (e.g., separate `ADD`/`SUB`), models cannot satisfy line-level requirements via semantically equivalent rewrites, and thus must follow the intended sequence of state updates under the language's control-flow (Figures 1 and 2).

3.1 Experimental Setup

We evaluate models with access to a black-box command-line interpreter for the task language in a sandboxed workspace. Models may read the provided example files, create additional files, and

Model	Pass. Acc.	Pass. Runs	Pass. Turns	Act. Acc.	Act. Runs	Act. Turns	Overall
gpt-5.2	94.7%	7.42	17.71	60.5%	13.68	38.47	77.6%
claude-opus-4.5	94.7%	6.55	16.18	42.1%	15.24	40.18	68.4%
gemini-3-pro	84.2%	11.50	29.21	31.6%	12.74	33.76	57.9%
gemini-3-flash	73.7%	11.74	33.11	26.3%	11.50	35.82	50.0
o4-mini	71.1%	8.29	31.76	7.9%	11.50	51.39	39.5
deepseek-r1-0528	47.4%	8.21	34.97	2.6%	10.45	55.11	25.0
gpt-4o	23.7%	3.97	18.79	0.0%	11.37	35.95	11.8

Table 2: Main results on CFLBench with an interpreter budget of 20 runs. All reported accuracies are Pass@1. Accuracies are computed over 38 passive tasks (T1–T2), 38 active tasks (T3–T4), and 76 total tasks; runs/turns are means over the corresponding task subsets.

inspect their workspace via standard shell commands (e.g., `ls`, `cat`). The interpreter implementation is not available in the workspace.

To execute a program, the agent must issue a dedicated interpreter call as a standalone message of the form `lang-ran <file-name>`. The harness executes the file, returns the program output, and increments the interpreter run counter.

Models are given an interpreter run limit which is incremented each time `lang-ran` is called. We set the interpreter run limit to 20 and the step limit to 60. They are explicitly told the interpreter limit and can observe the interpreter runs used, but are not given the step limit. Once the model reaches the max number of interpreter runs and attempts to run the interpreter more, it receives the message: `ERROR: INTERPRETER RUN LIMIT REACHED. USE echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT TO SUBMIT YOUR FINAL SOLUTION.` Once a model reaches the maximum number of steps, the solution is automatically submitted for evaluation. Each task is evaluated independently in a fresh workspace; models do not carry state across tasks beyond what is provided in the prompt.

Models. We report results for seven models on all 19 languages and all four tasks per language (76 tasks total).

Metrics. We evaluate tasks as pass/fail, Pass@1. A task is solved if and only if running `solution.<ext>` produces the exact target output and satisfies all task constraints. We report (i) overall accuracy, and (ii) passive (T1–T2) vs. active (T3–T4) accuracy.

LLM-Powered Agent

We evaluate each task with a text-only agent implemented by adapting `mini-swe-agent` (SWE-agent Team, 2026; Yang et al., 2024) which employs a standard ReAct (Yao et al., 2023) setup. The agent decides when to inspect files, when to execute probe programs, and when to submit a final `solution.<ext>`; the interaction interface and budget rules are described in Section 3.1.

4 Results

Overall Performance

We report results in Table 2. GPT 5.2 has the highest accuracy on active tasks, achieving a 60.5% solve rate, while Claude Opus 4.5 achieves 42.1% and Gemini-3-Pro achieves 31.6%. Gemini-3-Pro and Gemini-3-Flash reach comparable accuracies on active tasks, while Gemini-3-Pro achieves 10.5% higher accuracy on passive tasks. Overall, the results suggest an apparent separation between stronger reasoning models (e.g., GPT 5.2, Claude Opus 4.5, Gemini-3-Pro) and weaker reasoning baselines (e.g., Gemini-3-Flash, o4-mini, deepseek-r1-0528) and non-reasoning models (e.g., GPT-4o) on active tasks: some of the strongest reasoning models achieve double-digit active accuracy, whereas o4-mini and deepseek-r1 remain below 10%, and gpt-4o struggles on both passive and active tasks.

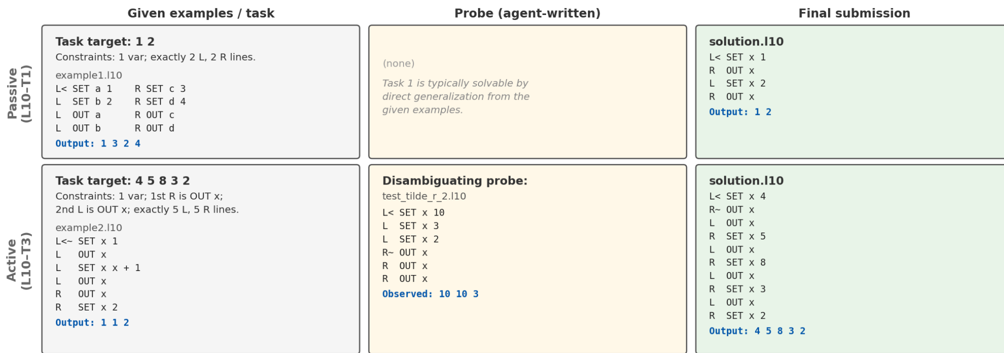
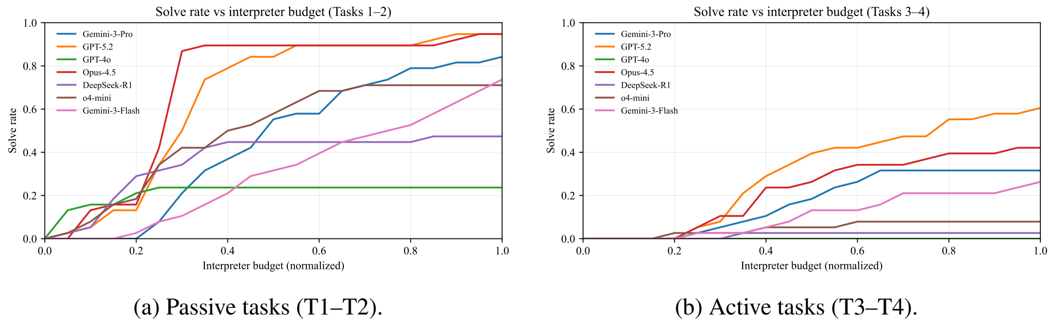


Figure 2: Examples of passive vs. active tasks in a given language. In L10, programs live on two rails (L/R). Passive tasks can often be solved directly from the provided examples. Active tasks benefit from targeted probes that reveal stateful modifiers (here, `R~` desynchronizes the rails by repeating the same L-instruction on the next step).



(a) Passive tasks (T1–T2).

(b) Active tasks (T3–T4).

Figure 3: Solve rate vs. interpreter budget, split by passive (T1–T2) and active (T3–T4) tasks.

Analysis

Experimental efficiency

On passive tasks, performance tends to plateau and most gains occur in the first half of the interpreter budget. The active-task curves show a similar early improvement pattern, but with diminishing returns as interpreter usage increases, suggesting that additional runs alone often fail to resolve remaining semantic uncertainty.

Turn Length and Interpreter Usage

Most models show an increase in mean interpreter runs and turn length from passive to active tasks, consistent with active tasks requiring more probing and iteration. Gemini-3-Pro and Gemini-3-Flash are notable exceptions: their average interpreter usage and turn length stay roughly similar across passive vs. active tasks, which is consistent with their higher rate of early submissions that do not yet solve the task (Figure 4). Across models, Tasks 3–4 elicit substantially more interaction than Tasks 1–2 (Table 2). For example, Claude Opus 4.5 averages 6.55 interpreter runs / 16.18 turns on Tasks 1–2 versus 15.24 runs / 40.18 turns on Tasks 3–4; GPT 5.2 similarly increases from 7.42 runs / 17.71 turns to 13.68 runs / 38.47 turns. Gemini-3-Pro and Gemini-3-Flash use fewer interpreter runs on Tasks 3–4 (12.74 and 11.50, respectively—about 7% and 25% fewer than GPT 5.2 and Claude Opus 4.5) yet achieve lower Task 3–4 accuracy (31.6% and 26.3%), suggesting that closing the gap likely requires not only more runs but also more informative probes and better hypothesis updates.

378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397

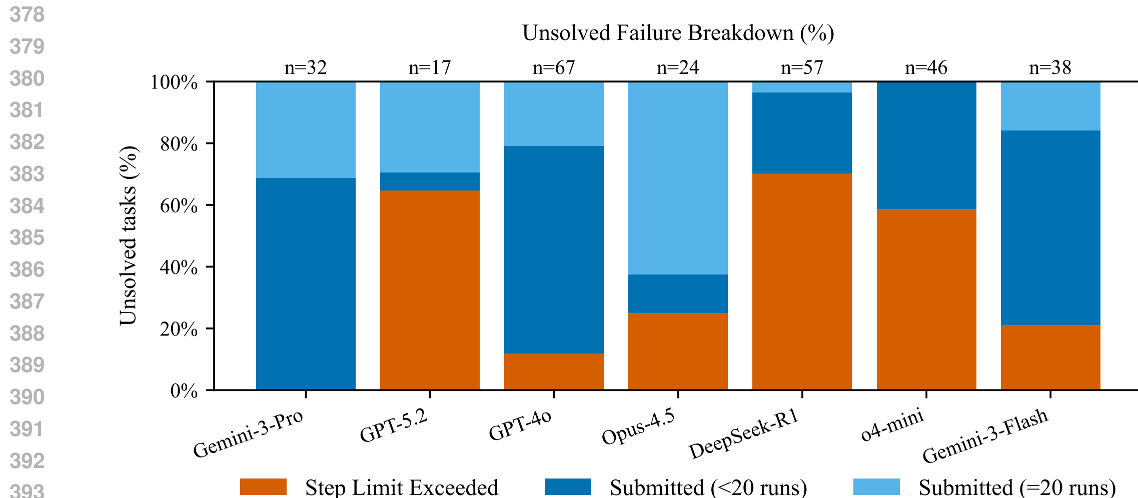


Figure 4: Breakdown of unsolved tasks by failure category.

398 PASSIVE ACTIVE GAP

399 400 401 402 403 404 405 406 407 408 409 410

The headline pattern—high accuracy on Tasks 1–2 but much lower accuracy on Tasks 3–4—should not be read as "models cannot use tools." Rather, it suggests that the *information-acquisition problem* may be a key difficulty: Tasks 3–4 require selecting probes that can perceive and set apart competing semantic hypotheses, then translating the favored hypothesis into a constrained program. This is qualitatively different from Tasks 1–2, where the expected outputs are often achievable with understanding the structure of the provided examples alone and the main challenge is maintaining the integrity of such structure for differing outputs. In our reasoning traces, unsolved tasks are typically longer and involve more iteration, suggesting that failures are not solely due to early termination. One possible explanation (which we treat as a hypothesis rather than a conclusion) is that the agent sometimes explores *within* an initially chosen conceptual frame of limited variety (e.g., standard single-instruction-step control flow) and fails to sufficiently challenge that frame after unsuccessful probes.

411 412

5 DISCUSSION AND LIMITATIONS

413 414 415 416 417 418 419 420

Benchmark scope and representativeness. Our languages are intentionally small and deterministic, which helps isolate semantic induction but may under represent other real world difficulties (e.g., rich type systems, large libraries, undocumented APIs, or stochastic environments). Conversely, many real DSLs are also small and ad-hoc, and the black-box-interpreter setting is common when semantics are embedded in closed systems (Mora et al., 2024; Joel et al., 2025). We view CFLBench as complementary to large-scale software engineering benchmarks (Jimenez et al., 2024): it targets a narrower but important capability axis.

421 422 423 424 425 426

How CFLBench can be used. We expect CFLBench to be most useful for (i) comparing agent designs for interactive semantic induction (probe selection, hypothesis management, verification strategies), (ii) studying generalization across execution mechanisms (e.g., whether success transfers from multi-rail to scheduler semantics), and (iii) diagnosing failure modes via trajectories (e.g., where agents get stuck, which probes are chosen, and how hypotheses evolve).

427 428 429 430 431

Future directions. The benchmark readily supports extensions: adding more language families; increasing the confusability of semantics; introducing partial documentation to study hybrid learning (docs + experimentation); and adding controlled chaos (e.g., noisy outputs or randomized initial states) to test robustness. Another promising direction is to evaluate agent improvements aimed specifically at *information gain*: selecting probes that maximize the expected discrimination between semantic hypotheses under a limited interaction budget.

432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485

6 CONCLUSION

We presented CFLBench, a benchmark for black-box acquisition of novel execution semantics in small programming languages. By splitting tasks into passive generalization (T1–T2) and active, experiment-driven induction (T3–T4), the benchmark helps characterize a capability gap that is not visible from passive coding benchmarks alone. Our initial results show strong passive performance but substantially weaker active performance, motivating future work on agent designs that better support probe selection, hypothesis testing, and constraint-aware synthesis.

7 BROADER IMPACT

CFLBench is an evaluation benchmark intended to make progress on interactive semantic induction more measurable and reproducible. A realistic positive impact is improved diagnosis of when tool-using agents fail: the benchmark can help researchers separate failures of (i) semantic discovery (choosing informative probes and updating hypotheses) from (ii) constraint satisfaction and implementation. This may lead to more reliable agent designs and evaluation practices for workflows that involve querying black-box systems.

Potential negative impacts are primarily about misuse in evaluation rather than direct real-world harm. First, as with many benchmarks, CFLBench may incentivize overfitting: systems can be tuned to this specific suite of synthetic languages (or to the harness) without improving robustness on real interpreters, which could distort progress signals. Second, the benchmark may increase compute usage via repeated agent rollouts; we encourage reporting run budgets and limiting unnecessary probing when comparing methods.

REFERENCES

Zai-org/humaneval-x · Datasets at Hugging Face. https://huggingface.co/datasets/zai-org/humaneval-x, December 2025.

Saraswathy Amjith, Michael Wang, Arul Kolla, Jayson Lynch, and Neil Thompson. In-Context Learning for Esoteric Programming Languages: Evaluating and Enhancing LLM Reasoning Without Fine-Tuning.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models, August 2021.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs, March 2017.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation, December 2022.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q. Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs, September 2024.

Yimeng Chen, Piotr Piękos, Mateusz Ostaszewski, Firas Laakom, and Jürgen Schmidhuber. PhysGym: Benchmarking LLMs in Interactive Physics Discovery with Controlled Priors, October 2025.

François Chollet. On the Measure of Intelligence, November 2019.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O, March 2017.

Andor Diera, Abdelhalim Dahou, Lukas Galke, Fabian Karl, Florian Sihler, and Ansgar Scherp. GenCodeSearchNet: A Benchmark Test Suite for Evaluating Generalization in Programming Language Understanding, November 2023.

486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A Robust Framework for Learning Invariants. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, Armin Biere, and Roderick Bloem (eds.), *Computer Aided Verification*, volume 8559, pp. 69–87. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08866-2 978-3-319-08867-9. doi: 10.1007/978-3-319-08867-9_5.

Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples.

Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection. *Artifact for OOPSLA'23: Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection*, 7(OOPSLA1):103:819–103:847, April 2023. doi: 10.1145/3586055.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?, November 2024.

Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages, September 2025.

Jia Li, Ge Li, Chongyang Tao, Jia Li, Huangzhao Zhang, Fang Liu, and Zhi Jin. Large Language Model-Aware In-Context Learning for Code Generation, October 2023.

Wen-Ding Li and Kevin Ellis. Is Programming by Example solved by LLMs?, November 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation, October 2023.

Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E. Gonzalez, Elizabeth Polgreen, and Sanjit A. Seshia. Synthetic Programming Elicitation for Text-to-Code in Very Low-Resource Programming and Formal Languages, October 2024.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The ConceptARC Benchmark: Evaluating Understanding and Generalization in the ARC Domain, May 2023.

Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis.

Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. *SIGPLAN Not.*, 50(10):107–126, October 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814310.

Scott Reed and Nando de Freitas. Neural Programmer-Interpreters, February 2016.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D. Doan, and Chandan K. Reddy. LLM-SRBench: A New Benchmark for Scientific Equation Discovery with Large Language Models, June 2025.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures.

SWE-agent Team. mini-swe-agent: A Minimal, Configurable Agent Loop for Interactive Tool Use. `https://github.com/swe-agent/mini-swe-agent`, 2026. Accessed: 2026-01-27.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago S. F. X. Teixeira, Diyi Yang, Ke Wang, and Alex Aiken. EquiBench: Benchmarking Large Language Models' Reasoning about Program Semantics via Equivalence Checking, September 2025a.

Anjiang Wei, Tarun Suresh, Jiannan Cao, Naveen Kannan, Yuheng Wu, Kai Yan, Thiago S. F. X. Teixeira, Ke Wang, and Alex Aiken. CodeARC: Benchmarking Reasoning Capabilities of LLM Agents for Inductive Program Synthesis, August 2025b.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback, October 2023.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, November 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models, March 2023.

Wojciech Zaremba and Ilya Sutskever. Learning to Execute, February 2015.

ZiXiao Zhao and Fatemeh H. Fard. Do Current Language Models Support Code Intelligence for R Programming Language?, May 2025.

Tianshi Zheng, Kelvin Kiu-Wai Tam, Newt Hue-Nam K. Nguyen, Baixuan Xu, Zhaowei Wang, Jiayang Cheng, Hong Ting Tsang, Weiqi Wang, Jiaxin Bai, Tianqing Fang, Yangqiu Song, Ginny Y. Wong, and Simon See. NewtonBench: Benchmarking Generalizable Scientific Law Discovery in LLM Agents, December 2025.

A APPENDIX OVERVIEW

This appendix contains the minimal supplemental material referenced from the main paper: an example task `README.md`, the agent system prompt, and the full language inventory.

B REPRODUCIBILITY ARTIFACTS

B.1 EXAMPLE TASK README.MD

```
Use ‘lang-ran‘ as a standalone command (do not chain it with other commands).

Interpreter runs are limited to 20. Use ‘lang-ran --count‘ to see how many runs you have used.

How to run files:
lang-ran example1.l14
See available example files by running: ls -la

TASK:
Output exactly: 6 5 4 5 4 6

Write your final solution to: solution.l14

Constraints:
- Only one variable may be used in the entire program
- Your program must contain exactly these lines: 2, 10, 20, 30, 40, 45, 50, 70, 100, 120, 130, 140
- Must use exactly 3 OUTs in total
```

B.2 AGENT SYSTEM PROMPT (USED FOR INTERACTIVE EVALUATION)

For reproducibility, we include the core system prompt used for the interactive agent setup.

```
Please solve this task: {{task}}

   You can execute bash commands and edit files to implement the necessary changes.

   ## Recommended Workflow

   This workflow should be done step-by-step. Learn from each experiment before proceeding.

   1. Read examples to understand the system’s behavior. It is highly recommended to reverse-
        engineer the provided examples to verify your understanding of the system’s mechanics.
   2. Form a hypothesis and predict what your code will output BEFORE running it
   3. Run your experiment and compare actual output to your prediction
   4. If output differs from prediction: STOP and analyze why. What does this reveal?
   5. Update your mental model, then design the next experiment based on what you learned
   6. ONLY when your solution produces the CORRECT output, submit by issuing: ‘echo
        COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT‘.
     Do not combine it with any other command. <important>After this command, you cannot
          continue working on this task.</important>

   <important>
   Do NOT submit until your solution WORKS and produces the expected output. If your code
        fails or loops infinitely, keep experimenting and debugging. Submitting a broken
        solution is worse than continuing to try.
```

````
</important>

## Important Rules

1. Every response must contain exactly one action
2. The action must be enclosed in triple backticks
3. Directory or environment variable changes are not persistent. Every action is executed
     in a new subshell.
   However, you can prefix any action with ‘MY_ENV_VAR=MY_VALUE cd /path/to/working/dir &&
        ...‘ or write/load environment variables from files

## Formatting your response

Here is an example of a correct response:

<example_response>
THOUGHT: Based on example2, I hypothesize that setting variable N changes where line N
     jumps to. I predict this program will print "A" then loop forever because line 20 will
      redirect to line 10.

```bash
ruby interpreter.rb test.fo
```
</example_response>

## Useful command examples

### Create a new file:

<important>
Do NOT chain commands with && after a heredoc EOF. This will cause a syntax error.
Instead, create the file in one command, then run a separate command to execute it.
</important>

```bash
cat <<’EOF’ > newfile.py
import numpy as np
hello = "world"
print(hello)
EOF
```

### Edit files with sed:

{%- if system == "Darwin" -%}
<important>
You are on MacOS. For all the below examples, you need to use ‘sed -i ’’‘ instead of ‘sed -
     i‘.
</important>
{%- endif -%}

```bash
# Replace all occurrences
sed -i ’s/old_string/new_string/g’ filename.py

# Replace only first occurrence
sed -i ’s/old_string/new_string/’ filename.py

# Replace first occurrence on line 1
sed -i ’1s/old_string/new_string/’ filename.py

# Replace all occurrences in lines 1-10
sed -i ’1,10s/old_string/new_string/g’ filename.py


### View file content:

```bash
# View specific lines with numbers
nl -ba filename.py | sed -n ’10,20p’
```

### Any other command you want to run

```bash
anything
```
````
````

C LANGUAGE INVENTORY

Language	Control-flow mechanism	Core execution rule (ground truth)	Primary induction challenge
`L1`	3-lane cyclic scheduler	Three independent lane instruction pointers; `UDA` rotates the active lane A→B→C; execution halts when the current lane is exhausted.	Discover multi-IP interleaving and the semantics of `UDA` ("switch lanes", not a jump).
`L2`	Dependency gating (`XIH`)	Single-use labeled lines with precedence constraints; each step runs the smallest eligible label; deadlocks/ties are runtime errors.	Infer hidden dependency constraints from I/O and error patterns.
`L3`	Active-set wrap scheduler	Labels are Active/Dormant; scheduler picks the next Active label $>$ current (else wraps); executed labels become Dormant; `HEX`/`DVX` act non-locally.	Infer stateful activity and wraparound selection (plus `HEX`/`DVX` effects).
`L4`	Stride register over sorted list	Program is sorted by label into a list; an index advances by a persistent stride (`RCS`); out-of-bounds halts (no wrap).	Infer index-based stepping and the termination condition.
`L5`	Replay batches (chorus+`NBA`)	`[*]` marks chorus lines; `NBA k` replays chorus batch k defined by `NBA` positions; no nesting during replay.	Recover batch segmentation and replay semantics from executions.
`L6`	Extreme selection (min/max)	Each step selects either the min or max active label based on a toggle (`CGH`); executed labels incur a one-step cooldown.	Infer a hidden scheduler state (toggle + cooldown) rather than a static order.
`L7`	Delayed echo injection (`~k`)	Main pass executes lines once in ascending order; `~k` schedules one extra "echo" immediately before the line k positions later; echoes do not recurse.	Infer position-based delayed re-execution and its non-recursive nature.
`L8`	Cursor-guided nearest-neighbor jumps	Maintain an integer cursor; after executing label L, target $T = L +$ `cursor` and execute the label closest to T (ties to smaller); `QTA` sets cursor.	Infer geometric "closest-to-target" control flow and tie-breaking.
`L9`	Penalty scheduler (`RGW`)	Each label has a penalty; each step runs the unexecuted label with minimum penalty (ties to smallest label); `RGW` increments a label's penalty.	Infer latent per-label scores that modulate execution order.
`L10`	Dual-rail interleaving	Two rails (L/R) with separate instruction pointers; alternate by default; `~` delays switching for one step; < chooses the start rail.	Infer two concurrent pointers and modifier semantics from traces.
`L11`	Two-pass with marker reordering	Pass 1 executes all lines once; pass 2 replays only marked lines, but reorders them by swapping adjacent `v`-blocks followed by `^`-blocks.	Infer multi-pass execution and the non-local pass-2 ordering rule.
`L12`	Scoped record/replay (`UWR`/`NHO`)	`UWR sym` records a ribbon starting after the marker; `NHO sym` replays from that ribbon up to the next `UWR`; nested `NHO` during replay errors.	Infer replay boundaries, symbol scoping, and non-reentrancy.
`L13`	Quota-driven round-robin (`SIZ`)	Each label has an execution quota; the interpreter cycles ascending through labels with remaining quota; each execution decrements quota; `SIZ` increases quota.	Infer repeated execution under quotas and dynamic quota updates.
`L14`	Deferred return stack (`AYZ`/`STOP`)	`AYZ` pushes pinned labels; main path is ascending single-use; `STOP` jumps to the most recent pin if any; remaining pins execute in LIFO order.	Infer that `STOP` can be non-terminal and that pins are LIFO.
`L15`	Consumptive subroutines (`FWN`/`KIE`)	`FWN` pushes call-site and jumps; labels are consumptive; `KIE` returns to the next unexecuted label after the call-site.	Infer call/return semantics under "execute at most once".
`L16`	Direction flip via `[!]`	A global direction selects the next label among remaining labels (with wrap); executing a `[!]` line flips direction for the next step; labels are single-use.	Infer hidden direction state and flip triggers from behavior.
`L17`	Bracket tournament (`AMO`)	Adjacent labels "match"; the winner executes and advances; `AMO` toggles whether low/high label wins, but applies to the next match; byes may occur.	Infer match-based control flow and delayed toggle effects.
`L18`	Guarded execution (relative guards)	Each line has a guard; lines execute only if the guard condition holds over previously executed labels; `[*]` always executes first; star-prefixed guards implement relative offsets; one `[!]` never executes.	Infer the guard language, especially star-offset semantics.
`L19`	Countdown interrupts (`XYZ`/`ABC`)	Labels may carry timers; timers decrement each step; when any hits 0, that label interrupts normal flow (ties to smallest); timers reset; `XYZ`/`ABC` disable/enable timers.	Infer interrupt priority and periodic timer reset behavior.

Table 3: Language inventory: control-flow semantics and the primary induction challenge. File extensions are omitted for readability (they follow `.l#` by language index).

Task package example (L1)

README.md (Task 2)

```
TASK:
Output exactly: 6 10 7

Write your final solution to
     : solution.l1

Constraints:
- Only one variable may be
     used in the entire
     program
- Must have exactly 4 lines
     labeled with A and
     exactly 4 lines labeled
      with B
```

README.md (Task 4)

```
TASK:
Output exactly: 3 7 9 10 2 1

Write your final solution to
     : solution.l1

Constraints:
- Only one variable may be
     used in the entire
     program
- Must include labels A, B,
     and C
- Must have exactly 4 lines
     labeled with A, exactly
      7 lines labeled with B,
      and exactly 4 lines
     labeled with C
```

example1.l1

```
A SET x 1
B SET x 1
B OUT x
C SET x 2
A OUT x
A UDA
C OUT x
```

Output: `1 1`

example2.l1

```
A SET x 4
B SET x 1
C SET x 4
A OUT x
```

Output: `4`

example3.l1

```
A SET x 4
B SET x 1
C SET x 4
A UDA
B OUT x
C OUT x
B SET x 3
```

Output: `1`

example4.l1

```
A SET x 4
A SET x 3
A OUT x
B OUT x
```

Output: `3`

solution.l1 (Task 2)

```
A SET x 6
A OUT x
A SET x 10
A UDA
B OUT x
B SET x 7
B OUT x
```

Output: `6 10 7`

solution.l1 (Task 4)

```
A SET x 3
A OUT x
A SET x 7
A UDA
B OUT x
B SET x 9
B OUT x
B SET x 10
B OUT x
B SET x 2
B UDA
C OUT x
C SET x 1
C OUT x
```

Output: `3 7 9 10 2 1`

Execution model (L1).

- Three lanes (A/B/C), each runs top-to-bottom within its own lane list.
- Each lane has its own instruction pointer.
- Execution starts in lane A at its first instruction.
- UDA switches the active lane in a fixed cycle: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A $\rightarrow \ldots$
- When switching, a lane resumes where it left off.
- If the current lane is exhausted, execution halts.
- STOP halts immediately.

Task 2 explanation. *Special instructions:* UDA.

1. Lane A (line 1): SET x 6 sets x = 6.
2. Lane A (line 2): OUT x prints 6. Output: 6.
3. Lane A (line 3): SET x 10 sets x = 10.
4. Lane A (line 4): UDA switches the active lane to the next one in the cycle.
5. Lane B (line 5): OUT x prints 10. Output: 6 10.
6. Lane B (line 6): SET x 7 sets x = 7.
7. Lane B (line 7): OUT x prints 7. Output: 6 10 7.

Final output: `6 10 7`.

Task 4 explanation. *Special instructions:* UDA.

1. Lane A (line 1): SET x 3 sets x = 3.
2. Lane A (line 2): OUT x prints 3. Output: 3.
3. Lane A (line 3): SET x 7 sets x = 7.
4. Lane A (line 4): UDA switches the active lane to the next one in the cycle.
5. Lane B (line 5): OUT x prints 7. Output: 3 7.
6. Lane B (line 6): SET x 9 sets x = 9.
7. Lane B (line 7): OUT x prints 9. Output: 3 7 9.
8. Lane B (line 8): SET x 10 sets x = 10.
9. Lane B (line 9): OUT x prints 10. Output: 3 7 9 10.
10. Lane B (line 10): SET x 2 sets x = 2.
11. Lane B (line 11): UDA switches the active lane to the next one in the cycle.
12. Lane C (line 12): OUT x prints 2. Output: 3 7 9 10 2.
13. Lane C (line 13): SET x 1 sets x = 1.
14. Lane C (line 14): OUT x prints 1. Output: 3 7 9 10 2 1.

Final output: `3 7 9 10 2 1`.

Figure 5: Task package example for L1: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L2)

README.md (Task 2)

```
TASK:
Output exactly: 1 7

Write your final solution to
     : solution.l2

Constraints:
- Your program must contain
     exactly these lines: 5,
      10, 20, 30, 40
- Line 20 must be SET x 7
- Line 30 must be OUT x
- Line 40 must be OUT x
- Only one variable may be
     used in the entire
     program
```

README.md (Task 4)

```
TASK:
Output exactly: 3 7 10 4 4

Write your final solution to
     : solution.l2

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110
- Must use exactly 4 SETs in
      total
- Line 110 must be SET x 4
```

example1.l2

```
5 SET x 1
10 XIH 30 35
20 OUT x
30 SET x 7
35 SET x 1
40 OUT x
```

Output: 1 7

example3.l2

```
10 SET x 1
20 XIH 40 60
40 OUT x
50 SET x 9
55 SET x 1
60 OUT x
```

Output: 1 1

example2.l2

```
5 XIH 20 30
10 SET x 4
20 OUT x
30 SET x x + 1
```

Output: 5

example4.l2

```
10 SET x 2
20 SET x x + 1
30 OUT x
40 OUT x
```

Output: 3 3

solution.l2 (Task 2)

```
5 XIH 20 30
10 SET x 1
20 SET x 7
30 OUT x
40 OUT x
```

Output: 1 7

solution.l2 (Task 4)

```
10 SET x 3
20 OUT x
30 XIH 50 100
40 SET x 7
50 OUT x
55 OUT x
60 SET x 10
70 OUT x
90 XIH 100 110
100 OUT x
110 SET x 4
```

Output: 3 7 10 4 4

Execution model (L2).

- Labels are positive integers and execute at most once.
- A dependency edge $a \leftarrow b$ means label a can only execute after label b.
- Dependencies are declared with <label> XIH a b.
- Each step executes the smallest eligible label.
- If no eligible labels exist while unexecuted labels remain, it's a deadlock error.
- If multiple XIH-dependent labels become eligible at the same time, it's a tie error.
- STOP halts immediately.

Task 2 explanation. *Special instructions:* XIH a b.

1. Label 5: XIH 20 30 adds dependency 20 $\leftarrow$ 30 (label 20 waits for 30).
2. Label 10: SET x 1 sets x = 1.
3. Label 30: OUT x prints 1. Output: 1.
4. Label 20: SET x 7 sets x = 7.
5. Label 40: OUT x prints 7. Output: 1 7.

Final output: 1 7.

Task 4 explanation. *Special instructions:* XIH a b.

1. Label 10: SET x 3 sets x = 3.
2. Label 20: OUT x prints 3. Output: 3.
3. Label 30: XIH 50 100 adds dependency 50 ¡- 100 (label 50 waits for 100).
4. Label 40: SET x 7 sets x = 7.
5. Label 55: OUT x prints 7. Output: 3 7.
6. Label 60: SET x 10 sets x = 10.
7. Label 70: OUT x prints 10. Output: 3 7 10.
8. Label 90: XIH 100 110 adds dependency 100 ¡- 110 (label 100 waits for 110).
9. Label 110: SET x 4 sets x = 4.
10. Label 100: OUT x prints 4. Output: 3 7 10 4.
11. Label 50: OUT x prints 4. Output: 3 7 10 4 4.

Final output: 3 7 10 4 4.

Figure 6: Task package example for L2: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L3)

README.md (Task 2)

```
TASK:
Goal output:
5

Write your final solution to
     : solution.l3

Constraints:
- Only one variable may be
     used in the entire
     program
- Must use exactly 3 OUTs in
      total
```

README.md (Task 4)

```
TASK:
Goal output (tokens): 2 1 3
     4 5

Write your final solution to
     : solution.l3

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 45, 50, 55,
      60, 70, 80, 90, 110
- Line 30 must be SET x 4
- Must use exactly 5 SETs in
      total
```

example1.l3

```
10 SET a 1
20 OUT a
30 SET a 2
40 OUT a
```

Output: `1 2`

example3.l3

```
10 SET a 1
15 DVX 30
20 OUT a
30 OUT a
```

Output: `1`

example5.l3

```
10 SET a 1
20 OUT a
```

Output: `1`

example2.l3

```
10 SET a 1
20 OUT a
30 HEX 20
```

Output: `1 1`

example4.l3

```
10 SET a 1
15 DVX 20
20 OUT a
```

Output: `(no output)`

solution.l3 (Task 2)

```
10 SET x 5
20 OUT x
30 DVX 50
40 DVX 60
50 OUT x
60 OUT x
```

Output: `5`

solution.l3 (Task 4)

```
10 SET x 2
20 DVX 30
30 SET x 4
40 OUT x
45 SET x 1
50 OUT x
55 SET x 3
60 OUT x
70 HEX 30
80 SET x 5
90 OUT x
```

Output: `2 1 3 5`

Execution model (L3).

- Labels are positive integers and are either Active or Dormant.
- Start at the smallest label.
- The scheduler picks the next Active label greater than the current label; if none, it wraps to the smallest Active label.
- After a label executes, it becomes Dormant (cannot run again).
- Program halts when no Active labels remain.

Task 2 explanation. *Special instructions:* DVX label (marks the target label Dormant without executing it).

1. Label 10: SET x 5 sets x = 5.
2. Label 20: OUT x prints 5. Output: 5.
3. Label 30: DVX 50 marks label 50 Dormant (so it will not execute).
4. Label 40: DVX 60 marks label 60 Dormant (so it will not execute).
5. No Active labels remain, so execution halts.

Final output: `5`.

Task 4 explanation. *Special instructions:* DVX label (marks the target label Dormant without executing it), HEX label (immediately executes the target label's instruction, then makes it Dormant).

1. Label 10: SET x 2 sets x = 2.
2. Label 20: DVX 30 marks label 30 Dormant (so it will not execute in the normal schedule).
3. Label 40: OUT x prints 2. Output: 2.
4. Label 45: SET x 1 sets x = 1.
5. Label 50: OUT x prints 1. Output: 2 1.
6. Label 55: SET x 3 sets x = 3.
7. Label 60: OUT x prints 3. Output: 2 1 3.
8. Label 70: HEX 30 immediately executes label 30's instruction (SET x 4), then makes label 30 Dormant.
9. Label 80: SET x 5 sets x = 5.
10. Label 90: OUT x prints 5. Output: 2 1 3 5.

Final output: `2 1 3 5`.

Figure 7: Task package example for L3: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L4)

README.md (Task 2)

```
TASK:
Output exactly: 3

Write your final solution to
     : solution.l4

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50
- Line 20 must be STOP
```

README.md (Task 4)

```
TASK:
Output exactly: 3 2 1 0

Write your final solution to
     : solution.l4

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110, 120,
      130
- Line 60 must be STOP
- Line 80 must be STOP
- Line 100 must be STOP
```

example1.l4

```
10 SET x 1
20 SET x x + 1
30 OUT x
40 STOP
```

Output: `2`

example2.l4

```
10 RCS 2
20 SET x 1
30 SET x 1
40 SET x 99
50 OUT x
60 SET x 99
70 STOP
```

Output: `1`

example3.l4

```
10 SET x 1
20 OUT x
30 RCS 2
40 SET x 2
50 OUT x
60 OUT x
70 STOP
```

Output: `1 1`

example4.l4

```
10 SET x 1
15 RCS 2
40 SET x 2
45 OUT x
70 SET x 1
75 OUT x
100 STOP
```

Output: `1 1`

solution.l4 (Task 2)

```
10 RCS 2
20 STOP
30 SET x 3
40 STOP
50 OUT x
```

Output: `3`

solution.l4 (Task 4)

```
10 SET x 3
20 OUT x
30 SET x 2
40 OUT x
50 RCS 2
60 STOP
70 SET x 1
80 STOP
90 OUT x
100 STOP
110 RCS 1
120 SET x 0
130 OUT x
```

Output: `3 2 1 0`

Execution model (L4).

- Lines are sorted by label and stored in a list.
- A STRIDE register controls which index executes next.
- Start at the smallest label (index 0) with stride = 1.
- After each instruction, index += stride.
- If index goes out of bounds, execution halts (no wraparound).
- STOP halts immediately.

Task 2 explanation. *Special instructions:* RCS expr (sets stride; must be positive and non-zero).

1. Label 10: RCS 2 sets stride = 2.
   Note: stride = 2; move: 0 + 2 = 2.
2. Label 30: SET x 3 sets x = 3.
   Note: x = 3; move: 2 + 2 = 4.
3. Label 50: OUT x prints 3. Output: 3.
   Note: OUTPUT: 3; move: 4 + 2 = 6 (out of bounds, halt).

Final output: `3`.

Task 4 explanation. *Special instructions:* RCS expr (sets stride; must be positive and non-zero).

1. Label 10: SET x 3 sets x = 3.
   Note: x = 3; move: 0 + 1 = 1.
2. Label 20: OUT x prints 3. Output: 3.
   Note: OUTPUT: 3; move: 1 + 1 = 2.
3. Label 30: SET x 2 sets x = 2.
   Note: x = 2; move: 2 + 1 = 3.
4. Label 40: OUT x prints 2. Output: 3 2.
   Note: OUTPUT: 2; move: 3 + 1 = 4.
5. Label 50: RCS 2 sets stride = 2.
   Note: stride = 2; move: 4 + 2 = 6.
6. Label 70: SET x 1 sets x = 1.
   Note: x = 1; move: 6 + 2 = 8.
7. Label 90: OUT x prints 1. Output: 3 2 1.
   Note: OUTPUT: 1; move: 8 + 2 = 10.
8. Label 110: RCS 1 sets stride = 1.
   Note: stride = 1; move: 10 + 1 = 11.
9. Label 120: SET x 0 sets x = 0.
   Note: x = 0; move: 11 + 1 = 12.
10. Label 130: OUT x prints 0. Output: 3 2 1 0.
    Note: OUTPUT: 0; move: 12 + 1 = 13 (out of bounds, halt).

Final output: `3 2 1 0`.

Figure 8: Task package example for L4: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

Task package example (L5)

README.md (Task 2)

```
TASK:
Output exactly: 5 5

Write your final solution to
     : solution.l5

Constraints:
- Only one variable may be
     used in the entire
     program
- Must use exactly 1 OUT in
     total
```

README.md (Task 4)

```
TASK:
Output exactly: 7 7 9 4 9 8
     8

Write your final solution to
     : solution.l5

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110
- Must use exactly 4 OUTs in
      total
```

example1.l5

```
10 SET x 1
20* OUT x
30 SET x x + 1
40 NBA 1
```

Output: `1 2`

example2.l5

```
10 SET x 1
20* OUT x
30 SET x x + 4
40* OUT x
50 NBA 1
60* SET x x + 1
70 NBA 2
```

Output: `1 5 5 5`

example3.l5

```
10 SET x 1
20* SET x x + 1
30 OUT x
40 NBA 1
50 OUT x
```

Output: `2 3`

example4.l5

```
10 SET x 2
20 OUT x
30 SET x x - 1
40 OUT x
```

Output: `2 1`

solution.l5 (Task 2)

```
10 SET x 5
20* OUT x
30 NBA 1
```

Output: `5 5`

solution.l5 (Task 4)

```
10 SET x 7
20* OUT x
30 NBA 1
40* SET x 9
50 OUT x
60 SET x 4
70* OUT x
80 NBA 2
90* SET x 8
100* OUT x
110 NBA 3
```

Output: `7 7 9 4 9 8 8`

Execution model (L5).

- Lines execute in ascending label order.
- A * after the label marks that line as part of the chorus.
- Chorus lines are grouped into batches based on NBA positions:
- batch 1 is before the first NBA, batch 2 between first and second, etc.
- NBA k immediately replays chorus batch k (ascending label order), then resumes.
- NBA is illegal during replay (no nesting).
- STOP halts immediately.

Task 2 explanation. *Special instructions:* NBA k.

1. Label 10: SET x 5 sets x = 5.
2. Label 20*: OUT x prints 5. Output: 5.
3. Label 30: NBA 1 replays chorus batch 1.
4. Label 20*: OUT x prints 5. Output: 5 5.
   Note: chorus replay.

Final output: `5 5`.

Task 4 explanation. *Special instructions:* NBA k.

1. Label 10: SET x 7 sets x = 7.
2. Label 20*: OUT x prints 7. Output: 7.
3. Label 30: NBA 1 replays chorus batch 1.
4. Label 20*: OUT x prints 7. Output: 7 7.
   Note: chorus replay.
5. Label 40*: SET x 9 sets x = 9.
6. Label 50: OUT x prints 9. Output: 7 7 9.
7. Label 60: SET x 4 sets x = 4.
8. Label 70*: OUT x prints 4. Output: 7 7 9 4.
9. Label 80: NBA 2 replays chorus batch 2.
10. Label 40*: SET x 9 sets x = 9.
    Note: chorus replay.
11. Label 70*: OUT x prints 9. Output: 7 7 9 4 9.
    Note: chorus replay.
12. Label 90*: SET x 8 sets x = 8.
13. Label 100*: OUT x prints 8. Output: 7 7 9 4 9 8.
14. Label 110: NBA 3 replays chorus batch 3.
15. Label 90*: SET x 8 sets x = 8.
    Note: chorus replay.
16. Label 100*: OUT x prints 8. Output: 7 7 9 4 9 8 8.
    Note: chorus replay.

Final output: `7 7 9 4 9 8 8`.

Figure 9: Task package example for L5: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025

Task package example (L6)

README.md (Task 2)

```
TASK:
Output exactly: 1 2

Write your final solution to
     : solution.l6

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 35, 40, 50
- Line 35 must be STOP
```

README.md (Task 4)

```
TASK:
Output exactly: 9 3 3 7 10
     10

Write your final solution to
     : solution.l6

Interpreter call budget: You
      have {N} total
     interpreter runs for
     this task.

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110, 120,
      130
- Line 50 must be SET x 10
- Line 70 must be OUT x
- Line 80 must be STOP
```

example1.l6

```
10 SET x 1
20 OUT x
30 SET x 2
40 STOP
```

Output: `1`

example3.l6

```
10 SET x 0
15 CGH
20 SET x 0
30 OUT x
40 SET x 1
```

Output: `1`

example5.l6

```
10 SET x 1
20 OUT x
30 OUT x
40 STOP
```

Output: `1 1`

example2.l6

```
10 SET x 5
20 OUT x
30 STOP
40 SET x 0
```

Output: `5`

example4.l6

```
10 SET x 1
20 OUT x
30 SET x 2
40 OUT x
```

Output: `1 2`

solution.l6 (Task 2)

```
10 SET x 1
20 OUT x
30 CGH
35 STOP
40 OUT x
50 SET x 2
```

Output: `1 2`

solution.l6 (Task 4)

```
10 SET x 9
20 CGH
30 SET x 7
40 OUT x
50 SET x 10
60 OUT x
70 OUT x
80 STOP
90 CGH
100 OUT x
110 OUT x
120 SET x 3
130 OUT x
```

Output: `9 3 3 7 10 10`

Execution model (L6).

- Labels are positive integers and execute at most once.
- At each step, consider the min and max active labels.
- A state bit pick chooses which runs (0 = min, 1 = max).
- CGH toggles pick.
- After executing a label, it goes on cooldown for 1 step (can't run next step).
- STOP halts immediately.

Task 2 explanation. *Special instructions:* CGH.

1. Label 10: SET x 1 sets x = 1.
2. Label 20: OUT x prints 1. Output: 1.
3. Label 30: CGH toggles the min/max pick state for the next step.
   Note: CGH → pick=1.
4. Label 50: SET x 2 sets x = 2.
5. Label 40: OUT x prints 2. Output: 1 2.
6. Label 35: STOP halts execution.

Final output: `1 2`.

Task 4 explanation. *Special instructions:* CGH.

1. Label 10: SET x 9 sets x = 9.
2. Label 20: CGH toggles the min/max pick state for the next step.
   Note: CGH → pick=1.
3. Label 130: OUT x prints 9. Output: 9.
4. Label 120: SET x 3 sets x = 3.
5. Label 110: OUT x prints 3. Output: 9 3.
6. Label 100: OUT x prints 3. Output: 9 3 3.
7. Label 90: CGH toggles the min/max pick state for the next step.
   Note: CGH → pick=0.
8. Label 30: SET x 7 sets x = 7.
9. Label 40: OUT x prints 7. Output: 9 3 3 7.
10. Label 50: SET x 10 sets x = 10.
11. Label 60: OUT x prints 10. Output: 9 3 3 7 10.
12. Label 70: OUT x prints 10. Output: 9 3 3 7 10 10.
13. Label 80: STOP halts execution.

Final output: `9 3 3 7 10 10`.

Figure 10: Task package example for L6: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L7)

README.md (Task 2)

```
TASK:
Output exactly: 1 2 2

Write your final solution to
     : solution.l7

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40
```

README.md (Task 4)

```
TASK:
Output exactly: 10 7 10 7 13
      8 3 4 8

Write your final solution to
     : solution.l7

Constraints:
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110, 120
- Only one variable may be
     used in the entire
     program
- Must use exactly 6 SETs in
      total
```

example1.l7

```
10 SET x 1
20~2 OUT x
30 SET x 2
40 OUT x
```

Output: `1 2 2`

example3.l7

```
10 SET x 1
20~1 OUT x
30 SET x 9
40 OUT x
```

Output: `1 1 9`

example2.l7

```
10 SET x 3
20~1 OUT x
30 OUT x
40~2 SET x 4
50 OUT x
```

Output: `3 3 3 4`

example4.l7

```
10 SET x 2
20~1 OUT x
30 SET x 7
40 OUT x
```

Output: `2 2 7`

solution.l7 (Task 2)

```
10 SET x 1
20~2 OUT x
30 SET x 2
40 OUT x
```

Output: `1 2 2`

solution.l7 (Task 4)

```
10~4 SET x 10
20~3 OUT x
30~2 SET x 7
40~1 OUT x
50 SET x 13
60 OUT x
70~5 SET x 8
80~2 OUT x
90 SET x 3
100 SET x 4
110 OUT x
120 OUT x
```

Output: `10 7 10 7 13 8 3 4 8`

Execution model (L7).

- Main pass executes each line once in ascending label order.
- A line with delay k ¿ 0 schedules a single echo of itself to run immediately before the line that is k positions after it in label order.
- Echoes execute at their scheduled label and do not schedule further echoes.
- When the main pass completes, the program halts.
- STOP halts immediately (even if echoes are pending).

Task 2 explanation.

1. Label 10: SET x 1 sets x = 1.
2. Label 20˜2: OUT x prints 1. Output: 1.
3. Label 30: SET x 2 sets x = 2.
4. Label 20˜2: OUT x prints 2. Output: 1 2.
   Note: echo of a delayed line.
5. Label 40: OUT x prints 2. Output: 1 2 2.

Final output: `1 2 2`.

Task 4 explanation.

1. Label 10˜4: SET x 10 sets x = 10.
2. Label 20˜3: OUT x prints 10. Output: 10.
3. Label 30˜2: SET x 7 sets x = 7.
4. Label 40˜1: OUT x prints 7. Output: 10 7.
5. Label 10˜4: SET x 10 sets x = 10.
   Note: echo of a delayed line.
6. Label 20˜3: OUT x prints 10. Output: 10 7 10.
   Note: echo of a delayed line.
7. Label 30˜2: SET x 7 sets x = 7.
   Note: echo of a delayed line.
8. Label 40˜1: OUT x prints 7. Output: 10 7 10 7.
   Note: echo of a delayed line.
9. Label 50: SET x 13 sets x = 13.
10. Label 60: OUT x prints 13. Output: 10 7 10 7 13.
11. Label 70˜5: SET x 8 sets x = 8.
12. Label 80˜2: OUT x prints 8. Output: 10 7 10 7 13 8.
13. Label 90: SET x 3 sets x = 3.
14. Label 80˜2: OUT x prints 3. Output: 10 7 10 7 13 8 3.
    Note: echo of a delayed line.
15. Label 100: SET x 4 sets x = 4.
16. Label 110: OUT x prints 4. Output: 10 7 10 7 13 8 3 4.
17. Label 70˜5: SET x 8 sets x = 8.
    Note: echo of a delayed line.
18. Label 120: OUT x prints 8. Output: 10 7 10 7 13 8 3 4 8.

Final output: `10 7 10 7 13 8 3 4 8`.

Figure 11: Task package example for L7: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L8)

README.md (Task 2)

```
TASK:
Write a program that outputs
      exactly: 4 5

Write your final solution to
     : solution.l8

Constraints:
- Only one variable may be
     used in the entire
     program
```

README.md (Task 4)

```
Task:
Output exactly: 4 6 7 9

Write your final solution to
     : solution.l8

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 0,
      8, 22, 50, 78, 81, 88,
      100, 106, 109, 139,
     179, 200, 227
- Line 78 must be STOP
```

example1.l8

```
10 QTA 7
20 SET a 5
30 OUT a
40 STOP
```

Output: 5

example3.l8

```
0 QTA 6
10 SET x 1
20 QTA 13
30 SET x x + 4
50 OUT x
60 STOP
```

Output: 5

example5.l8

```
0 QTA 10
10 SET x 1
11 SET x 2
25 OUT x
40 STOP
```

Output: 1

example2.l8

```
0 QTA 14
10 SET a 2
25 SET b a * 3 + 1
40 OUT b
55 OUT a
70 STOP
```

Output: 7 2

example4.l8

```
0 QTA 13
10 SET a 1
24 QTA 10
35 SET a a + 2
47 OUT a
60 STOP
```

Output: 3

solution.l8 (Task 2)

```
0 QTA 10
10 SET a 4
20 OUT a
30 SET a 5
40 OUT a
50 STOP
```

Output: 4 5

solution.l8 (Task 4)

```
0 QTA 10
8 SET x 4
22 QTA 30
50 OUT x
78 STOP
81 QTA 7
88 SET x 6
100 OUT x
106 SET x 7
109 QTA 33
139 OUT x
179 SET x 9
200 OUT x
227 STOP
```

Output: 4 6 7 9

Execution model (L8).

- Execution starts at the smallest label.
- A cursor integer guides control flow.
- After executing a line with label L and cursor C:
- target T = L + C
- next label N minimizes —N - T—; ties break to the smaller label.
- STOP halts immediately.

Task 2 explanation. *Special instructions:* QTA expr (sets cursor).

1. Label 0: QTA 10 sets the cursor to 10.
   Note: target = 10, next label = 10.
2. Label 10: SET a 4 sets a = 4.
   Note: target = 20, next label = 20.
3. Label 20: OUT a prints 4. Output: 4.
   Note: target = 30, next label = 30.
4. Label 30: SET a 5 sets a = 5.
   Note: target = 40, next label = 40.
5. Label 40: OUT a prints 5. Output: 4 5.
   Note: target = 50, next label = 50.
6. Label 50: STOP halts execution.
   Note: target = 60, next label = None.

Final output: `4 5`.

Task 4 explanation. *Special instructions:* QTA expr (sets cursor).

1. Label 0: QTA 10 sets the cursor to 10.
   Note: target = 10, next label = 8.
2. Label 8: SET x 4 sets x = 4.
   Note: target = 18, next label = 22.
3. Label 22: QTA 30 sets the cursor to 30.
   Note: target = 52, next label = 50.
4. Label 50: OUT x prints 4. Output: 4.
   Note: target = 80, next label = 81.
5. Label 81: QTA 7 sets the cursor to 7.
   Note: target = 88, next label = 88.
6. Label 88: SET x 6 sets x = 6.
   Note: target = 95, next label = 100.
7. Label 100: OUT x prints 6. Output: 4 6.
   Note: target = 107, next label = 106.
8. Label 106: SET x 7 sets x = 7.
   Note: target = 113, next label = 109.
9. Label 109: QTA 33 sets the cursor to 33.
   Note: target = 142, next label = 139.
10. Label 139: OUT x prints 7. Output: 4 6 7.
    Note: target = 172, next label = 179.
11. Label 179: SET x 9 sets x = 9.
    Note: target = 212, next label = 200.
12. Label 200: OUT x prints 9. Output: 4 6 7 9.
    Note: target = 233, next label = 227.
13. Label 227: STOP halts execution.
    Note: target = 260, next label = None.

Final output: `4 6 7 9`.

Figure 12: Task package example for L8: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L9)

README.md (Task 2)

```
TASK:
Output exactly: 4 4

Write your final solution to
     : solution.l9

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40
- Line 30 must be SET x 4
- Must use exactly 1 SET in
     total
```

example1.l9

```
10 SET x 1
20 RGW 40
30 SET x 9
40 OUT x
```

Output: `9`

example2.l9

```
10 SET x 2
30 RGW 40
40 OUT x
50 SET x 3
```

Output: `3`

example3.l9

```
10 SET x 1
20 RGW 30
25 OUT x
30 OUT x
40 SET x 9
50 OUT x
```

Output: `1 9 9`

example4.l9

```
10 SET x 5
20 OUT x
30 RGW 50
40 SET x 7
50 OUT x
60 OUT x
```

Output: `5 7 7`

solution.l9 (Task 2)

```
10 RGW 20
20 OUT x
30 SET x 4
40 OUT x
```

Output: `4 4`

README.md (Task 4)

```
TASK:
Output exactly: 10 3 4 7 9

Write your final solution to
     : solution.l9

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110, 120,
      130, 140, 150, 160
- Line 30 must be SET x 9
- Line 50 must be SET x 7
- Must use exactly 5 SETs in
      total
```

solution.l9 (Task 4)

```
10 RGW 30
20 RGW 30
30 SET x 9
40 RGW 50
50 SET x 7
60 SET x 10
70 OUT x
80 RGW 80
90 OUT x
100 RGW 110
110 RGW 110
120 OUT x
130 SET x 3
140 OUT x
150 SET x 4
160 OUT x
```

Output: `10 10 10 3 4`

Execution model (L9).

- Labels are positive integers and execute at most once.
- Each label has a penalty score (starts at 0).
- Each step selects the unexecuted label with the minimum penalty; ties break to smallest label.
- RGW ¡label¿ increments the penalty of that target label by 1.
- STOP halts immediately.

Task 2 explanation. *Special instructions:* RGW label.

1. Label 10: RGW 20 increases the penalty for label 20, making it less likely to be chosen next.
2. Label 30: SET x 4 sets x = 4.
3. Label 40: OUT x prints 4. Output: 4.
4. Label 20: OUT x prints 4. Output: 4 4.

Final output: `4 4`.

Task 4 explanation. *Special instructions:* RGW label.

1. Label 10: RGW 30 increases the penalty for label 30, making it less likely to be chosen next.
2. Label 20: RGW 30 increases the penalty for label 30, making it less likely to be chosen next.
3. Label 40: RGW 50 increases the penalty for label 50, making it less likely to be chosen next.
4. Label 60: SET x 10 sets x = 10.
5. Label 70: OUT x prints 10. Output: 10.
6. Label 80: RGW 80 increases the penalty for label 80, making it less likely to be chosen next.
7. Label 90: OUT x prints 10. Output: 10 10.
8. Label 100: RGW 110 increases the penalty for label 110, making it less likely to be chosen next.
9. Label 120: OUT x prints 10. Output: 10 10 10.
10. Label 130: SET x 3 sets x = 3.
11. Label 140: OUT x prints 3. Output: 10 10 10 3.
12. Label 150: SET x 4 sets x = 4.
13. Label 160: OUT x prints 4. Output: 10 10 10 3 4.
14. Label 50: SET x 7 sets x = 7.
15. Label 110: RGW 110 increases the penalty for label 110, making it less likely to be chosen next.
16. Label 30: SET x 9 sets x = 9.

Final output: `10 10 10 3 4`.

Figure 13: Task package example for L9: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L10)

README.md (Task 2)

```
TASK:
Output exactly: 4 5 9

Write your final solution to
     : solution.l10

Constraints:
- Only one variable may be
     used in the entire
     program
- Must have exactly 4 lines
     labeled with L
- Must have exactly 2 lines
     labeled with R
```

README.md (Task 4)

```
TASK:
Output exactly: 1 2 7 8 3

Write your final solution to
     : solution.l10

Constraints:
- Only one variable may be
     used in the entire
     program
- The third R label must be
     OUT x
- The second L label must be
      OUT x
- Must have exactly 5 lines
     labeled with R
- Must have exactly 5 lines
     labeled with L
```

example1.l10

```
L< SET a 1
R SET c 3
L SET b 2
R SET d 4
L OUT a
R OUT c
L OUT b
R OUT d
```

Output: 1 3 2 4

example3.l10

```
L SET x 1
L< SET x 2
R OUT x
L OUT x
```

Output: 2 2

example5.l10

```
L< SET x 7
R OUT x
L STOP
R OUT x
```

Output: 7

example2.l10

```
L<~ SET x 1
L OUT x
L SET x x + 1
L OUT x

R OUT x
R SET x 2
```

Output: 1 1 2

example4.l10

```
L< SET x 1
L OUT x
L SET x x + 1
L OUT x

R OUT x
```

Output: 1 1 2

solution.l10 (Task 2)

```
L< SET x 4
R SET y 5
L OUT x
R OUT y
L SET x 9
L OUT x
```

Output: 4 5 9

solution.l10 (Task 4)

```
L OUT x
L~ OUT x
L SET x 7
L SET x 8
L SET x 3

R< SET x 1
R SET x 2
R OUT x
R OUT x
R OUT x
```

Output: 1 2 7 8 3

Execution model (L10).

- Two rails (L and R) with separate instruction pointers.
- Execution alternates between rails by default.
- A sticky marker (˜) on the rail token prevents switching for one step.
- START marker (¡) selects the initial rail and instruction pointer.
- STOP halts immediately.

Task 2 explanation.

1. Lane L (line 1): SET x 4 sets x = 4.
2. Lane R (line 2): SET y 5 sets y = 5.
3. Lane L (line 3): OUT x prints 4. Output: 4.
4. Lane R (line 4): OUT y prints 5. Output: 4 5.
5. Lane L (line 5): SET x 9 sets x = 9.
6. Lane L (line 6): OUT x prints 9. Output: 4 5 9.

Final output: 4 5 9.

Task 4 explanation.

1. Lane R (line 7): SET x 1 sets x = 1.
2. Lane L (line 1): OUT x prints 1. Output: 1.
3. Lane R (line 8): SET x 2 sets x = 2.
4. Lane L (line 2): OUT x prints 2. Output: 1 2.
5. Lane L (line 3): SET x 7 sets x = 7.
6. Lane R (line 9): OUT x prints 7. Output: 1 2 7.
7. Lane L (line 4): SET x 8 sets x = 8.
8. Lane R (line 10): OUT x prints 8. Output: 1 2 7 8.
9. Lane L (line 5): SET x 3 sets x = 3.
10. Lane R (line 11): OUT x prints 3. Output: 1 2 7 8 3.

Final output: 1 2 7 8 3.

Figure 14: Task package example for L10: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L11)

README.md (Task 2)

```
TASK:
Output exactly: 1 2 1

Write your final solution to
    : solution.l11

Constraints:
- Only one variable may be
    used in the entire
    program
- Must use exactly 2 OUTs in
    total
```

README.md (Task 4)

```
TASK:
Output exactly: 2 10 3 7 5
    10 7 3

Write your final solution to
    : solution.l11

Constraints:
- Only one variable may be
    used in the entire
    program
- Your program must contain
    exactly these lines: 5,
    10, 15, 20, 25, 30, 35,
    40, 45
- Must use exactly 4 OUTs in
    total
```

example1.l11

```
10 SET x 1
20^ OUT x
30 SET x x + 1
40 OUT x
50 OUT x
```

Output: `1 2 2 2`

example3.l11

```
10 SET x 1
20 OUT x
30v OUT x
40^ SET x x + 1
50 OUT x
```

Output: `1 1 2 3`

example2.l11

```
10 SET x 1
20 OUT x
30 SET x x + 1
40 OUT x
50 OUT x
```

Output: `1 2 2`

example4.l11

```
10 SET x 1
20 OUT x
40 OUT x
```

Output: `1 1`

solution.l11 (Task 2)

```
10 SET x 1
20 OUT x
30 SET x x + 1
40^ OUT x
50 SET x 1
```

Output: `1 2 1`

solution.l11 (Task 4)

```
5 SET x 2
10v OUT x
15v SET x 10
20v OUT x
25^ SET x 5
30v SET x 3
35v OUT x
40^ SET x 7
45^ OUT x
```

Output: `2 10 3 7 5 10 7 3`

Execution model (L11).

- Two-pass execution.
- Pass 1: execute all lines once in ascending label order.
- Pass 2: execute only marked lines once more. The order is derived by:
- - taking the subsequence of marked lines (keeping relative order)
- - grouping consecutive identical markers into blocks
- - swapping each adjacent v-block followed by a ^-block
- Variables persist across both passes.
- STOP halts immediately.

Task 2 explanation.

1. Label 10: SET x 1 sets x = 1.
2. Label 20: OUT x prints 1. Output: 1.
3. Label 30: SET x x + 1 sets x = 2.
4. Label 40^: OUT x prints 2. Output: 1 2.
5. Label 50: SET x 1 sets x = 1.
6. Label 40^: OUT x prints 1. Output: 1 2 1.

Final output: `1 2 1`.

Task 4 explanation.

1. Label 5: SET x 2 sets x = 2.
2. Label 10v: OUT x prints 2. Output: 2.
3. Label 15v: SET x 10 sets x = 10.
4. Label 20v: OUT x prints 10. Output: 2 10.
5. Label 25^: SET x 5 sets x = 5.
6. Label 30v: SET x 3 sets x = 3.
7. Label 35v: OUT x prints 3. Output: 2 10 3.
8. Label 40^: SET x 7 sets x = 7.
9. Label 45^: OUT x prints 7. Output: 2 10 3 7.
10. Label 25^: SET x 5 sets x = 5.
11. Label 10v: OUT x prints 5. Output: 2 10 3 7 5.
12. Label 15v: SET x 10 sets x = 10.
13. Label 20v: OUT x prints 10. Output: 2 10 3 7 5 10.
14. Label 40^: SET x 7 sets x = 7.
15. Label 45^: OUT x prints 7. Output: 2 10 3 7 5 10 7.
16. Label 30v: SET x 3 sets x = 3.
17. Label 35v: OUT x prints 3. Output: 2 10 3 7 5 10 7 3.

Final output: `2 10 3 7 5 10 7 3`.

Figure 15: Task package example for L11: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L12)

README.md (Task 2)

```
TASK:
Output exactly: 9 9

Write your final solution to
     : solution.l12

Constraints:
- Only one variable may be
     used in the entire
     program
- Must use exactly 1 OUT in
     total
```

README.md (Task 4)

```
TASK:
Output exactly: 4 3 10 9 4 9
      3 10

Write your final solution to
     : solution.l12

Constraints:
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110, 120,
      130, 140, 150
- Only one variable may be
     used in the entire
     program
- Must use exactly 4 OUTs in
      total
```

example1.l12

```
10 SET x 0
20 UWR A
30 SET x x + 1
40 OUT x
50 UWR B
60 SET x x + 10
70 OUT x
80 NHO A
90 SET x 4
100 OUT x
```

Output: 1 11 12 4

example3.l12

```
10 SET x 2
20 OUT x
30 SET x x + 1
40 OUT x
```

Output: 2 3

example2.l12

```
10 UWR A
20 SET x 4
30 OUT x
40 SET x 2
50 OUT x
```

Output: 4 2

example4.l12

```
10 SET x 10
12 OUT x
```

Output: 10

solution.l12 (Task 2)

```
10 SET x 9
20 UWR A
30 OUT x
40 UWR B
50 NHO A
```

Output: 9 9

solution.l12 (Task 4)

```
10 UWR A
20 SET x 4
30 OUT x
40 UWR B
50 SET x 3
60 OUT x
70 SET x 10
80 OUT x
90 UWR C
100 SET x 9
110 OUT x
120 UWR D
130 NHO A
140 NHO C
150 NHO B
```

Output: 4 3 10 9 4 9 3 10

Execution model (L12).

- UWR sym records a ribbon for sym starting at the line immediately after the UWR.
- NHO sym replays from that ribbon up to (but not including) the next UWR in source order.
- After replay, execution resumes at the line after the NHO.
- NHO during replay is a runtime error.
- STOP halts immediately.

Task 2 explanation. *Special instructions:* NHO sym, UWR sym.

1. Label 10: SET x 9 sets x = 9.
2. Label 20: UWR A records a ribbon starting after this line.
3. Label 30: OUT x prints 9. Output: 9.
4. Label 40: UWR B records a ribbon starting after this line.
5. Label 50: NHO A replays the recorded ribbon for A.
6. Label 30: OUT x prints 9. Output: 9 9.

Final output: 9 9.

Task 4 explanation. *Special instructions:* NHO sym, UWR sym.

1. Label 10: UWR A records a ribbon starting after this line.
2. Label 20: SET x 4 sets x = 4.
3. Label 30: OUT x prints 4. Output: 4.
4. Label 40: UWR B records a ribbon starting after this line.
5. Label 50: SET x 3 sets x = 3.
6. Label 60: OUT x prints 3. Output: 4 3.
7. Label 70: SET x 10 sets x = 10.
8. Label 80: OUT x prints 10. Output: 4 3 10.
9. Label 90: UWR C records a ribbon starting after this line.
10. Label 100: SET x 9 sets x = 9.
11. Label 110: OUT x prints 9. Output: 4 3 10 9.
12. Label 120: UWR D records a ribbon starting after this line.
13. Label 130: NHO A replays the recorded ribbon for A.
14. Label 20: SET x 4 sets x = 4.
15. Label 30: OUT x prints 4. Output: 4 3 10 9 4.
16. Label 140: NHO C replays the recorded ribbon for C.
17. Label 100: SET x 9 sets x = 9.
18. Label 110: OUT x prints 9. Output: 4 3 10 9 4 9.
19. Label 150: NHO B replays the recorded ribbon for B.
20. Label 50: SET x 3 sets x = 3.
21. Label 60: OUT x prints 3. Output: 4 3 10 9 4 9 3.
22. Label 70: SET x 10 sets x = 10.
23. Label 80: OUT x prints 10. Output: 4 3 10 9 4 9 3 10.

Final output: 4 3 10 9 4 9 3 10.

Figure 16: Task package example for L12: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L13)

README.md (Task 2)

```
TASK:
Output exactly: 3 4

Write your final solution to
     : solution.l13

Constraints:
- Only one variable may be
     used in the entire
     program
- Must use exactly 4 OUTs in
      total
```

README.md (Task 4)

```
TASK:
Output exactly: 13 4 5 9 8 5
      9 8 8

Write your final solution to
     : solution.l13

Constraints:
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110
- Only one variable may be
     used in the entire
     program
- Must use exactly 4 OUTs in
      total
- Line 10 must be SET x 8
```

example1.l13

```
10{1} SET x 7
20{3} OUT x
```

Output: `7 7 7`

example2.l13

```
10{1} SET x 1
20{1} OUT x
30{1} SET x 2
40{1} OUT x
50{1} SET x 3
60{1} OUT x
```

Output: `1 2 3`

example3.l13

```
10{1} SET x 0
20{1} SET x 1
30{1} OUT x
40{1} SIZ 30 2
50{1} SET x 2
60{2} OUT x
```

Output: `1 2 2 2 2`

example4.l13

```
10{1} SET x 1
20{0} OUT x
```

Output: `(no output)`

solution.l13 (Task 2)

```
10{1} SET x 3
20{1} OUT x
30{0} OUT x
40{1} SET x 4
50{1} OUT x
60{0} OUT x
```

Output: `3 4`

solution.l13 (Task 4)

```
10{0} SET x 8
20{1} SET x 13
30{3} OUT x
40{1} SET x 4
50{1} OUT x
60{1} SET x 5
70{2} OUT x
80{1} SIZ 10 2
90{2} SET x 9
100{3} OUT x
110{1} SIZ 50 1
```

Output: `13 4 5 9 8 8 8 9 8 8`

Execution model (L13).

- Labels are sorted ascending.
- The interpreter repeatedly executes the next label with remaining quota in ascending order (wrapping).
- Each execution consumes 1 quota for that label.
- Program halts when all quotas are exhausted or STOP executes.

Task 2 explanation.

1. Label 10{0}: SET x 3 sets x = 3.
2. Label 20{0}: OUT x prints 3. Output: 3.
3. Label 40{0}: SET x 4 sets x = 4.
4. Label 50{0}: OUT x prints 4. Output: 3 4.

Final output: `3 4`.

Task 4 explanation. *Special instructions:* SIZ label n (adds n to that label's remaining quota; n must be ¿= 0).

1. Label 20{0}: SET x 13 sets x = 13.
2. Label 30{2}: OUT x prints 13. Output: 13.
3. Label 40{0}: SET x 4 sets x = 4.
4. Label 50{0}: OUT x prints 4. Output: 13 4.
5. Label 60{0}: SET x 5 sets x = 5.
6. Label 70{1}: OUT x prints 5. Output: 13 4 5.
7. Label 80{0}: SIZ 10 2 adds 2 quota to label 10.
8. Label 90{1}: SET x 9 sets x = 9.
9. Label 100{2}: OUT x prints 9. Output: 13 4 5 9.
10. Label 110{0}: SIZ 50 1 adds 1 quota to label 50.
11. Label 10{1}: SET x 8 sets x = 8.
12. Label 30{1}: OUT x prints 8. Output: 13 4 5 9 8.
13. Label 50{0}: OUT x prints 8. Output: 13 4 5 9 8 8.
14. Label 70{0}: OUT x prints 8. Output: 13 4 5 9 8 8 8.
15. Label 90{0}: SET x 9 sets x = 9.
16. Label 100{1}: OUT x prints 9. Output: 13 4 5 9 8 8 8 9.
17. Label 10{0}: SET x 8 sets x = 8.
18. Label 30{0}: OUT x prints 8. Output: 13 4 5 9 8 8 8 9 8.
19. Label 100{0}: OUT x prints 8. Output: 13 4 5 9 8 8 8 9 8 8.

Final output: `13 4 5 9 8 8 8 9 8 8`.

Figure 17: Task package example for L13: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L14)

README.md (Task 2)

```
TASK:
Output exactly: 3 3 3

Write your final solution to
     : solution.l14

Constraints:
- Only one variable may be
    used in the entire
    program
- Must use exactly 1 OUT in
    total
```

README.md (Task 4)

```
TASK:
Output exactly: 6 5 4 5 4 6

Write your final solution to
     : solution.l14

Constraints:
- Only one variable may be
    used in the entire
    program
- Your program must contain
    exactly these lines: 2,
     10, 20, 30, 40, 45, 50,
     70, 100, 120, 130, 140
- Must use exactly 3 OUTs in
     total
```

example1.l14

```
10 SET x 1
20 OUT x
30 SET x 2
40 OUT x
50 STOP
```

Output: `1 2`

example3.l14

```
10 SET x 1
20 AYZ 60
30 AYZ 70
40 OUT x
50 STOP
60 SET x 2
70 OUT x
```

Output: `1 1`

example5.l14

```
10 SET x 5
20 OUT x
30 STOP
```

Output: `5`

example2.l14

```
10 SET x 1
20 AYZ 50
30 OUT x
40 SET x 2
50 OUT x
60 STOP
```

Output: `1 2`

example4.l14

```
10 SET x 1
20 OUT x
30 AYZ 50
40 OUT x
50 SET x x + 1
60 OUT x
70 STOP
```

Output: `1 1 2`

solution.l14 (Task 2)

```
10 SET x 3
20 AYZ 60
30 AYZ 60
40 AYZ 60
50 STOP
60 OUT x
```

Output: `3 3 3`

solution.l14 (Task 4)

```
2 SET x 6
10 OUT x
20 SET x 5
30 OUT x
40 AYZ 10
45 AYZ 2
50 SET x 4
70 OUT x
100 AYZ 70
120 AYZ 50
130 AYZ 30
140 AYZ 20
```

Output: `6 5 4 5 4 6`

Execution model (L14).

- Labels execute in ascending order; each label executes at most once.
- AYZ ¡label¿ pushes a return label on a stack.
- When the main path ends, execution returns to pinned labels in LIFO order.
- STOP immediately checks the AYZ stack: if non-empty, jumps to the most recent pinned label; otherwise halts.

Task 2 explanation. *Special instructions:* AYZ label.

1. Label 10: SET x 3 sets x = 3.
2. Label 20: AYZ 60 pushes label 60 onto the return stack.
3. Label 30: AYZ 60 pushes label 60 onto the return stack.
4. Label 40: AYZ 60 pushes label 60 onto the return stack.
5. Label 50: STOP halts execution.
6. Label 60: OUT x prints 3. Output: 3.
7. Label 60: OUT x prints 3. Output: 3 3.
8. Label 60: OUT x prints 3. Output: 3 3 3.

Final output: `3 3 3`.

Task 4 explanation. *Special instructions:* AYZ label.

1. Label 2: SET x 6 sets x = 6.
2. Label 10: OUT x prints 6. Output: 6.
3. Label 20: SET x 5 sets x = 5.
4. Label 30: OUT x prints 5. Output: 6 5.
5. Label 40: AYZ 10 pushes label 10 onto the return stack.
6. Label 45: AYZ 2 pushes label 2 onto the return stack.
7. Label 50: SET x 4 sets x = 4.
8. Label 70: OUT x prints 4. Output: 6 5 4.
9. Label 100: AYZ 70 pushes label 70 onto the return stack.
10. Label 120: AYZ 50 pushes label 50 onto the return stack.
11. Label 130: AYZ 30 pushes label 30 onto the return stack.
12. Label 140: AYZ 20 pushes label 20 onto the return stack.
13. Label 20: SET x 5 sets x = 5.
14. Label 30: OUT x prints 5. Output: 6 5 4 5.
15. Label 50: SET x 4 sets x = 4.
16. Label 70: OUT x prints 4. Output: 6 5 4 5 4.
17. Label 2: SET x 6 sets x = 6.
18. Label 10: OUT x prints 6. Output: 6 5 4 5 4 6.

Final output: `6 5 4 5 4 6`.

Figure 18: Task package example for L14: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L15)

README.md (Task 2)

```
TASK:
Output exactly: 4 7

Write your final solution to
     : solution.l15

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 55, 60
- Line 40 must be STOP
```

README.md (Task 4)

```
TASK:
Output exactly: 3 8 10

Write your final solution to
     : solution.l15

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60, 70,
      80, 90, 100, 110, 120,
      130, 140, 150
- Line 30 must be STOP
- Line 70 must be STOP
- Line 100 must be STOP
```

example1.l15

```
10 SET x 1
20 FWN 40
30 OUT x
40 OUT x
50 SET x x + 5
60 KIE
70 OUT x
```

Output: `1 6 6`

example3.l15

```
10 SET x 10
20 FWN 50
30 SET x 2
50 SET x 3
70 KIE
90 OUT x
```

Output: `2`

example2.l15

```
10 SET x 3
20 FWN 30
30 OUT x
40 KIE
```

Output: `3`

example4.l15

```
10 SET x 2
20 OUT x
30 SET x x + 1
40 STOP
50 OUT x
```

Output: `2`

solution.l15 (Task 2)

```
10 SET x 4
20 OUT x
30 FWN 50
40 STOP
50 SET x 7
55 OUT x
60 KIE
```

Output: `4 7`

solution.l15 (Task 4)

```
10 SET x 3
20 FWN 40
30 STOP
40 OUT x
50 SET x 8
60 FWN 80
70 STOP
80 OUT x
90 FWN 110
100 STOP
110 SET x 10
120 OUT x
130 KIE
140 KIE
150 KIE
```

Output: `3 8 10`

Execution model (L15).

- Labels are sorted ascending and are consumptive (each executes at most once).
- Default flow falls through to the next higher unexecuted label.
- FWN ¡label¿ begins a block: push the current label and jump to the target label.
- KIE ends the block: resume at the next higher unexecuted label after the call-site.
- STOP halts immediately.

Task 2 explanation. *Special instructions:* FWN label, KIE.

1. Label 10: SET x 4 sets x = 4.
2. Label 20: OUT x prints 4. Output: 4.
3. Label 30: FWN 50 jumps to label 50 and pushes a return point.
4. Label 50: SET x 7 sets x = 7.
5. Label 55: OUT x prints 7. Output: 4 7.
6. Label 60: KIE ends the block and returns to the instruction after the call site.
7. Label 40: STOP halts execution.

Final output: `4 7`.

Task 4 explanation. *Special instructions:* FWN label, KIE.

1. Label 10: SET x 3 sets x = 3.
2. Label 20: FWN 40 jumps to label 40 and pushes a return point.
3. Label 40: OUT x prints 3. Output: 3.
4. Label 50: SET x 8 sets x = 8.
5. Label 60: FWN 80 jumps to label 80 and pushes a return point.
6. Label 80: OUT x prints 8. Output: 3 8.
7. Label 90: FWN 110 jumps to label 110 and pushes a return point.
8. Label 110: SET x 10 sets x = 10.
9. Label 120: OUT x prints 10. Output: 3 8 10.
10. Label 130: KIE ends the block and returns to the instruction after the call site.
11. Label 100: STOP halts execution.

Final output: `3 8 10`.

Figure 19: Task package example for L15: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L16)

README.md (Task 2)

```
TASK:
Output exactly: 5 5 5

Write your final solution to
     : solution.l16

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50
- Line 30 must be STOP
```

README.md (Task 4)

```
TASK:
Output exactly: 4 7 7 10 5 5
      13

Write your final solution to
     : solution.l16

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these lines: 10,
      15, 20, 25, 30, 40, 45,
      50, 55, 60, 65, 70
- Line 20 must be SET x x +
     3
- Line 45 must be SET x 13
```

example1.l16

```
10 SET x 1
20 OUT x
40! SET x x + 10
45 STOP
50 OUT x
60 OUT x
```

Output: `1 11 11`

example2.l16

```
10 SET x 1
30! SET x x + 1
40 OUT x
50 SET x x + 10
55 OUT x
60 OUT x
```

Output: `2 2 12`

example3.l16

```
10 SET x 7
40 OUT x
50 OUT x
60 STOP
```

Output: `7 7`

example4.l16

```
10 SET x 1
20 SET x x + 10
45 OUT x
50 SET x x - 3
55 OUT x
```

Output: `11 8`

solution.l16 (Task 2)

```
10 SET x 5
20! OUT x
30 STOP
40 OUT x
50 OUT x
```

Output: `5 5 5`

solution.l16 (Task 4)

```
10 SET x 4
15 OUT x
20! SET x x + 3
25 OUT x
30! SET x 5
40 OUT x
45 SET x 13
50 OUT x
55 OUT x
60! SET x 10
65 OUT x
70 OUT x
```

Output: `4 7 7 10 5 5 13`

Execution model (L16).

- Start at the smallest label.
- Each label executes at most once.
- A global direction controls which label executes next:
- - forward: smallest remaining label ¿ current (wraps to smallest)
- - backward: largest remaining label ¡ current (wraps to largest)
- If the executed line has !, the direction flips for the next step.
- STOP halts immediately.

Task 2 explanation.

1. Label 10: SET x 5 sets x = 5.
2. Label 20!: OUT x prints 5. Output: 5.
3. Label 50: OUT x prints 5. Output: 5 5.
4. Label 40: OUT x prints 5. Output: 5 5 5.
5. Label 30: STOP halts execution.

Final output: `5 5 5`.

Task 4 explanation.

1. Label 10: SET x 4 sets x = 4.
2. Label 15: OUT x prints 4. Output: 4.
3. Label 20!: SET x x + 3 sets x = 7.
4. Label 70: OUT x prints 7. Output: 4 7.
5. Label 65: OUT x prints 7. Output: 4 7 7.
6. Label 60!: SET x 10 sets x = 10.
7. Label 25: OUT x prints 10. Output: 4 7 7 10.
8. Label 30!: SET x 5 sets x = 5.
9. Label 55: OUT x prints 5. Output: 4 7 7 10 5.
10. Label 50: OUT x prints 5. Output: 4 7 7 10 5 5.
11. Label 45: SET x 13 sets x = 13.
12. Label 40: OUT x prints 13. Output: 4 7 7 10 5 5 13.

Final output: `4 7 7 10 5 5 13`.

Figure 20: Task package example for L16: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619

Task package example (L17)

README.md (Task 2)

```
TASK:
Output exactly: 4 4

Write your final solution to
     : solution.l17

Constraints:
- Your program must contain
     exactly these lines: 10,
      20, 30, 40, 50, 60
- Line 20 must be SET x 9
- Line 60 must be OUT x
- Only one variable may be
     used in the entire
     program
```

README.md (Task 4)

```
TASK:
Output exactly: 5 7 5

Write your final solution to
     : solution.l17

Constraints:
- Your program must contain
     exactly these labels:
     10, 20, 30, 40, 50, 60,
      70, 80, 90, 100, 110,
     120
- Labels 70 must be STOP
- Label 120 must be STOP
- Must use exactly 2 SETs in
      total
- Only one variable may be
     used in the entire
     program
```

example1.l17

```
10 SET x 1
20 SET x 0
30 OUT x
40 SET x 0
```

Output: `1`

example2.l17

```
10 SET x 5
20 SET x 0
30 OUT x
40 SET x 0
```

Output: `5`

example3.l17

```
10 AMO
20 SET x 0
30 SET x 0
40 SET x 1
50 SET x 0
60 OUT x
```

Output: `1 1 1`

example4.l17

```
10 STOP
20 SET x 1
30 OUT x
40 SET x 0
```

Output: `(no output)`

example5.l17

```
10 SET x 1
20 SET x 0
30 OUT x
40 SET x 0
50 SET x x + 1
60 OUT x
```

Output: `1`

solution.l17 (Task 2)

```
10 SET x 4
20 SET x 9
30 OUT x
40 STOP
50 OUT x
60 OUT x
```

Output: `4 4`

solution.l17 (Task 4)

```
10 SET x 5
20 STOP
30 OUT x
40 STOP
50 AMO
60 OUT x
70 STOP
80 SET x 7
90 OUT x
100 AMO
110 OUT x
120 STOP
```

Output: `5 7 5`

Execution model (L17).

- All labels are sorted into a bracket.
- Each match compares adjacent labels; the winner executes and advances.
- Mode '0' means the lower label wins; mode '1' means the higher label wins.
- 'AMO' toggles the mode, but the change applies to the **next** match.
- If there is an odd label in a round, it gets a bye and advances without executing.
- When one label remains, it executes once and the program halts.
- 'STOP' halts immediately.

Task 2 explanation.

1. Label 10: SET x 4 sets x = 4.
2. Label 30: OUT x prints 4. Output: 4.
3. Label 50: OUT x prints 4. Output: 4 4.
4. Label 10: SET x 4 sets x = 4.
5. Label 10: SET x 4 sets x = 4.

Final output: `4 4`.

Task 4 explanation. *Special instructions:* AMO.

1. Label 10: SET x 5 sets x = 5.
2. Label 30: OUT x prints 5. Output: 5.
3. Label 50: AMO toggles match mode for the next bracket match.
4. Label 80: SET x 7 sets x = 7.
5. Label 100: AMO toggles match mode for the next bracket match.
6. Label 110: OUT x prints 7. Output: 5 7.
7. Label 10: SET x 5 sets x = 5.
8. Label 50: AMO toggles match mode for the next bracket match.
9. Label 110: OUT x prints 5. Output: 5 7 5.
10. Label 50: AMO toggles match mode for the next bracket match.
11. Label 50: AMO toggles match mode for the next bracket match.

Final output: `5 7 5`.

Figure 21: Task package example for L17: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L18)

README.md (Task 2)

```
TASK:
Output exactly: 3 6

Write your final solution to
     : solution.l18

Constraints:
- Only one variable may be
     used in the entire
     program
```

README.md (Task 4)

```
TASK:
Output exactly: 3 6 6 6 15
     15

Write your final solution to
     : solution.l18

Constraints:
- Only one variable may be
     used in the entire
     program
- Your program must contain
     exactly these labels: 1,
      2, 3, 4, 5, 6, 7, 8, 9,
      10, 11, 12
- Label 4 must be STOP
- Label 8 must be STOP
- Label 9 must be STOP
```

example1.l18

```
1[*] SET x 1
2[1] SET x x + 1
3[!] SET x x + 10
4[3] OUT x
5[*,*,2] SET x x + 1
6[5] OUT x
```

Output: 3

example3.l18

```
1[*] SET c 1
2[1] SET c c + 1
3[2] SET c c + 1
4[3] SET c c + 1
5[4] SET c c + 1
6[*,*,*,2] OUT c
7[6] STOP
```

Output: 5

example5.l18

```
1[*] SET w 3
2[1] OUT w
3[2] SET w w - 1
4[3] OUT w
5[4] STOP
6[5] OUT w
```

Output: 3 2

example2.l18

```
1[*] SET b 5
2[1] OUT b
3[!] SET b b + 5
4[3] OUT b
5[*,*,2] SET b b + 1
6[5] OUT b
7[6] STOP
```

Output: 5 6

example4.l18

```
1[*] SET z 1
2[1] SET z z + 2
3[!] OUT z
4[3] SET z z + 5
5[*,*,2] OUT z
```

Output: 3

solution.l18 (Task 2)

```
1[*] SET x 3
2[1] OUT x
3[2] SET x x + 3
4[3] OUT x
5[4] STOP
```

Output: 3 6

solution.l18 (Task 4)

```
1[*] SET x 1
2[1] SET x x + 2
3[!] SET x x + 5
4[3] STOP
5[*,*,2] OUT x
6[5] SET x x + 3
7[6] OUT x
8[*,*,*,4] STOP
9[*,7] OUT x
10[9] OUT x
11[10] SET x x + 9
12[11] OUT x
13[*,*,*,*,8] STOP
14[*,12] OUT x
```

Output: 3 6 6 6 15 15

Execution model (L18).

- Lines execute in ascending label order.
- Every line must include a guard in brackets.
- The first line must use '[*]' and always executes.
- A guard '[k]' means: execute only if label 'k' executed.
- A guard '[* ,k]' (one '*') means: execute only if the label two positions
- earlier executed. More '*'s increase the distance by 1 each time.
- Exactly one line may use '[!]', which never executes.
- 'STOP' halts immediately; otherwise execution ends after the last line.

Task 2 explanation.

1. Label 1[*]: SET x 3 sets x = 3.
   Note: guard [*] always executes.
2. Label 2[1]: OUT x prints 3. Output: 3.
   Note: guard [1] satisfied.
3. Label 3[2]: SET x x + 3 sets x = 6.
   Note: guard [2] satisfied.
4. Label 4[3]: OUT x prints 6. Output: 3 6.
   Note: guard [3] satisfied.
5. Label 5[4]: STOP halts execution.
   Note: guard [4] satisfied.

Final output: 3 6.

Task 4 explanation.

1. Label 1[*]: SET x 1 sets x = 1.
   Note: guard [*] always executes.
2. Label 2[1]: SET x x + 2 sets x = 3.
   Note: guard [1] satisfied.
3. Label 3[!]: SET x x + 5 sets x = 8.
   Note: guard [!] never executes.
4. Label 4[3]: STOP halts execution.
   Note: guard [3] satisfied.
5. Label 5[*,*,2]: OUT x prints 8. Output: 8.
   Note: guard [,*,2] satisfied.*
6. Label 6[5]: SET x x + 3 sets x = 11.
   Note: guard [5] satisfied.
7. Label 7[6]: OUT x prints 11. Output: 8 11.
   Note: guard [6] satisfied.
8. Label 8[*,*,*,4]: STOP halts execution.
   Note: guard [,*,*,4] satisfied.*
9. Label 9[*,7]: OUT x prints 11. Output: 8 11 11.
   Note: guard [,7] satisfied.*
10. Label 10[9]: OUT x prints 11. Output: 8 11 11 11.
    Note: guard [9] satisfied.
11. Label 11[10]: SET x x + 9 sets x = 20.
    Note: guard [10] satisfied.
12. Label 12[11]: OUT x prints 20. Output: 8 11 11 11 20.
    Note: guard [11] satisfied.
13. Label 13[*,*,*,*,8]: STOP halts execution.
    Note: guard [,*,*,*,8] satisfied.*
14. Label 14[*,12]: OUT x prints 20. Output: 8 11 11 11 20 20.
    Note: guard [,12] satisfied.*

Final output: 3 6 6 6 15 15.

Figure 22: Task package example for L18: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.

Task package example (L19)

README.md (Task 2)

```
TASK:
Output exactly: 10 15

Write your final solution to
     : solution.l19

Constraints:
- Only one variable may be
    used in the entire
    program
- Your program must contain
    exactly these labels: 1,
     2, 3, 4, 5
- Must use exactly 2 SETs in
     total
- Line 1 must be SET x 0
```

README.md (Task 4)

```
TASK:
Output exactly: 2 3 4 5 6 6
     6 7 8

Write your final solution to
     : solution.l19

Constraints:
- Only one variable may be
    used in the entire
    program
- Your program must contain
    exactly these labels: 1,
     2, 3, 4, 5, 6, 7, 8, 9,
     10, 11
- Label 3 is SET x x - 1
- Label 9 must be STOP
- Must use exactly 4 SETs in
     total
```

example1.l19

```
1{3} SET x 10
2 SET x 1
3 SET x x + 1
4 OUT x
5 STOP
```

Output: 10

example3.l19

```
1 SET a 1
2 SET b 2
3 SET c 3
4 OUT c
5 OUT a
6 OUT b
7 STOP
```

Output: 3 1 2

example5.l19

```
1 SET x 5
2{3} SET x 10
3 OUT x
4 XYZ 2
5 SET x x + 1
6 ABC 2
7 SET x x + 1
8 OUT x
9 STOP
```

Output: 10 12

example2.l19

```
1 SET x 1
2{3} SET x x + 10
3 OUT x
4 XYZ 2
5 SET x x + 1
6 OUT x
7 STOP
```

Output: 11 22

example4.l19

```
1{4} SET x 3
2 SET x 0
3 SET x x + 1
4 OUT x
5 STOP
```

Output: 1

solution.l19 (Task 2)

```
1 SET x 0
2{2} SET x x + 5
3 OUT x
4 OUT x
5 STOP
```

Output: 10 15

solution.l19 (Task 4)

```
1 SET x 2
2 OUT x
3 SET x x - 1
4 SET x x - 1
5 XYZ 10
6 OUT x
7 ABC 10
8 OUT x
9 STOP
10{2} SET x x + 1
11{3} OUT x
```

Output: 2 3 4 5 6 6 6 7 8

Execution model (L19).

- Labels are sorted and normally execute sequentially.
- A label may include ‘{T}‘ to set its countdown timer.
- Every step, all active timers decrement by 1.
- If any timer reaches 0, that label executes first (smallest label wins ties).
- After a timer-triggered execution, its timer resets to the original value.
- ‘XYZ k‘ disables label ‘k‘’s timer; ‘ABC k‘ re-enables it.
- ‘STOP‘ halts immediately.

Task 2 explanation.

1. Label 1: SET x 0 sets x = 0.
2. Label 2{2}: SET x x + 5 sets x = 5.
3. Label 2{2}: SET x x + 5 sets x = 10.
   Note: timer triggered (interrupts normal order).
4. Label 3: OUT x prints 10. Output: 10.
5. Label 2{2}: SET x x + 5 sets x = 15.
   Note: timer triggered (interrupts normal order).
6. Label 4: OUT x prints 15. Output: 10 15.
7. Label 2{2}: SET x x + 5 sets x = 20.
   Note: timer triggered (interrupts normal order).
8. Label 5: STOP halts execution.

Final output: 10 15.

Task 4 explanation. *Special instructions:* ABC, XYZ.

1. Label 1: SET x 2 sets x = 2.
2. Label 2: OUT x prints 2. Output: 2.
3. Label 10{2}: SET x x + 1 sets x = 3.
   Note: timer triggered (interrupts normal order).
4. Label 11{3}: OUT x prints 3. Output: 2 3.
   Note: timer triggered (interrupts normal order).
5. Label 10{2}: SET x x + 1 sets x = 4.
   Note: timer triggered (interrupts normal order).
6. Label 3: SET x x - 1 sets x = 3.
7. Label 10{2}: SET x x + 1 sets x = 4.
   Note: timer triggered (interrupts normal order).
8. Label 11{3}: OUT x prints 4. Output: 2 3 4.
   Note: timer triggered (interrupts normal order).
9. Label 10{2}: SET x x + 1 sets x = 5.
   Note: timer triggered (interrupts normal order).
10. Label 4: SET x x - 1 sets x = 4.
11. Label 10{2}: SET x x + 1 sets x = 5.
    Note: timer triggered (interrupts normal order).
12. Label 11{3}: OUT x prints 5. Output: 2 3 4 5.
    Note: timer triggered (interrupts normal order).
13. Label 10{2}: SET x x + 1 sets x = 6.
    Note: timer triggered (interrupts normal order).
14. Label 5: XYZ 10 disables the timer on label 10.
15. Label 11{3}: OUT x prints 6. Output: 2 3 4 5 6.
    Note: timer triggered (interrupts normal order).
16. Label 6: OUT x prints 6. Output: 2 3 4 5 6 6.
17. Label 7: ABC 10 re-enables the timer on label 10.
18. Label 11{3}: OUT x prints 6. Output: 2 3 4 5 6 6 6.
    Note: timer triggered (interrupts normal order).
19. Label 10{2}: SET x x + 1 sets x = 7.
    Note: timer triggered (interrupts normal order).
20. Label 8: OUT x prints 7. Output: 2 3 4 5 6 6 6 7.
21. Label 10{2}: SET x x + 1 sets x = 8.
    Note: timer triggered (interrupts normal order).
22. Label 11{3}: OUT x prints 8. Output: 2 3 4 5 6 6 6 7 8.
    Note: timer triggered (interrupts normal order).
23. Label 10{2}: SET x x + 1 sets x = 9.
    Note: timer triggered (interrupts normal order).
24. Label 9: STOP halts execution.

Final output: 2 3 4 5 6 6 6 7 8.

Figure 23: Task package example for L19: README constraints (Task 2 and Task 4), example programs, candidate submissions, and explanations.