TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Trace-Guided Program Synthesis Using Large Language Model Priors

Rutger KLAASSEN

Supervisors:
Dr. Sebastijan DUMANČIĆ
Dr. Matthijs SPAAN

8th January 2026

Delft University of Technology

Trace-Guided Program Synthesis Using Large Language Model Priors

Master's Thesis in Computer Science

Algorithmics group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Rutger Klaassen

8th January 2026

Author
Rutger Klaassen

Title
Trace-Guided Program Synthesis Using Large Language Model Priors

MSc presentation
15-01-2025

Graduation Committee

Dr. Sebastijan Dumančić	Delft University of Technology
Dr. Matthijs Spaan	Delft University of Technology
T. Hinnerichs, Msc.	Delft University of Technology

Abstract

Learning from Demonstration allows robot behaviour to be specified by examples rather than manual programming, but demonstrations are often noisy and provide limited structure for efficient search. Framing the problem as program synthesis allows us to search for programs that follow the demonstrated behaviour, but enumerative synthesis suffers from an enormous search space and inefficient exploration. Recent work shows that large language models (LLMs) can guide synthesis by providing probabilistic priors over program structure, though such methods typically ignore execution traces and structural depth.

In this thesis, we propose a synthesis framework that combines LLM-guided probabilistic grammars with trace-based feedback and depth-sensitive costs. We derive a depth-aware probabilistic grammar from LLM-generated programs, guide bottom-up enumeration using the induced costs, and dynamically update rule probabilities based on how closely candidate programs follow demonstrated execution traces. The results show that the effectiveness of LLM guidance and trace-based feedback depends on structural depth: while LLM priors mainly improve reachability in depth-agnostic settings, trace-based updates performs better when depth-aware costs are used, and their combination yields the strongest gains in search efficiency.

Contents

Chapter 1

Introduction

Robots are increasingly used in human environments, where they are expected to carry out diverse and often unpredictable tasks, from household chores to industrial manipulation. These tasks are difficult to specify manually as programming every possible behavior is impossible due to real world variables. There are several approaches to addressing this problem, and one of them is Learning from Demonstration (LfD). Using this method, robots acquire skills by observing demonstrations of a task. Rather than requiring explicit programming, LfD allows a task to be communicated through examples. This approach lowers the barrier to programming, allowing non-experts to teach robots through examples.

At the same time, LfD faces several challenges that limit its usefulness. Demonstrations are often noisy, incomplete, or tied to a specific environment. A demonstration of moving a cup from a table to a shelf, for instance, specifies one trajectory under one arrangement of objects, but does not make the underlying steps of the task explicit, such as "pick up the cup and place it on the shelf." Without additional structure, the robot has no way of distinguishing between the essential steps and the details of the demonstration. More fundamentally, demonstrations consist of raw trajectories or observations rather than structured task representations. This makes it difficult to reason about the demonstration, to verify whether a candidate program reproduces it correctly, or to search efficiently for a solution.

One way to address these limitations is to frame the problem of learning from demonstration as a problem of program synthesis. Rather than treating a demonstration as a raw trajectory to be replayed, we can attempt to infer a structured program that reproduces the observed behavior. A program provides a symbolic representation of the task, capturing the essential steps while abstracting away from incidental details of a single demonstration. This representation makes it possible to check whether a candidate solution correctly reproduces the demonstration, to compare different candidate solutions, and to reason about the structure of the task in a way that is not possible with unstructured trajectories.

One way to deal with these difficulties, is to introduce bias into the search process. Rather than enumerating all programs uniformly, the search can be guided towards program structures that are more likely to solve the task. HYSYNTH [Barke et al., 2025] does this by assigning probabilities to grammar production rules and using these probabilities to steer enumerative synthesis. By doing this, the grammar no longer only describes which programs are syntactically valid, but also which programs are considered more likely to be a solution. This allows the synthesizer to focus its effort on a small but promising subset of the program space without actually cutting out any solutions. In HYSYNTH, the guidance is obtained by sampling candidate programs from a large language model for the given task and extracting statistics about the used production rules to form a task specific probabilistic grammar. This provides a useful prior about program structure, but by itself it does not exploit trace-based feedback during search, nor does it model how rule frequency changes with depth.

In this thesis, we study program synthesis from demonstrations in a robotic programming language by extending LLM-guided enumeration with trace-aware updates and depth-sensitive probabilities. The objective is to synthesize a program that satisfies the demonstration-derived specification while reducing the search space, in other words we want to improve efficiency without sacrificing success. Concretely, we (i) derive a depth-dependent probabilistic grammar from multiple LLM samples, (ii) guide a bottom-up enumerator by the induced costs, and (iii) update those probabilities during search based on how many demonstration steps a candidate program reaches. While this framework defines our approach, it is motivated by several fundamental challenges in program synthesis from demonstrations. Before describing the individual components in detail, we first outline the main problems that arise when applying naive enumeration to this setting.

Motivating problems

Problem 1 : The search space is vast, and naive enumeration explores it inefficiently Even with a compact domain-specific language (DSL), the number of candidate programs grows exponentially with depth. For example, if the grammar allows five possible actions at each position, then at depth d the number of possible action sequences already exceeds 5^d. In practice, grammars for robotic tasks also contain control-flow constructs such as `IF` and `WHILE`, which multiply the number of possible derivations even further. As a result, even toy problems quickly give rise to enormous search spaces. Naive bottom-up enumeration attempts to traverse this space by combining simpler programs into larger ones, ordered only by size or cost. This ensures completeness, but it provides no mechanism for prioritizing programs that are more likely to satisfy the demonstration. In the context of robotics, this inefficiency is especially problematic: the DSL must be expressive enough to encode non-trivial behaviors. Without additional guidance, bottom-up search spends the majority of its effort exploring low-value regions of the program space.

Problem 2 : Correct programs are rare, but partial progress is common. Although the set of valid programs is enormous, only a very small fraction will fully satisfy the demonstrations. However, many incorrect programs are not entirely wrong, they often match parts of the demonstration before diverging. For instance, consider a demonstration where a robot must move forward, pick up a cup, and then place it at a different location. A candidate program might correctly move forward and pick up the cup, but fail to place it. Such a program is not a solution, but it is closer to the goal than another program that simply turns in place indefinitely. Naive enumeration treats both of these programs the same: they are equally discarded because they do not fully satisfy the specification. This wastes potentially useful information. If we could utilise a mechanism that exploits partial progress, we could distinguish between "near misses" and "hopeless failures," and bias the search accordingly.

Problem 3: Demonstrations provide more than input-output pairs, but most synthesis frameworks ignore this. In many synthesis domains, specification comes only in the form of input-output examples: the synthesizer must map an input to the correct output, with no feedback until the program is executed to completion. In our robotics domain, however, demonstrations provide more information in the form of execution traces. A trace records the sequence of intermediate states encountered during a demonstration, such as the robot's positions, the objects it is holding, or the state of the environment. These traces provide information about the intended behavior, indicating not only the final goal but also the intermediate waypoints that must be reached along the way.

Problem 4: Production rules are not equally useful at every depth.
Another challenge arises from the structure of programs themselves: not all production rules play the same role at different depths of a derivation. For example, control-flow constructs such as `IF` or `WHILE` are far more common near the root of a program, where they structure the overall behavior, while primitive actions such as `move` typically appear at the leaves. Treating probabilities as depth-independent ignores this structural bias, and as a result the synthesizer may waste time applying rules in contexts where they rarely occur. In practice, this means that even with a probabilistic grammar, the search can be misled if depth information is not taken into account.

Taken together, these problems highlight the limitations of naive enumeration in program synthesis from demonstrations. The search space is large, solutions are rare, partial progress is underutilized, traces are ignored, and structural biases are not captured. To overcome these challenges, we propose a synthesis framework that integrates large language model (LLM) guidance, depth-dependent probabilistic updates, and trace-based supervision.

LLM-guided priors. To handle the large search space (Problem 1), we use the HY-

SYNTH method [Barke et al., 2025]. The HYSYNTH method uses large language models to generate candidate programs and extract statistics over their production rule usage. These statistics are used to construct a probabilistic context-free grammar (PCFG), which biases enumeration toward program structures that the LLM uses to solve the problem.

Cost updates. To address the problem of rare correct solutions and partial progress (Problem 2), we adopt a mechanism for updating rule probabilities during search based on the PROBE mechanism [Barke et al., 2020]. When a program partially matches the demonstration, the rules it uses are reinforced at the depths where they occur. This ensures that the synthesizer can learn from near misses instead of discarding them, gradually refining the probability distribution toward rules and contexts that lead to success.

Trace-based supervision. To exploit all the information available in demonstrations (Problem 3), we evaluate candidates not only on final input-output correctness but also on how many of the intermediate steps they follow. This provides more frequent feedback during search and allows the system to reward programs that are "on the right track" even if they do not yet complete the task. In combination with the update mechanism, this enables continuous refinement of the grammar distribution.

Depth-aware structural bias. This thesis proposes a synthesis framework that integrates LLM-derived probabilistic guidance, trace-based supervision and depth-sensitive updates to improve program synthesis from demonstrations. We introduce depth-aware extension of both HYSYNTH and Probe, and empirically evaluate the approach on our self made benchmark, demonstrating improved search efficiency without sacrificing synthesis success.

Chapter 2

Background

To understand the foundation of this thesis, it is important to first introduce the relevant theoretical background. This chapter covers key concepts from program synthesis, probabilistic grammars, and learning from demonstration. These form the building blocks for the methods used later in the thesis, where large language models and program synthesis are combined to improve the structure and generalization of robot programs inferred from demonstrations.

2.1 Program Synthesis

Program synthesis is the task of automatically constructing a program that satisfies a given specification of its intended behavior. [Gulwani et al., 2017] Rather than writing code manually, the user provides some description of *what* the program should do, and a synthesizer attempts to construct the corresponding *how*. This makes synthesis a promising approach in domains where writing code is difficult, time-consuming or susceptible to errors. While the idea of automatically generating code is not new [Manna and Waldinger, 1971], it has seen growing interest in recent years, particularly alongside increasing capabilities and availability of large language models. The core of a program synthesis system consists out of three parts :

- a **specification**, which encodes the desired behaviour.
- a **program space**, which defines the set of candidate programs the system can explore.
- a **search strategy**, which determines how to traverse the program space.

In the upcoming subsection we will explain these parts accordingly. For each of these components we will use the *Robot environment* as an example.

Example 1 (Robot Environment):

The Robot environment consists of a robot and a ball placed on an $n \times n$ grid. A `RobotState` defines the environment at a given moment and includes:

- `robot_x`, `robot_y`: the robot's position,
- `ball_x`, `ball_y`: the ball's position,
- `holds_ball`: a boolean indicating whether the robot is holding the ball (0 or 1),
- `size`: the size of the grid (n).

Each synthesis task consists of an input state and a desired output state. The goal is to generate a program—a sequence of abstract robot actions—that transforms the input state into the output state.

2.1.1 Specification

In program synthesis, the **specification** describes the desired behaviour of the program, providing constraints that any valid solution must satisfy. There are various ways to specify the intent of a program, ranging from concrete input-output examples to formal logical assertions or natural language descriptions. In this thesis, we'll focus on the first kind, called **programming by example** (PBE). Programming by example is a way of specifying the intended behaviour of a program by providing input-output examples for a given problem.

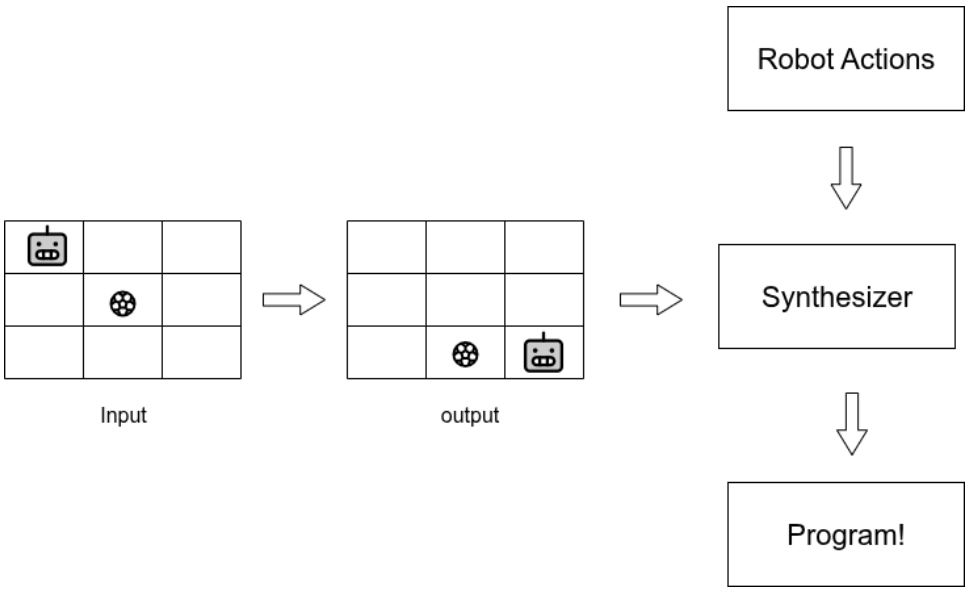

Figure 2.1: PBE schematic

Programming by Demonstration (PBD) [Cypher and Halbert, 1993] can be seen as an extension of PBE, where the input-output examples are extended with intermediate steps. These intermediate steps, known as *traces*, provide additional structure and information that can help guide the synthesizer more efficiently toward a solution. Figure 2.2 shows the traces for the input-output example of Figure 2.1.

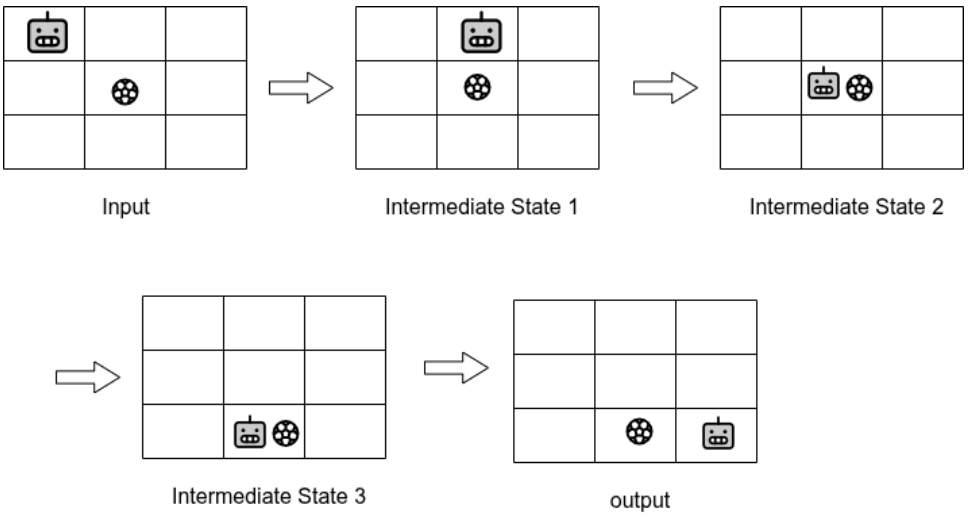

Figure 2.2: PBD schematic

2.1.2 Program Space

The **program space** defines the set of all candidate programs that a synthesizer can consider. This space is typically constrained to one language, also called a **Domain-specific language (DSL)**.

Context-free grammars

The syntax and semantics of a DSL specify what programs are valid. Usually, the DSL is defined using something called a **Context-free grammar (CFG)**. A CFG consists of the basic building blocks that can be used to generate every single valid program. Typically, a CFG consists of the following four elements :

- **Starting symbol :** A non-terminal symbol that represents the root of all valid derivations.
- **Terminal symbols :** These are the concrete elements of the language — pieces of code or actions — that appear in the final program. In the con-

text of the Robot environment, terminal symbols could include actions like `moveLeft`, `grab`, or `returnState`. Terminal symbols may also contain non-terminal symbols that need to be expanded into terminal symbols (i.e `IF(Condition, Sequence, Sequence)` where `Condition` and `Sequence` are non-terminal symbols that still need to be extended, and `IF` is a terminal symbol.

- **Non-terminal symbols:** These are placeholders or abstract symbols that do not directly appear in the final program but are used during derivation. Each non-terminal can be replaced by one or more sequences of terminals and non-terminals, as defined by the grammar's production rules. For example, in the robot language, the non-terminal symbol `Condition` can be expanded using production rules into the terminal symbol `AtTop()`.

- **Production rules :** These specify how non-terminal symbols can be rewritten. Each rule takes the form of: $A \rightarrow \alpha$, where A is a non-terminal, and α is a sequence of terminals and/or non-terminals. A grammar may define multiple rules for each non-terminal, representing different ways that part of the program could be constructed.

The CFG for the robot example is portrayed in 2.3

```
Start := Sequence

Sequence := Operation

Sequence := (Operation; Sequence)

Operation := Transformation

Operation := ControlStatement

Transformation := moveRight() | moveDown() | moveLeft()  |
                  moveUp() | drop() | grab()

ControlStatement := IF(Condition, Sequence, Sequence)

ControlStatement := WHILE(Condition, Sequence)

Condition := atTop() | atBottom() | atLeft() | atRight() | notAtTop() |
             notAtBottom() | notAtLeft() | notAtRight()
```

Figure 2.3: robot CFG

The program space is equivalent to all programs that can be derived from a CFG. The problem with a search space like this, and with that the main problem with program synthesis, is that it's exponentially large. This makes searching for a program within this search space a hard, sometimes near-impossible task.

Weighted Context-Free Grammars

A **Weighted Context-Free Grammar** is an extension of a context-free grammar in which each production rule is associated with a numerical weight. These weights are used to guide or bias the generation of derivations, influencing which derivations are explored first during search.

The cost of a program P is defined as the sum of the weights of the production rules used in its derivation trace. Let $r_1, r_2, \ldots, r_n$ be the sequence of production rules applied to derive P, and let $w(r_i)$ denote the weight of rule r_i. Then the total cost of the program is given by:

$$\text{cost}(P) = \sum_{i=1}^{n} w(r_i)$$

This cost function allows the synthesizer to compare different programs and prioritize those with lower total weights, which are assumed to be more likely, more efficient, or better aligned with prior knowledge.

Probabilistic Context-Free Grammars

In this thesis, we use another extension of a CFG, called a **Probabilistic Context Free Grammar (PCFG)**. A PCFG is the same as a CFG, except with probabilities added to production rules. For example, the production rule `Operation := Transformation` and `Operation := ControlStatement` could have probability 20% and 80% respectively. That would mean that a synthesizer selecting which of the two production rules to use on the non-terminal `Operation` would select `Operation := ControlStatement` sooner. A WCFG can be derived from any PCFG where $w(r) = -\log p(r)$, so the weight of a production rule in a WCFG is equal to the negative log probability of a production rule in a PCFG.

2.1.3 Search procedures

Once the specification and program space are defined, the final component of a synthesis system is the **search procedure**. The task of the search procedure is to traverse the search space to find a program that satisfies the given specification. Search procedures are computationally expensive, as the search space they must explore is usually enormous.

Bottom-up Search

There are multiple strategies for traversing the search space, and in this thesis, we will mainly be using the **Bottom-up Search**. The key idea behind bottom-up search is to generate programs in order of increasing cost or size, starting with the simplest expressions and incrementally building more complex ones by combining

existing programs using the grammar's production rules. This program is implemented using a dynamic programming-style algorithm, where a data structure – called the **program bank** – is maintained to store all generated programs, indexed by their cost. At each cost level c, all programs of cost c are generated by the synthesizer by combining previously generated programs using production rules.

Generating new programs using bottom-up works as follows : Suppose you have the production rule `Sequence → Operation ; Sequence`, which has a weight of 1, and you are currently generating programs at cost level 3. To use this rule, the combined cost of its subprograms must be $c_1 + c_2 = 2$, where c_1 and c_2 are the costs of the subprograms that will fill the `Operation` and `Sequence` nonterminals, respectively.

Assume that cost level 1 in the program bank contains the following programs: `Transformation()`, `ControlStatement()`, and `Operation()`. The synthesizer then enumerates all combinations `Sequence →` $(P_1$ `;` $P_2)$, where each P_x is a subprogram from a lower level in the bank such that $c_1 + c_2 = 2$.

For example, it may consider the candidate `Sequence → (Transformation(); Operation())`. The synthesizer checks whether `Operation → Transformation()` and `Sequence → Operation()` are valid derivations according to the grammar. If so, the constructed program is added to the program bank at cost level 3.

This process is repeated for all applicable production rules and for all compatible subprogram combinations, in order to exhaustively enumerate all valid programs at the current cost level.

Each candidate program is then evaluated on the examples to see if it is a solution or not. The search procedure also maintains a cache of evaluation results and discards any newly constructed programs if they're **observationally equivalent** to a program already stored in the program bank. In the context of Programming by demonstration, this means that if two different programs result in the same sequence of state transitions (traces) for all demonstration inputs, they are treated as equivalent. The pseudocode for the full bottom-up search algorithm is down below (Algorithm 1). The Bottom-up search is based on the algorithm described in the HYSYNTH paper [Barke et al., 2025].

Algorithm 1 Bottom-Up Search Algorithm

Input: Input-output examples $\mathcal{E}$, a WCFG $\mathcal{G}_w = (\mathcal{N}, \Sigma, S, \mathcal{R}, w)$
Output: A program P consistent with $\mathcal{E}$ or failure ($\perp$)

procedure BOTTOM-UP-SEARCH($\mathcal{G}_w, \mathcal{E}$)
  LVL, $\mathbb{B}, \mathbb{E} \leftarrow 1, \emptyset, \emptyset$ ▷ Initialize state of the search
  while true **do**
    for all $P \in$ NEW-PROGRAMS($\mathcal{G}_w$, LVL, $\mathbb{B}$) **do**
      Eval $\leftarrow \{\langle i, P(i)\rangle \mid \langle i, o\rangle \in \mathcal{E}\}$ ▷ Evaluate on inputs from $\mathcal{E}$
      if Eval $= \mathcal{E}$ **then**
        return P ▷ P fully satisfies $\mathcal{E}$, solution found!
      else if Eval $\in \mathbb{E}$ **then**
        continue ▷ P is semantically equiv to another program in $\mathbb{B}$
      end if
      $\mathbb{B}$[LVL] $\leftarrow \mathbb{B}$[LVL] $\cup \{P\}$ ▷ Add to the bank, indexed by cost
      $\mathbb{E} \leftarrow \mathbb{E} \cup$ Eval ▷ Cache evaluation result
    end for
    LVL $\leftarrow$ LVL $+ 1$
  end while
  return $\perp$ ▷ Cost limit reached
end procedure
procedure NEW-PROGRAMS($\mathcal{G}_w$, LVL, $\mathbb{B}$)
for all $R = N \to s_0 N_1 s_1 \dots N_k s_k \in \mathcal{R}$ **do** ▷ R is a production rule with k non-terminals
  for all $(c_1, \dots, c_k) \in \{1 \dots \text{LVL} - 1\}^k \mid \sum c_i = \text{LVL} - w(R)$ **do**
    for all $(P_1, \dots, P_k) \in \mathbb{B}[c_1] \times \dots \times \mathbb{B}[c_k] \mid \forall i,\ N_i \Rightarrow^* P_i$ **do**
      yield $s_0 P_1 s_1 \cdots P_k s_k$ ▷ Substitute subexpressions into R's RHS
    end for
  end for
end for

Dynamic cost updates (PROBE)

While weighted and probabilistic grammars provide a mechanism for biasing enumeration, the effectiveness of this bias depends on how well the assigned costs reflect the structure of correct programs. In practice, such information is often unavailable beforehand. This motivates synthesis approaches that can refine their guidance dynamically during search based on observed progress.

PROBE [Barke et al., 2020] is a bottom-up enumerative synthesis framework that introduces an online learning mechanism for updating rule costs during search. The central observation behind PROBE is that partial solutions, programs that satisfy a subset of the specification, often share syntactic structure with the final solution. Rather than discarding these programs entirely, PROBE treats them as informative signals about which production rules are useful.

PROBE operates in a loop consisting of three phases. First, programs are enumerated in order of increasing cost according to the current weighted grammar. Second, programs that partially satisfy the specification are collected as partial solutions. Third, the grammar is updated by lowering the costs of production rules that occur in these promising candidates, after which enumeration is restarted using the updated cost model. Over successive iterations, this process biases the search toward program structures that increasingly resemble a correct solution.

A key property of PROBE is that it does not require any training data or pre-existing model of the domain. Instead, it learns a probabilistic cost model solely from observations made during search itself. This makes it broadly applicable across synthesis domains while retaining the completeness guarantees of bottom-up enumeration.

In this thesis, PROBE provides the baseline mechanism for dynamic cost updates during search. While the original formulation updates rule probabilities based on whether programs satisfy subsets of input–output examples, we later extend this mechanism by incorporating information from intermediate execution traces and by conditioning updates on derivation depth.

Chapter 3

Related work

Program synthesis has been a challenge in artificial intelligence for a long time [Manna and Waldinger, 1971], aiming to automatically generate programs that satisfy a given expectation. Traditional approaches, such as enumerative and constraint based synthesis, offer guarantees of correctness but struggle with enormous search spaces. One recurring challenge in these methods is deciding how to guide the search toward promising regions of the program space. This can be done by assigning costs or probabilities to grammar rules, so that the synthesizer tries more promising programs earlier. These costs can come from a pre-trained model [Zhang et al., 2018], or can be adjusted while the search is running [Odena et al., 2020]. More recently, large language models (LLMs) have emerged as a new form of pre-trained models, as they are capable of producing code directly from natural language. However, LLM-generated code lacks the correctness that is guaranteed by the methods mentioned before.
In this thesis, we explore the possibility of combining these different methods to leverage the benefits of live updating and LLM guidance together with our own contribution of adding traces to improve search efficiency when learning from a demonstration. In this chapter, aside from a general overview of the research surrounding our problem setting, we focus on prior work that is closely related to our approach and helps position it within the broader literature. In particular, we discuss HYSYNTH, which directly informs our use of LLM-derived guidance. Mechanisms for dynamic cost updates during bottom-up enumeration, such as PROBE, are treated as background and are discussed in Chapter 2.

3.1 Programming by example

Programming by example (PBE) is an approach to program synthesis that focuses on a generating user-specified program. It tries to do this by inferring program behaviour based on input-output examples specified by the user. [Halbert, 1984] This setup makes PBE appealing for end-users as it allows non-programmers to communicate their desired behaviour through examples without interacting with any of the

code themselves. However, PBE is also inherently ambiguous, as multiple candidate programs may satisfy the same examples [Gulwani, 2016]. FlashFill is a known example of PBE in practice. Introduced by Gulwani et al. [Gulwani, 2011] and deployed in Microsoft Excel, FlashFill automatically learns string transformation programs from just a few input-output examples. Its success comes from combining a carefully designed domain-specific language (DSL) with an efficient deductive search strategy and a ranking mechanism that selects the most likely intended program among many consistent candidates. This combination enables FlashFill to operate interactively and at scale, despite the ambiguity inherent to example-based specifications. Beyond FlashFill, research has explored ways of addressing the challenges in PBE. A common approach is to use reasoning to prune the search space in enumerative program synthesis. For example, oracle-guided synthesis is an approach that generate additional test cases that help tell competing programs apart [Jha et al., 2010]. ther techniques focus on how programs are built from smaller pieces. The component-based approach by Alur et al. [Alur et al., 2017] uses the meaning of each component to quickly discard combinations that cannot match the examples. Finding conflicts is an approach that more researchers have taken, as conflict-driven analysis is another promising approach [Feng et al., 2018]. The Neo system learns from mistakes by analysing why a partial program fails and then avoiding that program but also similar ones in the future. While programming by example shares the goal of inferring programs from observed behaviour, it differs fundamentally from the setting considered in this thesis. Classical PBE formulations rely exclusively on input-output examples and provide no information about the intermediate states encountered during execution. As a result, candidate programs can only be evaluated once they are complete, and partial programs offer no feedback to the synthesizer. In contrast, our work assumes that we have access to the full execution traces from a demonstration and exploits these traces to guide search. Rather than using examples solely as specification, we use intermediate trace information to evaluate partial programs and update probabilistic guidance during enumeration.

3.2 Programming by demonstration for robotics

Programming by Demonstration (PbD), also referred to as Learning from Demonstration (LfD), is mostly studied in the robotics community, where a robot learns a skill by observing one or more demonstrations rather than through manual programming. As described in Argall et al.'s survey [Argall et al., 2009], much of this research focuses on learning a control policy from a sequence of demonstrated states and actions. Typical PbD systems collect demonstrations, derive a policy that reproduces the observed behaviour and then execute this policy on the robot. Although our work does not deal with robotics or policy learning, it draws on the same idea: intermediate execution traces provide information that is not available from input–output examples alone. In our setting, traces are used to guide the

search of a bottom-up synthesizer, helping us discard partial programs that do not follow the demonstrated behaviour and updating our probabilistic guidance during search. In this way, our approach connects to the core motivation behind PbD while applying it in a symbolic program synthesis context rather than robotic skill acquisition.

There has, however, also been work that combines program synthesis and robotics as a PbD problem. This work is mostly concerned with inferring high-level manipulation programs from demonstrations rather than learning control policies directly. For example, the Prolex system [Patton et al., 2024] synthesizes long-horizon robot programs by abstracting a demonstration into a symbolic action trace, constructing a program sketch from that trace, and completing it through top-down search. Although the initial environment is used to prune impossible branches, the demonstration itself is treated as a sequence of high-level actions and does not provide intermediate states for guiding the synthesis process.

In contrast, our work uses demonstrations in a different way. We assume access to full execution traces in a small gridworld domain and integrate these traces directly into a bottom-up synthesizer. Instead of relying on demonstrations to produce a sketch or action segmentation, we evaluate partial programs against the demonstrated intermediate states and use this feedback to update a depth-aware probabilistic grammar during search. This offers a form of trace-level guidance that is not present in prior PbD-style synthesis systems and fills a gap between robotic PbD and symbolic program synthesis: demonstrations are used not just to specify behaviour, but to shape the enumeration process itself.

3.3 HYSYNTH

The piece of prior work most directly connected to this thesis is HYSYNTH. It addresses the problem of guiding enumerative program synthesis by extracting probabilistic information from large language models and encoding it in a grammar-based cost model. Our work builds on this idea by refining how such probabilistic guidance is constructed and by integrating additional sources of feedback during search.

HYSYNTH

HYSYNTH [Barke et al., 2025] is a synthesis framework that uses large language models (LLMs) to construct a probabilistic grammar for guiding bottom-up search. The system begins by sampling many programs from an LLM by giving it a textual representation of the problem. These samples are parsed into abstract syntax trees, and the frequencies of the grammar rules used in the samples are collected. From these statistics, HYSYNTH builds a probabilistic context-free grammar (PCFG) whose rule probabilities reflect the LLM's preferences. This PCFG is then

converted into a weighted grammar by assigning each rule a cost proportional to $-\log p(R)$, allowing bottom-up enumeration to prioritise programs that the LLM implicitly considers more likely. In domains where the LLM produces invalid code, HYSYNTH falls back on analysing operator occurrences directly in the sampled text to approximate rule usage. Across several benchmarks, the paper shows that this LLM-derived weighted grammar significantly improves search efficiency compared to unguided enumeration. In our work, this mechanism provides the static prior: the LLM gives us an initial distribution over grammar rules that shapes how the bottom-up synthesizer explores the space of programs.

Chapter 4

Problem Statement

The goal of this thesis is to extend LLM-guided program synthesis with behavioral traces inferred from demonstrations, such as robot task executions. These traces specify how a desired behaviour unfolds over time and provide information that is not captured by simple input-output examples alone. In this chapter, we formally define the synthesis problem considered in this thesis.

The example-driven synthesis problem

Formally, each example-driven synthesis problem instance is defined by:

- A set of input-output specifications $\mathcal{S} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$.
- A set of behavioral traces $D = \{d_1, \ldots, d_n\}$, where each $d_i = (s_0^i, s_1^i, \ldots, s_{T_i}^i)$ is the sequence of world states observed during a demonstration on input x_i.
- A domain-specific language (DSL) $\mathcal{L}(\mathcal{G})$ defined by a grammar $\mathcal{G}$, describing the space of candidate programs and associating probabilities with production rules, potentially depending on their depth in the derivation.

Executing a program $P \in \mathcal{L}(\mathcal{G})$ on an input x yields both a final output and a sequence of intermediate states. We write

$$P(x) = (y, \tau),$$

where y is the final output and $\tau = (s_0, s_1, \ldots, s_T)$ is the execution trace.

A program P is considered *correct* if and only if, for every example (x_i, y_i) with demonstrated trace d_i, executing P on x_i produces both the correct output and the demonstrated trace:

$$\forall i, \quad P(x_i) = (y_i, d_i).$$

Beyond correctness, we are interested in the *efficiency* of the synthesis process. Specifically, we measure efficiency by the number of search nodes visited during synthesis.

Let $\mathcal{A}$ be a synthesis algorithm that, given $(\mathcal{S}, \mathcal{G}, D)$, searches over the program space $\mathcal{L}(\mathcal{G})$ and outputs a correct program P. The synthesis problem can then be formulated as the following optimization problem:

$$\min_{P \in \mathcal{L}(\mathcal{G})} \quad \mathcal{N}_{\mathcal{A}}(P) \quad \text{subject to} \quad P(x_i) = (y_i, d_i) \quad \forall i.$$

Chapter 5

Methods

This section describes a hybrid program synthesis system that combines probabilistic guidance from large language models (LLMs) with feedback from execution traces in a learning-from-demonstration (LfD) setting. The goal is to efficiently synthesize correct programs by integrating syntactic priors—extracted from LLM-generated solutions — with semantic feedback derived from observed traces. We build upon the probabilistic grammar formulation introduced in HYSYNTH and extend it with a probability updating mechanism inspired by Probe [Barke et al., 2020], enabling continual updates to the grammar based on trace coverage during search.

In essence, the system consists out of the following four phases :

1. **LLM prompting**
   An LLM is prompted multiple times with input-output examples and the domain-specific language (DSL). The resulting candidate programs are parsed to extract the number of production rules at each depth. These frequencies are then smoothed using a kernel based smoothing function to prevent overfitting .

2. **Construction of a depth-sensitive PCFG**
   A probabilistic context-free grammar (PCFG) is constructed from the collected production rule statistics, assigning probabilities to production rules based on their frequency at each depth.

3. **Probabilistically guided bottom-up enumeration**
   The PCFG is used to guide a bottom-up synthesizer. Production rules are selected according to their probabilities, with costs assigned as $-\log_2(p)$ to prioritize likely derivations.

4. **Trace-based update of the probabilities**
   Each enumerated program is evaluated to determine how many demonstration traces it reaches. The PCFG is then updated by reinforcing production

rules used in successful programs, enabling a form of learning during synthesis.

Figure 5.1 gives a high level overview of the system this thesis aims to implement.

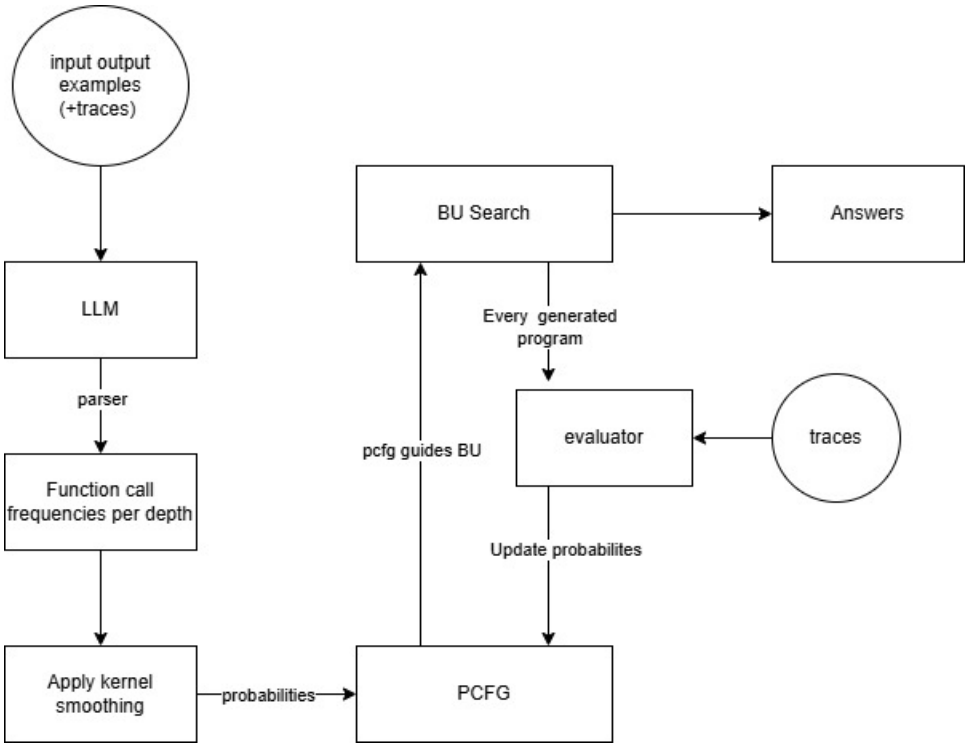

Figure 5.1: Architecture of the hybrid synthesis method

5.1 LLM prompting

We begin by querying a large language model (LLM) multiple times with a prompt that includes a set of input-output examples, a high-level description of the task, and a definition of the domain-specific language (DSL). The DSL consists of a small set of function calls that the synthesizer is allowed to use, typically representing control flow (e.g., `IF`, `WHILE`) and primitive operations (e.g., `moveLeft()`, `grab()`). The prompts used for both experiments can be found in appendix A

5.1.1 Parsing answers into frequencies

Each LLM response is parsed into an abstract syntax tree (AST), and we extract all function calls used in the program, annotated by their depth in the AST.

By collecting statistics over n samples (where n is a tunable hyperparameter, typically between 5 and 100), we build a frequency histogram of which productions occur at each depth level. These histograms serve as the basis for estimating the initial probabilistic grammar. An example of two different ASTs and the frequencies for production rules is given in figure 5.2 and table 5.1.

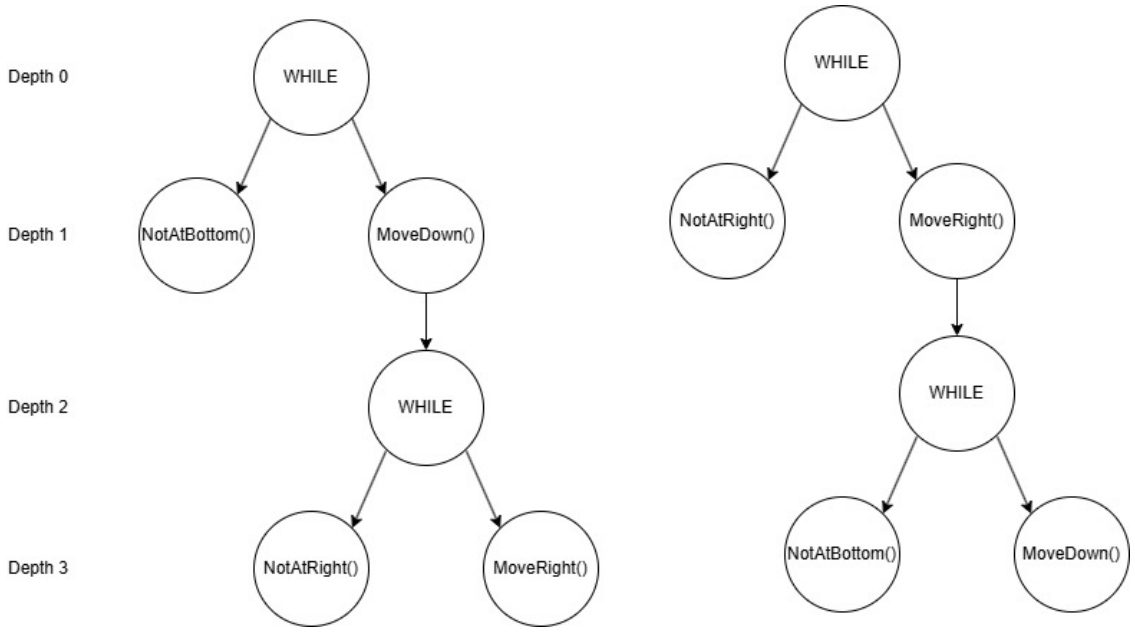

Figure 5.2: example AST

In this example, the function `WHILE(NotAtBottom, Movedown()); WHILE(NotAtRight(), MoveRight();` and `WHILE(NotAtRight(), MoveRight();WHILE(NotAtBottom, Movedown()));` are modeled. We extract the function calls per depth and record them in the table below.

depth \Function	WHILE	NotAtBottom()	MoveDown()	MoveRight()	NotAtRight()
0	2	0	0	0	0
1	0	1	1	1	1
2	2	0	0	0	0
3	0	1	1	1	1

Table 5.1: Frequency per depth table

5.1.2 Kernel-based smoothing

The raw frequencies collected from LLM-generated programs are often sparse, especially at greater depths or when only a small number of samples are available. This can lead to overfitting: certain production rules may receive artificially high or low probabilities simply due to over/under-sampling at certain depths.

To mitigate this, we apply a kernel-based smoothing technique across the depth dimension. The intuition is that the probability of using a particular production rule at a given depth should not change abruptly from one level to the next.

We define the smoothed frequency $P_d(r)$ of a production rule r at depth d as:

$$P_d(r) = \frac{\sum_{d'}(K(d-d') \cdot Z_{d'}(r))}{\sum_{d'} K(d-d')}$$

where $Z_{d'}(r)$ is the normalized (and optionally smoothed) relative frequency of rule r at depth d':

$$Z_{d'}(r)) = \frac{C_{d'}[r] + \alpha}{\sum_{r'}(C_{d'[r']} + \alpha)}$$

Here, $C_{d'}[r]$ is the raw count of how often rule r appeared at depth d' in the LLM-generated programs, and $\alpha > 0$ is a smoothing parameter (e.g., $\alpha = 1$ for Laplace smoothing) that prevents division by zero and handles unseen rules.

The kernel $K(d, d')$ weights the influence of depth d' on depth d according to their distance. We use an exponential decay kernel:

$$K(d, d') = e^{\lambda|d-d'|}$$

where λ is a variable controlling how quickly the influence of nearby depths decays. A larger λ results in narrower smoothing (closer to the raw counts), while a smaller λ spreads the influence more evenly across depths. This smoothing strategy ensures that the probability distribution $P_d(r)$ varies smoothly with depth and is robust to limited LLM-generated data. Applying the smoothing function to table 5.1 leaves us with the following probabilities :

depth \Function	WHILE	NotAtBottom()	MoveDown()	MoveRight()	NotAtRight()
0	0.327	0.168	0.168	0.168	0.168
1	0.218	0.196	0.196	0.196	0.196
2	0.284	0.179	0.179	0.179	0.179
3	0.182	0.205	0.205	0.205	0.205

Table 5.2: Smoothed frequency per depth table

5.2 Construction of a depth-sensitive PCFG

A probabilistic context-free grammar (PCFG) augments a context-free grammar $G = (N, \Sigma, R, S)$ with a probability assignment over production rules. Each production rule $r \in R$ has the form $A \to \alpha$, where $A \in N$ is a nonterminal and α is a sequence of terminals and/or nonterminals. In a standard PCFG, a probability $P(r)$ is assigned to every production rule, such that for each nonterminal $A \in N$, the probabilities of all rules with left-hand side A sum to one.

In this work, we introduce a minimal extension of this formulation by allowing the probability of a production rule to depend on the depth at which the rule is

applied in the abstract syntax tree. Importantly, the underlying grammar remains context-free: depth-dependent probabilities are used solely to guide the search process and do not affect which programs are syntactically valid. Formally, we define a depth-sensitive rule probability function

$$P : R \times \mathbb{N} \to [0, 1],$$

where $P(r, d)$ denotes the probability of applying production rule r at depth d in the abstract syntax tree. The depth d corresponds to the depth of the node at which the rule is applied, with the root node having depth 0. For every nonterminal $A \in N$ and every depth $d \in \mathbb{N}$, the probabilities of all production rules with left-hand side A sum to one:

$$\sum_{\substack{r \in R \\ \text{lhs}(r)=A}} P(r, d) = 1.$$

The depth-sensitive probabilities $P(r, d)$ are estimated from LLM-generated programs by extracting production rule frequencies per depth and applying kernel-based smoothing across depth levels, as described in the previous sections. To guide bottom-up enumeration, the probabilities are converted into costs using the standard negative log-likelihood formulation:

$$\text{cost}(r, d) = -\log_2 P(r, d).$$

This cost model ensures that production rules that are more likely under the LLM-derived prior are assigned lower cost and are therefore explored earlier during synthesis.

5.3 Probabilistically guided bottom-up enumeration

Once the depth-sensitive PCFG is constructed, it is used to guide a bottom-up enumeration strategy, which starts from simple programs such as constants, variables, or primitive function calls, and incrementally builds larger programs by combining smaller subtrees. At each step, the synthesizer considers all ways to combine existing subtrees using the production rules defined in the grammar. These derivations are scored using the PCFG defined in the previous section, where the cost of applying a rule at a given depth is determined by its probability. The synthesizer maintains a priority queue of candidate programs in the program bank, ordered by their total cost. This induces a search over the program space that prioritizes programs that align better with the structural patterns seen in the LLM samples. While Algorithm 25 describes the baseline bottom-up enumeration procedure, the method proposed in this thesis modifies several components of this algorithm.
First, the rule cost function $w(R)$ is replaced by a depth-sensitive cost $\text{cost}(R, d)$, where the cost of applying a production rule depends on the depth d at which it is used in the derivation tree.

Second, partial solutions are identified based on trace coverage rather than solely on input-output consistency.
Finally, rather than using a fixed weighted grammar throughout search, the cost model is updated dynamically. After evaluating a batch of candidate programs, production rules occurring in programs that successfully match parts of the demonstrations are assigned lower cost, following a PROBE-style update mechanism. Apart from these modifications, the overall structure of the bottom-up search algorithm remains unchanged.

5.4 Trace-based update of the probabilities

After a program is synthesized, it is evaluated against the demonstrations. Each trace describes a required step in the execution path we're searching for, so a program is only considered successful if it reproduces the correct sequence of actions.

For every program that reaches at least one intermediate state of a trace, we update the probabilities based on the PROBE update mechanism [Barke et al., 2020]. Only the probabilities of production rules used in the successful program are updated. Our update formula is a modified version of the PROBE strategy, adapted to a trace-based setting. In the original formulation, partial solutions are selected based on how many input-output examples they satisfy. In our case, we instead count how many traces a program reaches. The number of traces matched by a program determines the strength of the update applied to each production rule it uses
For updating costs, we used the following formula :

$$P_d(r) = \frac{P_{\text{base},d}(r)^{1-\text{Fit}_d(r)}}{\sum_{r' \in \mathcal{R}(N)} P_{\text{base},d}(r')^{1-\text{Fit}_d(r')}}$$

Here $P_d(r)$ is the updated probability of rule r at depth d, and $P_{base,d}(r)$ is the base probability before the update. The denominator ensures normalization over all rules $\mathcal{R}(N)$ that expand the same nonterminal N. The term $\text{Fit}_d(r)$ represents how well rule r at depth d explains the traces, where higher values indicate a better fit. The exponent $1 - \text{Fit}_d(r)$ downweighs poorly fitting rules. Fit is denoted by :

$$\text{Fit}_d(r) = \max_{\substack{p \in \text{PSol} \\ r \in \text{tr}_d(p)}} \frac{\tau_s}{|\tau|} \text{Fit}_d(r) = \max_{\substack{p \in \text{PSol} \\ r \in \text{tr}_d(p)}} \frac{\tau_s}{|\tau|}$$

Here $p \in Psol$ means that program p is a partial solution (reaches a certain amount of traces, but not all of them), $r \in tr_d(p)$ means that rule r is used in the evaluated program p ,$|\tau|$ is the total number of traces and τ_s is the number of traces reached by the program. It is important to note that programs can only reach traces in order, which means that if a program skips the first trace but reaches the second trace, it will count as 0 traces reached.

Chapter 6

Experiments & Results

In this chapter, we evaluate the proposed methods from chapter 5 by answering the following research questions :

1. **Do LLM guidance, simple cost updates and trace-based cost updates separately improve search efficiency in PS?**
2. **Does the combination of these components improve search efficiency?**
3. **Do more intermediate states in traces lead to better search efficiency?**

To answer these questions, we designed three experiments. Together, they provide a structured view of how each component contributes individually, how the components interact when combined, and how varying the number of intermediate states influences performance.

6.1 Experimental setup

All of the experiments were conducted on a system running an Intel i9-13900H processor with 16 GB of ram. The bottom up search methods were implemented in Julia with the Herb.jl program synthesis library. For seeding the pCFGs we used the DeepSeek-V3.2-Exp Large language model on default settings with the prompt provided in appendix A.

6.1.1 Benchmarks

To evaluate our proposed synthesis methods across different problem settings, we conduct our experiments on two benchmarks : our own synthetic KAREL benchmark and a subset of the Syntax-Guided Synthesis (SyGuS) benchmark. These different benchmarks allow us to assess the generality of the proposed approaches. Both of these benchmarks are compared to the baseline, for which we used a standard bottom-up enumerator guided by a uniform PCFG, and the standalone HYSYNTH and Probe techniques.

All synthesis runs use a beam-guided bottom-up enumeration strategy with a fixed beam width of $n = 10$, meaning that at each combination step only the ten lowest-cost partial programs are retained. Enumeration is subject to a global maximum budget of 100,000 generated programs per problem. However, runs may terminate before reaching this budget if the candidate pool is exhausted under the current cost model and constraints. Consequently, differences in the reported number of enumerated programs can reflect both guidance quality and early termination due to grammar exhaustion, rather than differences in the imposed search limits. Throughout this section, mean steps refers to the average number of programs enumerated per problem, counting unsolved instances by the total number of programs enumerated before termination.

KAREL

The KAREL benchmark consist of robot control programs in a grid-world environment. This dataset consists of 150 synthesis problems with known solution depths of 6, 7, and 8. Each synthesis problem is specified by input–output examples together with full execution traces. A world state is defined by the robot's grid position and orientation, the locations of all markers in the grid, and the number of markers currently carried by the robot. Given a candidate program, we define an intermediate state as the world state after executing each atomic action in the program (e.g., move, turn, pick, drop), yielding a sequence of state snapshots from the initial world to the final output state. The access to these intermediate states makes the KAREL dataset particularly suitable for evaluating trace-based feedback. For this reason, KAREL serves as the primary benchmark for evaluating the effects of intermediate state information.

SyGuS

The SyGuS benchmark represents a more traditional formal synthesis setting. Problems are specified using input-output examples and a context-free grammar. In contrast to KAREL, SyGuS problems do not provide explicit execution traces or intermediate states. Feedback during synthesis is therefore limited to input–output consistency. This benchmark allows us to evaluate whether LLM-guided probabilistic grammars and cost updates remain effective in the absence of explicit behavioural traces. This specific SyGuS benchmark can be found in the herb benchmark repository [Herb-AI]

6.1.2 Data Generation

For our experiments we used a synthetic KAREL dataset of input-output examples that includes intermediate states. We generated this dataset ourselves in julia using the Herb.jl library. Given the grammar, we used the RandomIterator (which iterates all programs in a random order) to generate a program. We initialised five different

random KAREL worlds of configurable size, wall density and at least one marker. The robot's position and orientation are also randomized. The random program is executed on each of these 5 worlds using an interpreter that records full execution traces. Each trace consists out of a sequence of world states encountered during evaluation, including robot position, orientation, marker bag contents and marker locations. To reduce redundancy, only states that differ from the previous snapshot are appended to the trace. We also filter on the following criteria to prevent programs that are too simple :

- Pick-drop contraint : At least one world per program must contain a program execution chain where the robot picks up a marker and later drops it in a different cell.
- Action-count constraint : At least one world per program must produce a trace of a specified minimum number of intermediate states when executing the program.

For each accepted program we store :

- Its abstract syntax tree and syntax
- The randomly generated worlds
- The input, output and intermediate states.
- The per-world intermediate states count

The resulting dataset is serialized in a compact format (.jls) for use during experiments. We constructed four main datasets corresponding to different action targets, each containing N programs across N x 5 worlds.

6.2 Results

6.2.1 RQ 1: Individual components

This experiment was designed to evaluate the contribution of each individual component towards improving search efficiency. The results show that the effectiveness of each component depends strongly on whether depth information is used. In the depth-agnostic setting, LLM-based priors provide the strongest overall improvement by biasing the search towards globally plausible program structures. In contrast, when a depth-aware PCFG is enabled, trace-based feedback is the best source of guidance, reducing the number of programs enumerated and scaling better to hard synthesis tasks. To isolate the effect of each mechanism, we evaluate each component independently on the datasets described in Section 6.1.1, disabling all other guidance mechanisms.

the KAREL benchmark

We first report results on the KAREL benchmark, since it provides execution traces and therefore allows trace-based feedback to be evaluated directly. All methods described in this section share the same grammar, evaluator, and synthesis parameters, ensuring that observed differences arise solely from the active component.

Individual Components without Depth Awareness
Figure 6.1 shows the performance of each individual component in the depth-agnostic setting. For problems solved within approximately 10^3 enumerated programs, all methods exhibit similar behaviour. These correspond primarily to the depth-6 subset, where the search space is sufficiently small that guidance plays a limited role.

Beyond this region, the effects of guidance become more visible. LLM-based priors solve the largest number of problems, indicating that the LLM captures useful global semantic biases and prioritizes broadly plausible program shapes. However, this guidance remains coarse: by assigning low cost to a wide range of syntactically reasonable production rules, the LLM does not sharply prune the search space. As a result, enumeration frequently continues until the maximum budget is reached. In contrast, the uninformed baseline often terminates earlier due to grammar exhaustion, which explains why it can exhibit a lower mean number of enumerated programs despite solving fewer problems overall.

Probe-based probability updates offer slight improvements over the baseline. Since probe updates are applied only when an improvement in world-level execution coverage is observed, enumeration without improvement is capped at 20,000 programs. This limits the ability of probe-based updates to steer the search effectively on harder problems, where partial semantic progress is rare.

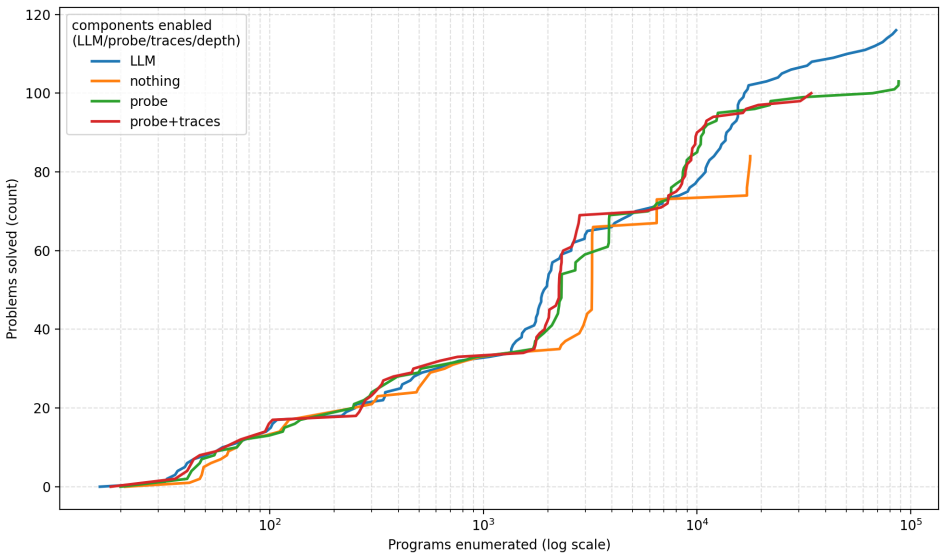

Figure 6.1: An ablation study on 150 problems comparing the contribution of the individual components on the KAREL dataset

Method	**Solved**	**Solve rate (%)**	**Mean steps**
HYSYNTH	**116**	77.33	29806.30
Baseline	84	56.00	19462.13
Probe	103	68.67	16442.06
Probe + Traces	100	66.67	**15582.72**

Table 6.1: Overall synthesis performance.

Method	**Depth 6**		**Depth 7**		**Depth 8**	
	Solved	Mean steps	Solved	Mean steps	Solved	Mean steps
HYSYNTH	47	4327.26	**47**	13875.10	**22**	71216.54
Baseline	**48**	**2604.48**	27	20903.74	9	34878.18
Probe	**48**	3687.52	43	**11353.74**	12	34284.92
Probe + Traces	**48**	3076.56	42	12334.88	10	**31336.72**

Table 6.2: Performance by solution depth.

Individual Components with Depth Awareness

The second part of this experiment evaluates how the behaviour of the individual components changes when depth-aware costs are introduced. Incorporating depth information fundamentally alters the role of guidance: rather than merely prioritizing globally plausible programs, the search is now biased toward specific structural

shapes. As a result, guidance signals that align with the true depth distribution of the target programs are strongly amplified, while mismatched signals are penalized more severely.

Figure 6.2 shows that depth awareness increases the separation between methods, especially for deeper searches. The biggest improvement is observed for trace-based updating. When combined with depth-aware costs, trace feedback outperforms all other individual components both in the number of problems solved and in the number of programs enumerated. This behaviour is consistent with the intuition behind depth-sensitive trace updates: matching intermediate execution states at the correct structural depth provides a more informative signal about proximity to the target program. As a result, the model can focus on the more relevant program shapes, allowing probe with traces to solve more problems at every enumeration budget and achieve substantially higher final solve counts.

In contrast, pure LLM guidance behaves differently in the depth-aware setting than in the depth-agnostic case discussed earlier. Instead of improving performance, depth-aware LLM guidance performs worst among the evaluated methods. This suggests that while the LLM captures which production rules are useful, the inferred rule-depth distributions often do not match the true structural depth at which these rules should occur. In a depth-sensitive PCFG, such misalignment is unforgiving: when a frequently used rule is assigned high probability at an incorrect depth, the resulting bias is applied systematically, causing incorrect program shapes to be prioritized throughout the search. Compared to the depth-agnostic setting, this indicates that the LLM tends to generate solutions with the right components but an incorrect overall structure.

Applying kernel smoothing partially mitigates this issue a little bit for LLM guidance. Smoothing reduces overconfidence in the extracted depth distributions, making the search less brittle by allowing frequencies to spread across nearby depths. While this improves performance relative to unsmoothed depth-aware LLM guidance, it does not fully correct the underlying mismatch between the LLM's depth predictions and the true program structures.

Probe-based updates with depth awareness also improve over the uninformed baseline, but remain less effective than their depth-agnostic counterpart. This suggests that depth alone provides a strong structural constraint, and that simple world-coverage-based feedback does not supply sufficient information about partial correctness to guide the search toward the correct structure. Trace feedback, which provides more accurate information about intermediate behaviour, does supply this information, explaining its strong performance in the depth-aware setting.

Overall, these results show that depth awareness amplifies both good and bad guidance. Components that align well with the true depth distribution of the target

programs benefit substantially from depth-sensitive costs, while components that introduce noisy or mismatched depth information suffer significant performance degradation. This observation motivates the combined systems evaluated in the next experiment.

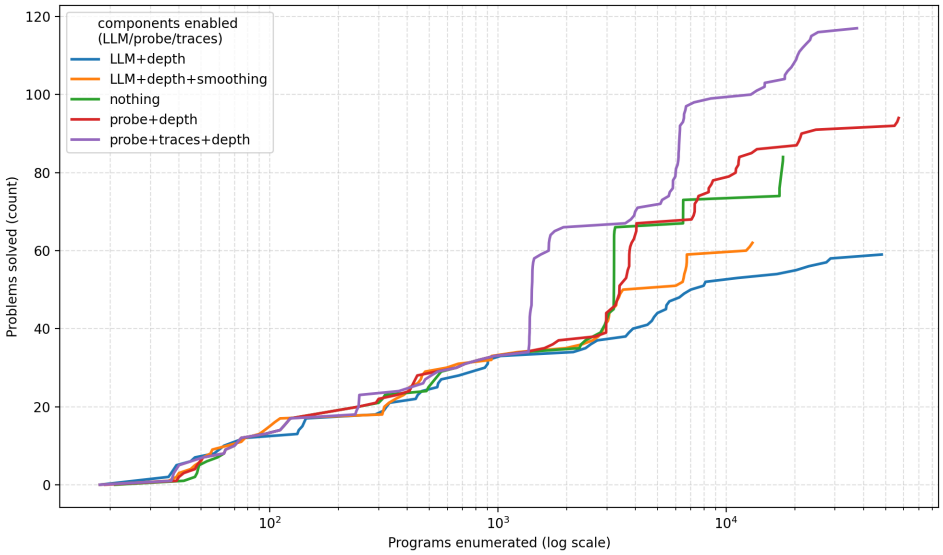

Figure 6.2: An ablation study on 150 problems comparing the contribution of the individual components with depth enabled on the KAREL dataset

Method	**Solved**	**Solve rate (%)**	**Mean steps**
HYSYNTH + Depth	59	39.33	22441.11
HYSYNTH + Depth + Smoothing	62	41.33	14098.66
Baseline	84	56.00	19462.13
Probe + Depth	94	62.67	14075.76
Probe + Traces + Depth	**117**	**78.00**	**12704.61**

Table 6.3: Overall performance with depth-aware configurations on the KAREL dataset.

Method	Depth 6		Depth 7		Depth 8	
	Solved	Mean steps	Solved	Mean steps	Solved	Mean steps
HYSYNTH + Depth	38	8266.50	13	28241.28	8	30815.56
HYSYNTH + Depth + Smoothing	41	4374.76	19	16247.44	2	**21673.78**
Baseline	**48**	2604.48	27	20903.74	9	34878.18
Probe + Depth	47	2797.12	37	12765.22	10	26664.94
Probe + Traces + Depth	**48**	**1824.86**	**47**	**7644.90**	**22**	28644.06

Table 6.4: Performance by solution depth for depth-aware configurations on the KAREL dataset.

The SYGUS benchmark

We next report results on the SyGuS benchmark. Unlike KAREL, these instances do not include intermediate execution states, so trace-based feedback is not evaluated here. We therefore focus on the two remaining individual mechanisms: LLM-seeded probabilistic grammars (HYSYNTH) and PROBE-style cost updates, and compare them against the uniform-PCFG baseline.

Individual Components without Depth Awareness
Figure 6.3 and Table 6.5 summarize the individual component ablation study we did on 100 SyGuS problems in the depth-agnostic setting. A key difference from the results on the KAREL dataset is that the overall success rates are lower for the SyGuS dataset. This tells us that, for this subset and enumeration budget, the difficulty is not in the ranking of programs among many near-miss candidates, but rather entering the correct region of the search space that contains a solution at all.

More often than not, during the experiments we saw the enumerator hit the enumeration budget without satisfying any extra input-output examples. This led to Probe being a lot less effective, most of the time not contributing to the search at all. This can be seen in the graph, as the curve that represents Probe-like updating often aligns with our baseline, telling us there's no difference between the two for a lot of problems. The difference in mean steps is due to the termination technique used in Probe-style updating. It uses a smaller enumeration budget, so that it can reset the search with a new PCFG if it has satisfied more than 1 new input-output example since the last check. If it has not satisfied any new examples, it stops. In the SyGuS dataset, a lot of the problems have very specific input-output examples where a bottom up technique like the one we use will either solve all of the examples or none. Because it's usually the latter, it tends to terminate quicker, leading to a lower mean steps.

What is consistent across this dataset is that HYSYNTH performs best among the individual components in the depth-agnostic setting, achieving a 31% relative in-

crease in solved instances over the baseline (compared to a 38% relative increase on KAREL). This suggests that the LLM captures which production rules are more likely to appear in valid SyGuS solutions. By biasing the grammar toward these more likely constructions, HYSYNTH increases the chance that the enumerator encounters a correct program within the fixed search budget. However, just like on the KAREL dataset, this guidance primarily affects whether a solution is found at all, and has only a limited effect on the average enumeration effort required to reach it.

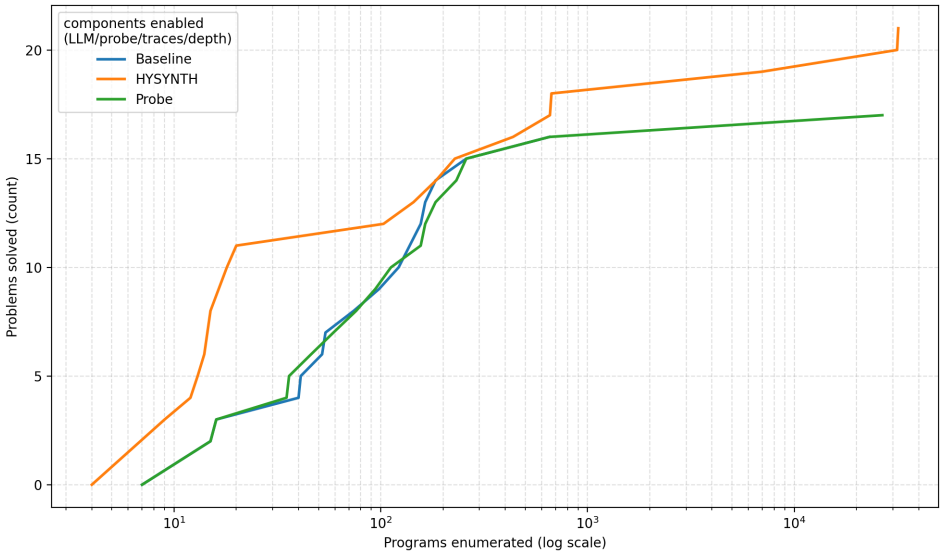

Figure 6.3: An ablation study on 100 Sygus problems comparing the contribution of the individual components

Method	**Solved**	**Solve rate (%)**	**Mean steps**
Baseline	16	16.00	80636.78
HYSYNTH	**21**	**21.00**	79241.38
Probe	17	17.00	**37258.49**

Table 6.5: Overall synthesis performance on the SyGuS benchmark for individual components.

Individual Components with Depth Awareness

Table 6.6 and figure 6.4 summarize the performance of the individual components on the SyGuS benchmark when depth-aware costs are enabled. In contrast

to the KAREL results, introducing depth sensitivity does not lead to a substantial seperation between the baseline and the HYSYNTH-based configurations. All methods show very similar behaviour, showing that adding depth awareness is not as impactful on the SyGuS benchmark as on the KAREL benchmark.

This outcome is best understood by considering the structural characteristics of the SyGuS problems evaluated in this experiment. The majority of solutions in this subset are very shallow, typically requiring only a few levels of derivation depth. At such shallow depths, the grammar imposes strong structural constraints on the search: only a small number of production rules are applicable at each depth, and in some cases the derivation is effectively forced regardless of the assigned rule probabilities.

As a consequence, depth-aware probabilistic grammars derived from LLM outputs can become close to uniform at these early depths, but for a different reason than simply ”agreeing” on a single best rule. At shallow depths, only a small subset of production rules is ever observed in the LLM samples. The remaining rules are unrepresented (frequency zero), and in our construction they receive a shared fallback probability (equivalently, a shared fallback cost) at that depth. Since the majority of rules fall into this unrepresented category, large parts of the grammar effectively have equal cost early in the derivation.

This reduces the ability of the LLM prior to meaningfully shape the search ordering at low depths: the cost model distinguishes only a handful of observed rules, while treating the rest almost uniformly. As a result, the baseline and HYSYNTH+Depth configurations explore very similar regions of the search space in nearly the same order during the initial stages of enumeration.

This behaviour weakly mirrors the trend observed on the KAREL benchmark, where depth-aware HYSYNTH can perform worse than its depth-agnostic counterpart due to structural misalignment. On SyGuS, however, the limited depth of the target programs prevents such misalignment from having a large negative effect. Instead, HYSYNTH+Depth differentiates itself from the baseline only marginally, solving a small number of additional problems without substantially changing overall search behaviour.

Probe-style updating with depth awareness follows a similar pattern. As in the depth-agnostic SyGuS setting, Probe does not substantially increase the number of solved instances, but mainly affects search expenditure by terminating unsuccessful runs earlier. Because SyGuS problems provide no intermediate behavioural signal, updates are triggered only when a candidate program satisfies additional input-output examples, which occurs rarely on this subset. As a result, depth-aware Probe behaves similarly to the baseline in terms of reachability, while differing primarily in when the search terminates rather than in which programs are

explored.

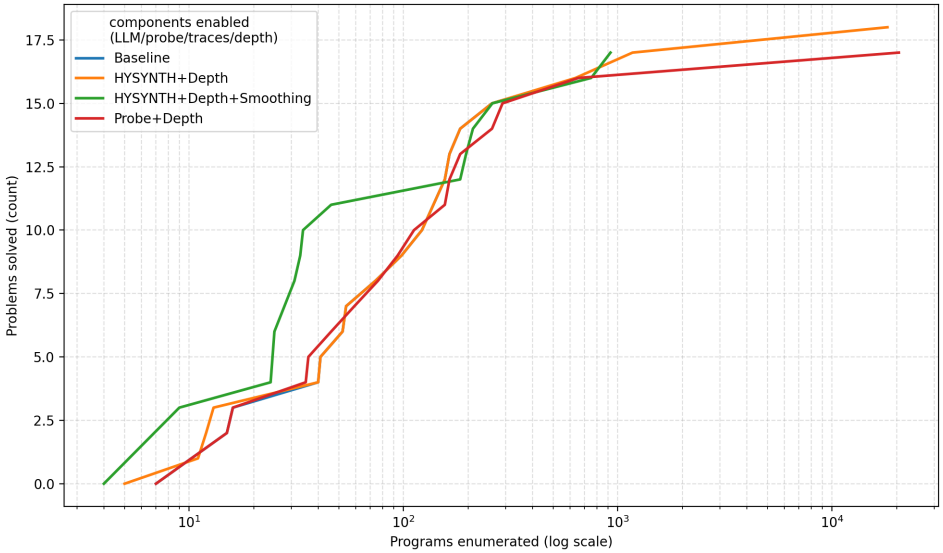

Figure 6.4: An ablation study on 100 Sygus problems comparing the contribution of the individual components with depth enabled

Method	**Solved**	**Solve rate (%)**	**Mean steps**
Baseline	16	16.00	80636.78
HYSYNTH + Depth	**18**	**18.00**	79986.32
HYSYNTH + Depth + Smoothing	17	17.00	80388.92
Probe + Depth	17	17.00	**34311.71**

Table 6.6: Overall synthesis performance on the SyGuS benchmark for depth-aware individual components.

6.2.2 RQ 2: The combined system

The second experiment evaluates whether combining the individual components studied in RQ1 leads to further improvements in search efficiency. More specifically, this experiment investigates whether LLM-guided priors, probabilistic updates, and trace-based feedback interact constructively, or whether their effects interfere when applied simultaneously. This directly addresses RQ2 by testing whether the full system outperforms both the uninformed baseline and the best individual components. As in the previous experiment, all methods are evaluated on the same datasets, grammars, and synthesis parameters. Performance is measured in terms of the number of problems solved as a function of the enumeration budget, as well as aggregate statistics over solved instances. The results are shown in Figures 6.5- 6.6, with corresponding summaries in Tables 6.7–6.10.

Depth-agnostic combinations.

In the depth-agnostic setting, combining LLM guidance with probe-based updates does not yield an improvement over LLM guidance alone. While LLM priors remain the dominant source of guidance, adding probe or trace-based updates leads to a reduction in the number of problems solved and an increase in the average number of enumerated programs. This suggests that, without structural constraints, the coarse global bias provided by the LLM already dominates the search ordering, leaving little room for improvements from probe updating. In some cases, these additional updates appear to interfere with the LLM prior by introducing biases that do not align with the target program.

These results are consistent with the findings from RQ1: in the absence of depth awareness, trace-based feedback is precise but sparse, and probe-based updates provide only weak guidance. When combined with an already strong global prior, their contribution is insufficient to outweigh the added noise introduced into the cost model.

Method	**Solved**	**Solve rate (%)**	**Mean steps**
HYSYNTH	**116**	77.33	29806.30
HYSYNTH + Probe	97	64.67	40236.07
HYSYNTH + Traces + Probe	97	64.67	38726.32
Baseline	84	56.00	19462.13
Probe	103	68.67	**16442.06**

Table 6.7: Overall performance for LLM and hybrid probe variants on the KAREL dataset.

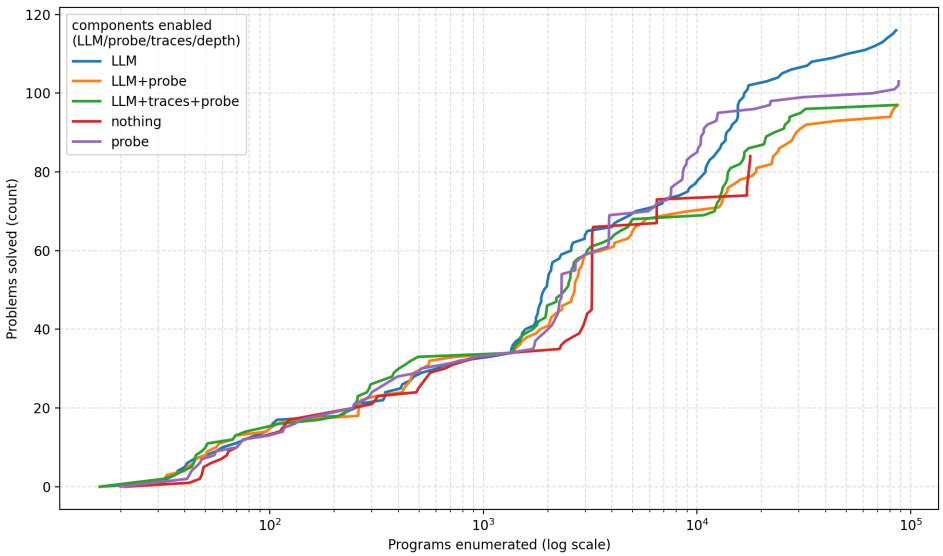

Figure 6.5: An ablation study on 150 problems comparing the combination of different components on the KAREL dataset

Method	**Depth 6**		**Depth 7**		**Depth 8**	
	Solved	Mean steps	Solved	Mean steps	Solved	Mean steps
HYSYNTH	47	4327.26	**47**	13875.10	**22**	71216.54
HYSYNTH + Probe	**49**	3300.74	38	36954.00	10	80453.46
HYSYNTH + Traces + Probe	47	5033.30	38	34134.44	12	77011.22
Baseline	48	**2604.48**	27	20903.74	9	34878.18
Probe	48	3687.52	43	**11353.74**	12	**34284.92**

Table 6.8: Performance by solution depth for LLM and hybrid probe variants on the KAREL dataset.

Depth-aware combinations.

The behaviour changes again once we introduce depth-aware costs. In this setting, the combined system integrating LLM priors, probe updates, and trace-based feedback substantially outperforms all individual components. The system with all components (HYSYNTH, PROBE, depth and smoothing) enabled solves the largest number of problems and does so with fewer enumerated programs than any other configuration. This improvement is consistent across all solution depths, with most of the improvement happening in the depth-7 and depth-8 problems.

This result can be understood by relating back to RQ1. Depth awareness makes the cost model highly sensitive to structural correctness. As observed earlier, trace-

based feedback provides a precise signal about partial correctness for specific structures, while LLM guidance supplies a useful global prior over which production rules are likely to appear. When combined, these signals become complementary rather than competing: the LLM prior biases the search toward plausible rule choices, while trace feedback refines this bias by reinforcing only those rule applications that occur at the correct structural depth.

Applying kernel smoothing further improves the combined depth-aware system by reducing brittleness in the learned depth distributions. Smoothing mitigates overconfident or slightly misaligned depth assignments coming fom the LLM, allowing probabilities to propagate to neighbouring depths. This leads to a slight but consistent reduction in enumeration cost without affecting the final solve rate, indicating improved robustness rather than a change in the underlying search strategy.

Overall, these results show that combining guidance mechanisms is only effective when their inductive biases are compatible. In the depth-agnostic setting, LLM guidance dominates and additional feedback provides little benefit. In contrast, when depth-aware costs are enabled, the interaction between LLM priors and trace-based feedback becomes strongly synergistic, producing a system that outperforms all individual components. This confirms that the benefits of combination are conditional on the presence of structural constraints that allow different guidance signals to reinforce rather than contradict each other.

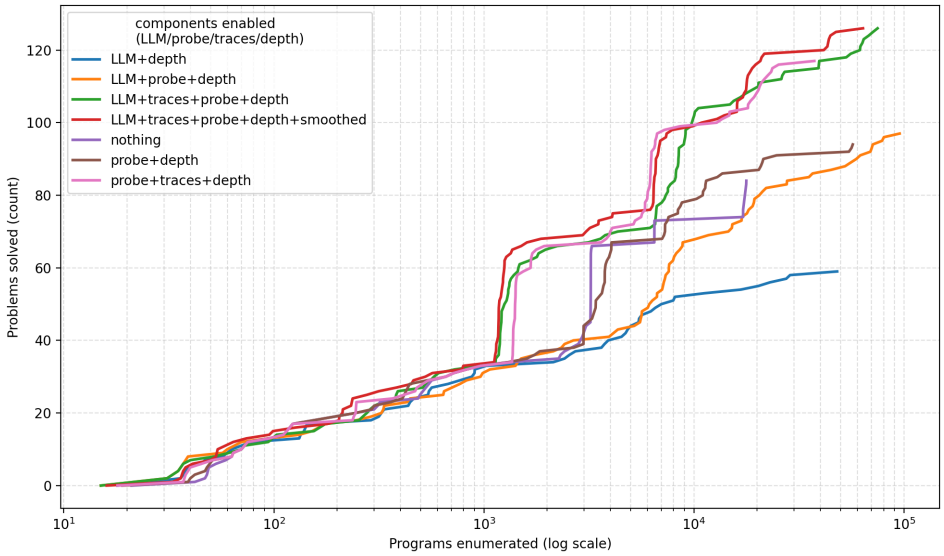

Figure 6.6: An ablation study on 150 KAREL problems comparing the different combined systems with depth enabled

Method	Solved	Solve rate (%)	Mean steps
HYSYNTH + Depth	59	39.33	22441.11
HYSYNTH + Probe + Depth	97	64.67	36468.45
HYSYNTH + Traces + Probe + Depth	**126**	**84.00**	22306.18
HYSYNTH + Traces + Probe + Depth + Smoothed	**126**	**84.00**	19235.86
Baseline	84	56.00	19462.13
Probe + Depth	94	62.67	14075.76
Probe + Traces + Depth	117	78.00	**12704.61**

Table 6.9: Overall performance for depth-aware hybrid configurations.

Method	Depth 6		Depth 7		Depth 8	
	Solved	Mean steps	Solved	Mean steps	Solved	Mean steps
HYSYNTH + Depth	38	8266.50	13	28241.28	8	30815.56
HYSYNTH + Probe + Depth	46	10065.02	38	32883.12	13	66457.22
HYSYNTH + Traces + Probe + Depth	**48**	3437.46	**49**	8482.24	**29**	54998.84
HYSYNTH + Traces + Probe + Depth + Smoothed	**48**	**1724.24**	**49**	**5884.54**	**29**	50098.80
Baseline	**48**	2604.48	27	20903.74	9	34878.18
Probe + Depth	47	2797.12	37	12765.22	10	**26664.94**
Probe + Traces + Depth	**48**	1824.86	47	7644.90	22	28644.06

Table 6.10: Performance by solution depth for depth-aware hybrid configurations.

The Sygus Dataset

Figure 6.7 and table 6.11 show the results of combining LLM guidance, probe-based updates, depth aware costs and smoothing on the SyGuS benchmark. In contrast to the KAREL results, the difference between the configurations are modest. All combined systems solve the same number of problems, and the performance curve converge as the enumeration budget increases.

The combined configurations do not substantially change the set of problems that can be solved on the SyGuS benchmark. All methods converge to similar final solve counts, and the performance curves show only minor separation across enumeration budgets. Differences in mean enumeration cost are therefore not primarily due to earlier discovery of solutions, but rather to differences in when the search terminates.

This is consistent with the SyGuS observations from RQ1: because the benchmark provides no intermediate behavioural signal, updates are triggered only when a candidate program satisfies additional input–output examples, which occurs rarely. As a result, combining HYSYNTH with probe-style updating mainly affects when the

search terminates rather than which solutions become reachable. Depth-awareness and smoothing reduce enumeration effort slightly, but do not change the final solve rate, suggesting again that the limiting factor in this setup is entering the region of the search space that contains the solution rather than re-ranking many near-miss candidates.

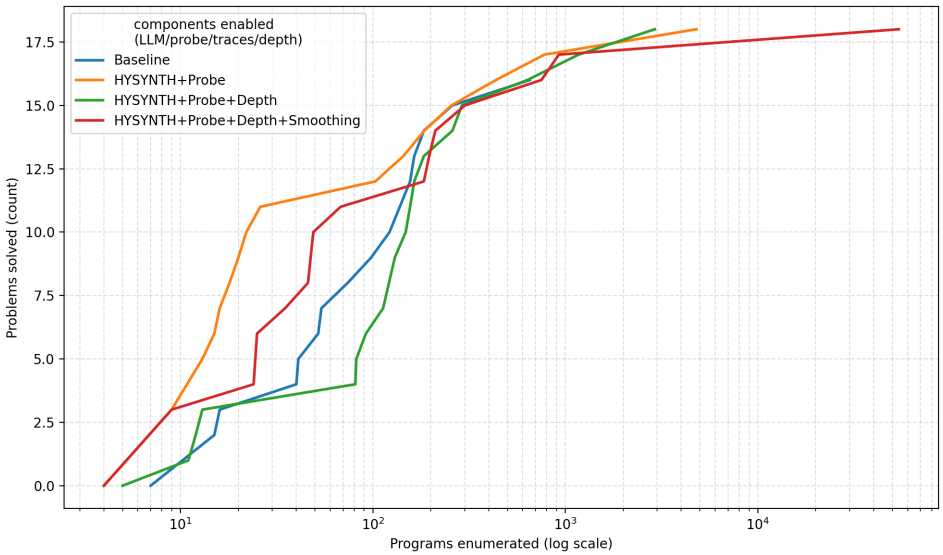

Figure 6.7: An ablation study on 100 Sygus problems comparing the different combined systems

Method	**Solved**	**Solve rate (%)**	**Mean steps**
Baseline	16	16.00	80636.78
HYSYNTH + Probe	**18**	**18.00**	45619.27
HYSYNTH + Probe + Depth	**18**	**18.00**	37968.84
HYSYNTH + Probe + Depth + Smoothing	**18**	**18.00**	**37032.80**

Table 6.11: Overall synthesis performance on the SyGuS benchmark for combined depth-aware configurations.

6.2.3 RQ 3: Effect of Intermediate State Quantity (RQ3)

The previous experiments demonstrated that intermediate execution states can substantially improve search efficiency when used as feedback during synthesis. However, these experiments assume that the full set of intermediate states is available to the update mechanism. In practice, the feedback signal used by our method depends on how many intermediate states are exposed, a trace may contain only a few, or a dense sequence of them.

This raises a natural question: to what extent does the effectiveness of trace-based guidance depend on the quantity of intermediate states? Does providing more intermediate states always lead to better guidance, or is there a point where additional states no longer meaningfully improve the search?

In this experiment we ran a search strategy with both probe-like updating and traces enabled, but with a limit on the amount of intermediate states an example in a problem can have. This experiment uses the Probe+Traces+Depth configuration, as well as beam width of 10 and a budget of 100k programs, just like all the other experiments. For this experiment we are also restricted to the KAREL benchmark, as SyGuS problems do not provide execution traces. The intermediate states are counted from the beginning, so when the limit is three and an example has five intermediate states then the last two are ignored.

Figure 6.8 and table 6.12 show that increasing the number of available intermediate states generally improves the search efficiency, but this effect does not stay the same as you increase the number of states. For depth-7 and depth-8 problems in particular, moving from zero or very few intermediate states to a moderate number (e.g., one to five) leads to more solved problems. This confirms our general intuition behind trace-based updating : More behavioural checkpoints make it more likely that partial programs match some examples of the demonstrated behaviour, allowing the cost model to reinforce relevant production rules earlier in the search.

What's interesting here is that some problems in depth 7 and 8 could be solved when setting the max number of states to 5, but not when using all available states. This indicates that additional intermediate states do not always provide more useful guidance, and can sometimes even hinder the search.

One explanation for this effect lies in how reinforcement interacts with beam-limited bottom-up enumeration. Early intermediate states are typically easy to satisfy and are shared by many shallow candidate programs. As a result, production rules corresponding to simple, non-branching behaviour can receive strong reinforcement early in the search, even when such programs do not contain the control-flow structure required to reach the final goal.

For example, consider a target program of the form *While(RightIsClear, MoveRight())*. Partial programs such as *MoveRight()* or short sequences of movement actions can satisfy a large fraction of the intermediate states produced by the demonstration. When many intermediate states are exposed, these shallow programs repeatedly receive cost reductions and therefore dominate the beam. Control-flow constructs such as *While* are introduced later in bottom-up enumeration and typically have higher initial cost. As a result, they may never enter the beam once it is filled with reinforced linear programs.

Limiting the number of intermediate states might reduce this effect by decreasing the relative advantage of shallow programs that merely reproduce execution prefixes. With fewer checkpoints, reinforcement is more closely tied to structural decisions that are necessary for reaching the final behaviour, allowing control flow to enter and remain in the beam. This explains why moderate numbers of intermediate states can outperform fully dense traces on deeper problems.

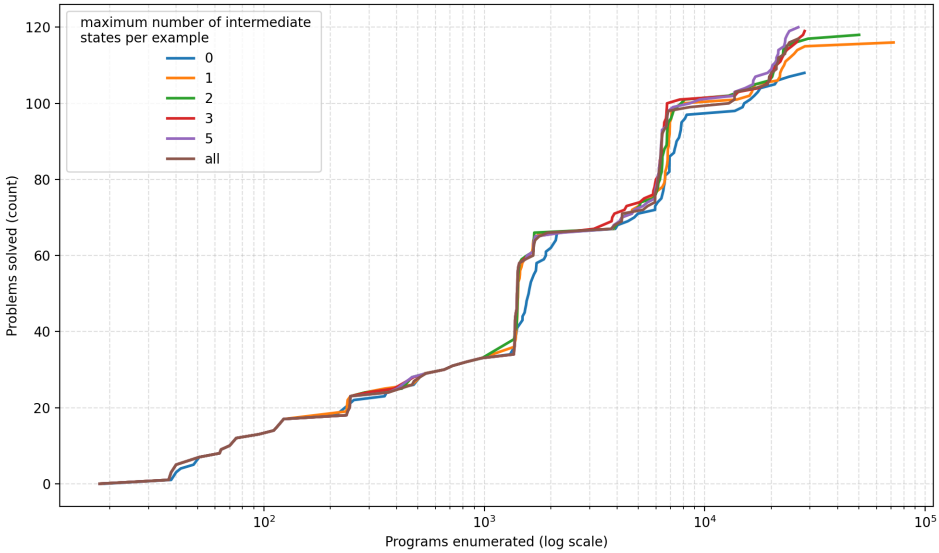

Figure 6.8: Effect of Intermediate State Availability on Synthesis Performance

Max Intermediate states	**Depth 6**		**Depth 7**		**Depth 8**	
	Solved	Mean steps	Solved	Mean steps	Solved	Mean steps
0	**48**	3077.62	44	8298.58	16	**27177.40**
1	**48**	3052.34	47	8191.36	21	33013.20
2	**48**	**3047.94**	**48**	7534.66	22	37220.72
3	**48**	3057.84	**48**	**7373.96**	23	35168.98
5	**48**	3059.78	**48**	7448.24	**24**	28884.44
all	**48**	3060.00	47	7757.86	22	28592.40

Table 6.12: Performance by solution depth across different runs for the depth-aware probe with trace feedback configuration.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

This thesis investigated how multiple sources of guidance, including statistical priors, structural bias, and behavioral feedback, can be combined to improve the efficiency of enumerative program synthesis. Rather than treating these sources as separate and standalone components, this work focused on integrating them into a unified synthesis framework.

We presented a hybrid synthesis approach that combines probabilistic guidance from large language models with bottom-up enumerative search and learning from demonstration. Building on existing grammar and feedback based cost models, we extended these mechanisms with depth-sensitive costs. This allowed production rule probabilities to vary depending on their position in the derivation tree.

Using this representation, we extracted (depth-dependent) production rule statistics from LLM-generated programs and constructed probabilistic context-free grammars that bias enumeration towards parts of the search space more likely to contain the solution. In addition, execution traces obtained from demonstration were used to evaluate partial programs and update rule costs at specific depths during search, enabling feedback-driven refinement based on trace coverage rather than solely on input-output consistency.

The experimental results show that the effectiveness of each guidance mechanism depends strongly on whether depth-aware costs are used. Without depth-awareness, LLM-seeded priors are the most effective individual component on both benchmarks, mainly by increasing the chance that the search enters the region of the grammar which contains a solution. However, in this setting the prior remains coarse, and does not substantially reduce the overall enumeration effort. Combining it with probe-style updates and trace feedback does not yield further gains and can even degrade performance, suggesting interference rather than being comple-

mentary.

Once depth-aware costs are introduced, this behaviour changes. Depth sensitivity amplifies guidance by making the cost model sensitive to program structure. This makes trace-based updating more effective, while causing LLM-based priors to perform worse as usually their inferred depth distributions are structurally misaligned. Kernel smoothing helps mitigate this brittleness by reducing the impact of such misalignment. The strongest results occur in the depth-aware combined system, where the prior and trace-based feedback reinforce rather than contradict each other.

The experiments also show that trace information is not automatically beneficial: its usefulness depends on how it interacts with the search procedure. On KAREL, traces give a strong signal once the update mechanism can attach that signal to the right structural decisions (which is why trace-based updates become most effective in the depth-aware setting). However, RQ3 shows that adding more intermediate states does not always improve performance. Dense traces can over-reward shallow programs that reproduce easy execution prefixes, causing them to dominate the beam and crowd out candidates that introduce the control-flow structure needed to reach the final outputs. In this sense, our results suggest that traces help most when intermediate checkpoints align with the structural decisions required by the target program, rather than when simply as many checkpoints as possible are provided.

Overall, this thesis demonstrates how probabilistic priors, structural bias, and behavioural feedback can be integrated within an enumerative synthesis framework, and clarifies the conditions under which these sources of guidance reinforce or interfere with one another.

7.2 Future Work

7.2.1 Richer structural representations

A natural direction for future work is to extend the structural representation used by the cost model beyond derivation depth alone. While depth-aware costs provide a simple and effective way to bias the search toward specific program shapes, the experimental results show that this representation can be brittle. Guidance that is misaligned with the true depth distribution of target programs can easily mislead the search and prevent the discovery of a solution. This suggests that depth captures only a limited aspect of program structure.

An interesting extension would be to incorporate richer structural context into the grammar and cost model, such as parent–child relationships, root-to-node derivation paths, or other local structural features of the syntax tree. Such representations could allow the model to distinguish not only where a production rule

occurs in terms of depth, but also the structural context in which it is applied. This may provide a more expressive and robust notion of the program shape, reducing sensitivity to misaligned depth information while retaining the benefits of structure-aware guidance. Exploring how such context-sensitive structural abstractions interact with probabilistic priors and trace-based feedback remains an open and promising direction for future work.

7.2.2 Heuristic-guided search

In this thesis, trace-based feedback is used as a source of graded information about partial program correctness. While effective on the KAREL benchmark, this approach relies on the availability of execution traces and domain-specific intermediate behaviour. An interesting direction for future work is to generalize this mechanism by framing guidance in terms of a heuristic search rather than trace matching.

From this perspective, both probabilistic priors and feedback-driven cost updates can be interpreted as heuristic signals that estimate how close a partial program is to a target solution. This suggests an extension in which the synthesis process is guided by explicit heuristic functions, like those used in A* search. Instead of relying on execution traces, such heuristics could score partial programs based on how much of the specification they already satisfy. This could be based on partial input–output agreement, constraint satisfaction, or other measures of progress toward the final program. Using an explicit notion of distance to the goal would allow feedback to be applied even in domains where traces are unavailable, and may provide a more direct signal than trace matching. How such heuristics interact with bottom-up enumeration and cost-based pruning is an open question.

7.2.3 Search dynamics and beam-limited enumeration

An important direction for future work concerns the interaction between guidance mechanisms and the underlying search dynamics. Throughout this thesis, synthesis is performed using a beam-limited bottom-up enumeration strategy, where only a fixed number of lowest-cost partial programs are kept at each step. While this makes the search tractable, the size of the beam can strongly influence which kinds of programs are explored.

In particular, the experiments on intermediate state availability demonstrate that strong early reinforcement can cause shallow programs to dominate the beam, preventing structurally richer candidates from being considered later in the search. This effect becomes more pronounced when feedback is dense or when cost updates strongly favour early partial matches. As a result, even informative guidance signals can lead to suboptimal search behaviour due to interactions with beam pruning.

A more systematic study of how beam width, pruning strategies, and cost update schedules affect the exploration of the program space would help clarify these dynamics. Understanding these interactions may lead to more robust search strategies that preserve the benefits of guidance while reducing sensitivity to early reinforcement effects.

Bibliography

R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 319–336. Springer, 2017.

B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

S. Barke, H. Peleg, and N. Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–29, 2020.

S. Barke, E. Anaya Gonzalez, S. R. Kasibatla, T. Berg-Kirkpatrick, and N. Polikarpova. Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems*, 37:15612–15645, 2025.

A. Cypher and D. C. Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

S. Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *International Joint Conference on Automated Reasoning*, pages 9–14. Springer, 2016.

S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL `https://doi.org/10.1561/2500000010`.

D. C. Halbert. *Programming by example*. University of California, Berkeley, 1984.

Herb-AI. Herb-ai/herbbenchmarks.jl: Benchmarks and problems for herb.jl. URL `https://github.com/Herb-AI/HerbBenchmarks.jl`.

S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224, 2010.

Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

A. Odena, K. Shi, D. Bieber, R. Singh, C. Sutton, and H. Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

N. Patton, K. Rahmani, M. Missula, J. Biswas, and I. Dillig. Programming-by-demonstration for long-horizon robot tasks. *Proceedings of the ACM on Programming Languages*, 8(POPL):512–545, 2024.

L. Zhang, G. Rosenblatt, E. Fetaya, R. Liao, W. Byrd, M. Might, R. Urtasun, and R. Zemel. Neural guided constraint logic programming for program synthesis. *Advances in Neural Information Processing Systems*, 31, 2018.

Appendix A

LLM Prompt Templates

This appendix documents the prompt templates used to query large language models throughout this thesis. We include them verbatim (except for placeholder blocks such as {GRAMMAR_BLOCK}), so that future work can reproduce or adapt our experimental setup.

A.1 Karel synthesis with trace constraints

Purpose. Synthesize a single Karel program in the target DSL such that it satisfies (i) the input–output grids and (ii) the provided intermediate states / traces.

Template.

```
You are a program synthesizer. Output ONLY one program in the target DSL
Do not explain. Do not add comments or extra text. If no solution fits th

# TASK
Given:
1) A domain-specific language (DSL) as a context-free grammar (CFG).
2) A set of input→output examples.
3) Execution traces / intermediate states that MUST be matched

Produce ONE program in the DSL that:
- Is syntactically valid w.r.t. the CFG.
- When executed, returns the required outputs for ALL examples.
- follows the same step-by-step behavior for those examples.

# OUTPUT FORMAT (STRICT)
Return ONLY the program, nothing else.
- return the complete program block only.
- Do NOT include backticks or prose.
```

```
# MINIMALITY & VALIDITY
- Prefer the simplest correct program (fewest nodes / constructs) when mu
- Use ONLY terminals and productions allowed by the provided CFG.
- Do NOT invent identifiers, predicates, or library calls outside the CFG
- Respect arities and types exactly.

# DSL (CFG)
    Start = Block

    Block = Action
    Block = (Action; Block)
    Block = ControlFlow

    Action = move
    Action = turnLeft
    Action = turnRight
    Action = pickMarker
    Action = putMarker

    ControlFlow = IF(Condition, Block)
    ControlFlow = IFELSE(Condition, Block, Block)
    ControlFlow = WHILE(Condition, Block)
    ControlFlow = REPEAT(R=INT, Block)
    INT = |(1:5)

    Condition = frontIsClear
    Condition = leftIsClear
    Condition = rightIsClear
    Condition = markersPresent
    Condition = noMarkersPresent
    Condition = NOT(Condition)

# WORLD REPRESENTATION
Each example takes place in a Karel world | a 2D grid surrounded by walls
- `.` represents an empty cell.
- `o` represents a cell containing a marker.
- `^`, `v`, `<`, `>` denote Karel's position and orientation (north, south
- Walls (`#`) cannot be traversed.
- Actions like `move`, `turnLeft`, `pickMarker`, and `putMarker` modify th
The goal is to transform the input world into the output world exactly, m

# SPECIFICATION
# Input→Intermediate→Output states the program MUST satisfy:
```

```
{EXAMPLES_BLOCK}

# CONSTRAINTS
- No I/O; compute only via allowed DSL primitives.
- Execution must terminate on all provided examples.

# SELF-CHECK BEFORE YOU ANSWER (NO OUTPUT OF THIS THINKING):
1) Parse your candidate against the CFG.
2) Simulate it on ALL inputs; confirm intermediate states and outputs mat
4) If any check fails, revise. Otherwise, output ONLY the final program.
```

Notes on placeholders.

- {EXAMPLES_BLOCK} is replaced by the concrete input → intermediate → output sequences for all training examples.

A.2 SyGuS function synthesis from grammar and I/O

Purpose. Synthesize a single SyGuS-style expression (as an s-expression) that completes a function body consistent with a provided grammar and input–output examples.

Template.

```
You are a coding assistant. Be precise and terse.
You will be given a SyGuS grammar and a set of input{output examples.
Your task is to complete the provided function definition with an impleme
Use only symbols and productions allowed by the grammar.
Your answer must be a single s-expression only, with no explanation or co

[GRAMMAR]
{GRAMMAR_BLOCK}

[EXAMPLES]
{EXAMPLES_BLOCK}
```

Notes on placeholders.

- {GRAMMAR_BLOCK} is replaced by the SyGuS grammar (nonterminals, productions, terminals).
- {EXAMPLES_BLOCK} is replaced by the input–output examples for the target function.