Diagnosing CFG Interpretation in LLMs

Anonymous ACL submission

Abstract

As LLMs are increasingly integrated into agentic systems, they must adhere to dynamically defined, machine-interpretable interfaces. We evaluate LLMs as in-context interpreters: *given a novel context-free grammar, can LLMs generate syntactically valid, behaviorally functional, and semantically faithful outputs?* We introduce RoboGrid, a framework that disentangles **syntax**, **behavior**, and **semantics** through controlled stress-tests of recursion depth, expression complexity, and surface styles. Our experiments reveal a consistent hierarchical degradation: LLMs often maintain surface syntax but fail to preserve structural semantics. Despite the partial mitigation provided by CoT reasoning, performance collapses under structural density, specifically deep recursion and high branching, with semantic alignment vanishing at extreme depths. Furthermore, "`Alien`" lexicons reveal that LLMs rely on semantic bootstrapping from keywords rather than pure symbolic induction. These findings pinpoint critical gaps in hierarchical state-tracking required for reliable, grammar-agnostic agents.

1 Introduction

Large language models (LLMs) are increasingly deployed as autonomous agents that interact with their environment through tool-calling (Yao et al., 2022), API orchestration (Hou et al., 2025), and the generation of domain-specific languages (Mazrouei, 2025). In these agentic workflows, the LLM's output must conform to rigid, machine-interpretable interfaces (Shorten et al., 2024), such as JSON schemas or function signatures defined dynamically within the prompt (Geng et al., 2025). This practical requirement implicitly asks LLMs to function as **in-context grammar interpreters**: given a novel context-free grammar (CFG) (Chomsky and Schützenberger, 1959) specification at inference time, can a LLM produce outputs that are syntactically valid, behaviorally functional, and semantically faithful?

While LLMs show apparent fluency in common languages like Python (Jain et al., 2024), much of this competence may be attributable to extensive pre-training exposure (Liang et al., 2025). This raises a fundamental challenge for agentic reliability: *do LLMs truly internalize the formal logic of a provided grammar, or are they merely retrieving surface regularities from memory?* In real-world tool-use, "almost correct" is insufficient (Yehudai et al., 2025). A single missing delimiter or a logical misalignment in a nested loop can lead to catastrophic execution failure (Dou et al., 2024). To build robust agents, we must understand the LLM's capacity for **systematic generalization**: the ability to induce and apply structural rules from novel, in-context definitions without relying on familiar semantic cues.

To rigorously investigate this, we propose a controlled evaluation hierarchy that disentangles grammar interpretation into three distinct layers: (1) **Syntax Validity**: Does the output conform to the CFG production rules? (2) **Behavioral Equivalence**: Does the executed program achieve the intended state change? (3) **Semantic Correctness**: Is the induced abstract syntax tree (AST) structurally identical to the ground-truth logic? This decomposition enables a fine-grained diagnosis of failure modes: whether a LLM fails due to syntactic fragility, planning errors, or a reliance on semantic anchors.

We operationalize this hierarchy through three tasks: grammaticality judgment, goal-conditioned generation, and instruction-to-code generation, conducted within RoboGrid. RoboGrid is a deterministic grid-world environment designed to provide unambiguous execution semantics and a

Code and evaluation framework will be made publicly available upon publication.

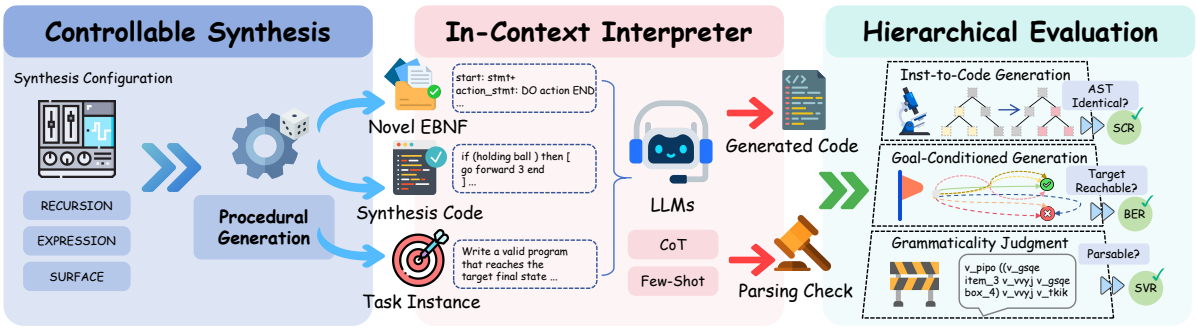

Figure 1: Architecture of the evaluation framework. It illustrates the transition from controllable synthesis of novel grammars to the hierarchical assessment of LLMs' in-context interpretation capability.

clear mapping between code and behavior, following recent trends in utilizing synthetic environments for precise behavioral evaluation (Yang et al., 2025b). Our framework supports systematic stress-testing across recursion depth, expression complexity, and surface realizations. Notably, we introduce **"Alien" lexicons**, which replace familiar keywords with opaque tokens, to force the LLM to rely exclusively on the provided grammar, thereby isolating pure symbolic reasoning from semantic priors.

Through systematic experiments, we find that LLMs exhibit a consistent hierarchical degradation in grammar interpretation, where syntactic validity does not guarantee behavioral or semantic correctness under complex constraints. Specifically, even the strongest LLMs struggle with structural alignment, showing a significant gap between surface parsing and logical execution. Furthermore, our ablations reveal that performance is highly sensitive to recursion depth and lexical familiarity. Sharp non-linear decay occurs as depth increases, consistent with findings on the limitations of LLMs in maintaining deep reasoning chains (Rodkin et al., 2025). The sharp decline in performance when using `Alien` tokens further indicates that LLMs struggle with pure symbolic induction once stripped of familiar lexical priors.

In summary, our contributions are as follows:

- We develop RoboGrid, a diagnostic benchmark that utilizes a three-layer evaluation hierarchy (**Syntax, Behavior, Semantics**) to localize failures in structural reasoning.
- We formalize the task of **in-context grammar interpretation** for LLMs, shifting the evaluation focus from surface pattern matching to formal rule induction.
- We characterize the operational limits of SOTA LLMs, revealing a **massive gap between syntactic adherence and semantic alignment**. While LLMs may follow formal rules at the surface level, their ability to maintain logical fidelity collapses as complexity increases, exposing a heavy reliance on semantic priors rather than pure symbolic induction.

2 Preliminaries

2.1 RoboGrid Environment

RoboGrid is a deterministic grid-world designed to decouple a LLM's understanding of task logic from its adherence to formal syntax. In this environment, a robot navigates the grid to pick up objects which are then stored in its inventory. The robot maintains a state $(\texttt{x}, \texttt{y}, \texttt{facing}, \texttt{inventory})$, representing its 2D coordinates, orientation, and the collection of held items respectively. The instruction set is as follows:

```
Move(dir, n)            #forward/backward n steps
Turn(dir)               #left/right 90 deg
Grab(item)              #pick up item|key|box
Loop(n, body)           #repeat body n times
If(cond, then, else?)   #conditional branch
Holding(item)           #state-query predicates
And | Or | Not          #logical operators
```

2.2 Grammar Specification via EBNF

To interact with RoboGrid, LLMs must generate code conforming to a specific CFG provided in the prompt. We specify these grammars using **Extended Backus-Naur Form (EBNF)** (McCracken and Reilly, 2003). A representative EBNF snippet for RoboGrid is shown below:

Category	Parameter	Conf. Space	Evaluation Purpose
RECURSION	Max Nesting Depth (D)	$\{2, \ldots, 20\}$	Probes hierarchical state tracking horizon
	Else-Branch Prob. (p)	$[0.0, 1.0]$	Modulates control flow branching density
EXPRESSION	Expression Depth (E)	$\{1, 2, 3\}$	Controls nested arithmetic and boolean predicates
SURFACE	Syntactic Style	{`Block`, `C-style`, `S-expr`}	Tests robustness to structural delimiters
	Lexical Familiarity	{`Natural`, `Alien`}	Isolates syntactic reasoning from semantic priors

Table 1: Controllable parameters for procedural data generation. By systematically varying these dimensions (D, p, E), we isolate specific aspects of grammar interpretation capability.

```
start: stmt+
stmt: action_stmt | loop | if_stmt

action_stmt: DO action END
loop: LOOP INT TIMES LBR stmt+ RBR
if_stmt: IF cond THEN LBR stmt+ RBR (ELSE LBR
    stmt+ RBR)?

action: MOVE MOVE_DIR INT? | TURN TURN_DIR |
    GRAB ITEM
cond: HOLDING ITEM

DO: "exec"      END: "end"
LOOP: "loop"    TIMES: "times"
IF: "when"      THEN: "after"
LBR: "["        RBR: "]"
```

This EBNF defines the legal hierarchical structure of the language. Specifically, it specifies:

- **Recursive Composition**: The `stmt` nonterminal expands into actions, loops, or conditionals, where `loop` and `if_stmt` can recursively contain `stmt+` to enable nesting.
- **Explicit Delimiters**: Actions are wrapped by `DO`/`END` and control blocks by `LBR`/`RBR`, providing anchors for tracking recursion levels.
- **Dynamic Terminals**: Bottom-level mappings (e.g., `LOOP: "loop"`) define the lexicon.

By substituting terminals, e.g., replacing `"loop"` with an opaque token like `"v_xkqm"`, we decouple pure syntactic induction from semantic priors. This formalizes LLMs as **in-context interpreters**, where the EBNF acts as a dynamic protocol that LLMs must interpret on the fly.

3 Controllable Data Generation

To rigorously evaluate LLMs' ability to induce novel CFGs, we developed a controllable generation pipeline that disentangles structural complexity from surface representation. Unlike static benchmarks, our framework dynamically synthesizes instances by sweeping across a multi-dimensional configuration space (see Table 1).

3.1 Structural Complexity: Recursion and Expressions

The core difficulty of a CFG language lies in its hierarchical depth. We control this via two primary dimensions.

Hierarchical Depth The *Max Nesting Depth* (D) limits the recursion level of control structures such as `Loop` and `If`. This parameter probes the model's ability to maintain a deep stack of nested execution states. We further modulate control-flow density using *Else-Branch Probability*, which determines the structural branching factor of the program.

Expression Complexity To decouple high-level logic from local computation, we parameterize the *Expression Depth* (E), which controls the nesting level of arithmetic and boolean predicates. This ensures that a LLM's failure in deep recursion can be distinguished from its inability to resolve complex nested logic. By varying E, we isolate the impact of local predicate resolution on the overall program's behavioral and semantic correctness.

3.2 Surface Realization: Styles and Lexicons

To isolate symbolic reasoning from the effects of pre-training exposure, we vary the surface realization of program through two key mechanisms.

Syntactic Style We implement three distinct formatting conventions: `Block` (verbose keywords like `do`/`end`), `C-style` (braces `{}` and semicolons), and `S-expr` (Lisp-like prefix notation). This variation tests whether LLMs are biased toward specific structural delimiters inherited from common programming languages.

Lexical Familiarity This is a critical control in our framework. While `Natural` mode employs standard keywords (e.g., `loop`), `Alien` mode maps all grammar terminals to randomized, opaque tokens (e.g., `v_xkqm`). This `Alien` setting necessitates pure in-context induction, as the LLM cannot

Algorithm 1 Controllable Grammar and Program Generation

Require: Styles $\mathcal{S}$, Lexicons $\mathcal{L}$, Max depth $D_{\max}$, Max block size $B_{\max}$, Else-prob p_e
Ensure: Grammar $\mathcal{G}$, Code y, AST T

function GENERATEINSTANCE
  $s \sim \mathcal{S}, \ell \sim \mathcal{L}$ ▷ Sample style and lexicon
  $\mathcal{K} \leftarrow \text{MAPLEXICON}(\ell)$
  $\mathcal{G} \leftarrow \text{BUILDGRAMMAR}(s, \mathcal{K})$
  $T \leftarrow \text{SAMPLENODE}(d = 0)$
  $y \leftarrow \text{LINEARIZE}(T, s, \mathcal{K})$
  return $(\mathcal{G}, y, T)$
end function

function SAMPLENODE(d)
  if $d \geq D_{\max}$ **then**
    return SAMPLEACTION ▷ Base case: leaf node
  end if
  $t \sim$ Categorical({Act, Loop, If})
  if $t =$ Act **then**
    return SAMPLEACTION
  else if $t =$ Loop **then**
    $e \leftarrow \text{SAMPLEEXPR}(d)$
    $B \leftarrow \text{SAMPLEBLOCK}(d + 1)$
    return Node(Loop, e, B)
  else if $t =$ If **then**
    $c \leftarrow \text{SAMPLECOND}(d)$
    $T_b \leftarrow \text{SAMPLEBLOCK}(d + 1)$
    $E_b \leftarrow$ Bernoulli(p_e)?SAMPLEBLOCK($d + 1$) : None
    return Node(`If`, c, T_b, E_b)
  end if
end function

function SAMPLEBLOCK(d)
  $n \sim$ Uniform$(1, B_{\max})$ ▷ Number of statements
  return [SAMPLENODE(d) for $i = 1 \ldots n$]
end function

leverage semantic priors to infer the function of a command, forcing a reliance on the provided EBNF specification.

3.3 Synthesis Procedure

Algorithm 1 formalizes the generation process. The pipeline first samples a grammar variant $\mathcal{G}$ by binding a syntactic style to a lexicon. It then performs a constrained recursive walk to construct a valid AST within the specified depth limits. Finally, the AST is linearized into a code string y according to the chosen style and mapping. This ensures that every generated instance is guaranteed to be syntactically valid and possesses a deterministic execution trace.

4 Evaluation Framework

To rigorously assess LLMs as in-context grammar interpreters, we propose a diagnostic framework that disentangles the multi-faceted process of grammar induction. Our framework combines three progressively difficult tasks with a three-layered hierarchy of evaluation metrics.

4.1 Task Definitions

We design three tasks to evaluate the LLM's ability to recognize, plan, and implement logic under a novel grammar $\mathcal{G}$ provided in the prompt. Detailed examples of each task are provided in Appendix A.2.

Task 1: Grammaticality Judgment This task probes the LLM's capacity for *syntactic recognition*. Given $\mathcal{G}$ and a candidate string s, the LLM must classify it as `VALID` or `INVALID`. We construct the test set with a balanced distribution of valid programs and strings containing various perturbations (e.g., delimiter mismatches, illegal nesting, or lexical violations). This task isolates pure syntactic induction from semantic-related reasoning.

Task 2: Goal-Conditioned Generation This task evaluates *behavioral planning* under syntactic constraints. Given $\mathcal{G}$ and a target environment state g (e.g., the robot's final coordinates and inventory), the LLM must synthesize a program $\hat{y}$ that reaches g. This assesses whether the LLM can map abstract goals to the provided EBNF primitives, even if the synthesized program structure differs from ground-truth solution.

Task 3: Instruction-to-Code Generation The most demanding task, requiring *procedural alignment*. Given $\mathcal{G}$ and a natural language instruction x (e.g., "Move forward 3 steps, then repeat the grab action twice"), the LLM must generate code $\hat{y}$ that faithfully implements the specified logic. To ensure a precise mapping between instructions and ground-truth code, these natural language descriptions are deterministically synthesized using templates corresponding to the underlying AST structure. This requires the LLM to map the hierarchical structure of natural language to the nested syntax of $\mathcal{G}$ while maintaining semantic fidelity.

4.2 Evaluation Metrics

We evaluate performance across the tasks using a hierarchy of three metrics.

Syntax Validity Rate (SVR) SVR measures the proportion of LLM's outputs that conform to the production rules of $\mathcal{G}$. For Task 1, SVR is the classification accuracy of the judgment. For Tasks 2 and 3, it is defined as the parseable rate:

$$\text{SVR} = \mathbb{E}[\mathbf{1}\{\text{parse}(\hat{y}, \mathcal{G}) \neq \text{error}\}]. \quad (1)$$

where $\hat{y}$ denotes the program string synthesized by LLM. The parse$(\cdot)$ function is implemented via

Lark library, which provides a standard parser to validate $\hat{y}$ against $\mathcal{G}$.

Behavioral Equivalence Rate (BER) Applied to Tasks 2 and 3, BER measures the ratio of generated programs that, when executed from an initial state s_0, result in a final environment state identical to the ground truth:

$$\text{BER} = \mathbb{E}[\mathbf{1}\{\text{exec}(\hat{y}, s_0) = \text{exec}(y^*, s_0)\}]. \quad (2)$$

Here, y^* represents the ground-truth program, and $\text{exec}(\cdot, s_0)$ denotes a deterministic execution function that maps a program and an initial environment state s_0 to a final state. This metric captures functional correctness, rewarding any valid program $\hat{y}$ that achieves the intended outcome regardless of its internal implementation.

Semantic Correctness Rate (SCR) The strictest metric, primarily utilized for Task 3. SCR measures whether the induced AST of the generated output is structurally identical to the ground-truth AST:

$$\text{SCR} = \mathbb{E}[\mathbf{1}\{\text{AST}(\hat{y}) = \text{AST}(y^*)\}]. \quad (3)$$

SCR ensures that a LLM has correctly internalized the hierarchical logic and structural intent defined by the grammar, precluding "semantic shortcuts" where a LLM reaches the correct state through an incorrect execution path.

By construction, these metrics follow a strict containment: SCR $\leq$ BER $\leq$ SVR. This allows us to localize exactly where the LLM's reasoning fails—whether at the level of surface syntax, functional execution, or structural semantics.

We further define the conditional metrics as CBER $=$ BER/SVR and CSCR $=$ SCR/SVR. These scores isolate the LLM's logical and structural performance from its basic syntactic proficiency, revealing its reasoning quality specifically on validly formatted outputs.

5 Experimental Results and Findings

We conduct a comprehensive evaluation of LLMs on RoboGrid to assess their ability to handle recursive syntax and semantic constraints.

5.1 Main Experiments

We evaluate LLMs across three progressively difficult tasks: **Grammaticality Judgment**, which tests syntactic recognition; **Goal-Conditioned Generation**, which assesses behavioral planning under constraints; and **Instruction-to-Code Generation**, which requires full alignment across syntax, behavior, and semantics.

Configurations. We establish a high-difficulty benchmark with 200 distinct samples per task, fixing a recursion depth of $D = 10$ and a 0-shot Chain of Thought (CoT) (Wei et al., 2022) prompting strategy. To isolate symbolic reasoning from semantic memorization, we utilize the `block` syntactic style paired with an `Alien` lexicon, where all functional keywords are replaced by opaque tokens. We evaluate a diverse suite of LLMs, including the Qwen3 family (8B, 32B, 235B) (Yang et al., 2025a), reasoning LLMs (DeepSeek-V3.2 (Team, 2025a), Mimo-V2-flash (Xiaomi, 2025), GLM4.7 (Team, 2025b), MiniMax M2.1) (Team, 2025c), and the GPT-5 series (OpenAI, 2025).

Results and Analysis. As reported in Table 2, performance consistently follows a hierarchical degradation of SCR $<$ BER $<$ SVR, highlighting the challenge of maintaining structural fidelity under deep recursion. Three prominent findings emerge: (1) **Model Scale vs. Task Consistency:** Increased model scale does not consistently translate to better structural alignment. For instance, GPT-5-mini significantly outperforms the larger GPT-5.2 in Task 2 BER (90.0% vs. 60.0%). (2) **Syntactic Mastery vs. Logical Fidelity:** The conditional metrics CBER and CSCR reveal that syntax is a necessary but insufficient condition for correctness. Notably, DeepSeek-V3.2 maintains a high Task 3 CBER (87.6%), whereas GPT-5-mini exhibits a severe semantic collapse with a CSCR of only 9.23% despite its high SVR. (3) **Functional Heuristics vs. Structural Alignment:** The sharp drop in CSCR across all models under the `Alien` lexicon suggests that without semantic anchors, LLMs rely on surface-level workarounds rather than pure symbolic state-tracking, failing to maintain a consistent mapping between nested instructions and recursive production rules.

5.2 Ablation Studies

To further isolate the factors governing in-context grammar induction, we conduct a series of systematic ablations across the three dimensions defined in Table 1. For these experiments, we focus on the *Instruction-to-Code Generation* task as our primary testbed, as it requires the most comprehensive alignment across syntax, behavior, and semantics.

Model	Grammaticality Judgment	Goal-Conditioned Generation			Instruction-to-Code Generation				
	SVR	SVR	BER	CBER	SVR	BER	SCR	CBER	CSCR
Open Source Models									
Qwen3-8B	47.0	1.00	1.00	**100**	0.00	0.00	0.00	–	–
Qwen3-32B	51.0	62.5	58.5	93.6	0.50	0.50	0.00	**100**	0.00
Qwen3-235B	**92.0**	96.0	76.5	79.7	40.0	34.5	19.0	86.3	47.5
Mimo-V2-flash	79.5	69.5	54.0	77.7	21.5	17.5	9.00	81.4	41.9
DeepSeek-V3.2	86.0	**100**	**96.5**	96.5	**68.5**	60.0	**39.5**	87.6	**57.7**
GLM4.7	65.0	74.5	68.5	91.9	44.5	38.5	24.5	86.5	55.1
MiniMax M2.1	41.0	91.5	85.5	93.4	8.50	6.00	2.50	70.6	29.4
Closed Source Models									
GPT-5-nano	70.0	76.5	71.5	93.5	9.50	1.50	0.00	15.8	0.00
GPT-5-mini	83.5	**100**	90.0	90.0	65.0	**60.5**	6.00	93.1	9.23
GPT-5.2	75.0	95.5	60.0	62.8	38.0	16.0	10.0	42.1	26.3

Table 2: Main results under the `Alien` Lexicon setting at Depth 10. All metrics are reported in percentage (%). Bold indicates the best performance per column.

Dimension	Configuration	SVR	BER	SCR
Baseline	**Standard**	**21.5**	**17.5**	**9.00**
Control Flow Density (p)	No Else ($p = 0.0$)	24.5	22.5	15.0
	All Else ($p = 1.0$)	13.0	11.5	8.50
Expression Complexity	Shallow ($E = 1$)	22.5	20.0	10.5
	Deep ($E = 3$)	16.5	14.0	4.00
Syntactic Style	`C-style`	19.5	16.0	6.50
	`S-expr`	16.0	13.5	4.50
Lexical Fam.	`Natural`	24.5	21.5	10.5

Table 3: Ablation results on Mimo-V2-flash ($D = 10$, 0-shot CoT). All metrics are in %. E denotes Expression Depth and p denotes Else-branch probability. The **Standard** baseline uses `Block` style with `Alien` lexicon, $E = 2$, and $p = 0.5$.

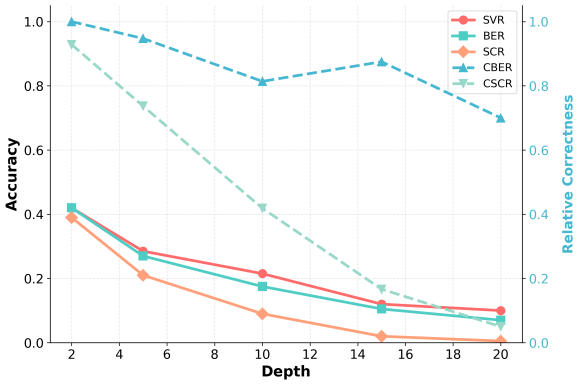

Figure 2: Impact of recursion depth on model performance under the `Alien` lexicon.

We employ Mimo-V2-flash as the representative model for ablation experiments.

Hierarchical Depth and Recursive Scaling The recursive nature of CFGs poses a significant challenge for hierarchical state-tracking. By evaluating LLMs across *Max Nesting Depths* ($D \in \{2, 5, 10, 15, 20\}$), we observe a sharp, non-linear decay across all metrics (Figure 2). At $D = 2$, the model achieves a moderate SVR of 42.0% and an SCR of 39.0%, indicating relatively stable structural alignment at shallow levels. However, as the depth increases to 20, SVR drops to 10.0%, while SCR nearly vanishes at 0.50%. This widening gap between syntax and semantics, where the CSCR falls from 92.8% to a negligible 5.0%, suggests that while LLMs can occasionally produce locally parseable code at high depths, they almost entirely lose the global state-tracking required to faithfully implement the intended hierarchical logic.

This structural collapse is further exacerbated by the *Else-Branch Probability* (p). As shown in Table 3, forcing $p = 1.0$ causes SVR to plunge to 13.0%, a nearly 40% reduction compared to the baseline. In contrast, removing else-branches ($p = 0.0$) restores SCR to 15.0%. This confirms that tracking multiple execution paths significantly exhausts the LLM's capacity to maintain the current context. As the density of these branches increases, the model often fails to track the active scope, resulting in incomplete control blocks or the premature termination of nested structures.

Predicate Complexity and Logic Resolution Beyond the global structure, the complexity of nested predicates within control flow serves as a critical bottleneck. By modulating the *Expression Depth* (E), we find that BER and SCR are more sensitive than SVR. As shown in Table 3, increasing the complexity from the baseline ($E = 2$) to

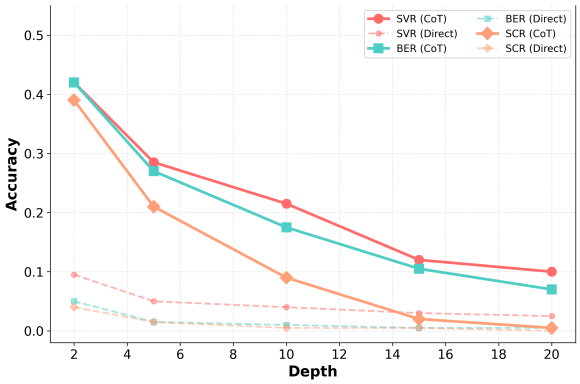

Figure 3: Comparison of CoT and Direct generation across recursion depths.

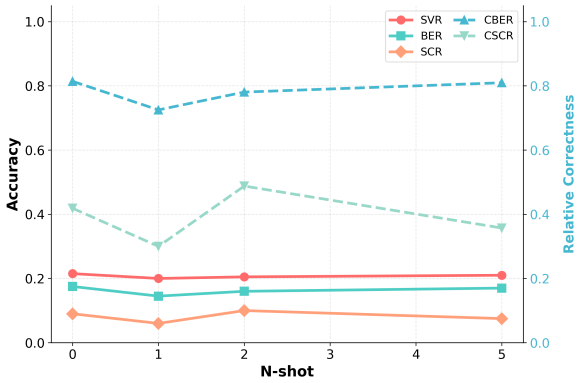

Figure 4: Impact of few-shot examples on performance at fixed recursion depth with CoT reasoning.

the deep setting ($E = 3$) causes SCR to drop by more than half, from 9.00% to 4.00%. Conversely, simplifying predicates to $E = 1$ improves SCR to 10.5% and BER to 20.0%. The fact that SVR remains relatively high (16.5%) even at $E = 3$ indicates that while LLMs can still generate syntactically valid shells, they fail to resolve the underlying logical operations. This suggests that nested arithmetic and boolean predicates constitute a distinct reasoning hurdle that persists even when the model successfully adheres to the global grammar rules.

Structural Delimiters and Lexical Priors The impact of surface realization reveals the extent to which LLMs rely on pre-training biases versus formal rules. As shown in Table 3, among the tested *Syntactic Styles*, the `Block` used in our baseline is the most robust, while `C-style` and `S-expr` lead to consistent performance drops. Specifically, switching to `S-expr` reduces SVR from 21.5% to 16.0% and nearly halves the SCR to 4.50%. This suggests that without familiar semantic cues, the repetitive parentheses of `S-expr` become harder for the model to track than explicit keywords. A more profound effect is observed through *Lexical Familiarity*. Moving from the `Alien` baseline to the `Natural` lexicon improves SVR to 24.5% and BER to 21.5%. This performance gain confirms that LLMs employ semantic bootstrapping, leveraging familiar keywords like `loop` to guess behavior, rather than performing pure in-context induction from the provided EBNF specification.

6 Diagnostic Analysis and Discussion

6.1 Prompting Strategies and ICL Efficiency

We evaluate the impact of different prompting configurations, specifically comparing zero-shot and few-shot in-context learning (ICL) (Brown et al., 2020) both with and without CoT reasoning.

The Dominance of CoT in Novel CFGs. Figure 3 reveals that CoT is a critical factor for enabling LLMs to function as grammar interpreters. In the direct (without CoT) setting, performance is consistently abysmal across all depths, with SVR rarely exceeding 10% and SCR dropping to near zero even at shallow depths ($D = 5$). In stark contrast, enabling CoT provides a massive performance boost, lifting SVR at $D = 2$ from 9.5% to 42.0% with zero-shot setting. This suggests that the external reasoning trace is a functional necessity for the model to maintain the hierarchical state-tracking required by EBNF production rules.

Diminishing Returns of Few-shot Prompting. To isolate the impact of few-shot examples, we analyze performance variations under the CoT setting specifically at a fixed complexity of $D = 10$ (Figure 4). We observe diminishing returns as the number of shots increases. Moving from 0-shot (21.5% SVR, 9% SCR) to 1-shot actually results in a performance dip (20.0% SVR, 6.0% SCR). While 2-shot and 5-shot trials show minor recoveries in SVR (20.5% and 21.0% respectively), they fail to consistently surpass the 0-shot baseline. This indicates that while ICL might assist with surface-level alignment in simpler tasks, for deep recursive structures, additional examples do not provide a clear reasoning advantage and may even introduce distracting noise into the context window.

Universal Fragility at Depth. Regardless of the prompting strategy, all configurations exhibit a sharp performance decay as recursion depth increases (Figure 3). Even with CoT, SCR plunges from 39.0% at $D = 2$ to a negligible 0.5% at

$D = 20$. This universal downward trend confirms that current LLMs remain fundamentally limited in their ability to maintain deep symbolic abstractions, with neither CoT nor ICL fully mitigating the structural collapse at extreme hierarchical depths.

6.2 Failure Mode and Case Studies

To diagnose the operational limits of LLMs, we categorize the observed failures into a three-layered taxonomy, analyzing how reasoning chains break down at each level of the evaluation hierarchy. A detailed taxonomy and abstracted case studies for each category are provided in Appendix A.4.

Syntactic Fragility Syntactic failures manifest as a fundamental breakdown in following formal production rules. LLMs frequently suffer from **lexical mapping collapses**, **operator confusion**, **scoping violations**, and **delimiter imbalances**, exposing LLMs' struggle to maintain structural integrity under strict symbolic constraints.

Behavioral Misalignment Behavioral failures occur when predicted code is syntactically valid but fails to achieve the target. These typically manifest as **logical predicate inversions** or **arithmetic simulation collapses** where mathematical expressions are miscalculated. we also observe a **planning horizon collapse** where LLMs prematurely terminate the execution flow after navigating sub-tasks, leaving the final actions unimplemented.

Semantic Drift The most subtle failure mode is semantic drift, where the generated code is behaviorally functional but semantically non-aligned. This is primarily driven by two phenomena: (1) **AST flattening bias** in arithmetic and boolean contexts, where LLMs prematurely evaluate expressions into simplified results. (2) **Autoregressive Echoing**, where the model loses its position within deep nested sequences and redundantly duplicates entire logical blocks. These failures reveal that LLMs often prioritize simulative success over structural fidelity, acting as independent executors rather than faithful in-context grammar interpreters.

7 Related Work

7.1 Computational Expressivity and Grammar Interpretation

While Transformers with polynomial-length CoT theoretically possess the expressivity to recognize CFGs (Merrill and Sabharwal, 2023), potentially via attention heads simulating dynamic programming (Allen-Zhu and Li, 2023) or MLP neurons tracking states (Zhang et al., 2025), a significant gap exists in practice. LLMs often struggle with deep recursion (Schulz et al., 2025) and revert to shallow heuristics over compositional parsing (Petty et al., 2025), failing at explicit metalinguistic deduction from formal specifications (Liu et al., 2025). These limitations suggest a reliance on statistical pattern matching rather than abstract rule internalization. We extend these insights by using a 3-layered evaluation and `Alien` lexicons to isolate pure symbolic induction from lexical priors.

7.2 Formal Constraint Enforcement

To bridge the gap between probabilistic generation and rigid machine interfaces, researchers have developed enforcement frameworks that ensure syntactic validity. Grammar-Constrained Decoding (GCD) engines, such as XGrammar (Dong et al., 2025), SynCode (Ugare et al., 2024), and TreeCoder (Princis et al., 2025), use pushdown automata to mask output distributions, while Grammar-Aligned Decoding (Park et al., 2024) aims to guarantee EBNF adherence without semantic distortion. Complementarily, Rule-Following Fine-Tuning (RFFT) methods (Hu et al., 2024, 2025) attempt to internalize these constraints by shifting model behavior from pattern matching toward deterministic symbolic execution.

8 Conclusion

In this work, we formalize the task of in-context grammar interpretation, evaluating LLMs across the three layers of syntax, behavior, and semantics. Through our RoboGrid framework, we demonstrate that while SOTA LLMs can maintain surface syntactic validity, they suffer from a severe syntax-logic gap as structural density increases. Our findings highlight that CoT reasoning is a functional necessity for following formal rules, yet LLMs remain heavily dependent on semantic bootstrapping from familiar keywords. The rapid collapse of semantic alignment under deep recursion and high branching factors pinpoints a fundamental limitation in current hierarchical state-tracking. Ultimately, our results suggest that achieving reliable, grammar-agnostic agents requires moving beyond surface-level prompt engineering toward capabilities of robust symbolic induction and reliable state-tracking across dense hierarchical structures.

Limitations

Despite the insights provided by our evaluation hierarchy, this work has certain limitations that open avenues for future research.

First, while RoboGrid provides a deterministic and controllable environment for isolating structural reasoning, it relies on template-synthesized instructions. In real-world agentic deployments, LLMs often encounter a complex interleaving of nuanced natural language and highly intricate structured interfaces. The interplay between linguistic ambiguity and formal rigidity may introduce additional failure modes. Future work will extend our framework to more heterogeneous environments to evaluate LLMs as in-context interpreters under open-domain constraints.

Second, although our study exposes a massive gap between syntactic adherence and semantic alignment in novel grammars, we do not propose a specialized architecture or decoding strategy to bridge this divide. Current solutions often either rely on model-internal knowledge or sacrifice semantic flexibility for syntactic validity. Developing a grammar-agnostic reasoning mechanism that maintains both structural integrity and logical fidelity without over-reliance on semantic priors remains a critical challenge for the community.

Ethics Statement

All experiments in this work are conducted on fully synthetic datasets generated by predefined rules. These datasets do not correspond to real individuals or real-world text and therefore contain no personally identifiable information or offensive content.

During the preparation of this manuscript, AI-assisted tools were used for minor tasks, including text polishing and code debugging.

References

Zeyuan Allen-Zhu and Yuanzhi Li. 2023. Physics of language models: Part 1, learning hierarchical language structures. *arXiv preprint arXiv:2305.13673*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Noam Chomsky and Marcel P Schützenberger. 1959. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 118–161. Elsevier.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems*, 7.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, and 1 others. 2024. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. 2025. Jsonschemabench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278*.

Yi Hu, Shijia Kang, Haotong Yang, Haotian Xu, and Muhan Zhang. 2025. Beyond single-task: Robust multi-task length generalization for llms. *arXiv preprint arXiv:2502.11525*.

Yi Hu, Xiaojuan Tang, Haotong Yang, and Muhan Zhang. 2024. Case-based or rule-based: how do transformers do the math? ICML'24. JMLR.org.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025. The swe-bench illusion: When state-of-the-art llms remember instead of reason. *arXiv preprint arXiv:2506.12286*.

Fenghua Liu, Yulong Chen, Yixuan Liu, Zhujun Jin, Solomon Tsai, and Ming Zhong. 2025. The gold medals in an empty room: Diagnosing metalinguistic reasoning in llms with camlang. *arXiv preprint arXiv:2509.00425*.

Saif Khalfan Saif Al Mazrouei. 2025. Anka: A domain-specific language for reliable llm code generation. *arXiv preprint arXiv:2512.23214*.

Daniel D McCracken and Edwin D Reilly. 2003. Backus-naur form (bnf). In *Encyclopedia of computer science*, pages 129–131.

William Merrill and Ashish Sabharwal. 2023. The expressive power of transformers with chain of thought. *arXiv preprint arXiv:2310.07923*.

OpenAI. 2025. Introducing gpt-5.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-aligned decoding. *Advances in Neural Information Processing Systems*, 37:24547–24568.

Jackson Petty, Michael Y Hu, Wentao Wang, Shauli Ravfogel, William Merrill, and Tal Linzen. 2025. Relic: Evaluating compositional instruction following via language recognition. *arXiv preprint arXiv:2506.05205*.

Henrijs Princis, Arindam Sharma, and Cristina David. 2025. Treecoder: Systematic exploration and optimisation of decoding and constraints for llm code generation. *arXiv preprint arXiv:2511.22277*.

Ivan Rodkin, Daniil Orel, Konstantin Smirnov, Arman Bolatov, Bilal Elbouardi, Besher Hassan, Yuri Kuratov, Aydar Bulatov, Preslav Nakov, Timothy Baldwin, and 1 others. 2025. Beyond memorization: Extending reasoning depth with recurrence, memory and test-time compute scaling. *arXiv preprint arXiv:2508.16745*.

Laura Ying Schulz, Daniel Mitropolsky, and Tomaso Poggio. 2025. Unraveling syntax: How language models learn context-free grammars. *arXiv preprint arXiv:2510.02524*.

Connor Shorten, Charles Pierse, Thomas Benjamin Smith, Erika Cardenas, Akanksha Sharma, John Trengrove, and Bob van Luijt. 2024. Structuredrag: Json response formatting with large language models. *arXiv preprint arXiv:2408.11061*.

DeepSeek Team. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

GLM Team. 2025b. Glm-4.7.

MiniMax Team. 2025c. Minimax m2.1.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Syncode: Llm generation with grammar augmentation. *Transactions on Machine Learning Research*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

LLM-Core Xiaomi. 2025. Mimo-v2-flash technical report.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Rem Yang, Julian Dai, Nikos Vasilakis, and Martin Rinard. 2025b. Evaluating the generalization capabilities of large language models on code reasoning. *arXiv preprint arXiv:2504.05518*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. 2025. Survey on evaluation of llm-based agents. *arXiv preprint arXiv:2503.16416*.

Yifan Zhang, Wenyu Du, Dongming Jin, Jie Fu, and Zhi Jin. 2025. Finite state automata inside transformers with chain-of-thought: A mechanistic study on state tracking. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13603–13621, Vienna, Austria. Association for Computational Linguistics.

A Appendix

A.1 EBNF Examples

A.1.1 Different Syntactic Styles

We implement three distinct syntactic styles to evaluate the LLMs' sensitivity to structural delimiters and formatting conventions.

The `block` style utilizes explicit, verbose keywords to demarcate action and control boundaries, providing clear anchor points for hierarchical parsing:

```
start: stmt+
action_stmt: DO action END
loop: LOOP expr TIMES LBR stmt+ RBR
if_stmt: IF cond THEN LBR stmt+ RBR (ELSE LBR
    stmt+ RBR)?
...
```

The `C-style` mimics common imperative programming languages, employing curly braces for scoping and semicolons for statement termination:

```
start: stmt*
action_stmt: action SEMI
loop: LOOP PAR_L expr PAR_R LBR stmt* RBR
if_stmt: IF PAR_L cond PAR_R LBR stmt* RBR
    (ELSE LBR stmt* RBR)?
...
```

The `S-expr` style adopts a Lisp-like prefix notation with extensive parenthesization:

```
start: stmt+
loop: PAR_L LOOP expr stmt+ PAR_R
if_stmt: PAR_L IF cond THEN stmt+ (ELSE
    stmt+)? PAR_R
...
```

A.1.2 Different Surface Realization

To investigate the impact of lexical priors, we vary the mapping of grammar terminals while keeping the underlying production rules constant.

The `Natural` lexicon employs familiar, semantically meaningful English keywords to represent operations and control flow.

```
start: stmt+
action_stmt: DO action END
...
DO: "do";  END: "stop";
LOOP: "repeat";  TIMES: "times";
IF: "when";  THEN: "next";
...
```

The `Alien` lexicon replaces these familiar terms with randomized, opaque tokens. This configuration forces the model to rely exclusively on the provided EBNF specification for symbolic induction, as the semantic cues are entirely stripped.

```
start: stmt+
action_stmt: DO action END
...
DO: "v_qnuc";  END: "v_ojrp";
LOOP: "v_takv";  TIMES: "v_gqge";
IF: "v_abgh";  THEN: "v_xhfm";
...
```

A.2 Task Examples

In this section, we provide concrete examples for the three tasks.

Task 1: Grammaticality Judgment The model is provided with a complete EBNF grammar and a code snippet. It must determine if the code is syntactically valid according to the production rules.

```
[PROMPT INPUT]
You are a strict syntax checker
Your task is to determine if the provided code is strictly
      valid according to the EBNF grammar.
EBNF:
start: stmt+
stmt: action_stmt | loop | if_stmt
action_stmt: "act" action "fin"
loop: "repeat" expr "iters" "{" stmt+ "}"
if_stmt: "when" cond "next" "{" stmt+ "}"
...

Code to Check:
repeat (3 + 4) iters {
  repeat ((1 + 0) * 4) iters {
    ...
  }
}

If the code can be parsed by the grammar, output 'VALID'
```

Task 2: Goal-Conditioned Generation The model receives the EBNF grammar, a start state and a target state description. It must synthesize a valid program that achieves this state.

```
You are a strict code generator.
You are given a NEW programming language definition (EBNF).
EBNF:
start: stmt+
stmt: action_stmt | loop | if_stmt
action_stmt: "act" action "fin"
loop: "repeat" expr "iters" "{" stmt+ "}"
if_stmt: "when" cond "next" "{" stmt+ "}"
...

Start state: pos (0, 0), facing N, inventory empty. Target
      final state: pos (0, 46), facing W, inventory empty.

Your task is to synthesize a valid code according to the
      EBNF that guides the robot to the target state.
```

Task 3: Instruction-to-Code Generation The model is given the EBNF grammar and a natural language instruction generated by template. It must

generate a program that is both syntactically valid and semantically faithful.

```
[PROMPT INPUT]
You are a strict code generator.
You are given a NEW programming language definition (EBNF).
EBNF:
start: stmt+
stmt: action_stmt | loop | if_stmt
action_stmt: "act" action "fin"
loop: "repeat" expr "iters" "{" stmt+ "}"
if_stmt: "when" cond "next" "{" stmt+ "}"
...

Instructions:
Step 1: Move backward ((4 times 4) times (4 plus 3)) steps.
      Step 2: Move forward (5 plus 1) steps. Step 3: Repeat
      ((4 times 0) plus 4) times: [ If (not (holding key))
      and ((holding box_0) and (holding cube)), then: [ Turn
       left. ] Otherwise: [ Move backward 3 steps. ] ] Step
      4: If not ((holding box_2) and (holding ball_0)), then:
       [ If not ((holding box) and (holding key)), then: [
      Grab the key_2. ] ] Otherwise: [ Turn right. ]

Your task is to faithfully translate the instructions into
      code. You must strictly adhere to the provided EBNF
      syntax and ensure the hierarchical structure of the
      generated program exactly matches the logical nesting
      of the instructions.
```

A.3 Generated Data Examples

We illustrate the diversity of our generated data with examples covering different recursion depths, syntactic styles, and lexical variations.

A.3.1 Recursion Depth Variations

We vary the maximum nesting depth of the AST to probe the LLM's ability to handle hierarchical structures. At depth 2, programs consist of shallow control structures, typically single loops or conditionals.

```
# Low Recursion Depth (Depth 2)
loop (3 + 4) iters {
  loop ((1 + 0) * 4) iters {
    act turn left end
  }
}
loop 3 iters {
  loop (4 * (4 * 2)) iters {
    act turn left end
  }
}
act turn left end
act take box_4 end
if not ((has cube_0 and has cube_4)) after {
  act turn left end
}
```

At depth 5, programs exhibit moderate nesting, requiring the model to track state across multiple indentations.

```
# Medium Recursion Depth (Depth 5)
if (no (has ball_3) alt (has item and has
     ball_2)) next {
  if (no (has key_4) alt (has ball_2 alt has
     ball)) next {
    do go forward (3 * (1 * 1)) end
  }
} else {
  if ((has box and has key_2) and (has ball_0
     and has box_0)) next {
    do go backward ((2 + 1) * 4) end
  }
}
loop ((4 * 4) + 3) times {
  do turn left end
}
...
```

At depth 10, programs feature deep recursion chains, challenging the LLM's ability to maintain hierarchical state-tracking.

```
# High Recursion Depth (Depth 10)
loop ((1 * 1) + (0 + 0)) x [
  exec go forward 5 stop
]
exec go forward 5 stop
loop ((0 * 3) * (3 * 3)) x [
  exec turn left stop
]
loop ((0 + 4) + (0 + 1)) x [
  loop 1 x [
    loop 4 x [
      loop ((2 * 1) * 1) x [
        loop 0 x [
          loop 1 x [
            if ((holding ball_4 alt holding
    item_1) alt not (holding key_2)) then [
              ...
            ]
          ]
        ]
      ]
    ]
  ]
]
```

A.3.2 Expression Complexity Variations

We also control the complexity of arithmetic and boolean expressions nested within the control flow. Shallow expressions (depth 1) involve simple arithmetic or boolean operations.

```
# Shallow Expressions (Depth 1)
run (3 + 2) times [
  if (holding ball or holding box) then [
    go forward 3 end
  ]
]
```

Deep expressions (depth 3) involve nested operations, requiring correct operator precedence handling.

```
# Deep Expressions (Depth 3)
run ((3 * (2 + 1)) + (4 * (0 + 1))) times [
  if ((holding ball and holding box) or (not
     (holding key) and holding item)) then [
    go forward (2 * (1 + 1)) end
  ]
]
```

A.3.3 Syntactic Style Variations

We generate the same underlying AST in different surface forms to test robustness to syntax. The `Block` style uses explicit keywords like do/end and `repeat` to delimit scope.

```
# Block Style
run ((0 + 1) * 4) times {
  if ((has cube plus_and has key_1) plus_and
     (has box_4 or has key_1)) after {
    do go forward 2 end
  }
}
```

The `C-style` mimics C-like syntax with curly braces {} and semicolons.

```
// C-Style
run(((0 + 1) * 4)){
  if(((holding(cube) and holding(key_1)) and
     (holding(box_4) alt holding(key_1)))){
    go(forward,2);
  }
}
```

The `S-expr` style uses Lisp-like prefix notation with fully parenthesized expressions.

```
; S-Expression Style
(repeat (* (+ 0 1) 4)
  (if (and (and (has cube) (has key_1)) (or
     (has box_4) (has key_1))) next
    (go forward 2)
  )
)
```

A.3.4 Lexical Familiarity Variations

We vary the lexicon to disentangle syntactic reasoning from semantic priors. `Natural` lexicon uses familiar English keywords.

```
# Natural Lexicon
if ((has item_3 and has box_4) and not (has
     ball_1)) after [
  if ((has ball_4 and has ball) and (has
     box_0 alt has key)) after [
    repeat (0 + (1 * 0)) x [
      repeat 4 x [ ... ]
    ]
  ]
]
```

`Alien` lexicon replaces keywords with randomized nonsense tokens, stripping semantic cues.

```
# Alien Lexicon
v_pipo ((v_gsqe item_3 v_vvyj v_gsqe box_4)
     v_vvyj v_tkik (v_gsqe ball_1)) v_oklp [
  v_pipo ((v_gsqe ball_4 v_vvyj v_gsqe ball)
     v_vvyj (v_gsqe box_0 v_lwmz v_gsqe key))
     v_oklp [
    v_zmew (0 v_vyht (1 v_lbke 0)) v_buty [
      v_zmew 4 v_buty [ ... ]
    ]
  ]
]
```

A.4 Typical Failure Modes and Case Studies

Following the three-layered evaluation hierarchy, we provide abstracted case studies to localize specific reasoning failures. For clarity of presentation, the examples below utilize the `Natural` lexicon style rather than the `Alien` mode to ensure the underlying logic remains interpretable. Furthermore, to accommodate the structural complexity of programs at extreme depths, we present only the critical segments of the code where errors occur. Non-essential instructions are omitted to focus exclusively on the breakdown of syntactic, behavioral, or semantic reasoning.

A.4.1 Syntactic Fragility

Case Study 1: Lexical Mapping Failure This failure identifies a symbolic binding gap where the model tracks the AST structure but fails to replace EBNF non-terminal with their terminal strings.

```
# [Model Output]
DO MOVE forward 5 END
IF ( NOT ( HOLDING item_2 ) ) ...
THEN [ DO GRAB cube_3 END ]

# [Ground Truth]
do move forward 5 stop
if (not (has item_2)) ...
next [ do take cube_3 stop ]
```

Case Study 2: Hierarchical Scoping Failure This failure demonstrates a breakdown in recursive state-tracking, where the model applies a flat encapsulation heuristic to hierarchical structures.

```
# [Model Output]
run 3 iters {
  exec if (no has box_0) next {
    exec go backward 5 fin
  } fin
}

# [Ground Truth]
run 3 iters {
  if (no (has box_0)) next {
    exec go backward 5 fin
  }
}
```

Case Study 3: Lexical Operator Confusion This failure occurs when the model fails to translate arithmetic descriptors into the formal mathematical operators required by the grammar, despite correctly identifying the calculation logic.

```
# [Model Output]
exec go forward (3 times 2) fin
exec go backward ((2 times 4) plus (1 plus 3))
     fin

# [Ground Truth]
exec go forward (3 * 2) fin
exec go backward ((2 * 4) + (1 + 3)) fin
```

Case Study 4: Premature Scope Termination This failure occurs when the model loses track of deep nesting levels, causing it to close an outer control block prematurely or omit trailing delimiters as the sequence length increases.

```
# [Model Output]
if (has item_1) {
    if (no (has box_2)) {
        ...
    }
}
// Missing second brace '}'

# [Ground Truth]
if (has item_1) {
    if (no (has box_2)) {
        ...
    }
}
```

A.4.2 Behavioral Misalignment

Case Study 1: Logical Predicate Inversion This failure occurs when a model produces syntactically valid code that adheres to all EBNF rules but fails to faithfully simulate the conditional logic of the instructions. The model correctly identifies the predicates but substitutes the wrong logical connective, leading to an incorrect execution path.

```
# [Model Output]
when (not (has ball)) plus_and ((has key_3
    plus_and has key_4)) next [
    ...
]

# [Ground Truth]
when (not (has ball) alt (has key_3 plus_and
    has key_4)) next [
    ...
]
```

Case Study 2: Arithmetic Simulation Collapse This failure occurs when the model generates syntactically correct expressions but fails to accurately simulate the underlying mathematical or procedural state, leading to a divergence in the final environment state.

```
# [Model Output]
exec go backward (4 * 5 + 1) end

# [Ground Truth]
exec go backward (4 * 5 * 1) end
```

Case Study 3: Planning Horizon Collapse This failure identifies a collapse where the model successfully navigates a high-complexity sub-task but fails to resume the global execution plan, leaving the robot stranded before reaching the goal.

Abstracted Behavioral Truncation

```
# [Model Output (Syntactically Valid)]
if (has box plus_and has key) then {
    if no (has item_2) then { ... }
}
// ERROR: Program terminates here

# [Ground Truth / Expected Behavior]
if (has box plus_and has key) then { ... }
act move forward 5 fin
```

A.4.3 Semantic Drift

Case Study 1: Arithmetic AST Flattening This failure identifies a "computation-over-alignment" bias where the model prioritizes behavioral success over structural adherence. While the resulting program achieves the correct environment state, it fails the **Semantic Correctness** check because the model simplifies complex expressions into literal results, discarding the original AST structure.

```
# [Model Output]
# The model evaluates the expression instead
    of preserving its structure
run 19 iters {
    exec turn left end
}

# [Ground Truth]
run ((4 * 4) + 3) iters {
    exec turn left end
}
```

Case Study 2: Boolean AST Flattening This failure occurs when the model correctly identifies all logical predicates but collapses the hierarchical nesting of a boolean expression into a single-level chain.

Abstracted Boolean Structure Merging

```
# [Model Output]
if (not holding item_1) plus_and (holding
    cube_4) plus_and (holding box_1) then {
    ...
}

# [Ground Truth]
if (not (holding item_1) plus_and (holding
    cube_4 plus_and holding box_1)) then {
    ...
}
```

Case Study 3: Structural Redundancy via Autoregressive Echoing This failure stems from the nature of autoregressive decoding, where previously generated complex blocks act as strong distractors in the context. The model's attention mechanism may loop by re-predicting the same high-probability sequence instead of advancing the execution plan. While behaviorally idempotent, this redundancy creates a structural mismatch with the ground-truth AST.

Abstracted Structural Redundancy

```
# [Model Output]
if no ( ... ) then { run 4 times { ... } }
if no ( ... ) then { run 4 times { ... } }
do turn left fin

# [Ground Truth]
if no ( ... ) then { run 4 times { ... } }
do turn left fin
```