DO LLMS UNDERSTAND PROGRAM SEMANTICS?

Anonymous authors
Paper under double-blind review

ABSTRACT

LLMs are remarkably good at writing code. But do they understand what code means—i.e., its semantics—or are they just exploiting syntactic patterns? We introduce a family of benchmarks that isolates reasoning about program semantics. Each task provides three of four components—a program, its inputs, its outputs, and a semantic interpretation—and asks the model to infer the fourth. All problems share a fixed (syntactic) language: programs are drawn from the same grammar, but are equipped with multiple semantic interpretations. This approach isolates semantic reasoning because models cannot rely on syntactic differences to distinguish behaviors. We construct these tasks via an automated procedure that samples programs, interpretations, and behaviors, and filters out degenerate cases to ensure non-trivial dependence on both inputs and semantics. Across a range of models and task types, we find that performance varies significantly between models, and falls off as the language's semantics get more difficult—revealing a gap between coding ability and semantic understanding. Moreover, performance on our benchmarks correlates with success on program synthesis, suggesting that this gap reflects a limitation that transfers to practical settings.

1 INTRODUCTION

Do large language models (LLMs) understand program semantics, or do they rely primarily on syntactic and statistical cues? Despite strong performance on programming tasks (Chen et al., 2021; Nijkamp et al., 2023; Roziere et al., 2023), it remains unclear whether models reason about program behavior or simply exploit surface-level syntactic patterns. A key challenge is that, in most benchmarks, semantic differences are entangled with syntactic ones. When program behavior changes, the syntax typically changes alongside it, which could allow models to succeed without reasoning about semantics. As a result, benchmark performance alone does not reliably measure semantic understanding.

We propose a simple evaluation principle: given a customizable language with a fixed syntax, a model demonstrates semantic understanding if it can infer a missing component of a program specification from the others. In addition to a specification of the language's syntax, each problem consists of four components: a specification of the language's semantics, inputs, outputs, and a target program in the language. We withhold one of the latter four components, which the model must infer, given the other three. For example, the model may be tasked with predicting the outputs given the language, inputs, and a program. This formulation directly targets semantic reasoning: our evaluation framework can be viewed as isolating a single variable, semantics, while holding syntax fixed. By constructing languages over different semantic interpretations and evaluating models across them, we directly test whether model behavior tracks semantic variation rather than syntactic form.

The remaining challenge is constructing problem instances where such reasoning is necessary. We use Semantics-guided Synthesis (SemGuS) (Kim et al., 2021) specifications as our vehicle of choice, which lets us provide customizable syntax and semantics for our problem languages, as well as use pre-existing tooling (Johnson et al., 2024) for checking the model's work. We develop an automated generation procedure that samples semantics, programs, and input/output behaviors under a fixed syntax.

Using this framework, we evaluate state-of-the-art LLMs across task types and levels of semantic complexity. We further examine whether performance on these tasks correlates with success on real-world SemGuS synthesis problems, testing whether the benchmark captures the semantic capabilities required for these problems.

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

This paper makes the following contributions:

- **Task.** We introduce a family of inference tasks in which models must infer a held-out component from {semantics, program, inputs, outputs} given the other three, requiring reasoning about program semantics.
- **Dataset.** We construct a fully automated benchmark generator that samples semantically diverse program-synthesis problems under a fixed grammar, with controllable semantic complexity.
- **Analysis.** We find that performance varies significantly between models, falling as the expressions used to describe the language's semantics grow in size. We further demonstrate that performance on our benchmark suite correlates with performance on existing program-synthesis tasks.

2 Related Work

LLMs for code are often evaluated through code-generation benchmarks, where a model is given a natural-language specification and asked to produce a program that passes tests. HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021) established this style of functional-correctness evaluation, while broader benchmarks such as CodeXGLUE (Lu et al., 2021), CodeScope (Yan et al., 2024), LiveCodeBench (Jain et al., 2025), and SWE-bench (Jimenez et al., 2024) extend evaluation to more languages, tasks, contamination-aware settings, and real-world software-engineering problems. These benchmarks measure whether models can write useful code, but they generally operate in existing languages with fixed, familiar semantics, making it difficult to separate semantic reasoning from memorized syntax, idioms, and benchmark-specific patterns.

More closely related are benchmarks that test code execution and semantic reasoning. CRUXEval (Gu et al., 2024) asks models to predict inputs or outputs of Python functions, and CRUXEval-X (Xu et al., 2025) extends this setting across programming languages. CodeI/O (Li et al., 2025) similarly uses input-output prediction to study procedural reasoning in code, while NExT (Ni et al., 2024) trains models with execution traces and rationales. Other work targets semantic properties more directly: FormalBench (Le-Cong et al., 2025) evaluates formal-specification inference, EquiBench (Wei et al., 2025a) evaluates program equivalence, CodeCrash (Lam et al., 2025) stress-tests code reasoning under perturbations, and Agentic Code Reasoning (Ugare & Chandra, 2026) studies whether LLM agents can perform semantic code analysis without execution. These works show that LLMs often struggle with program behavior beyond surface form; however, they all evaluate well-known programming languages whose semantics are fixed and overrepresented in training data.

Our work differs by making the semantic interpretation itself part of the benchmark instance: models infer one of four components (semantics, program, inputs, outputs) from the other three. With syntax fixed and semantics varying, we isolate whether predictions track the provided semantics rather than syntactic regularities. PLSemanticsBench (Thimmaiah et al., 2026) similarly mutates the semantics of an imperative language to test whether LLMs can act as interpreters, but its mutations only swap or obfuscate operator names: each operator still denotes a familiar primitive (this corresponds to a meta-language expression size of 3 in our work). We instead generate fresh semantics of increasing complexity and evaluate more task types, such as program synthesis. SATBench (Wei et al., 2025b) is methodologically related, using solver-validated generation to evaluate logical reasoning, but our formal objects differ.

3 Overview

A natural way to evaluate someone's understanding of the semantics of a programming language is to present them with a program and an input and ask them to predict the output, or to provide several input/output examples and ask for a program consistent with those examples. When testing the understanding of programming-language concepts (e.g., call-by-name vs. call-by-value in parameter passing), one could even hide part of the language definition and ask the student to infer a rule consistent with observed behavior of the "mystery language" (Pombrio et al., 2017). In all of these cases, understanding is evaluated behaviorally: a person demonstrates understanding of the language to the extent that their answers are consistent with the language's semantics. This behavioral criterion is natural for programming languages because semantics itself defines observable behavior—it specifies how programs transform inputs into outputs. We use the same principle to evaluate whether LLM predictions remain consistent with program semantics across controlled semantic variations.

Table 1: Semantic-reasoning tasks obtained by withholding one component of $y = [\![s]\!]_\sigma(x)$.

Task	**Input** (x_1, x_2)	**Output** y	**Semantics** σ	**Program** s
Predict Input	?	5	Sem_L	`A;B;C`
Predict Output	$(5, 2)$	?	Sem_L	`A;B;C`
Predict Semantics	$(5, 2)$	5	?	`A;B;C`
Predict Program	$(5, 2)$	5	Sem_L	?

One Syntax, Many Semantics. Consider a small register language with registers (r_0, r_1, r_2), where programs take as input two values x_1 and x_2 and produce as output a value y. Programs are sequences of one or more statements drawn from the fixed syntax

$$\mathit{Statement} ::= \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D}.$$

Example programs are "`A;B;C`" and "`B;B`." Semantically, r_0 is initialized to 0, while r_1 and r_2 are initialized with the program inputs x_1 and x_2, respectively. Each statement updates one or more registers using arithmetic expressions over the current register values. After execution finishes, the value stored in r_0 is treated as the program output y.

The key feature of our setting is that the syntax exposes only the statement labels `A`, `B`, `C`, and `D`, while their meanings are allowed to vary across semantic interpretations. For example, consider the following two interpretations that assign different arithmetic expressions to the statements:

$$(Sem_L)\quad \begin{array}{ll} \texttt{A}: & r_0 := r_1 + r_2 \\ \texttt{B}: & r_1 := r_0 \\ \texttt{C}: & r_0 := r_0 - r_2 \\ \texttt{D}: & r_2 := r_1 \end{array} \qquad (Sem_R)\quad \begin{array}{ll} \texttt{A}: & r_1 := r_0 - r_1 \\ \texttt{B}: & r_2 := r_2 + r_1 \\ \texttt{C}: & r_0 := r_0 - r_2 \\ \texttt{D}: & r_0 := r_1 - r_2 \end{array}$$

Now consider the program "`A;B;C`." When the input is $(x_1, x_2) = (5, 2)$, this program outputs 5 under Sem_L, whereas it outputs 3 under Sem_R. Although the syntax of the program is identical in both cases, its behavior changes because the semantic interpretation changes. Computing the output of a program in this language on a given input therefore requires reasoning about the provided semantics rather than relying on syntax alone.

Beyond Computing Outputs. We can also ask inverse questions: for example, given the semantic interpretation Sem_L and the program "`A;B;C`," what input produces the output 5? Solving this task requires reasoning about the semantics in reverse to infer a valid input. Similarly, we can ask for a program consistent with a set of inputs, outputs, and semantics (program synthesis), or omit part of the semantic interpretation and ask for a completion consistent with the observed program behavior. Note that for these three problems there can be more than one possible answer—e.g., there could be multiple programs consistent with the same examples.

These task types, summarized in Table 1, illustrate the central idea of our benchmark. Each task contains four components—an input x, an output y, a semantic interpretation σ (which defines how programs map inputs to outputs), and a program s—related by the semantic equation $y = [\![s]\!]_\sigma(x)$, where $[\![s]\!]_\sigma$ denotes the semantics of s under interpretation σ. We reveal three components and ask the model to infer the fourth. The key control is that every benchmark instance uses the same fixed syntax. In the example above, the statements `A`, `B`, `C`, and `D` are always present in the language, but their meanings vary across semantic interpretations. As a result, a model cannot solve the task merely by associating the syntax of statements with memorized behaviors; it must instead reason using the semantics provided in the benchmark instance. This fixed-syntax, varying-semantics design operationalizes semantic reasoning behaviorally: we evaluate whether a model's predictions remain consistent with the provided semantics across many controlled instances.

Generating Many Semantics. A meaningful evaluation requires the ability to generate many languages with distinct semantic interpretations that are unlikely to have appeared in an LLM's training data. Using a small collection of existing programming languages would make this difficult: their semantics may already be familiar to the model, and their syntax often carries strong behavioral cues through identifier names, idioms, and libraries. Instead, we seek to construct families of synthetic languages in which the syntax is held fixed while the semantics varies across instances.

We draw on ideas from the programming-languages and program-synthesis literature, where, to accommodate custom languages, synthesis tasks are often specified by separating a language's syntax from its semantics, and the semantics is expressed in a *standardized logical form*. This specification mechanism is particularly useful because it enables automated reasoning about programs and benchmark instances. We can execute programs symbolically, validate candidate solutions, and automatically filter trivial or degenerate tasks. Concretely, we use Semantics-guided Synthesis (SemGuS) (Kim et al., 2021) as the framework for constructing these languages. A SemGuS specification separates a language into two components: a grammar describing the program syntax, and a set of constrained Horn clauses (CHCs) (i.e., logical rules) defining the semantics of that syntax. This separation closely matches our evaluation setting. We can sample a semantic interpretation, sample a program from the fixed syntax, execute it on inputs to obtain outputs, and then construct semantic-reasoning tasks by withholding one component of the resulting tuple (σ, s, x, y). Because semantic interpretations are sampled procedurally, models cannot rely on memorized language semantics.

While we use the SemGuS representation to conveniently generate diverse benchmarks, we render benchmark instances in two concrete formats: (1) the native SemGuS syntax (Johnson et al., 2024), exposing the specification directly in SMT-LIB format (Barrett et al., 2016), a standardized representation of logical formulas used in SMT solvers; and (2) an equivalent C-like syntax familiar to most code models. Because both representations encode the same semantics and tasks, this comparison lets us distinguish failures due to unfamiliar SemGuS syntax from failures in reasoning about semantics.

Finally, the SemGuS project provides an existing suite of synthesis benchmarks that allows us to test whether the semantic-reasoning difficulties isolated by our benchmark correspond to difficulties that arise in realistic synthesis problems. Therefore, in addition to evaluating models on our generated semantic-reasoning tasks, we evaluate them on the 277 standard SemGuS synthesis benchmarks (Johnson et al., 2024). This experiment lets us investigate the following question: Is semantic-reasoning difficulty predictive of synthesis difficulty across models? If models that struggle with increasingly complex semantic reasoning in our controlled setting also struggle on standard synthesis benchmarks, it would suggest that our benchmark isolates the semantic capabilities required for these problems.

4 Task-Generation Pipeline

As discussed in Section 3, we begin from a fixed program syntax and generate benchmark instances by sampling semantic interpretations, programs, and input/output examples. Each such instance defines four reasoning problems obtained by withholding one component of the tuple (σ, s, x, y), where s is a program, σ is a semantic interpretation, x is an input, and $y = [\![s]\!]_\sigma(x)$ is the output produced by executing s on x under σ.

Programs are defined as finite sequences of statements from a set $\{A, B, \ldots, N\}$. The syntax of the language therefore remains identical across all benchmark instances. Semantic variation is introduced entirely through the interpretation σ, which assigns operational meaning to each production. Concretely, for each statement, σ specifies a transformation over a fixed collection of registers $(r_0, r_1, \ldots, r_k)$. The register r_0 serves as the output register: after program execution terminates, the final value of r_0 is returned as the program output. At the start of execution, input values are loaded into $r_1, r_2, \ldots, r_j$, $j \leq k$, where j is the number of inputs, while all remaining registers are initialized to 0. Executing a program applies these transformations sequentially to produce the final output. We describe how σ is concretely defined in detail in Section 4.2.

Given a sampled tuple (σ, s, x, y), we construct evaluation tasks by withholding one component and asking the model to infer it from the remaining three. This approach yields four task families:

- **Predict Output**: infer y, given σ, s, and x;
- **Predict Input**: infer a valid x', given σ, s, and y, such that $y = [\![s]\!]_\sigma(x')$;
- **Predict Program (Program Synthesis)**: infer a valid program s', given σ, x, and y, such that $y = [\![s']\!]_\sigma(x)$;
- **Predict Semantics (Semantic Completion)**: infer a valid missing semantic rule σ'_p, given an incomplete set of semantic rules $\sigma^* = \sigma \setminus \sigma_p$, together with s, x, and y, such that $y = [\![s]\!]_{\sigma^* \cup \sigma'_p}(x)$.

Only Predict Output is guaranteed to have a unique correct answer. The remaining tasks may admit multiple valid solutions, and candidate answers are accepted whenever they satisfy the required semantic relation.

The remainder of this section describes how benchmark instances are generated. We first focus on loop-free languages, where difficulty arises purely from semantic variation in the meanings assigned to productions (Sections 4.1 to 4.4). We then describe an extension with loop constructs, which introduces an additional source of structural reasoning difficulty (Section 4.5).

4.1 Benchmark Construction as Sampling

We formulate benchmark generation as sampling from a family of program semantics, programs, and inputs. Specifically, each benchmark tuple (σ, s, x, y) is sampled by drawing the first three elements from distributions $\sigma \sim \mathcal{D}_{sem}$, $s \sim \mathcal{D}_{prog}$, and $x \sim \mathcal{D}_{in}$, and the output y is computed by evaluating $y = [\![s]\!]_\sigma(x)$. Programs with identical surface structure may therefore exhibit different behavior under different sampled semantics—our pipeline can generate syntactically identical but **semantically distinct languages**.

Concretely, $\mathcal{D}_{sem}$ is a distribution over semantic interpretations (Section 4.2); $\mathcal{D}_{prog}$ is a distribution over programs in the fixed grammar (Section 4.3); and $\mathcal{D}_{in}$ is a distribution over program inputs (Section 4.4). Each component is generated under structural constraints, ensuring consistency while maintaining diversity across instances. Modulo bugs in the meta-level infrastructure used to implement the (sampled) semantic function $[\![\cdot]\!]_\sigma$, there is no noise in the benchmarks generated: one always has the property that $y = [\![s]\!]_\sigma(x)$.

Benchmark difficulty is controlled through parameters of the generation process, including the structural complexity of semantic expressions, and the presence or absence of loop constructs. The following subsections describe how semantic interpretations, programs, and input/output examples are generated.

4.2 Semantic Generation

We generate semantic interpretations by assigning operational meanings to the statements $A, B, \ldots, N$. Each generated semantics uses two kinds of meta-language statements:

- Copy statements copy the value of one register into another register;
- Computational statements update a register using an arithmetic expression over the current register values.

Meta-language arithmetic expressions in a semantic specification are generated recursively from register references and arithmetic operators. For example, $r_1 + (r_4 - r_2)$ is an expression of size 5, consisting of three register leaves and two operators. Expression size serves as one axis of benchmark difficulty: larger expressions require more complex semantic reasoning. Each meta-language arithmetic expression that defines the semantics of a production is generated to be a fixed size (configurable, but required to be odd). The expression is built recursively: at each level, if the remaining size $2n + 1$ is one (i.e., $n = 0$), a register is selected uniformly at random; otherwise, a binary operator $(+, -)$ is selected uniformly at random, and the remaining size $2n$ is distributed to the left and right children by choosing a partition k uniformly at random between 0 and $n - 1$. The left child is allocated size $2k + 1$, and the right child $2n - (2k + 1)$.

To prevent models from associating particular productions with fixed behaviors, the generated transformations are randomly assigned to production symbols $A, B, \ldots, N$ independently for each sampled language. Consequently, the same production symbol may correspond to entirely different semantics across benchmark instances.

4.3 Program Generation

Given a sampled semantic interpretation σ, we generate programs from the fixed grammar by constructing finite sequences of productions from $\{A, B, \ldots, N\}$. The goal of program generation is not merely to produce syntactically valid programs, but to generate programs whose behavior depends meaningfully on the sampled semantics and input values.

To encourage non-trivial semantic reasoning, programs are generated using a backwards dataflow construction. We begin from the output register r_0, which determines the final program output, and iteratively construct statements whose values contribute to registers that are currently required. During this process, we maintain a set of *live* registers whose values influence the final output. At each step, we select productions whose outputs contribute to currently live registers and update the liveness set according to the register dependencies induced by the sampled semantics. This construction preferentially generates programs in which intermediate computations propagate information throughout the register state rather than producing unused or semantically irrelevant computations. Once a (configurable) target program length is reached, we add a minimal set of additional statements to ensure that all registers initialized from the program inputs influence the final output. Consequently, generated programs are guaranteed to depend non-trivially on their inputs and to exercise multiple components of the sampled semantics.

Program difficulty is controlled primarily through program length and the complexity of the sampled semantic interpretations. Longer programs require models to reason over longer chains of semantic transformations, while more complex semantic expressions increase the difficulty of reasoning about the effect of each individual production.

4.4 Input/Output Generation

Given a sampled semantic interpretation σ and program s, we generate input/output examples by executing the program on randomly sampled inputs from the range $[0, 100]$. Inputs are drawn uniformly from a bounded integer range, and outputs are computed by evaluating $y = [\![s]\!]_\sigma(x)$. Candidate examples are filtered to ensure that generated tasks exhibit meaningful semantic behavior. In particular, we reject examples that: (1) exceed a fixed execution-time bound (0.02s); (2) produce outputs outside a bounded integer range ($|y| \leq 2^{63} - 1$); (3) exhibit degenerate behavior, such as constant outputs or trivial identity mappings between inputs and outputs. These filtering steps prevent tasks from being solvable through superficial heuristics that do not require reasoning about the sampled semantics.

If the desired number (configurable) of non-trivial input/output examples cannot be generated for a sampled program after 20 tries for each example (200 if the program contains loops), we discard the program and sample a new one. If repeated program resampling fails 500 times, we regenerate the semantic interpretation and restart the generation process. This filtering procedure ensures that benchmark instances remain both semantically diverse and behaviorally informative.

4.5 Adding Loops

The benchmark construction described above generates loop-free languages in which semantic difficulty arises entirely from the meanings assigned to productions. To introduce an additional source of structural reasoning difficulty, we extend the language with loop constructs. We only add simple loops, and not other control structures with more complicated predicates, to isolate the impact of iterative semantics, rather than test arbitrary control structure semantics.

Our loops contain a body of one to three statements (picked uniformly). When executed, the body runs a fixed number of times, determined by the current value of the output register when first entering the loop. A loop is inserted at a predetermined point in the program (chosen uniformly from the target program size). To ensure that loops contribute meaningfully to program behavior, we require each generated input/output example to execute the loop body at least once. We additionally enforce execution-time bounds during example generation to avoid excessively large iteration counts or non-terminating behavior.

Loops increase reasoning difficulty in several ways. In loop-free programs, models only need to reason about each production once along a straight-line execution trace. In contrast, loops require reasoning about iterative state updates and the cumulative effect of repeatedly applying semantic transformations. Importantly, the addition of loops preserves the core benchmark design: syntax remains fixed across benchmark instances, while semantic variation continues to arise from the sampled interpretation σ. Loops therefore introduce structural complexity without changing the underlying evaluation framework.

Table 2: Model performance for tasks on loop-free languages, split by input format. The SemGuS format uses the native SemGuS S-expression format, and the C format encodes the same language into a more familiar C syntax.

Model	Predict Output (%)		Predict Inputs (%)		Predict Semantics (%)		Predict Program (%)		Overall (%)	
	SemGuS	C	SemGuS	C	SemGuS	C	SemGuS	C	SemGuS	C
o3	88	98	90.4	99.6	88.4	96.8	54.4	62.8	80.3	89.3
o4-mini	56.8	94	72.4	94.8	66.8	92.8	41.6	59.2	59.4	85.2
gpt-5.1	0	0.4	3.2	2	22.8	25.6	2.8	2	7.2	7.5
gpt-5	76.8	98.8	78	97.2	68.4	90.8	45.6	56.8	67.2	85.9
gpt-4.1	0	0	2.8	4	23.2	21.6	1.6	1.6	6.9	6.8
gpt-4.1-mini	0	0.4	1.2	0.8	20.4	22	1.6	3.6	5.8	6.7

Table 3: Model performance for tasks on languages with loops, evaluated only in the SemGuS format.

Model	Predict Output (%)	Predict Inputs (%)	Predict Semantics (%)	Predict Program (%)	Overall (%)
o3	86	87.6	85.2	31.2	72.5
o4-mini	49.2	59.6	58.8	20.4	47
gpt-5.1	0	2.4	20.8	0.4	5.9
gpt-5	64.4	64	64.8	34	56.8
gpt-4.1	0	0.8	20.8	0.4	5.5
gpt-4.1-mini	0	0.4	23.2	0.8	6.1

5 EVALUATION

Experimental Setup. We evaluate six models across 500 generated languages (half straight-line loop-free programs, half with loops) over four registers and two inputs, each with a sampled target program of 10–20 statements and five input/output examples. Each language is generated with a semantic specification in which all of the (meta-language) arithmetic expressions in computational statements are of the same size. These sizes are sampled between 3 to 79, with a bias towards smaller sizes to better differentiate performance among smaller sizes. Each language is used in four prediction tasks (predict output, predict inputs, predict semantics, and predict program), yielding a total of 2000 task instances for each model. Each instance is evaluated three times, counting as solved if any of the three runs succeeds; we report 95% Wilson score confidence intervals.

We evaluate six commercially-available models from OpenAI: o3, o4-mini, gpt-5.1, gpt-5, gpt-4.1, and gpt-4.1-mini. This selection includes reasoning models (o3, o4-mini), recent standard models (gpt-5, gpt-5.1), and older models (gpt-4.1, gpt-4.1-mini). These models were hosted on Microsoft Azure and accessed via standard network API calls. For each task, we generate a prompt for the model. This prompt includes (1) a description of the task (e.g., "predict the output for each set of inputs"), (2) a solved in-context example (generated with the same parameters but not included in any tasks), (3) the language specification (in SemGuS or C format), (4) the inputs/outputs/program as required for the task, and (5) a description of the desired output format. Our prompts are intentionally simple and do not include chain-of-thought prompting, code-analysis scaffolding (as might be found in coding-assistant tools), or other techniques. This prompting strategy focuses only on the inherent capabilities of the models themselves, not the tools and techniques that can be built on top of them.

Loop-free tasks are run in both the SemGuS and C formats. Tasks with loops are only evaluated in the SemGuS format: there, a loop is syntactically opaque (e.g., `($loop body continuation)`), with meaning given through the semantics definitions alone. A C rendering, however, would expose some semantic hints to the model through the syntax (e.g., a `for` or `while` statement).

Q1: What is the performance of models on different tasks? Table 2 reports each model's overall pass rate and per-task breakdown on loop-free languages, split on the SemGuS input format and C input format. We observe a sharp two-tier structure.

Within the high-performing tier, o3, a reasoning model, achieves the best overall result in both the SemGuS format (80.3% on loop-free programs) and the C-like format (89.3%). It is followed by gpt-5 (67.2% / 85.9%) and o4-mini (59.4% / 85.2%). Notably, o3's advantage over gpt-5 is substantially larger in the SemGuS format (13.1 percentage points) than in the C format (3.4 points), suggesting that reasoning ability is especially valuable when the input syntax is less familiar.

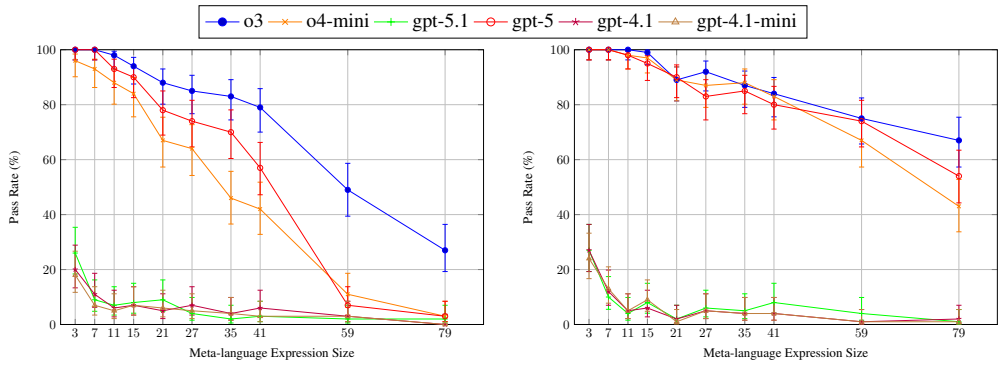


Figure 1: Task pass rates for the (a) SemGuS and (b) C input formats (loop-free) by meta-language expression size. Pass rates drop at higher sizes, with a more pronounced drop for the SemGuS format.

The remaining three models (gpt-4.1, gpt-4.1-mini, and gpt-5.1) cluster near the floor at 6–8%, essentially unable to reason semantically beyond the simplest tasks. Of note is gpt-5.1, which we run in the default non-reasoning mode. Despite being a newer model, with reasoning not enabled, it performs near the floor overall ($\sim$7%), comparable to gpt-4.1.

Table 3 reports performance for tasks over languages containing loops (SemGuS format only). The same tier structure holds with generally reduced success rates, suggesting that loops do add non-trivial difficulty at every level; notably, program synthesis performance drops for all models. In the high-performing tier, gpt-5 and o4-mini respectively suffer 14-point and 12.8-point drops on predicting inputs, compared to a 2.8-point drop on o3. This suggests that the additional reasoning capabilities of o3 help with tracing the program state through the loops to predict the input successfully.

The relative difficulty of predict output vs. predict inputs is notable. Although one might expect these to be symmetric tasks, output prediction is slightly harder on average: there is only one correct output for a given input, whereas a given output may be reachable from multiple inputs, giving input-prediction tasks a larger space of valid answers.

Another striking pattern is that the three weaker models (gpt-4.1, gpt-4.1-mini, gpt-5.1) achieve near 20–26% on predicting a missing meta-language expression in a semantic specification, roughly a quarter of that of the top models, but are at or near 0% on predict output and predict program. Likely this is because predicting semantics is a more open-ended task: many completions may be valid.

Q2: Does meta-language expression size impact performance? Our primary axis of semantic complexity is the meta-language expression size for the computational statements in the generated language. Figure 1 shows a clear trend of increasing difficulty as expression size grows, regardless of the input format (SemGuS versus C) or containing loops or not.

In the SemGuS format, on loop-free programs, the top performer, o3, maintains above 50% through size 41 (79% at size 41), but then drops sharply to 49% at size 59 and just 27% at size 79. No model achieves above 50% at size 79, and the three weaker models (gpt-4.1, gpt-4.1-mini, gpt-5.1) effectively reach 0% by size 79. Models with stronger semantic understanding exhibit a slower drop-off: the gap between o3 and gpt-5 is nil at size 3 but grows to 42 points by size 59. The C input format shows a similar, but less pronounced, trend.

Q3: Does the textual format of the input affect performance? Table 2 compares pass rates on the same loop-free instances encoded in the SemGuS format vs. a C language encoding. The format gap is substantial and related to model capability. The three most capable models show the largest gaps: o3 scores 9 percentage points lower in SemGuS than C, gpt-5 shows an 18.7-point gap (67.2% vs. 85.9%), and o4-mini shows a 25.8-point gap (59.4% vs. 85.2%). By contrast, the three weaker models show virtually no difference between formats (under 3 percentage points in all task categories).

This pattern suggests that weaker models are bottlenecked by semantic understanding regardless of how the problem is presented: they fail at the same rate in both formats because neither syntax unlocks any additional reasoning. Stronger models, on the other hand, do have latent semantic capability but are additionally sensitive to the syntactic presentation.

Q4: Does semantic understanding translate to better synthesis? As a practical test to see if our evaluation translates to real-world results, we ask our models under test to solve all 277 realizable SemGuS synthesis benchmarks from the standard SemGuS evaluation set (Johnson et al., 2024). We give the model access to a parsing tool (for output validation) but otherwise ask for one-shot solutions. To force the models to reason about semantics, we obscure the non-terminals, productions, and synthesis-task names (e.g., turning non-terminal names into `$NT001`, productions into `$PROD001`, etc.) to remove any syntactic clues. We run three trials for each model and benchmark with a 300s timeout, and we score it as "passed" if at least two out of three runs produce a valid solution.

A cactus plot of runtime vs. benchmarks solved is shown in Figure 2. At the top are gpt-5 and o3, solving 206 and 190 benchmarks, respectively, doing noticeably better than the baseline performance by the KS2 SemGuS synthesizer, at 161 benchmarks. About equal to KS2 is o4-mini, at 160 solved benchmarks. When the benchmarks are bucketed by an average semantic complexity size (semantics meta-language size divided by production count), we see similar trends as in the main task evaluation: gpt-4.1-mini, gpt-4.1, and gpt-5.1 can solve some less complex benchmarks (average size ≤ 40), but as complexity increases (> 40), they cannot solve any, while o3, o4-mini, and gpt-5 continue to solve 51%, 47%, and 69% (respectively) of benchmarks with average semantic complexity size greater than 40.

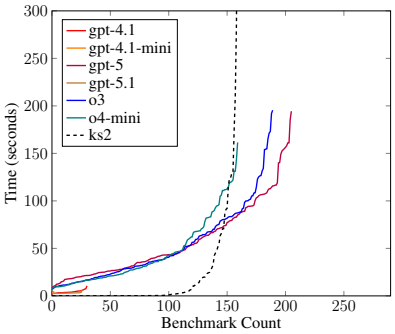


Figure 2: Cactus plot of synthesis runs. The dashed line shows the KS2 virtual best (Johnson et al., 2024). Two benchmarks solved by KS2 are off the scale, at 570 and 799 seconds. (Lower and to the right is better.)

A somewhat surprising result is that the traditional search-based synthesis used by KS2 is still competitive, despite the advanced capabilities of LLMs. However, one shouldn't read too much into the performance times, based on the different test conditions (KS2's local execution vs. remote LLMs).

Q5: How do models fail? Most failures are syntactically valid but semantically wrong answers: gpt-5.1 fails with a wrong answer 100% of the time, and o4-mini 99.4% of the time. These cases suggest syntactic pattern-matching without semantic understanding: the model returns something plausible-looking but wrong. Only a very small number of failures are caused by model output that cannot be parsed. We also accept responses that deviate from strict SemGuS syntax when the intended answer is still recoverable (e.g., C-style function-call notation instead of S-expressions, or unbalanced closing parentheses).

In some cases, the model produces no answer. This failure mode occurs primarily with the two strongest models: o3 abstains on 72% of its failures, and gpt-5 on 91%. The model responses do not expose specific details about the failures; however, the behavior is consistent with a timeout or token limit, implying that the model continues to reason about the answer until being cut off. In contrast, weaker models almost always attempt an answer regardless of whether they can reason about it correctly.

6 Discussion

Limitations. Although our results are corroborated by existing synthesis benchmarks, their generality beyond these settings remains an open question for future work. Moreover, it is hard to disentangle improved semantic understanding from general improvements in capability.

Conclusion and broader impact. More broadly, this work suggests a template for evaluating whether ML systems learn the governing semantics of a domain rather than surface correlations. In our domain of programming languages, we make this question precise by holding syntax fixed, varying the semantic interpretation, and testing whether model predictions track the resulting behavior, but this principle also generalizes to other areas. Our benchmarks are a controlled instance of this broader methodology: vary the latent rules while controlling superficial form, and test whether model behavior remains consistent with those rules.

AI USE STATEMENT

In this work, we used generative AI tools for coding assistance in implementing parts of the benchmark generation and evaluation pipeline, providing initial impressions of our benchmark results to guide our research direction and methodology, and troubleshooting errors and unexpected behaviors. We did not use generative AI tools to generate any part of our benchmarks: all benchmark instances are produced by the procedural generator described in Section 4, and all model answers are checked deterministically. We did not use generative AI tools to develop conceptual frameworks, formulate hypotheses, clean or reformat datasets, or interpret our final results. Formulating or proving mathematical claims, translation, and qualitative data analysis are not applicable to this work.

Additionally, we used generative AI tools to identify related literature, to write initial drafts of parts of this manuscript, and to edit for readability, consistency, and formatting. We have reviewed all AI-assisted work for correctness, including reading and verifying all cited works, reviewing and manually testing all code, and checking all claims, numbers, and figures in this work for accuracy against the underlying results. We take full responsibility for the final content of this work, including any text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide anonymized supplementary material containing all code for our benchmark generator and evaluation pipeline, instructions for running it, the full set of generated benchmark instances used in our experiments. The generation procedure, including all sampling parameters and filtering criteria, is described in Section 4, and the experimental setup is described in Section 5. Appendix A lists the exact versions of all evaluated models, and Appendix D gives an overview of the prompt structure (full prompts are in the supplementary material). The SemGuS synthesis benchmarks used in our evaluation are publicly available (Johnson et al., 2024), and our renaming procedure for them is described in Section 5. Because the evaluated models are commercial, API-hosted, and nondeterministic, re-running our evaluation may produce slightly different results.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024. URL `https://proceedings.mlr.press/v235/gu24c.html`.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL `https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c24cd76e1ce41366a4bbe8a49b02a028-Paper-round2.pdf`.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=chfJJYC3iL`.

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

Keith JC Johnson, Andrew Reynolds, Thomas Reps, and Loris D'Antoni. The SemGuS toolkit. In *International Conference on Computer Aided Verification*, pp. 27–40. Springer, 2024.

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434311. URL https://doi.org/10.1145/3434311.

Man Ho Lam, Chaozheng Wang, Jen-Tse Huang, and Michael R Lyu. CodeCrash: Exposing LLM fragility to misleading natural language in code reasoning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 120782–120828. Curran Associates, Inc., 2025. doi: 10.52202/085713-4029. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/aec5e2847c5ae90f939ab786774856cc-Paper-Conference.pdf.

Thanh Le-Cong, Bach Le, and Toby Murray. Can LLMs reason about program semantics? A comprehensive evaluation of LLMs on formal specification inference. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21991–22014, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1068. URL https://aclanthology.org/2025.acl-long.1068/.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. CodeI/O: Condensing reasoning patterns via code input-output prediction. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=feIaF6vYFl.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. NEXT: teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. Teaching Programming Languages by Experimental and Adversarial Thinking. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (eds.), *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 13:1–13:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-032-3. doi: 10.4230/LIPIcs.SNAPL.2017.13. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2017.13.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Aditya Thimmaiah, Jiyang Zhang, Jayanth Srinivasa, Junyi Jessy Li, and Milos Gligoric. LLMs lean on priors, not programming language semantics. In *ICML*, 2026.

Shubham Ugare and Satish Chandra. Agentic code reasoning. *arXiv preprint arXiv:2603.01896*, 2026.

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. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33868–33881, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1718. URL https://aclanthology.org/2025.emnlp-main.1718/.

Anjiang Wei, Yuheng Wu, Yingjia Wan, Tarun Suresh, Huanmi Tan, Zhanke Zhou, Sanmi Koyejo, Ke Wang, and Alex Aiken. SATBench: Benchmarking LLMs' logical reasoning via automated puzzle generation from SAT formulas. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33832–33849, Suzhou, China, November 2025b. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1716. URL https://aclanthology.org/2025.emnlp-main.1716/.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. CRUXEVAL-X: A benchmark for multilingual code reasoning, understanding and execution. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23762–23779, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1158. URL https://aclanthology.org/2025.acl-long.1158/.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and Shuiguang Deng. CodeScope: An execution-based multilingual multitask multidimensional benchmark for evaluating LLMs on code understanding and generation. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5511–5558, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.301. URL https://aclanthology.org/2024.acl-long.301/.

A LLM VERSION INFORMATION

See Table 4 for version information of LLMs used in this work.

Table 4: Large Language Model versioning used in this work

Model	Source	Version
gpt-4.1	OpenAI	2025-04-14
gpt-4.1-mini	OpenAI	2025-04-14
gpt-5	OpenAI	2025-08-07
gpt-5.1	OpenAI	2025-11-13
o3	OpenAI	2025-04-16
o4-mini	OpenAI	2025-04-16

B DETAILED SEMGUS SYNTHESIS BENCHMARK INFORMATION

See Table 5 for the breakdown of solved synthesis benchmarks by model and domain (as defined by Johnson et al. (2024)).

Table 5: Model performance on SemGuS synthesis benchmarks

Model	Sample Domains	Regular Expressions	Boolean	Bitvectors	**Total**
o3	17	48	58	67	190
o4-mini	16	47	44	53	160
gpt-5.1	2	0	4	21	27
gpt-5	17	62	60	67	206
gpt-4.1	2	0	6	23	31
gpt-4.1-mini	0	0	0	3	3
ks2 virtual best	15	54	49	43	161
Total	17	72	88	100	277

C MODEL PERFORMANCE TABLES WITH COMPLETE CONFIDENCE INTERVALS

A copy of Table 2 with 95% Wilson score intervals is shown in Table 6. A similar copy of Table 3 with confidence intervals is shown in Table 7.

Table 6: Model performance for tasks on loop-free languages, split by input format. The SemGuS format uses the native SemGuS S-expression format, and the C format encodes the same language into a more familiar C syntax. Includes 95% Wilson score intervals.

Model	**Predict Output (%)**		**Predict Inputs (%)**		**Predict Semantics (%)**		**Predict Program (%)**		**Overall (%)**	
	SemGuS	C	SemGuS	C	SemGuS	C	SemGuS	C	SemGuS	C
o3	$88.0^{+3.5}_{-4.6}$	$98.0^{+1.1}_{-2.6}$	$90.4^{+3.1}_{-4.3}$	$99.6^{+0.3}_{-1.8}$	$88.4^{+3.4}_{-4.6}$	$96.8^{+1.6}_{-3.0}$	$54.4^{+6.1}_{-6.2}$	$62.8^{+5.8}_{-6.1}$	$80.3^{+2.3}_{-2.6}$	$89.3^{+1.8}_{-2.1}$
o4-mini	$56.8^{+6.0}_{-6.2}$	$94.0^{+2.3}_{-3.7}$	$72.4^{+5.2}_{-5.8}$	$94.8^{+2.1}_{-3.5}$	$66.8^{+5.5}_{-6.1}$	$92.8^{+2.6}_{-3.9}$	$41.6^{+6.2}_{-5.9}$	$59.2^{+5.9}_{-6.2}$	$59.4^{+3.0}_{-3.1}$	$85.2^{+2.1}_{-2.3}$
gpt-5.1	$0.0^{+1.5}_{-0.0}$	$0.4^{+1.8}_{-0.3}$	$3.2^{+3.0}_{-1.6}$	$2.0^{+2.6}_{-1.1}$	$22.8^{+5.6}_{-4.8}$	$25.6^{+5.8}_{-5.0}$	$2.8^{+2.9}_{-1.4}$	$2.0^{+2.6}_{-1.1}$	$7.2^{+1.8}_{-1.4}$	$7.5^{+1.8}_{-1.5}$
gpt-5	$76.8^{+4.8}_{-5.6}$	$98.8^{+0.8}_{-2.3}$	$78.0^{+4.7}_{-5.5}$	$97.2^{+1.4}_{-2.9}$	$68.4^{+5.4}_{-6.0}$	$90.8^{+3.0}_{-4.2}$	$45.6^{+6.2}_{-6.1}$	$56.8^{+6.0}_{-6.2}$	$67.2^{+2.8}_{-3.0}$	$85.9^{+2.0}_{-2.3}$
gpt-4.1	$0.0^{+1.5}_{-0.0}$	$0.0^{+1.5}_{-0.0}$	$2.8^{+2.9}_{-1.4}$	$4.0^{+3.2}_{-1.8}$	$23.2^{+5.6}_{-4.8}$	$21.6^{+5.5}_{-4.7}$	$1.6^{+2.4}_{-1.0}$	$1.6^{+2.4}_{-1.0}$	$6.9^{+1.7}_{-1.4}$	$6.8^{+1.7}_{-1.4}$
gpt-4.1-mini	$0.0^{+1.5}_{-0.0}$	$0.4^{+1.8}_{-0.3}$	$1.2^{+2.3}_{-0.8}$	$0.8^{+2.1}_{-0.6}$	$20.4^{+5.4}_{-4.5}$	$22.0^{+5.5}_{-4.7}$	$1.6^{+2.4}_{-1.0}$	$3.6^{+3.1}_{-1.7}$	$5.8^{+1.6}_{-1.3}$	$6.7^{+1.7}_{-1.4}$

Table 7: Model performance for tasks on languages containing loops. Only the SemGuS format is supported for these tasks. Includes 95% Wilson score intervals.

Model	Predict Output (%)	Predict Inputs (%)	Predict Semantics (%)	Predict Program (%)	Overall (%)
o3	$86.0^{+3.8}_{-4.8}$	$87.6^{+3.5}_{-4.7}$	$85.2^{+3.9}_{-4.9}$	$31.2^{+6.0}_{-5.4}$	$72.5^{+2.7}_{-2.8}$
o4-mini	$49.2^{+6.2}_{-6.1}$	$59.6^{+5.9}_{-6.2}$	$58.8^{+5.9}_{-6.2}$	$20.4^{+5.4}_{-4.5}$	$47.0^{+3.1}_{-3.1}$
gpt-5.1	$0.0^{+1.5}_{-0.0}$	$2.4^{+2.7}_{-1.3}$	$20.8^{+5.5}_{-4.6}$	$0.4^{+1.8}_{-0.3}$	$5.9^{+1.6}_{-1.3}$
gpt-5	$64.4^{+5.7}_{-6.1}$	$64.0^{+5.7}_{-6.1}$	$64.8^{+5.7}_{-6.1}$	$34.0^{+6.1}_{-5.6}$	$56.8^{+3.0}_{-3.1}$
gpt-4.1	$0.0^{+1.5}_{-0.0}$	$0.8^{+2.1}_{-0.6}$	$20.8^{+5.5}_{-4.6}$	$0.4^{+1.8}_{-0.3}$	$5.5^{+1.6}_{-1.3}$
gpt-4.1-mini	$0.0^{+1.5}_{-0.0}$	$0.4^{+1.8}_{-0.3}$	$23.2^{+5.6}_{-4.8}$	$0.8^{+2.1}_{-0.6}$	$6.1^{+1.7}_{-1.3}$

D PROMPT STRUCTURE

Every task instance is posed as a single user message with the overall Markdown structure shown in Figure 3, with the instance block shown in Figure 4. Text in *italics* is a placeholder that is filled in per instance; all other text appears verbatim. Section D.1 gives the text that differs by task, for the SemGuS format (Section D.1.1) and the C format (Section D.1.2). The complete prompt for every instance is included in the supplementary material.

```
# Task: Task title

Given the following format name problem specification task summary.

# In-Context Examples

The following are worked examples -- sample problems of this same task with valid solutions.
Use them to calibrate the response format and the kind of reasoning expected. After reviewing
them, solve the problem at the bottom.

# Example 1

Problem instance (example)

## Answer

Example answer

# Problem to Solve

Problem instance (task)

## Output Format

IMPORTANT: Your response must contain ONLY answer shape with no additional text, explanation, or
formatting.
```

Figure 3: Prompt skeleton shared by all tasks and both input formats. Each *problem instance* block expands as shown in Figure 4.

```
## format name Specification

```
Language specification: syntax and semantics of all statements, plus I/O constraints (when synthesizing a program)
(omitted component replaced by an OMITTED marker)
```

Program (omitted when synthesizing a program)

```
Program in the format's program syntax
```

data heading (omitted when synthesizing a program)

Example 1: data line
...
Example 5: data line

Your Task

Task instruction
```

Figure 4: Instance block, used for both in-context examples and the task to solve.

D.1 TASK-SPECIFIC TEXT

The listings in the following sections show the text for each task for a sample instance; lines marked with ... are unchanged from Figures 3 and 4. Refer to the supplementary material for full prompt examples for all tasks and formats.

D.1.1 SEMGUS FORMAT

```
# Task: Predict Outputs

Given the following SemGuS problem specification and a program, predict the output for each
set of inputs.
...
## Inputs

Example 1: i0=81, i1=24
...
## Your Task

For each example, compute the output value that the program produces. The output is the final
value of r0 after executing all statements.

## Output Format

IMPORTANT: Your response must contain ONLY the output values in order (one per example),
separated by commas, with no additional text, explanation, or formatting.
```

```
# Task: Predict Inputs

Given the following SemGuS problem specification and a program, predict input values that
would produce each output.
...
## Outputs

Example 1: output=96
...
## Your Task

For each example, find input values (i0, i1) that would cause the program to produce the given
 output.

## Output Format

IMPORTANT: Your response must contain ONLY the input values for each example as: (i0 i1 ...),
separated by commas, with no additional text, explanation, or formatting.
```

```
# Task: Predict Missing Semantics

Given the following SemGuS problem specification with one statement's semantics omitted, a
program that uses that statement, and input/output examples, predict the missing semantics.
...
## Input/Output Examples

Example 1: i0=81, i1=24 -> output=96
...
## Your Task

The semantics for $F are omitted (marked with SEMANTICS OMITTED). Determine which register
this statement writes to and what expression it computes.
Available operators: +, -

## Output Format

IMPORTANT: Your response must contain ONLY the target register and expression as: rN = <expr>
with no additional text, explanation, or formatting.
```

```
# Task: Synthesize Program

Given the following SemGuS problem specification with input/output constraints, find a program
 that satisfies all constraints.
...
## Your Task

Write a program using the grammar that satisfies all the constraints. Programs have the form:
($prog ($seq $Statement ($seq $Statement ... $end))) where each $Statement is one of the
available statement productions (e.g., $A, $B, etc.).

## Output Format

IMPORTANT: Your response must contain ONLY the program as a single s-expression with no
additional text, explanation, or formatting.
```

D.1.2 C FORMAT

```
# Task: Predict Outputs

Given the following C language problem specification and a program, predict the output for
each set of inputs.
...
## Inputs

Example 1: i0=81, i1=24
...
## Your Task

For each example, compute the output value that the program produces. The output is the final
value of r[0] after executing all statements.

## Output Format

IMPORTANT: Your response must contain ONLY the output values in order (one per example),
separated by commas, with no additional text, explanation, or formatting.
```

```
# Task: Predict Inputs

Given the following C language problem specification and a program, predict input values that
would produce each output.
...
## Outputs

Example 1: output=96
...
## Your Task

For each example, find input values (i0, i1) that would cause the program to produce the given
 output.

## Output Format

IMPORTANT: Your response must contain ONLY the input values for each example as: (i0 i1 ...),
separated by commas, with no additional text, explanation, or formatting.
```

```
# Task: Predict Missing Semantics

Given the following C language problem specification with one statement's semantics omitted, a
 program that uses that statement, and input/output examples, predict the missing semantics.
...
## Input/Output Examples

Example 1: i0=81, i1=24 -> output=96
...
## Your Task

The semantics for stmt_F are omitted (marked with SEMANTICS OMITTED). Determine which register
 this statement writes to and what expression it computes.
Available operators: +, -

## Output Format

IMPORTANT: Your response must contain ONLY the target register and expression as: rN = <expr>
with no additional text, explanation, or formatting.
```

```
# Task: Synthesize Program

Given the following C language problem specification with input/output constraints, find a
program that satisfies all constraints.
...
## Your Task

Write a program using the grammar that satisfies all the constraints. Programs are a sequence
of statement names (e.g., A, B, C) listed in execution order, separated by commas. Registers
are initialized as: int64_t r[4] = {0, i0, i1, 0}. The program returns r[0].

## Output Format

IMPORTANT: Your response must contain ONLY the program as a comma-separated list of statement
names (e.g., A, B, C) with no additional text, explanation, or formatting.
```