Predicting Program Exit Code with LLMs and Programming Language Semantics

Lara Marinov
The University of Texas at Austin
Austin, USA
marinov@utexas.edu

Aditya Thimmaiah
The University of Texas at Austin
Austin, USA
auditt@utexas.edu

Jayanth Srinivasa
Cisco Research
San Francisco, USA
jasriniv@cisco.com

Junyi Jessy Li
The University of Texas at Austin
Austin, USA
jessy@utexas.edu

Milos Gligoric
The University of Texas at Austin
Austin, USA
gligoric@utexas.edu

Abstract

Large language models (LLMs) have shown proficiency in various software engineering tasks, such as code generation and translation. However, a key limitation in their performance may be their (lack of) understanding of programming-language semantics. Even when explicit semantics are given, it remains unclear whether LLMs apply those rules or lean on priors learned during pre-training instead. We study if LLMs lean on priors or given semantics with a novel task–Program Executability Prediction (PrEx)–that asks models to predict whether a program is semantically valid or invalid (and, if invalid, which formal rule it violates) given the program's syntax and operational semantics. Because PrEx requires both valid and invalid programs, we build a dataset with systematically generated invalid transformations derived from valid programs. We evaluate open-source coding LLMs under two semantic formalisms and two semantic shifts across Human-Written, LLM-Translated, and Fuzzer-Generated program splits. Our findings show that LLMs lean on pre-training priors rather than systematically applying the given rules, performing especially poorly on modified semantics and degrading further as program complexity increases. PrEx is available at https://github.com/EngineeringSoftware/prex.

CCS Concepts: • **General and reference** → *Empirical studies*; • **Theory of computation** → **Operational semantics**; • **Computing methodologies** → **Machine learning**.

Keywords: Operational Semantics, K Framework, Program Executability Prediction, Large Language Models

ACM Reference Format:
Lara Marinov, Aditya Thimmaiah, Jayanth Srinivasa, Junyi Jessy Li, and Milos Gligoric. 2026. Predicting Program Exit Code with LLMs and Programming Language Semantics. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '26), October 4–9, 2026, Oakland, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3843750.3843842

This work is licensed under a Creative Commons Attribution 4.0 International License.

LMPL '26, Oakland, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2986-7/2026/10
https://doi.org/10.1145/3843750.3843842

1 Introduction

LLMs have become widely used in software engineering, demonstrating remarkable proficiency in tasks like code generation [11, 24, 31], code translation [26, 43], bug fixing [15, 22], and summarizing complex codebases [23, 27, 38]. However, despite their practical success, it remains unclear to what extent these models truly comprehend the underlying semantics of the programs they process.

Much of an LLM's apparent coding ability seems to come from sophisticated pattern matching and the memorization of vast amounts of training data, rather than an inherent ability to logically simulate program execution [29, 44]. For example, consider a scenario where an LLM is presented with a standard sorting algorithm. Because the model has previously encountered this exact structure many times, it can easily summarize its purpose, generate missing lines of code, or predict the final outcome [8, 36]. However, if we introduce a subtle change to the program, such as slightly modifying the loop bounds, obfuscating variable names, or redefining how the comparison operator works, and ask the model to predict the result, it frequently fails [41]. Instead of systematically tracing the data flow step-by-step as an interpreter would, the model often falls back on its *priors* and simply guesses the standard sorted output [4, 28]. Such a situation clearly reveals a gap between rote memorization and genuine, systematic semantic reasoning.

Recent work introduced PLSemanticsBench [39] and established a framework for evaluating LLMs' semantic understanding through three tasks: predicting final variable states (PredState), identifying the formal semantic rules applied during execution (PredRule), and generating step-by-step execution traces (PredTrace) [39]. Evaluations revealed

that current LLMs struggle significantly across all three of these tasks. Because PLSemanticsBench presented a set of complex evaluations where models needed to articulate intermediate steps and output long execution traces, the models' uniformly low performance leaves an open question regarding exactly where their capability threshold is. To determine whether these failures are in-part due to the natural multi-step complexity of the tasks presented in PLSemanticsBench or a fundamental inability to reason about semantics, we establish a more foundational baseline that separates trace generation from semantic judgment. Building on this premise, we present PrEx: a task that asks models, given a program and formal programming language semantics, to predict whether execution succeeds or halts with a semantic error—and, when it fails, which rule was violated. The required output is deliberately small with just a discrete executability verdict rather than a long trace but the evaluation is not: we pair each valid program with invalid counterparts produced by semantics-aware invalid transformations, vary program source and structural complexity across three splits, and probe rule-following under alternate semantic formalisms and shifts. This design lets us ask whether models systematically apply the supplied rules or instead continue to rely on memorized priors.

Dataset. We build on the valid C* programs from PLSemanticsBench and extend them with semantically *invalid transformations* required by PrEx. Invalid programs are generated from valid programs using a semantics-aware *transformation procedure* that applies one of five errors: break-outside-loop, continue-outside-loop, divide-by-zero, modulo-by-zero, and variable-use-before-declare. Each valid program has a matched invalid counterpart in every error category. Programs come from three *splits*—Human-Written, LLM-Translated, and Fuzzer-Generated—with average length and complexity increasing across splits. Together, these design choices let us test whether semantic reasoning generalizes across program sources and complexity levels, not just memorized coding patterns.

Semantic formalisms. We evaluate models under two semantic formalisms: small-step operational semantics ($\mathbb{S}$) and the K-framework ($\mathbb{K}$). $\mathbb{S}$ defines execution through finer granularity inference rules, where each rule represents one atomic computation step [34]. $\mathbb{K}$ instead uses coarser rewriting rules that transform program configurations in larger chunks, relying on built-in rewriting to handle intermediate expression reduction [35]. We include both to test whether an LLM's ability to apply provided semantics depends on rule granularity or persists across formalisms.

Semantic shifts. We also evaluate models under two semantic shifts, KeywordSwap and KeywordObf, that alter the meaning of selected symbols while keeping execution defined by the provided formal rules. KeywordSwap interchanges

```
1  int i;
2  int ans;
3  int n;
4  i = 1;
5  n = 10;
6  while ((i * i) <= n) {
7     i = (i + 1);
8     ans = (ans + 1);
9  };
```

(a) Valid program

```
1  int i;
2  int ans;
3  int n;
4  i = 1;
5  n = 10;
6  while ((i * i) <= n) {
7     n = 0;
8     i = ((i / n) + 1);
9     ans = (ans + 1);
10 };
```

(b) Invalid program

Figure 1. Valid and invalid PrEx task examples.

the meanings of familiar operator symbols (e.g., + denotes subtraction), requiring models to override pre-trained associations and follow the supplied semantics. KeywordObf replaces standard operators and keywords with novel single-token symbols, removing recognizable cues from pre-training while preserving the same underlying rules. Together, these conditions test whether LLMs can reason from explicitly provided semantics rather than from their prior knowledge.

We evaluate open-source LLMs designed for coding tasks and enhanced reasoning using both non-chain-of-thought and chain-of-thought (CoT) reasoning. Our findings indicate that even on the simpler task (PrEx) of predicting program exit codes, LLMs continue to lean on priors learned during pre-training rather than systematically applying the supplied formal rules. Performance is especially poor under KeywordSwap and KeywordObf, where models must override familiar symbol meanings to follow the provided semantics. Accuracy also degrades significantly on the LLM-Translated and Fuzzer-Generated splits compared to Human-Written, indicating that semantic reasoning does not generalize well as structural complexity increases.

2 Task Description

In traditional program interpretation, the semantic analyzer identifies semantic errors in the input program, typically following parsing [1, 2]. As a step towards our long term goal–evaluating whether LLMs can function as interpreters–in this paper, we tasked LLMs with detecting semantic errors.

We not only provide the model with a program it is expected to analyze but also supply the formal semantics of the language, including the inference rules that govern valid and invalid executions, thereby encouraging it to reason about semantic correctness. If an LLM identifies a program as semantically invalid, it is further instructed to specify the rule that formalizes the error, assessing its understanding of the programming language semantics. Consequently, we test not only if these models can identify a program as semantically invalid, but also if they lean on priors learned in their training data or on the semantics provided to them.

To illustrate the task, consider the example in Figure 1 of a C* program which calculates the integer square root of the variable *n*. The program in Figure 1a is semantically valid and contains no rule violations, so an LLM successfully performing PrEx should classify it as valid, based on semantic reasoning over the inference rules rather than needing to simulate a full step-by-step execution. In contrast, Figure 1b shows an invalid variant of the same base program. Because line 8 of the invalid program contains a division-by-zero error, an LLM should successfully classify the program as invalid and accurately specify the violated rule.

3 Methodology

This section describes our experimental setup for PrEx. We describe the C* programming language (Section 3.1), the semantic formalisms used to specify program behavior ($\mathbb{S}$ and $\mathbb{K}$; Section 3.2), semantic shifts that test whether models follow provided rules or pre-training priors (KeywordSwap and KeywordObf; Section 3.3), and the full prompt given to models (Section 3.4). The C* language, semantic formalisms, and semantics shifts are introduced in PLSemanticsBench [39], but we recap them here together with the full prompt used for PrEx.

3.1 C* Language

We evaluate models on programs written in C*, a small imperative language with C-like syntax. The complete EBNF grammar used in our experiments is given in Figure 2. We use the same C* language as PLSemanticsBench.

```
1  <program>   ::= <stmt_list>
2  <stmt_list> ::= (<stmt> ';')*
3  <stmt>      ::= 'int' <id>
4               | <id> '=' <aexp>
5               | 'if' '(' <bexp> ')' '{' <stmt_list> '}'
                    'else' '{' <stmt_list> '}'
6               | 'while' '(' <bexp> ')' '{' <stmt_list> '
                    }'
7               | 'loop' '(' <bexp> ')' '{' <stmt_list> '}
                    '
8               | 'halt'
9               | 'continue'
10              | 'break'
11              | 'LE'
12 <aexp>      ::= <id>
13              | <literal>
14              | '(' <aexp>? <mathop> <aexp> ')'
15 <bexp>      ::= '(' <bool> ')'
16              | '(' <aexp> <relop> <aexp> ')'
17              | '(' <lognot> <bexp> ')'
18              | '(' <bexp> <logicalop> <bexp> ')'
19 <bool>      ::= 'true' | 'false'
20 <mathop>    ::= '+' | '-' | '*' | '/' | '%'
21 <relop>     ::= '<' | '<=' | '>' | '>=' | '==' | '!='
22 <lognot>    ::= '!'
23 <logicalop> ::= '&&' | '||'
24 <id>        ::= <letter> (<letter> | <digit>)*
25 <literal>   ::= <digit>+
```

Figure 2. Complete syntax of C* in EBNF (`<letter>` and `<digit>` rules are omitted and have their usual definitions).

3.2 Semantic Formalisms

We evaluate the models using two distinct semantic formalisms: Small-step operational semantics ($\mathbb{S}$) and the K-framework ($\mathbb{K}$). We use both formalisms to understand how the granularity and style of formal rules affect an LLM's ability to correctly reason about the programs it is given and to discover if presenting the semantic rules in a particular formulization maximizes the model's adherence to them.

Small-step operational semantics ($\mathbb{S}$). $\mathbb{S}$ is a fine-grained semantic formalism, where each rule represents one atomic computation [34]. Execution happens through repeated rule applications as expressions are broken down into tiny constituent parts and resolved one by one. Rules are written in Gentzen-style inference notation [9], with premises and side conditions written above the bar and conclusions below.

Table 1 shows a subset (due to the space reasons) of the $\mathbb{S}$ rules used in our experiments. To illustrate how the $\mathbb{S}$ formalism handles both valid execution and error states, we highlight several key rules. Rules 1 and 2 form a complementary pair describing variable lookup: Rule 1 defines successful evaluation when a variable is bound in the current state, while Rule 2 states that attempting to evaluate an undeclared variable results in an error. Similarly, Rules 18 and 19 describe the division operation, distinguishing between valid integer division and a division-by-zero error. Rule 9 highlights the fine-grained, small-step nature of the formalism by explicitly defining the addition transition only after both operands have been fully reduced to values.

K-framework. The K-framework is a coarser, rewriting-based framework [35, 40]. K-framework semantics evaluate code in slightly larger, higher-level chunks rather than atomic steps. Table 2 shows a subset of the K-framework rules used in our experiments, paralleling the $\mathbb{S}$ rules.

Like the $\mathbb{S}$ formalism, Rules 1 and 2 from $\mathbb{K}$ form a pair describing variable lookup. Rule 1 defines successful evaluation when a variable maps to a value in the current program state, and Rule 2 states that attempting to evaluate an undeclared variable stops execution. Likewise, Rules 6 and 7 (Table 2) parallel $\mathbb{S}$ Rules 18 and 19 (Table 1) to handle division. Rule 6 defines valid integer division when the right operand is not zero, and Rule 7 defines the error for division by zero.

The primary difference between the two formalisms is how they handle intermediate computations, which is best illustrated by the addition operation. In the $\mathbb{S}$ formalism, Rule 9 dictates the addition of two values, but its strictly small-step nature means this rule only applies after separate, explicitly defined atomic steps have reduced the operands into discrete values first. In contrast, the $\mathbb{K}$ semantics uses its

Table 1. Some $\mathbb{S}$-semantics rules of the C* language.

Rule	Formalization	Description
Rule 1	$\frac{\sigma(x) = v}{\langle x, \sigma, \chi\rangle \rightarrow v}$	If a variable is associated with a value in the current program state, then evaluating it yields that value.
Rule 2	$\frac{\sigma(x) = \bot}{\langle x, \sigma, \chi\rangle \rightarrow \langle \texttt{ERROR}, \sigma, \chi\rangle}$	If a variable has no associated value in the current program state, then attempting to evaluate it results in an error and the program execution stops immediately.
Rule 9	$\frac{v3 = v1 \ + \ v2}{\langle v1 \ + \ v2, \sigma, \chi\rangle \rightarrow v3}$	When both operands of a '+' expression are values, the result is obtained by adding those two values.
Rule 18	$\frac{v2 \neq 0 \qquad v3 = v1 \ / \ v2}{\langle v1 \ / \ v2, \sigma, \chi\rangle \rightarrow v3}$	When both operands of a '/' expression are values and the right operand value is not zero, the result is obtained by integer division of the left operand value by the right operand value.
Rule 19	$\frac{v2 = 0}{\langle v1 \ / \ v2, \sigma, \chi\rangle \rightarrow \langle \texttt{ERROR}, \sigma, \chi\rangle}$	When both operands of a '/' expression are values and the right operand value is zero, then an error occurs and the program execution stops immediately.

Table 2. Some $\mathbb{K}$-semantics rules of the C* language.

Rule	Formalization	Description
Rule 1	<k> X:Id ⇒ I . . . </k> <state> . . . X ↦ I . . . </state> <rules> L ⇒ L ListItem("Rule 1") </rules>	If a variable is associated with a value in the current program state, then evaluating it yields that value.
Rule 2	<k> X:Id ⇒ {ERROR} </k> <state> Rho:Map </state> <rules> L ⇒ L ListItem("Rule 2") </rules> requires notBool (X in_keys(Rho))	If a variable has no associated value in the current program state, then attempting to evaluate it results in an error and the program execution stops immediately.
Rule 3	<k> I1 {PLUS_OP} I2 ⇒ I1 + Int I2 . . . </k> <rules> L ⇒ L ListItem("Rule 3") </rules>	When both operands of a '+' expression are values, the result is obtained by adding those two values.
Rule 6	<k> I1 {DIV_OP} I2 ⇒ I1 / Int I2 . . . </k> <rules> L ⇒ L ListItem("Rule 6") </rules> requires I2 ≠ Int 0	When both operands of a '/' expression are values and the right operand value is not zero, the result is obtained by integer division of the left operand value by the right operand value.
Rule 7	<k> I1 {DIV_OP} I2 ⇒ {ERROR} . . . </k> <rules> L ⇒ L ListItem("Rule 7") </rules> requires I2 = Int 0	When both operands of a '/' expression are values and the right operand value is zero, then an error occurs and the program execution stops immediately.

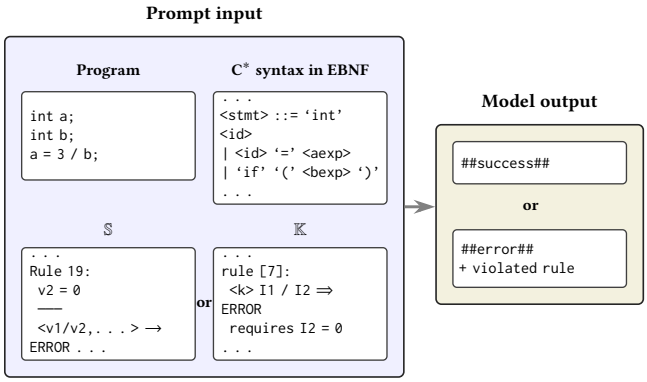


Figure 3. Overview of the PrEx prompt: the model receives the C* grammar, semantic rules ($\mathbb{S}$ or $\mathbb{K}$), and a program, and must predict executability.

built-in rewriting to handle this operand reduction implicitly. So, the corresponding Rule 3 for $\mathbb{K}$ encapsulates the addition evaluation into a single, coarser rewrite step, evaluating the expression once both operands are values.

3.3 Semantic Shifts

We use two semantic shifts, KeywordSwap and KeywordObf, to test whether models can trace programs using explicitly provided rules rather than relying on their prior knowledge.

KeywordSwap. KeywordSwap reassigns the semantic meanings of selected operator symbols to their counterparts (Table 3). For example, addition (+) and subtraction (-) are interchanged, such that the + symbol dictates subtraction. To reason correctly, models must strictly condition on the provided rules and override pre-trained semantic associations.

KeywordObf. KeywordObf replaces standard operator and keyword symbols with novel single-token symbols (Table 3). For instance, an expression using the ★ symbol is executed as addition. The KeywordObf condition ensures models can apply explicitly defined rules in the absence of recognizable patterns from prior exposure. Unlike PLSemanticsBench whose symbols are tokenized into multiple tokens by LLMs, we use a different set of symbols that are each one token. Ensuring each novel symbol maps to a single token prevents artificial inflation of the prompt length, reduces overall token overhead, and minimizes the risk of introducing unintended syntactic complexities that might confuse the model.

3.4 Full Prompt

For each program, we provide the model with the C* syntax in EBNF, the semantic rules ($\mathbb{S}$ or $\mathbb{K}$), and the program itself. Figure 3 summarizes this prompt structure. The model must predict whether the program is executable (`##success##`) or semantically invalid (`##error##`), and when invalid, identify the violated rule. For non-reasoning models, we evaluate both direct-answer and CoT using only variants with minor changes to this prompt (CoT is told to provide the reasoning).

Table 3. Transformations applied to the standard semantics to derive the nonstandard KeywordSwap and KeywordObf.

Type	Assignment	Arithmetic					Relational						Logical			Keyword				
Standard	=	+	-	*	/	%	<	<=	>	>=	==	!=	!	&&	||	if-else	while	halt	break	continue
KeywordSwap*	=	-	+	/	*	%	>	>=	<	<=	!=	==	!	||	&&	if-else	while	halt	break	continue
KeywordObf**	€	★	▲	$	■	▼	•	▽	◎	♡	♦	▶	◯	◇	△	▹-¥	₪	₹	✓	✓

* Swaps the semantics of standard operator/keyword symbols; ** Assigns semantics of standard operators/keywords to novel single-token symbols.

Table 4. Program size in lines of code (LOC) and token counts measured by the GPT-4o tokenizer per dataset split.

		LOC			Tokens		
Split	**# programs**	**Min**	**Median**	**Max**	**Min**	**Median**	**Max**
Human-Written	162	4	19	80	18	81	351
LLM-Translated	165	8	106	597	33	538	2,092
Fuzzer-Generated	164	85	786	1,961	982	9,081	22,239

Table 5. Introduced semantic errors and corresponding rules.

Executability	**Semantic error type**	**Corresponding Rule**		**Count**	**Percentage**
		$\mathbb{K}$	$\mathbb{S}$		
success	–	–	–	491	16.7%
error	Break outside loop	34	73	491	16.7%
error	Continue outside loop	31	76	491	16.7%
error	Divide by zero	7	19	491	16.7%
error	Modulo by zero	9	23	491	16.7%
error	Variable use before declare	2	2	491	16.7%

4 Dataset Construction

The set of programs used in our evaluation are comprised of three splits: Human-Written, LLM-Translated, and Fuzzer-Generated. We use the same valid programs as PLSemanticsBench and extend their dataset with invalid programs, because PrEx requires both valid and invalid programs. We obtain these invalid programs by applying semantics-aware transformations to valid programs, so that each invalid counterpart is derived directly from a matched valid one with consistent style and complexity.

Human-Written. The Human-Written split of C* programs are manually adapted from C++ programs sourced from LeetCode, HumanEval, CodeContests, and MBPP [3, 7, 18, 19, 45]. For each program, a single public test case and its corresponding oracle is selected.

LLM-Translated. The LLM-Translated split of C* programs are C++ programs translated to C* by LLMs. The C++ programs are sourced from CodeForces solutions published to HuggingFace [33]. Qwen2.5-Coder 32B is prompted with the C* syntax, semantics, the C++ solution, and one public test case to generate a valid C* program, and answers are filtered for successful test execution using the K-framework.

Fuzzer-Generated. The Fuzzer-Generated split of C* programs are constructed with a depth-controlled, semantics-aware, grammar-based fuzzer [13, 42]. The fuzzer picks statements from assign, if-else, while, break, continue, halt using depth-tapered probabilites to reduce the chance of generating new nested `if/while` blocks.

Table 4 shows program length across the three splits. Median lines of code (LOC) increases from Human-Written to LLM-Translated to Fuzzer-Generated. Even the shortest Fuzzer-Generated program is longer than the longest Human-Written program. We start with 491 valid programs[††]. These programs are transformed to create 2455 semantically invalid programs. The transformation process uses an ANTLR-based parser-visitor to apply the five predefined transformations to each valid program, corresponding to the five semantic error rules under $\mathbb{S}$ and $\mathbb{K}$ formalizations. These transformations introduce specific semantic errors: 1) inserting a break outside a loop, 2) inserting a continue outside a loop, 3) division by zero, 4) modulo by zero, and 5) using an undeclared variable. Each invalid program is generated by applying one random transformation rule to a valid C* program, ensuring each violates exactly one semantic error rule. For each valid program, we generate one corresponding semantically invalid program in each of the five error categories. Figure 4 illustrates these transformations on a shared valid program.

Table 5 shows the type of semantic errors introduced, their corresponding rules under $\mathbb{S}$ and $\mathbb{K}$ formalizations, and the number of programs in the dataset that violate each rule. The percentages are computed over the full set of 2946 programs.

5 Experiments and Results

We benchmark models under three settings by providing: the standard semantics and the unmodified program, KeywordSwap semantics and the transformed program, and KeywordObf semantics and the transformed program. Ground truths labels are derived from the transformation procedure. Valid programs are labeled ##success##, and each semantically invalid variant is labeled ##error## with the corresponding violated rule. We evaluate open-source LLMs designed for coding tasks and enhanced reasoning ability on

[††]The original PLSemanticsBench set had 492 valid programs, but we removed one Fuzzer-Generated program because its full PrEx prompt (C* syntax, semantic rules, and program) requires ~37K tokens and exceeds the 32K context window of our Qwen2.5-Coder models, leaving 491 semantically valid programs.

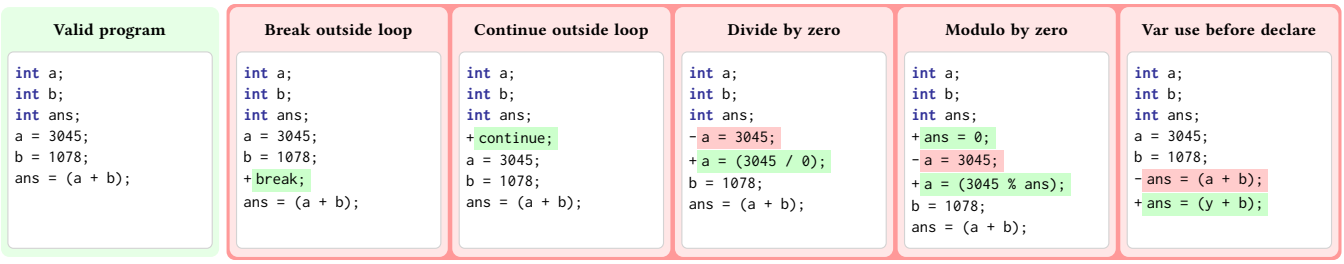


Figure 4. Example of a valid program and semantically invalid programs with their corresponding error categories.

PrEx, including models in the Qwen [16], DeepSeek [12], and Ministral [21] families. For non-reasoning models, we experiment with both CoT and non-CoT prompting. We run each model three times per configuration and report the mean accuracy across runs. Configurations are

$$\{\mathbb{S}, \mathbb{K}\} \times \{\mathit{Standard}, \mathit{KeywordSwap}, \mathit{KeywordObf}\} \times \{\text{non-}\mathit{CoT}, \mathit{CoT}\}.$$

Because reasoning models generate chain-of-thought traces by default, we evaluate them under one prompting condition only and do not run separate CoT and non-CoT variants.

Tables 6–8 summarize model performance on PrEx across the three dataset splits.

5.1 Performance Across Dataset Splits

Across all models, performance consistently drops between Human-Written and the other two splits, especially Fuzzer-Generated.

Each of Tables 6, 7, and 8 reports PrEx accuracy on one dataset split (Human-Written, LLM-Translated, and Fuzzer-Generated, respectively). Within each table, results are organized by semantics formalism: $\mathbb{K}$ semantics and $\mathbb{S}$ semantics (two groups of columns). For each formalism, we show accuracy under the three semantic shift configurations: Standard, KeywordSwap, and KeywordObf. Standard uses the unmodified programs; KeywordSwap and KeywordObf use the corresponding modified programs with swapped symbols and new symbols, respectively. For KeywordSwap and KeywordObf, we show, in parenthesis, percentage drop/increase compared to the Standard shift.

DeepSeek-Qwen 32B, Ministral 3 14B-CoT, and Qwen2.5-Coder 32B-CoT achieve the strongest Human-Written accuracy among all evaluated models (Table 6), but all three still degrade on other splits. On LLM-Translated programs (Table 7), mean accuracy—averaged across the six $\mathbb{K}/\mathbb{S}$ and semantic-shift columns in each model's row—falls by 2.9, 8.7, and 8.6 percentage points (pp) respectively when comparing to Human-Written (Table 6). On Fuzzer-Generated programs (Table 8) the decline is even sharper, with mean drops of 19pp, 24.8pp, and 33.3pp computed the same way across Tables 8 and 6. The largest single-configuration drop reaches 55pp for Qwen2.5-Coder 14B-CoT on KeywordObf semantics under $\mathbb{K}$ where it falls from 78% to 23% (Qwen2.5-Coder 14B-CoT row and KeywordObf column under $\mathbb{K}$ semantics in Tables 6 and 8).

This performance drop extends beyond the best three performing models: every "capable model" (mean Human-Written accuracy $\geq 45\%$) loses substantial accuracy on Fuzzer-Generated programs, with mean drops ranging from 13pp to 40pp (median 24pp). Qwen2.5-Coder 14B has a smaller absolute drop under non-CoT which drops by only 13.5pp on average (the Qwen2.5-Coder 14B row averaged across all six $\mathbb{K}/\mathbb{S}$ and semantic-shift columns when comparing Tables 8 and 6), but its Human-Written accuracy is much lower (mean 53.8%, the same six-column average in Table 6). Somewhat differently, DeepSeek-Qwen 14B does not perform as well on Human-Written (mean 77.5% vs. 88.7%–82.5%; each the six-column row average in Table 6) yet its mean Fuzzer-Generated drop is 3.2pp below the top-three average (again averaged over the six columns in Tables 8 and 6), and no configuration falls more than 29pp (its largest drop is at Table 6/DeepSeek-Qwen 14B/$\mathbb{K}$/KeywordObf, from 81% to 51% in Table 8, vs. up to 55pp at Table 6/Qwen2.5-Coder 14B-CoT/$\mathbb{K}$/KeywordObf above).

5.2 Performance on KeywordSwap and KeywordObf

Moving from Standard to KeywordSwap or KeywordObf semantics substantially reduces accuracy across models. On Human-Written programs (Table 6), KeywordObf is consistently harder than KeywordSwap: across all models, median accuracy falls by 19pp under KeywordSwap versus 32pp under KeywordObf (each median is the drop from Standard to KeywordSwap or KeywordObf, taken over all model rows and $\mathbb{K}/\mathbb{S}$ formalisms in Table 6).

The top three Human-Written models (DeepSeek-Qwen 32B, Ministral 3 14B-CoT, and Qwen2.5-Coder 32B-CoT) incur mean semantic-shift drops of 15.2pp, 14.8pp, and 23.3pp, respectively (each averaged over $\{\mathbb{K}, \mathbb{S}\} \times \{$KeywordSwap, KeywordObf$\}$ in Table 6).

The largest Human-Written drop reaches 59pp (Ministral 3 14B under $\mathbb{K}$, KeywordObf falls from 90% to 32%, Table 6), and the largest drop among the top three is Qwen2.5-Coder

Table 6. PrEx accuracy on Human-Written programs.

Models	$\mathbb{K}$-semantics			$\mathbb{S}$-semantics		
	Standard	KeywordSwap (pp)	KeywordObf (pp)	Standard	KeywordSwap (pp)	KeywordObf (pp)
Random	18	18 (000)	16 (−02)	18	18 (000)	16 (−02)
Ministral 3 3B	61	47 (−14)	14 (−47)	46	41 (−05)	21 (−25)
Ministral 3 8B	77	66 (−11)	24 (−53)	69	55 (−14)	20 (−49)
Ministral 3 14B	90	75 (−15)	32 (−58)	87	**78** (−09)	45 (−42)
Qwen2.5-Coder 3B	44	23 (−21)	7 (−37)	23	28 (+05)	18 (−05)
Qwen2.5-Coder 7B	45	22 (−23)	6 (−39)	55	35 (−20)	21 (−34)
Qwen2.5-Coder 14B	64	41 (−23)	26 (−38)	83	74 (−09)	37 (−46)
Qwen2.5-Coder 32B	70	57 (−13)	39 (−31)	92	71 (−21)	45 (−47)
DeepSeek-Qwen 14B	91	69 (−22)	81 (−10)	92	71 (−21)	61 (−31)
DeepSeek-Qwen 32B	**99**	77 (−22)	**98** (−01)	98	71 (−27)	88 (−10)
Ministral 3 3B-CoT	94	76 (−18)	54 (−40)	89	75 (−14)	64 (−25)
Ministral 3 8B-CoT	**99**	79 (−20)	77 (−22)	97	**78** (−19)	64 (−33)
Ministral 3 14B-CoT	**99**	79 (−20)	91 (−08)	**99**	**78** (−21)	**89** (−10)
Qwen2.5-Coder 3B-CoT	32	25 (−07)	16 (−16)	33	27 (−06)	17 (−16)
Qwen2.5-Coder 7B-CoT	69	49 (−20)	32 (−37)	71	46 (−25)	26 (−45)
Qwen2.5-Coder 14B-CoT	96	74 (−22)	78 (−18)	93	71 (−22)	55 (−38)
Qwen2.5-Coder 32B-CoT	**99**	**81** (−18)	72 (−27)	97	**78** (−19)	68 (−29)

32B-CoT on $\mathbb{S}$, KeywordObf, which loses 29pp (from 97% to 68%, Table 6). Interestingly, DeepSeek-Qwen 32B is an outlier under KeywordObf on $\mathbb{K}$, dropping only 2pp (from 99 to 98, Table 6) while its $\mathbb{S}$ drop is 10pp (Table 6).

On Fuzzer-Generated programs (results shown in Table 8), absolute semantic-shift drops remain large for the strongest models (e.g., Qwen2.5-Coder 32B-CoT loses 38.3pp on average), but the KeywordObf versus KeywordSwap gap narrows to 5.3pp on average (means of 26.3pp vs. 21pp) because the baseline standard accuracy is already much lower.

5.3 Rule Identification Performance by Error Type

Figure 5 breaks down per-error-type accuracy on invalid C* programs under $\mathbb{S}$ for all evaluated models.

The top row shows Human-Written programs and the bottom row shows Fuzzer-Generated programs; polygons shrink substantially on the bottom row, reflecting the split-wise accuracy drop in Table 8. Each radar axis is one model; a colored polygon that stays near the outer ring indicates reliable identification of that semantic error type across models. Across configurations, keyword-dependent errors (continue-outside-loop and break-outside-loop) shrink inward more under KeywordObf than arithmetic and scoping errors (divide-by-zero, modulo-by-zero, and variable-use-before-declare). Under KeywordSwap, divide-by-zero accuracy drops relative to Standard semantics, suggesting pretraining bias toward standard division.

5.4 Things That the Models Get Right

On Human-Written programs (Table 6), all three models are already strong under Standard (Table 6), and our inspection confirms that most remaining failures are subtle rather than a significant misunderstanding of the programs. When a model does fail on Human-Written, it typically still recognizes the relevant error category (e.g., that a break statement is illegal outside a loop), even if it cites the wrong rule number within that category.

5.5 Main Failure Modes by Dataset Split

The main failure modes differ by dataset split in ways that explain the accuracy drops reported in the previous subsections. Across all 6 configurations, Human-Written failures are relatively rare (661-1028 incorrect predictions per model out of 5,832 programs) and are split between false successes (509–779) and wrong-rule confusions (63–169). In contrast, Fuzzer-Generated and LLM-Translated failures are both more frequent and structurally different.

Human-Written. False success is the most common failure mode for DeepSeek-Qwen 32B and Qwen2.5-Coder 32B-CoT on Human-Written (549 and 779 cases respectively, aggregated across configurations). Wrong-rule errors are less frequent but when they do occur, models confuse nearby rules within the same error family, especially continue-outside-loop vs. break-outside-loop (Rules 76 vs. 73 under $\mathbb{S}$) and adjacent arithmetic-error rules.

LLM-Translated and Fuzzer-Generated. On programs from the LLM-Translated split (Table 7), false success is very frequent. For example, DeepSeek-Qwen 32B produces 695 false-success errors out of 822 total failures across configurations, suggesting that models often treat translated C-like code as executable even when it violates the supplied C* semantics. On Fuzzer-Generated programs (Table 8), false success remains common (372–1484 cases per model), but wrong-rule (597–1003) and false-error (341–453) predictions become much more frequent as well. The most common missed rules are modulo-by-zero, break-outside-loop, and variable-use-before-declare. Malformed outputs are also

Table 7. PrEx accuracy on LLM-Translated programs.

Models	$\mathbb{K}$-semantics			$\mathbb{S}$-semantics		
	Standard	KeywordSwap (pp)	KeywordObf (pp)	Standard	KeywordSwap (pp)	KeywordObf (pp)
Random	16	16 (000)	16 (000)	16	16 (000)	16 (000)
Ministral 3 3B	44	33 (−11)	10 (−34)	41	31 (−10)	18 (−23)
Ministral 3 8B	74	59 (−15)	22 (−52)	62	48 (−14)	19 (−43)
Ministral 3 14B	80	69 (−11)	29 (−51)	76	68 (−08)	32 (−44)
Qwen2.5-Coder 3B	35	18 (−17)	16 (−19)	21	23 (+02)	18 (−03)
Qwen2.5-Coder 7B	44	18 (−26)	8 (−36)	44	25 (−19)	19 (−25)
Qwen2.5-Coder 14B	55	39 (−16)	20 (−35)	74	69 (−05)	29 (−45)
Qwen2.5-Coder 32B	60	45 (−15)	29 (−31)	88	65 (−23)	38 (−50)
DeepSeek-Qwen 14B	90	71 (−19)	77 (−13)	88	69 (−19)	60 (−28)
DeepSeek-Qwen 32B	**98**	75 (−23)	**92** (−06)	**97**	70 (−27)	**83** (−14)
Ministral 3 3B-CoT	88	67 (−21)	42 (−46)	88	69 (−19)	48 (−40)
Ministral 3 8B-CoT	96	**78** (−18)	66 (−30)	96	**78** (−18)	54 (−42)
Ministral 3 14B-CoT	97	**78** (−19)	81 (−16)	88	71 (−17)	68 (−20)
Qwen2.5-Coder 3B-CoT	29	18 (−11)	14 (−15)	31	25 (−06)	16 (−15)
Qwen2.5-Coder 7B-CoT	58	36 (−22)	23 (−35)	59	37 (−22)	26 (−33)
Qwen2.5-Coder 14B-CoT	86	68 (−18)	48 (−38)	85	67 (−18)	37 (−48)
Qwen2.5-Coder 32B-CoT	95	75 (−20)	52 (−43)	95	77 (−18)	49 (−46)

Table 8. PrEx accuracy on Fuzzer-Generated programs.

Models	$\mathbb{K}$-semantics			$\mathbb{S}$-semantics		
	Standard	KeywordSwap (pp)	KeywordObf (pp)	Standard	KeywordSwap (pp)	KeywordObf (pp)
Random	17	17 (000)	16 (−01)	17	17 (000)	16 (−01)
Ministral 3 3B	24	26 (+02)	14 (−10)	29	17 (−12)	15 (−14)
Ministral 3 8B	43	30 (−13)	17 (−26)	35	26 (−09)	17 (−18)
Ministral 3 14B	65	37 (−28)	24 (−41)	71	31 (−40)	27 (−44)
Qwen2.5-Coder 3B	19	16 (−03)	12 (−07)	20	20 (000)	19 (−01)
Qwen2.5-Coder 7B	21	17 (−04)	16 (−05)	17	17 (000)	17 (000)
Qwen2.5-Coder 14B	58	30 (−28)	25 (−33)	65	39 (−26)	26 (−39)
Qwen2.5-Coder 32B	51	26 (−25)	28 (−23)	69	36 (−33)	34 (−35)
DeepSeek-Qwen 14B	78	43 (−35)	51 (−27)	78	42 (−36)	37 (−41)
DeepSeek-Qwen 32B	**89**	**58** (−31)	**68** (−21)	**90**	**57** (−33)	**57** (−33)
Ministral 3 3B-CoT	55	29 (−26)	28 (−27)	49	30 (−19)	24 (−25)
Ministral 3 8B-CoT	82	48 (−34)	40 (−42)	82	45 (−37)	37 (−45)
Ministral 3 14B-CoT	88	54 (−34)	66 (−22)	76	51 (−25)	52 (−24)
Qwen2.5-Coder 3B-CoT	20	14 (−06)	14 (−06)	22	18 (−04)	13 (−09)
Qwen2.5-Coder 7B-CoT	47	31 (−16)	22 (−25)	38	25 (−13)	25 (−13)
Qwen2.5-Coder 14B-CoT	68	47 (−21)	23 (−45)	69	45 (−24)	22 (−47)
Qwen2.5-Coder 32B-CoT	73	48 (−25)	24 (−49)	77	44 (−33)	29 (−48)

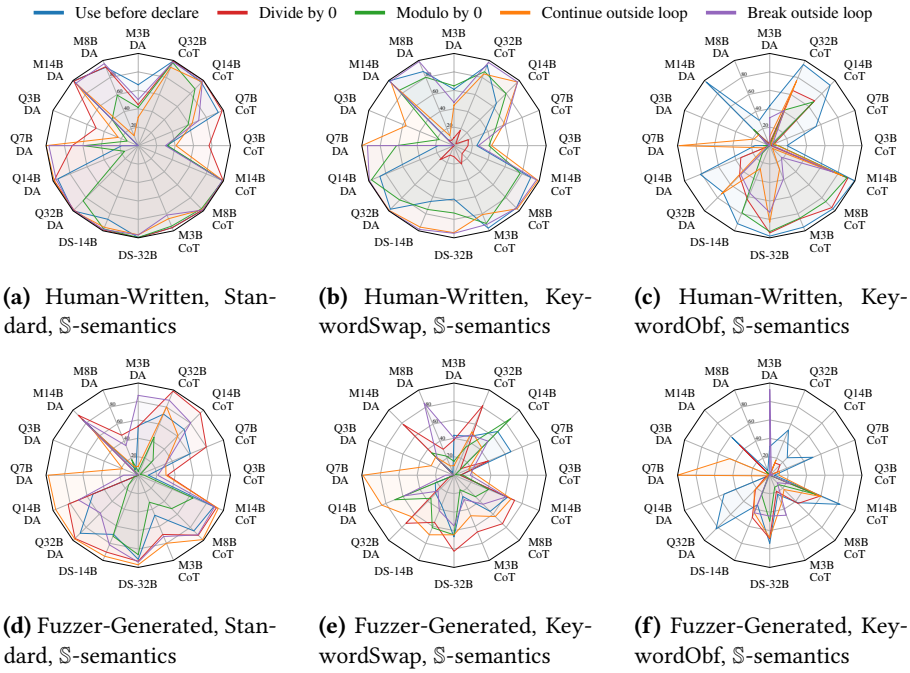
more frequent on LLM-Translated and Fuzzer-Generated, especially for Ministral 3 14B-CoT.

Overall, PrEx accuracy is highest on short Human-Written programs under Standard semantics but drops sharply on Fuzzer-Generated and LLM-Translated splits, under KeywordSwap and KeywordObf shifts, and on keyword-based error types such as continue-outside-loop and break-outside-loop. The split-wise and semantic-shift declines in Tables 6–8 and Figure 5 coincide with systematic failure modes. On Human-Written, remaining errors are relatively rare and tend toward wrong-rule confusions within the same error family; on Fuzzer-Generated and LLM-Translated, false success is more common, with models frequently declaring invalid programs executable despite the supplied C* semantics. Taken together, these results suggest that current LLMs lean on pre-training priors rather than systematically applying the provided formal semantics even for execution prediction.

6 Qualitative Study

The quantitative results in Section 5 report rule-level accuracy aggregated over all programs and runs. To understand *how* models fail, we complement those aggregates with a qualitative pass over the three strongest PrEx models (DeepSeek-Qwen 32B, Qwen2.5-Coder 32B-CoT, and Ministral 3 14B-CoT). We only consider programs in the Human-Written split. We classify every misprediction into one of 4 failure modes:

- **False success:** predicts `##success##` for invalid program.
- **False error:** predicts `##error##` for valid program.
- **Wrong rule:** correctly predicts invalidity but cites a different violated rule than the ground truth.
- **Malformed output:** the model response does not contain a parseable `<ans>` tag.



(a) Human-Written, Standard, $\mathbb{S}$-semantics

(b) Human-Written, KeywordSwap, $\mathbb{S}$-semantics

(c) Human-Written, KeywordObf, $\mathbb{S}$-semantics

(d) Fuzzer-Generated, Standard, $\mathbb{S}$-semantics

(e) Fuzzer-Generated, KeywordSwap, $\mathbb{S}$-semantics

(f) Fuzzer-Generated, KeywordObf, $\mathbb{S}$-semantics

Figure 5. Per-error-type accuracy on invalid C* programs across all evaluated models under $\mathbb{S}$-semantics. The top row shows Human-Written programs and the bottom row shows Fuzzer-Generated programs (Standard, KeywordSwap, and KeywordObf configurations). Radar axes are models and each colored polygon is a semantic error type. Radial grid lines mark accuracy percentages, using the same rule-level criterion as Tables 6 and 8.

Table 9. Minimal Human-Written PrEx failures under $\mathbb{S}$-semantics (seed 42). Both programs are 5 lines long. Highlighted lines introduce the semantic error. Even on these small examples, top models may predict the correct invalidity but cite the wrong rule.

Program	Expected	Model	Predicted

`int ans;`
`int a;`
`+ continue;`
`a = 156;`
`ans = (a * a);`
	Invalid; Rule 76 (continue-outside-loop)	Qwen2.5-Coder 32B-CoT	Predicted invalid but cited Rule 73 (break-outside-loop) instead of Rule 76 (*wrong rule*)

`int ans;`
`int a;`
`+ ans = 0;`
`+ a = (156 % ans);`
`ans = (a * a);`
	Invalid; Rule 23 (modulo-by-zero)	DeepSeek-Qwen 32B	Predicted invalid by modulo-by-zero, but cited Rule 24 instead of Rule 23 (*wrong rule*)

This taxonomy uses the same rule-level criterion as the main accuracy tables: a prediction counts as correct only when both executability *and* the violated rule (on invalid programs) match the ground truth.

For each of the 6 PrEx configurations ($\mathbb{K}/\mathbb{S}$ × Standard/KeywordSwap/KeywordObf), we select the 2 smallest (by line count) non-empty failing programs for each top model and inspect the full model responses. The goal is not to cherry-pick hard cases, but to examine minimal counterexamples of programs where even short, explicit code still triggers systematic errors. Table 9 shows 2 of the shortest Human-Written failures under $\mathbb{S}$-semantics (5 lines each). Both programs are semantically invalid for a single, localized reason; neither requires deep control-flow reasoning.

In the first example, `continue` appears at the top level with an empty control stack, violating Rule 76. DeepSeek-Qwen 32B and Ministral 3 14B-CoT identify this correctly, but Qwen2.5-Coder 32B-CoT predicts invalid while citing Rule 73 (break-outside-loop) instead. This matches the per-error-type pattern in Figure 5, where continue-outside-loop and break-outside-loop identification degrades more than arithmetic errors.

In the second example, `ans` is initialized to 0 and then used as a modulo divisor. All three models predict invalid and the correct error type (modulo-by-zero), but DeepSeek-Qwen 32B cites Rule 24 rather than Rule 23 (Rule 24 is an unrelated unary expression rule). The model therefore applies the right semantic reasoning (modulo-by-zero) but maps it to a neighboring rule in the $\mathbb{S}$ specification. Such off-by-one rule citations still count as incorrect under our metric even when the natural-language explanation is essentially right.

7 Related Work

LLM-Based predictive execution. A growing body of work asks whether LLMs can simulate program execution without running code. Lyu et al. [25] treat LLMs as direct code executors, feeding snippets to the model and evaluating the returned outputs on LeetCode programs. Ni et al. [30] introduced NExT, a self-training approach that teaches LLMs to inspect execution traces and reason about run-time behavior via CoT rationales, improving program repair on MBPP and HumanEval. Li et al. [20] proposed PredEx, a predictive executor for Python that combines program analysis with LLM prompting to predict full execution traces and static runtime errors. Patel et al. [32] enabled LLMs to act as predictive interpreters by autonomously navigating a program's control-flow graph and tracking variable states at branching points to simulate execution and statically detect runtime errors. Le et al. [17] proposed CodeFlow, a learned CFG-based model that combines static control-flow structure with dynamic dependencies extracted from execution traces to predict code coverage and localize runtime errors. These methods improve predictive execution through specialized training, program analysis, or learned graph models, but they evaluate implicit language semantics on real-world programs rather than testing whether models follow explicitly supplied formal rules.

Evaluating execution and runtime behavior. Complementary work evaluates code execution reasoning through benchmarks and empirical studies on real-world programs. Gu et al. [10] introduced CRUXEval, a benchmark of short Python functions with paired input–output examples and two tasks: predicting the output given an input or the input given an output. Chen et al. [5] proposed REval, a framework that extends this line of evaluation to intermediate runtime behavior (code coverage, program state, execution path, and final output) and to incremental consistency across these dependent sub-tasks. Hora [14] studied a related applied setting, asking GPT-4 to predict pass/fail outcomes for Python Standard Library test cases without execution. These evaluations measure reasoning under familiar Python semantics; none supplies an explicit operational rule set in the prompt. Our PrEx task is simpler than REval's multi-step runtime behavior reasoning, but adds matched valid/invalid C* programs, user-provided $\mathbb{S}$ and $\mathbb{K}$ semantics, and KeywordSwap/KeywordObf shifts to diagnose whether predictions follow the given rules or priors.

Semantics, identifiers, and formal properties. Other work examines whether LLMs reason from program logic or from surface cues and formal semantic properties. Wang et al. [41] showed that replacing variable, method, and function names with nonsense or misleading identifiers substantially degrades CodeBERT performance on code analysis tasks, indicating that models rely heavily on identifier semantics rather than logic alone. Sultan et al. [37] evaluated frontier LLMs on SV-Comp termination tasks, finding that models can approach specialized verifiers in termination classification yet frequently fail to produce machine-valid witness proofs. Chen et al. [6] applied LLMs to dead code elimination, using a small classifier to locate suspect lines and a fine-tuned model to judge, explain, and patch unreachable or unused code. The naming results motivate our semantic-shift conditions: KeywordSwap and KeywordObf perturb familiar symbol meanings to test whether models condition on provided rules rather than pre-trained associations. Together with the termination and dead-code studies, they highlight that LLM "understanding" of code semantics remains fragile; PrEx offers a controlled executability baseline in which success requires applying supplied formal rules to predict whether execution succeeds or violates a specific rule.

8 Conclusion

We introduced Program Executability Prediction (PrEx), a task that asks models, when given a program and formal programming language semantics, to predict whether execution succeeds or halts on a semantic error and, when it fails, which rule was violated. Through PrEx, we can evaluate whether LLMs apply explicitly provided programming-language semantics or fall back on pre-training priors. Building on valid C* programs from PLSemanticsBench, we extended the benchmark with invalid programs produced by five semantics-aware invalid transformations, and evaluated open-source coding LLMs under $\mathbb{S}$ and $\mathbb{K}$ formalisms, KeywordSwap and KeywordObf semantic shifts, and three program data splits (Human-Written, LLM-Translated, and Fuzzer-Generated). We find that current models struggle to follow the supplied semantics. Accuracy is highest on short Human-Written programs under standard semantics, but drops sharply under semantic shifts and on longer, structurally complex LLM-Translated and Fuzzer-Generated programs. Our results show that LLMs do not yet reliably reason from supplied formal semantics and still heavily rely on patterns learned in their pre-training.

Acknowledgments

We thank Cheng Ding, Ivan Grigorik, Yan Levin, Tong-Nong Lin, Karl Palmskog, Samuel Yuan, Linghan Zhong, and the anonymous reviewers for helpful feedback and discussions. Computational resources were provided by the Texas Advanced Computing Center (TACC[†]) at the University of Texas at Austin, and AMD (University Program AI & HPC Cluster). This work was supported in part by the U.S. National Science Foundation (NSF) Nos. CCF-2217696, CCF-2313027, CCF-2403036; the NSF–Simons AI Institute for Cosmic Origins (CosmicAI[‡]) funded by NSF award AST-2421782; the Simons Foundation (MPS-AI-00010515); and a sponsored

research award from Cisco. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring entities.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Andrew W. Appel. 1997. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, USA.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021). doi:10.48550/arXiv.2108.07732

[4] Shir Bernstein, David Beste, Daniel Ayzenshteyn, Lea Schonherr, and Yisroel Mirsky. 2026. Trust me, i know this function: hijacking LLM static analysis using bias. In *NDSS*. doi:10.14722/ndss.2026.242066

[5] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning runtime behavior of a program with LLM: How far are we?. In *ICSE*. 1869–1881. doi:10.1109/icse55347.2025.00012

[6] Minyu Chen, Guoqiang Li, Ling-I Wu, and Ruibang Liu. 2025. DCE-LLM: Dead code elimination with large language models. In *NAACL*. 9942–9955. doi:10.18653/v1/2025.naacl-long.501

[7] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374

[8] Djiré Albérick Euraste, Abdoul Kader Kaboré, Jordan Samhi, Earl T. Barr, Jacques Klein, and Tegawendé F. Bissyandé. 2026. Learned or memorized? Quantifying memorization advantage in code LLMs. In *ICSE*. doi:10.48550/arXiv.2604.13997

[9] Gerhard Gentzen. 1964. Investigations into logical deduction. *American philosophical quarterly* 1, 4 (1964), 288–306.

[10] Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. In *ICML*. 16568–16621. doi:10.48550/arXiv.2401.03065

[11] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. LongCoder: a long-range pre-trained language model for code completion. In *ICML*. doi:10.48550/arXiv.2306.14893

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *Nature* (2025), 633–638. doi:10.1038/s41586-025-09422-z

[13] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *NDSS*. doi:10.14722/ndss.2019.23263

[14] Andre Hora. 2024. Predicting test results without execution. In *FSE Companion*. 542–546. doi:10.1145/3663529.3663794

[15] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A deep dive into large language models for automated bug localization and repair. FSE (2024), 1471 – 1493. doi:10.1145/3660773

[16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024). doi:10.48550/arXiv.2409.12186

[17] Cuong Chi Le, Hoang Nhat Phan, Huy Nhat Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. CodeFlow: Program behavior prediction with dynamic dependencies learning. In *FORGE*. 192–203. doi:10.1109/forge66646.2025.00029

[18] LeetCode. 2024. LeetCode Online Judge. Accessed: 2025-05-16. https://leetcode.com

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* (2022), 1092–1097. doi:10.1126/science.abq1158

[20] Yi Li, Hridya Dhulipala, Aashish Yadavally, Xiaokai Rong, Shaohua Wang, and Tien N. Nguyen. 2025. Blended analysis for predictive execution. (2025), 2987 – 3008. doi:10.1145/3729402

[21] Alexander H. Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, Alexandre Sablayrolles, Amélie Héliou, Amos You, Andy Ehrenberg, Andy Lo, Anton Eliseev, Antonia Calvi, Avinash Sooriyarachchi, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Clémence Lanfranchi, Corentin Barreau, Cyprien Courtot, Daniele Grattarola, Darius Dabert, Diego de las Casas, Elliot Chane-Sane, Faruk Ahmed, Gabrielle Berrada, Gaëtan Ecrepont, Gauthier Guinet, Georgii Novikov, Guillaume Kunsch, Guillaume Lample, Guillaume Martin, Gunshi Gupta, Jan Ludziejewski, Jason Rute, Joachim Studnia, Jonas Amar, Joséphine Delas, Josselin Somerville Roberts, Karmesh Yadav, Khyathi Chandu, Kush Jain, Laurence Aitchison, Laurent Fainsin, Léonard Blier, Lingxiao Zhao, Louis Martin, Lucile Saulnier, Luyu Gao, Maarten Buyl, Margaret Jennings, Marie Pellat, Mark Prins, Mathieu Poirée, Mathilde Guillaumin, Matthieu Dinot, Matthieu Futeral, Maxime Darrin, Maximilian Augustin, Mia Chiquier, Michel Schimpf, Nathan Grinsztajn, Neha Gupta, Nikhil Raghuraman, Olivier Bousquet, Olivier Duchenne, Patricia Wang, Patrick von Platen, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Quentin Torroba, Romain Sauvestre, Roman Soletskyi, Rupert Menneer, Sagar Vaze, Samuel Barry, Sanchit Gandhi, Siddhant Waghjale, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Teven Le Scao, Théo Cachet, Theo Simon Sorg, Thibaut Lavril, Thiziri Nait Saada, Thomas Chabal, Thomas Foubert, Thomas Robert, Thomas Wang, Tim Lawson, Tom Bewley, Tom Bewley, Tom Edwards, Umar Jamil, Umberto Tomasini, Valeriia Nemychnikova, Van Phung, Vincent Maladière, Virgile Richard, Wassim Bouaziz, Wen-Ding Li, William Marshall, Xinghui Li, Xinyu Yang, Yassine El Ouahidi, Yihan Wang, Yunhao Tang, and Zaccharie Ramzi. 2026. Ministral 3. doi:10.48550/arXiv.2601.08584

[22] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. MarsCode agent: AI-native automated bug fixing. doi:10.48550/arXiv.2409.00899

[23] Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. 2024. ProConSuL: Project context for code summarization with LLMs. In *EMNLP*. 866–880. doi:10.18653/v1/2024.emnlp-industry.65

[24] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *ACL*. 6227–6240. doi:10.18653/v1/2022.acl-long.431

[25] Chenyang Lyu, Lecheng Yan, Rui Xing, Wenxi Li, Younes Samih, Tianbo Ji, and Longyue Wang. 2024. Large language models as code executors: an exploratory study. doi:10.48550/arXiv.2410.06667

[†] https://www.tacc.utexas.edu/

[‡] https://www.cosmicai.org/

[26] Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. 2025. INTERTRANS: Leveraging transitive intermediate translations to enhance LLM-based code translation. In *ICSE*. 1153–1164. doi:10.1109/ICSE55347.2025.00236

[27] Vladimir Makharev and Vladimir Ivanov. 2025. Code summarization beyond function level. In *LLM4Code*. 153–160. doi:10.1109/LLM4Code66737.2025.00024

[28] Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Code simulation challenges for large language models. doi:10.48550/arXiv.2401.09074

[29] Matthieu Meeus, Anil Ramakrishna, Matthew Grange, Zheng Xu, and Luca Melis. 2026. Detecting functional memorization in code language models. In *ICML*. doi:10.48550/arXiv.2606.12764

[30] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NEXT: teaching large language models to reason about code execution. In *ICML*. 37929 – 37956. doi:10.48550/arXiv.2404.14662

[31] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *ICLR*. doi:10.48550/arXiv.2203.13474

[32] Smit Patel, Aashish Yadavally, Hridya Dhulipala, and Tien N. Nguyen. 2025. Planning a large language model for static detection of runtime errors in code snippets. In *ICSE*. 872–884. doi:10.1109/ICSE55347.2025.00102

[33] Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. CodeForces. https://huggingface.co/datasets/open-r1/codeforces.

[34] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* (2004), 17–139.

[35] Grigore Roșu and Traian Florin Șerbănuță. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[36] Adam Štorek, Mukur Gupta, Samira Hajizadeh, Prashast Srivastava, and Suman Jana. 2026. Sense and sensitivity: examining the influence of semantic recall on long context code understanding. In *ACL*. 480–498. doi:10.18653/v1/2026.acl-long.19

[37] Oren Sultan, Jordi Armengol-Estapé, Pascal Kesseli, Julien Vanegue, Dafna Shahaf, Yossi Adi, and Peter O'Hearn. 2026. LLMs versus the halting problem: Characterizing program termination reasoning. In *EMNLP*. doi:10.48550/arXiv.2601.18987

[38] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source code summarization in the era of large language models. In *ICSE*. 1882–1894. doi:10.1109/ICSE55347.2025.00034

[39] Aditya Thimmaiah, Jiyang Zhang, Jayanth Srinivasa, Junyi Jessy Li, and Milos Gligoric. 2026. LLMs lean on priors, not programming language semantics. doi:10.48550/arXiv.2510.03415

[40] Runtime Verification. 2025. K: A Framework for Formal Semantics. Accessed: 2025-03-25. https://github.com/runtimeverification/k

[41] Zhilong Wang, Lan Zhang, Chen Cao, Nanqing Luo, Xinzhi Luo, and Peng Liu. 2024. How does naming affect LLMs on code analysis tasks?. In *JSEA*. 803–816. doi:10.4236/jsea.2024.1711044

[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283—-294.

[43] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. In *FSE*. 1585–1608. doi:10.1145/3660778

[44] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. 2024. Unveiling memorization in code models. In *ICSE*. 1 – 13. doi:10.1145/3597503.3639074

[45] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *KDD*. 5673–5684. doi:10.1145/3580305.3599790

Received 2026-07-10; accepted 2026-08-14