HySynth: Context-Free LLM Approximation for Guiding Program Synthesis

Shraddha Barke
UC San Diego
San Diego, USA
sbarke@ucsd.edu

Emmanuel Anaya Gonzalez
UC San Diego
San Diego, USA
fanayagonzalez@ucsd.edu

Saketh Ram Kasibatla
UC San Diego
San Diego, USA
skasibatla@ucsd.edu

Taylor Berg-Kirkpatrick
UC San Diego
San Diego, USA
tbergkirkpatrick@ucsd.edu

Nadia Polikarpova
UC San Diego
San Diego, USA
npolikarpova@ucsd.edu

Abstract

Many structured prediction and reasoning tasks can be framed as program synthesis problems, where the goal is to generate a program in a *domain-specific language* (DSL) that transforms input data into the desired output. Unfortunately, purely neural approaches, such as large language models (LLMs), often fail to produce fully correct programs in unfamiliar DSLs, while purely symbolic methods based on combinatorial search scale poorly to complex problems. Motivated by these limitations, we introduce a hybrid approach, where LLM completions for a given task are used to learn a task-specific, context-free surrogate model, which is then used to guide program synthesis. We evaluate this hybrid approach on three domains, and show that it outperforms both unguided search and direct sampling from LLMs, as well as existing program synthesizers.

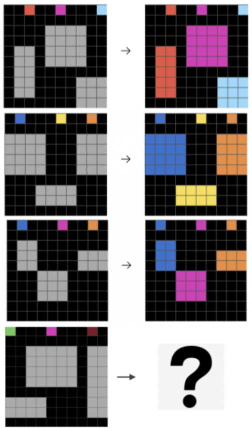
1 Introduction

Large language models (LLMs) demonstrate impressive capabilities in various domains, but they continue to struggle with tasks that require precision—e.g. structured prediction, reasoning, counting, or data transformation—when direct task examples are not prevalent in their training data [8, 12, 23, 31, 38, 40, 45]. As one example, consider the *Abstraction and Reasoning Corpus* (Arc) [14], which was designed as a benchmark for human-like structured reasoning. Arc tasks are grid-based puzzles, such as one depicted in Fig. 1a. This puzzle consists of three training examples, which are pairs of input and output grids; the goal is to infer the transformation that maps the input to the output, and then apply this transformation to the test grid. The Arc benchmark's emphasis on generalization and few-shot learning has rendered it challenging to solve with purely machine learning techniques: state-of-the-art generative models like GPT-4 hardly solve more than 10% of the tasks in the dataset when asked to predict the test output, even with the help of advanced prompting techniques [25].

In fact, the leading entries in the Arc Kaggle competition [1] tackle this task using *Programming-by-Example* (PBE): instead of predicting the output directly, they search for a program that captures the transformation occurring in the input-output examples. For example, the transformation in Fig. 1a might be represented as the following program:

$$\textbf{if}\ \texttt{color\_of}(\textbf{self}) = \texttt{GREY} \wedge \texttt{is\_neighbor}(\textbf{self},\ \texttt{other}) \wedge \texttt{size\_of}(\texttt{other}) = \texttt{MIN}\ \textbf{then}\ \texttt{update\_color}(\texttt{color\_of}(\texttt{other})) \quad (1)$$

38th Conference on Neural Information Processing Systems (NeurIPS 2024).



(a) ARC

```
Input Tensor:

in = [[[5., 2.],
       [1., 3.],
       [0., -1.]]]

Output Tensor:

out = [[[5.,5.],[1.,1.],[0.,0.]],
      [[2.,2.],[3.,3.],[-1.,-1.]]]

Natural language description
(optional):

“Index into the tensor”
```

(b) TENSOR

Input Strings	Outputs
"www.domain.com"	"com"
"mail.net"	"net"
"www.amazon.co.uk"	"uk"

(c) STRING

Figure 1: Example problems from the three PBE domains we evaluate HYSYNTH on: grid-based puzzles (ARC), tensor manipulation (TENSOR), and string manipulation (STRING).

This particular program is written in a *domain-specific language* (DSL) inspired by the ARGA tool [44]. It consists of a single *rule* of the form **`if`** *filter* **`then`** *transform*, which is applied to each object in the grid simultaneously; if the filter holds for the focus object **`self`** and another object `other`, then **`self`** undergoes the transform. In this case, the rule says that any grey object that has a neighbor of the grid's minimum size (here, a single pixel) should be colored with the color of that neighbor.

Beyond grid puzzles, PBE is a general paradigm for structured reasoning and data transformation tasks: for example, it can help spreadsheet users with systematic string manipulation [20], and help programmers use unfamiliar APIs [17, 18, 36]; Fig. 1 shows example PBE tasks from three domains.

Challenge: Harnessing the Power of LLMs for PBE How can we automatically learn programs from the input-output examples like those shown in Fig. 1? The traditional *program synthesis* approach is based on combinatorial search [2, 7, 34, 35, 39], which works well for small programs and restrictive DSLs, but becomes infeasible as the program size and the DSL complexity grow. At the other end of the spectrum, purely *neural* approaches [15, 42] use a neural model to predict the program from input-output examples; unfortunately, even state-of-art LLMs like GPT-4o [33] struggle to predict an entire program in an unfamiliar DSL: when we asked GPT-4o to generate 10 programs for the running example above, none of them were entirely correct.[1]

In the past, the limitations of both program synthesis and neural techniques have motivated a hybrid approach, where combinatorial search is *guided* by a learned probabilistic model [9, 24, 26, 32, 36, 37]. Existing hybrid techniques, however, use domain-specific models trained on datasets of similar PBE tasks, which limits their generalization to new domains. With the advent of LLMs, can we now use a single pre-trained model to guide program synthesis across a wide range of domains?

Interestingly, there is some tension in the hybrid approach between the efficiency of the search algorithm and the power of the model: a search algorithm is efficient when it *factorizes the search space* (*i.e.*, merges many search states into one), which often makes it incompatible with a powerful model that requires a lot of context to make a prediction. Specifically, one of the most widely used program synthesis techniques is *bottom-up search* [2, 11, 28, 36, 39], which is a dynamic programming algorithm, whose efficiency relies on reusing the work of constructing and evaluating subprograms in many different contexts. This essentially precludes using models with unlimited left-to-right context—like LLMs–to guide bottom-up search.

Our Solution: Context-Free LLM Approximation To bridge this gap and harness the power of LLMs to guide bottom-up search, we propose to approximate the LLM's conditional output distribution *for a given task* with a context-free surrogate model. Recent work in NLP [46] has found that a Hidden Markov Model (HMM) trained to match an LLM can be used as an efficient surrogate

[1] A detailed analysis of GPT-4o's performance on this task is provided in Appendix A.

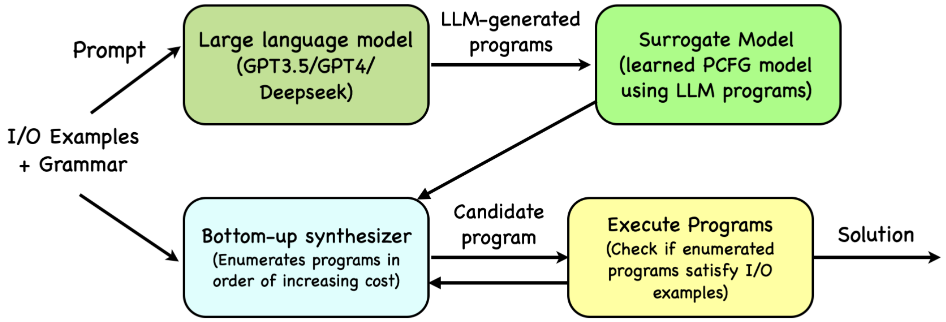

Prompt
Large language model (GPT3.5/GPT4/ Deepseek)
LLM-generated programs
Surrogate Model (learned PCFG model using LLM programs)
I/O Examples + Grammar
Bottom-up synthesizer (Enumerates programs in order of increasing cost)
Candidate program
Execute Programs (Check if enumerated programs satisfy I/O examples)
Solution

Figure 2: An overview of the hybrid program synthesis technique that uses a context-free LLM approximation. Programs generated by an LLM are used to learn a PCFG, which guides a bottom-up synthesizer to generate programs until a solution is found.

in style-controlled language generation. We extend this idea to program synthesis, replacing the HMM with a *probabilistic context-free grammar* (PCFG). The benefits of using a PCFG are twofold: (1) PCFGs are context-free, which makes them compatible with bottom-up search for PBE [11, 36], and (2) while a context-free model may make a poor approximation to an LLM's full joint, in a PBE setting it is able to reasonably approximate an LLM's conditional distribution over output programs *for a given prompt*. The overview of our approach is shown in Fig. 2.

Evaluation We implemented this technique in a tool HYSYNTH[2] and evaluated it on 299 PBE tasks from three domains: ARC grid-based puzzles [14], tensor manipulation tasks from TFCODER [36], and string manipulation tasks from the SYGUS benchmark [5], which are inspired by spreadsheet use cases. Example problems from these domains are shown in Fig. 1. Our evaluation shows that HYSYNTH outperforms both unguided search and LLMs alone, solving 58% of the tasks overall, compared to 40% for unguided search and 6% for LLMs without search. Our tool also outperforms baseline program synthesizers for these domains—ARGA, TFCODER, and PROBE [11], respectively; importantly, in the TENSOR domain, the guidance from the LLM not only speeds up the search, but also frees the user from having to explicitly provide any non-standard *constants* that the solution might use, thereby significantly improving the usability of the tool.

Contributions In summary, this paper makes the following contributions:

1. We propose a hybrid program synthesis approach that integrates LLMs with efficient bottom-up search via a task-specific context-free approximation.
2. We implement this approach in a tool HYSYNTH and instantiate it on three domains: grid-based puzzles (ARC), tensor manipulation (TENSOR), and string manipulation (STRING). While the latter two domains reuse off-the-shelf bottom-up synthesizers, for ARC we implement a custom synthesizer that uses a divide-and-conquer strategy [6] to leverage the structure of the rule-based DSL to further speed up the search.
3. We evaluate HYSYNTH on the three domains and show that it outperforms both the LLM alone and existing baseline synthesizers, which are not guided by LLMs.

2 Background

2.1 Programming-By-Example

Programming by Example (PBE) [21] is the task of synthesizing programs that satisfy a given set of input-output examples. To restrict the program space, the programs are typically drawn from a *domain-specific language* (DSL), which is specified by a *context-free grammar* and an *evaluation function*. This section provides a formal definition of these concepts.

Context-Free Grammars A *context-free grammar* (CFG) is a quadruple $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{R})$, where $\mathcal{N}$ is a set of non-terminal symbols, Σ is a set of terminal symbols, $\mathcal{S} \in \mathcal{N}$ denotes the starting

[2] The name stands for "HYbrid SYNTHesis" and is pronounced like the flower "hyacinth".

$$\begin{array}{rlrl}
\mathit{Rule} \rightarrow & \texttt{\textbf{if}}\ \mathit{Filter}\ \texttt{\textbf{then}}\ \mathit{Transform} & \mathit{Color} \rightarrow & \texttt{color\_of(}\mathit{Obj}\texttt{)} \mid \texttt{GREY} \mid \texttt{RED} \ldots \\
\mathit{Filter} \rightarrow & \mathit{Atom} \mid \texttt{not}\ \mathit{Atom} \mid \mathit{Atom} \land \mathit{Filter} \mid \ldots & \mathit{Size} \rightarrow & \texttt{size\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Atom} \rightarrow & \mathit{Color} =_c \mathit{Color} \mid \mathit{Size} =_s \mathit{Size} \mid \ldots & \mathit{Dir} \rightarrow & \texttt{dir\_of(}\mathit{Obj}\texttt{)} \mid \texttt{UP} \mid \texttt{DOWN} \mid \ldots \\
\mathit{Transform} \rightarrow & \texttt{update\_color(}\mathit{Color}\texttt{)} \mid \texttt{move(}\mathit{Dir}\texttt{)} \mid \ldots & \mathit{Obj} \rightarrow & \texttt{\textbf{self}} \mid \texttt{x} \mid \texttt{y} \mid \ldots
\end{array}$$

Figure 3: A fragment from the context-free grammar of our ARC DSL.

non-terminal, and $\mathcal{R}$ is the set of production rules. An example CFG is shown in Fig. 3. We denote with $\mathcal{R}(\mathrm{N})$ the set of all rules $\mathrm{R} \in \mathcal{R}$ whose left-hand side is N. A grammar $\mathcal{G}$ defines a (leftmost) *single-step derivation* relation on sequences of symbols: $s\mathrm{N}\alpha \Rightarrow s\beta\alpha$ if $\mathrm{N} \rightarrow \beta \in \mathcal{R}$, where $s \in \Sigma^*$ and $\alpha, \beta \in (\mathcal{N} \cup \Sigma)^*$. The transitive closure of this relation $\Rightarrow^*$ is called (leftmost) *derivation*.

Programs A *program* $P \in \Sigma^*$ is a terminal sequence derivable from some $\mathrm{N} \in \mathcal{N}$; we call a program *whole* if it is derivable from $\mathcal{S}$. The set of all programs is called the *language* of the grammar $\mathcal{G}$: $\mathcal{L}(\mathcal{G}) = \{s \in \Sigma^* \mid \mathrm{N} \Rightarrow^* s\}$. The *trace* of a program $\mathrm{tr}(P)$ is the sequence of production rules $\mathrm{R}_1, \ldots, \mathrm{R}_n$ used in its derivation ($\mathrm{N} \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow P$). The *size* of a program $|P|$ is the length of its trace. The semantics of a program P is defined by the evaluation function $[\![P]\!] \colon \mathrm{Val}^* \rightarrow \mathrm{Val}$, which maps the values of program variables to its output value.

Problem Statement A PBE problem is defined by a DSL with a grammar $\mathcal{G}$ and an evaluation function $[\![\cdot]\!]$, as well as a set of input-output examples $\mathcal{E} = \overrightarrow{\langle i, o \rangle}$ where $i \in \mathrm{Val}^*$, $o \in \mathrm{Val}$. A *solution* to the problem is a program $P \in \mathcal{L}(\mathcal{G})$ such that $\forall \langle i, o \rangle \in \mathcal{E}$, $[\![P]\!](i) = o$.

2.2 Assigning Costs to Programs

Weighted Context-free Grammar A *weighted context-free grammar* (WCFG) $\mathcal{G}_w$ is a pair of a CFG $\mathcal{G}$ and a function $w_{\mathbb{R}} : \mathcal{R} \rightarrow \mathbb{R}^+$ that maps each production rule $\mathrm{R} \in \mathcal{R}$ to a positive weight. Given a weighted grammar $\mathcal{G}_w$, we can define the *real cost* of a program P as the sum of weights of all the productions in its trace: $\mathrm{cost}_{\mathbb{R}}(P) = \sum_{\mathrm{R}_i \in \mathrm{tr}(P)} w_{\mathbb{R}}(\mathrm{R}_i)$.

For the purposes of search, it is convenient to define a *discrete weight* function $w : \mathcal{R} \rightarrow \mathbb{Z}^+$, which rounds weights up to the nearest integer: $w(\mathrm{R}) = \lceil w_{\mathbb{R}}(\mathrm{R}) \rceil$. The (discrete) *cost* of a program P is defined as the sum of discrete production weights: $\mathrm{cost}(P) = \sum_{\mathrm{R}_i \in \mathrm{tr}(P)} w(\mathrm{R}_i)$. Note that because of error accumulation, the discrete cost of a program can differ from its rounded real cost, but the difference can be made arbitrarily small by scaling all the costs by a constant factor $\alpha > 1$.

Probabilistic Context-free Grammar A popular way to assign weights to production rules is via a *probabilistic context-free grammar* (PCFG). A PCFG $\mathcal{G}_p$ is a pair of a CFG $\mathcal{G}$ and a function $p : \mathcal{R} \rightarrow [0, 1]$ that maps each production rule $\mathrm{R} \in \mathcal{R}$ to its probability, such that probabilities of all the rules for a given non-terminal $\mathrm{N} \in \mathcal{N}$ sum up to one: $\forall \mathrm{N}. \sum_{\mathrm{R} \in \mathcal{R}(\mathrm{N})} p(\mathrm{R}) = 1$. A PCFG defines a probability distribution on programs: $p(P) = \prod_{\mathrm{R}_i \in \mathrm{tr}(P)} p(\mathrm{R}_i)$.

Given a PCFG $(\mathcal{G}, p)$ we can derive a WCFG $\mathcal{G}_w$ where $w_{\mathbb{R}}(\mathrm{R}) = -\log(p(\mathrm{R}))$; to make sure that all weights are finite and positive, we exclude rules with $p(\mathrm{R}) = 0$ and inline rules with $p(\mathrm{R}) = 1$. In this WCFG, the real cost of a program is related to its probability: $\mathrm{cost}_{\mathbb{R}}(P) = -\log(p(P))$.

2.3 Bottom-up Search

Bottom-up search is a popular search technique in program synthesis [2, 11, 28, 36, 39], which enumerates programs from the DSL in the order of increasing costs until it finds a program that satisfies the given examples. The search is implemented as a dynamic programming algorithm (see Alg. 1), which maintains a program *bank* B mapping discrete costs to programs of that cost. Starting with an empty bank and current cost level $\textsc{Lvl} = 1$, the search iteratively creates all programs of cost 1, 2, 3, and so on; to create complex programs, the algorithm *reuses* simpler programs already stored in the bank, and combines them using the production rules of the grammar.

For example, consider the CFG in Fig. 3, and assume a uniform weight function $w(\cdot) = 1$. Then in the first iteration (cost level 1), the algorithm will enumerate programs consisting of a single literal or

Algorithm 1 Bottom-Up Search Algorithm

```
Input: Input-output examples $\mathcal{E}$, a WCFG $\mathcal{G}_w = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{R}, w)$
Output: A program $P$ consistent with $\mathcal{E}$ or failure ($\perp$)
 1: procedure BOTTOM-UP-SEARCH($\mathcal{G}_w, \mathcal{E}$)
 2:     $\textsc{Lvl}, \mathrm{B}, \mathrm{E} \leftarrow 1, \emptyset, \emptyset$                      ▷ Initialize state of the search
 3:     while true do
 4:         for $P \in \textsc{New-Programs}(\mathcal{G}_w, \textsc{Lvl}, \mathrm{B})$ do          ▷ For all programs of cost $\textsc{Lvl}$
 5:             $\textsc{Eval} \leftarrow [\langle i, [\![P]\!](i)\rangle \mid \langle i, o\rangle \in \mathcal{E}]$           ▷ Evaluate on inputs from $\mathcal{E}$
 6:             if ($\textsc{Eval} = \mathcal{E}$) then
 7:                 return $P$                              ▷ $P$ fully satisfies $\mathcal{E}$, solution found!
 8:             else if ($\textsc{Eval} \in \mathrm{E}$) then
 9:                 continue                  ▷ $P$ is semantically equivalent to another program in B
10:             $\mathrm{B}[\textsc{Lvl}] \leftarrow \mathrm{B}[\textsc{Lvl}] \cup \{P\}$                  ▷ Add to the bank, indexed by cost
11:             $\mathrm{E} \leftarrow \mathrm{E} \cup \textsc{Eval}$                               ▷ Cache evaluation result
12:         $\textsc{Lvl} \leftarrow \textsc{Lvl} + 1$
13:     return $\perp$                                              ▷ Cost limit reached
14: procedure NEW-PROGRAMS($\mathcal{G}_w$, $\textsc{Lvl}$, B)
15:     for $\mathrm{R} = \mathrm{N} \rightarrow s_0\mathrm{N}_1 s_1 \mathrm{N}_2 \dots \mathrm{N}_k s_k \in \mathcal{R}$ do     ▷ R is a production rule with $k$ non-terminals
16:         for $(c_1, \dots, c_k) \in \{\, [1..\textsc{Lvl} - 1]^k \mid \sum c_i = \textsc{Lvl} - w(\mathrm{R}) \,\}$ do    ▷ For all subexpression costs
17:             for $(P_1, \dots, P_k) \in \{\, \mathrm{B}[c_1] \times \dots \times \mathrm{B}[c_k] \mid \bigwedge_i \mathrm{N}_i \Rightarrow^* P_i \,\}$ do    ▷ For all subexpressions
18:                 yield $s_0 P_1 s_1 P_2 \dots\ P_k s_k$                ▷ Substitute subexpressions into R's RHS
```

variable—*e.g.* `self`, `GREY`, `UP`, *etc*—and store them in $\mathrm{B}[1]$. At cost level 2, it will enumerate unary operators applied to programs stored in $\mathrm{B}[1]$: *e.g.* `color_of(self)`, `move(UP)`, *etc*. More generally, at cost level $\textsc{Lvl}$, the algorithm considers all available productions, and for each production, enumerates all combinations of arguments whose costs sum up to $\textsc{Lvl} - 1$.

During search, each candidate expression is evaluated to see if it satisfies the examples (lines 5–7). Importantly, the search maintains a cache of all evaluation results E, and discards the newly constructed program if it is *observationally equivalent* to a program already in the bank (line 8), *i.e.* if it evaluates to the same output for all inputs in the examples. This step is the key to the efficiency of the bottom-up search algorithm: it allows the synthesizer to factorize the search space by evaluation result, significantly reducing the number of programs explored at each cost level.

3 The HySynth Approach

A key challenge in program synthesis is the astronomical size of the search space the synthesizer has to explore. For example, to find the program Eq. 1, the solution to the ARC task from the introduction, bottom-up search with a uniform weight function has to enumerate around 450K programs (all programs of size ≤ 16), which takes 4.5 minutes in our experiments.

On the other hand, sampling solutions to this task from an LLM yields programs that are *close* to the desired solution, even if not quite correct. As we show in Appendix A, GPT-4o uses relevant components `update_color`, `color_of`, and `is_neighbor` in nearly all of its solutions (usually missing some part of the filter or using the wrong color in the transform), and never uses irrelevant components like `move` or `rotate`. This suggests that the LLM generally has the right intuition about the components the solution needs to use; our insight is to leverage this intuition to guide bottom-up search by *assigning lower weights to the components that the LLM uses frequently*.

3.1 Guiding Bottom-up Search with Context-Free LLM Approximation

The overview of our approach, HySynth, is shown in Fig. 2. Given a PBE problem consisting of a DSL with grammar $\mathcal{G}$ and a set of input-output examples $\mathcal{E}$, HySynth proceeds in three steps.

Step 1: Sampling Solutions from an LLM HySynth starts by creating an LLM prompt that contains $\mathcal{G}$ and $\mathcal{E}$; the prompt can be optionally augmented with in-context examples if they are available for the given DSL. A complete prompt for the ARC running example can be found in Appendix B. The LLM is then used to sample a set $\{S_i\}_{i=1}^{N}$ of completions; the choice of N trades off computational cost and the faithfulness of the approximation to the true LLM conditional.

Step 2: Learning a PCFG from LLM Solutions Next, HySynth attempts to parse each completion S_i into a program P_i using the grammar $\mathcal{G}$. The resulting set of programs $\{P_i\}_{i=1}^{N'}$ (where $N' \leq N$) is used to learn a PCFG $\mathcal{G}_p$ via maximum likelihood estimation: $p(\mathrm{R}) = \frac{\mathrm{count}(\mathrm{R})+\alpha}{\sum_{\mathrm{R}\in\mathcal{R}} \mathrm{count}(\mathrm{R})+\alpha\times|\mathcal{R}|}$. Here $\mathrm{count}(\mathrm{R})$ is the frequency of rule R in all the derivations of the programs in $\{P_i\}$ and α is a smoothing parameter that ensures that every rule has a non-zero probability (typically set to 1).

Our experiments show that some models struggle to generate grammatical completions, leading to $N' \ll N$. To increase the sampling efficiency in those cases, HySynth implements *non-strict mode*, where ungrammatical completions S_i are not discarded. Instead the tool performs lexical analysis on S_i to convert it into a sequence of terminals and approximates the frequency of each production R based on the frequency of its *operator terminal*, a designated terminal of R, which represents a DSL operator; *e.g.* $\mathrm{count}(Atom \to \texttt{not}\ Atom) = \mathrm{count}(\texttt{not})$.[3]

Step 3: Guiding Bottom-up Search with PCFG Finally, HySynth uses the PCFG computed in the previous step to derive a weighted grammar $\mathcal{G}_w$ as explained in Sec. 2.2, and uses it to initialize the bottom-up search procedure in Alg. 1. As a result, the search is guided by the insights from the the LLM. For example, the WCFG learned from the GPT-4o completions for the Arc task above gives the relevant transform operator `update_color` weight 2, while all other *Transform* rules have weight 4; the relevant filter operators `color_of` and `is_neighbor` are similarly down-weighted. As a result, the search procedure only has to enumerate around 220K programs instead of 450K, achieving a 4x speedup, and solving the motivating example in just one minute with LLM guidance.

3.2 Domain-Specific Instantiations

We now describe how the HySynth approach is instantiated in three different domains: Arc grid puzzles, Tensor manipulations, and String manipulations.

Arc Domain An example task from this domain is shown in Fig. 1a and has been used as a running example throughout this paper. There is no established DSL for Arc, and arguably, DSL design is the biggest challenge when attempting to solve Arc using a PBE approach, since it is hard to capture the wide variety of tasks in this domain. Our DSL is inspired by the rule-based language of Arga [44], which we modified slightly to make it more compositional.

A program in our DSL is a sequence of rules of the form **`if`** *filter* **`then`** *transform*. A rule refers to the current object **`self`**, which is modified by the transform if the filter is satisfied in the current state of the grid. The rule can also refer to other objects in the grid, such as `other` in Eq. 1. This program is well-defined because its filter uniquely identifies the object `other`; if the filter is too weak to uniquely determine the effect of the transform, the program's output is considered undefined. The full grammar of our DSL can be found in Appendix H.

Instead of searching for a complete program using Alg. 1, we further optimize our synthesizer using a divide-and-conquer strategy inspired by [6], searching for filters and transforms *separately*. Specifically, HySynth-Arc first searches for transforms that are correct on some objects in the grid; once it has found a set of transforms that collectively describe all grid objects, it searches for filters that distinguish between the subsets of objects changed by each transform.

Consider once again our running example. When the transform synthesizer enumerates the expression `update_color(color_of(other))`, it detects that this transform works for all *grey objects*, because for each grey object **`self`** there exists a corresponding object `other` whose color can be copied. Now the goal of filter synthesis is to find a boolean expression that holds exactly for those pairs of objects (**`self`**, `other`) that make the transform work. See Appendix K for more details about this algorithm.

Tensor Domain This domain originates from the TFCoder synthesizer [36], which takes as input examples of a tensor transformation (with an optional natural language description) and synthesizes a TensorFlow program that performs the transformation. An example task from this domain is shown in Fig. 1b, whose solution is: `tf.gather_nd(in1, tf.stack((in2, in3), axis=-1))`. The main challenge, however, is that the TensorFlow grammar is very large (see Appendix G), and most importantly, the programs are allowed to use an *unbounded* set of constants. The original TFCoder

[3] Typically, the operator terminal uniquely identifies R, but when this is not the case, we can normalize $\mathrm{count}(\mathrm{R})$ by the number of rules in $\mathcal{R}$ that produce this terminal.

synthesizer requires the user to provide any non-standard constants that a task might require, and, according to their paper, this is the main barrier to the usability of their tool.

For program synthesis in this domain we use the TFCODER synthesizer off the shelf. TFCODER performs weighted bottom-up search, using a combination of hand-tuned weights and weights derived by two custom-trained neural models. HYSYNTH-TENSOR replaces these weights entirely with weights computed by sampling from an LLM. Importantly, our version of the tool does not require the user to provide any constants; instead we extract constants from the LLM completions, whereby significantly reducing the burden on the user.

STRING Domain Our third domain involves string manipulation tasks from the SYGUS competition [4], which are inspired by spreadsheet use cases. An example task, which requires extracting the top-level domain name from a URL, is shown in Fig. 1c. In this domain we use the PROBE [11] synthesizer off the shelf. PROBE performs weighted bottom-up search, starting with a uniform grammar and updating the weights on the fly; HYSYNTH-STRING instead initializes PROBE's search with weights derived from an LLM, and disables the weight updates during search.

4 Experiments and Results

4.1 Experimental Setup

We evaluate HYSYNTH on 299 PBE tasks from three different domains: ARC (160 tasks), STRING (70 tasks) and TENSOR (69 tasks).

ARC Benchmark The 160 ARC tasks are taken from the testing set of ARGA [44]. This *object-centric* subset of the full ARC corpus is known as OBJECT-ARC, and has been used to evaluate other ARC solvers [27]. ARC specifications consist of 2-7 input-output training grids and 1 testing grid. Correctness is based on whether the generated solution produces the correct output on the testing grid. Our ARC DSL has a total of 20 operations and 50 constants and variables across all types.

TENSOR Benchmark The 69 TENSOR tasks taken from TFCODER focus on tensor manipulation. 49 of them are sourced from StackOverflow inquiries, and 20 are from real-world scenarios faced by TensorFlow users at Google. The overall benchmark suite consists of 72 tasks. We use three of these tasks as in-context examples and evaluate on the rest. The grammar for this domain consists of 134 Tensorflow operations, primitives like `0, 1, -1, True` and other task-specific constants.

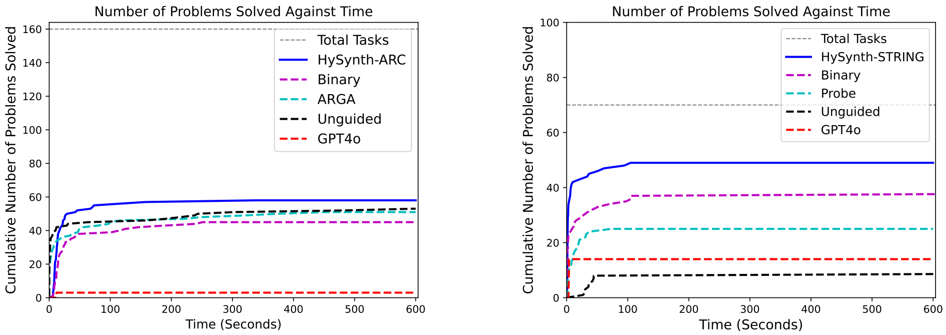
STRING Benchmark The 70 STRING tasks are taken from testing set of PROBE, which is derived from the SYGUS benchmark [4]. The number of examples ranges from 2 to 400. The original SYGUS benchmark have custom grammars for each task, but we use a union of all the grammars to make the search more challenging; the union grammar has 16 operations and 59 constants.

Configurations Our main HYSYNTH configuration uses GPT4O as the LLM, with 100 samples per task to learn a PCFG in non-strict mode (*i.e.* syntactically invalid completions are included in the PCFG learning process, as explained in Sec. 3.1). For each domain, we compare the performance of HYSYNTH with a baseline synthesizer for that domain (ARGA[4], PROBE, and TFCODER), as well as three ablations: (1) *no search*, *i.e.* using the 100 samples from the LLM directly, (2) *unguided search*, *i.e.* running the same synthesizer but with a uniform weighted grammar, and (3) *binary surrogate*, running the synthesizer but with a *binary PCFG*, *i.e.* a CFG that includes the components present in the LLM samples with equal probabilities, and excludes all other components completely. We also analyze the performance of HYSYNTH with different number of samples used to learn the PCFG (10, 20, and 50), with other LLMs (GPT3.5 and DEEPSEEK [22]), as well as in strict mode (which discards syntactically invalid LLM completions). The timeout is set to 10 minutes for all experiments and includes the search time and time to sample LLM completions (and compute PCFG). The average time to sample 100 solutions from GPT4O is 4 seconds, 12 seconds and 20 seconds per task for the STRING, ARC and TENSOR domains, respectively.

4.2 Results

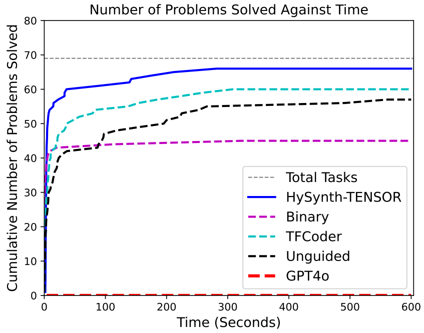
How does HYSYNTH compare to baselines and ablations? We compare the time to solution for the main HYSYNTH configuration, baseline synthesizers, and the three ablations; the results

[4] At the time of writing, ARGA is no longer state of the art on the OBJECT-ARC dataset; we explain in Sec. 5 why the comparison with ARGA is still relevant.



(a) HySynth-Arc results with GPT4o

(b) HySynth-String results with GPT4o



(c) HySynth-Tensor results with GPT4o

Domain/Model	% Valid completions
Tensor-GPT4o	99.9%
Tensor-DeepSeek	92.8%
String-GPT4o	37.5%
String-DeepSeek	0%
Arc-GPT4o	78.4%

(d) Percentage of syntactically valid completions

Figure 4: (a,b,c) Number of benchmarks solved by HySynth as a function of time for the Arc, Tensor, and String domains; timeout is 10 min. (d) Percentage of syntactically valid completions per domain.

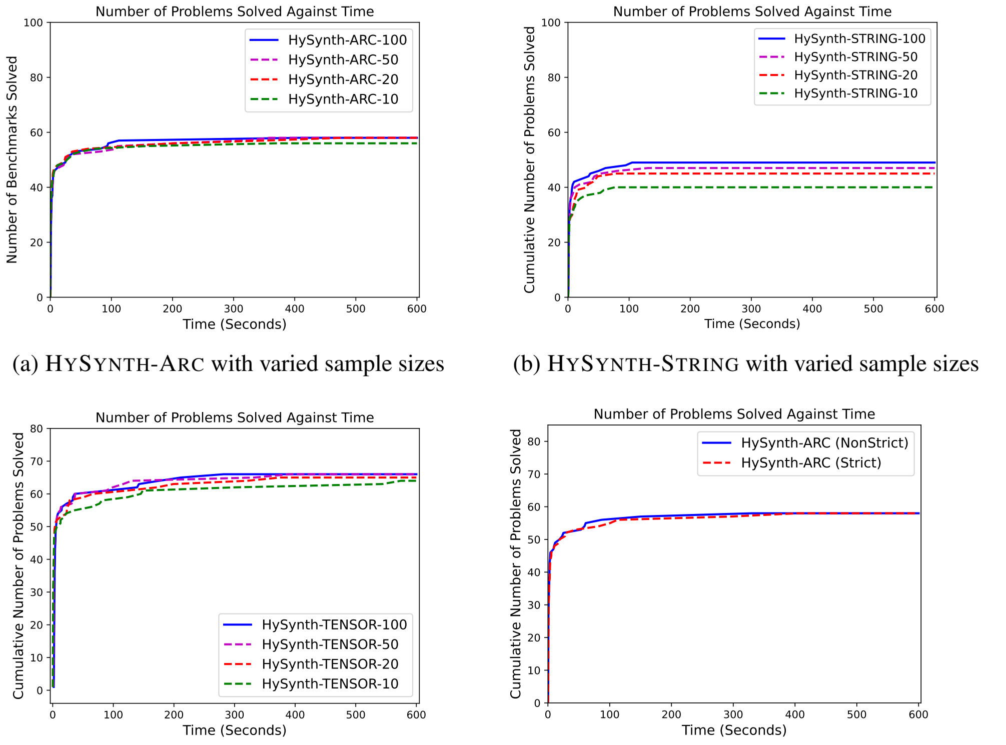
for the three domains are shown in Fig. 4a, Fig. 4b, and Fig. 4c. Overall, HySynth consistently outperforms both the baseline synthesizers and ablations, solving more tasks across all domains.

In more detail, direct LLM sampling performs very poorly on all domains, solving between 0 and 14 tasks; this confirms our hypothesis that LLMs struggle on PBE tasks in domain-specific languages, which are not prevalent in their training data. Interestingly, despite not being able to solve *any* Tensor tasks by itself, GPT4o provides excellent guidance for HySynth on that domain, helping it solve 96% of the total benchmark! On the other hand, synthesis guided by a binary surrogate model performs worse than HySynth (and even unguided search in case of Arc and Tensor) since the search excludes essential components from the grammar.

In String and Tensor domains, the baseline synthesizers predictably do better than unguided search, since both use the same search implementation, but with different weights. On Arc, however, our custom synthesizer outperforms Arga[5] even without LLM guidance; this speaks to the efficiency of the bottom-up search and the divide-and-conquer strategy we use, which are results of years of research in the program synthesis community.

How many samples are needed to learn a PCFG? To better understand how the number of samples affects the quality of PCFG guidance, we vary the number of GPT4o programs used in PCFG learning $N = 10, 20, 50, 100$, and once again measure the number of tasks solved over time. The results are shown in Fig. 5a, Fig. 5b, and Fig. 5c. As expected, larger sample sizes generally lead to better performance, but the difference is minimal: in Arc and Tensor, the difference between the best and worst performing versions of HySynth is only 2 problems each, while in String, HySynth solves 9 fewer problems with 10 samples than with 100. Despite these differences, all versions of HySynth still outperform the baseline and unguided search. This suggests that fewer samples are sufficient to effectively train a robust surrogate model, thereby optimizing costs.

[5] [44] report 57 tasks for Arga but we could only reproduce 51 on our hardware with a 10 minute timeout.



(a) HYSYNTH-ARC with varied sample sizes

(b) HYSYNTH-STRING with varied sample sizes

(c) HYSYNTH-TENSOR with varied sample sizes

(d) HYSYNTH-ARC with strict and non-strict modes

Figure 5: HYSYNTH-ARC, HYSYNTH-TENSOR and HYSYNTH-STRING results guided by a PCFG learned from different number of GPT4O samples (n=10, 20, 50, 100).

Do our results generalize to other models? To answer this question, we repeat our experiments on STRING and TENSOR domains with GPT3.5 and the open-source model `deepseek-coder-33b-instruct` (DEEPSEEK) [22]. The results with these models are detailed in Fig. 9 in Appendix C, and they corroborate the pattern observed with GPT4O, where the guided versions outperform the baseline, unguided search, and direct sampling from the LLM.

How important is non-strict mode? Fig. 4d shows the percentage of syntactically valid completions generated by GPT4O and DEEPSEEK (where applicable). You can see that while on TENSOR almost all completions are valid, this percentage falls to 78.4% for ARC and 37.5% for STRING; this is not surprising, given that the former are TensorFlow programs, which the model has seen during training, while the latter two are custom DSLs. In the STRING benchmark, the grammar is very restricted (*e.g.* only numeric constants allowed are 0-9), and the LLM has trouble adhering to this restricted grammar. But even if we were to relax the definition of syntactic validity, LLM solutions would achieve a syntactic validity of only 47%. Hence our non-strict mode proves especially helpful for low-resource domains, where otherwise we would have to discard a large proportion of completions. At the same time, we find that *given the same number of completions to learn from*, the PCFGs learned in non-strict mode are just as effective as those learned in strict mode: as shown in Fig. 5d, HYSYNTH-ARC with the guidance from 100 GPT4O completions solves 58 tasks *in either mode* (with the difference that strict mode has to sample more completions to get 100 valid ones).

4.3 Limitations

The main limitation of our hybrid approach *wrt.* purely neural approaches is that it requires implementing a synthesizer for each DSL of interest; although we have shown that the same bottom-up search can be used across different domains, some implementation effort is still required. On the other hand, compared to purely symbolic approaches, our method requires sampling from an LLM, which is costly; additionally, the guidance provided by our approach is only as good as the LLM's completions: if they contain many irrelevant operators, our guided search can be *slower* than unguided search. Finally, our experiments are subject to the usual threat that the LLMs might have seen our benchmarks in their training data; we do not consider it a major issue, however, given that our main result is the superior performance of guided search *relative* to using LLMs without search.

5 Related Work

Guiding Program Synthesis with Probabilistic Models The traditional approach to *program synthesis* is based on combinatorial search [7], augmented with pruning techniques based on program semantics [2, 6, 39]. To further speed up the search, researchers have proposed *guiding* the search with a learned probabilistic model. Most approaches to guided search use special-purpose models that have to be trained on a domain-specific corpus of programs [26] or PBE tasks [9, 24, 32, 37]. Although some of these models can be trained on synthetic data, the training process is still expensive and requires manual tuning, which makes it hard to apply these techniques to new domains.

With the advent of pretrained Large Language Models (LLMs), it seems only natural to use them to guide search-based program synthesis, thus alleviating the need for domain-specific training data. We are only aware of one other attempt to do this: concurrent work by Li et al. [29], which also extracts a PCFG from the LLM's samples, similarly to HySynth. An important difference is that they use the PCFG to guide *top-down* A* search, while we use it to guide *bottom-up* search, which is known to be more efficient (they also evaluate their tool on synthesis from logical formulas as opposed to PBE).

Solving the Abstraction and Reasoning Corpus All state-of-the-art solvers for this benchmark have relied on carefully curated DSLs for Arc [3, 13, 19, 27, 43]. Xu et al. [44] proposed the DSL we extend in our approach, and the Object-Arc subset we evaluate on. Lei et al. [27] embed their DSL as a subset of PDDL and use a Generalized Planning (GP) algorithm as their search component. They have the current best performance on Object-Arc, however they encode more domain-knowledge in the form of preconditions and per-abstraction restrictions on filters and transforms, to make GP viable. Our approach does not require this additional information. [3, 10] use DreamCoder [16], to perform execution-guided search over a DSL for grid manipulations, however they only provide proof-of-concept evaluations. [38, 41] also use an LLM to generate code given the spec of the task. Both of these approaches interact with the model across several rounds, while our technique uses the suggestions from the LLM only as a starting point. Our technique also performs a complete search guided by the LLM distribution, enabled by the structure of our DSL, whereas previous approaches only consider code directly generated by the LLM.

6 Conclusion and Future Work

Our approach introduces a robust technique for using both valid and invalid completions from an LLM to learn a surrogate model. By incorporating ungrammatical completions, we can extract useful insights that would otherwise be discarded. Overall, we provide an alternative to the conventional strategy of large-scale sampling from LLMs, proposing a more effective use of the available completions to guide the search process. An interesting future direction would be to guide search with a more expressive context-dependent surrogate model.

References

[1] 2020. Arc kaggle competition leaderboard. Accessed: 2024-05-19.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International Conference on Computer Aided Verification*, pages 934–950. Springer.

[3] Simon Alford, Anshula Gandhi, Akshay Rangamani, Andrzej Banburski, Tony Wang, Sylee Dandekar, John Chin, Tomaso Poggio, and Peter Chin. 2022. Neural-guided, bidirectional program search for abstraction and reasoning. In *Complex Networks & Their Applications X: Volume 1, Proceedings of the Tenth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2021 10*, pages 657–668. Springer.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa.

2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 1–8.

[5] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*.

[6] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. 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.

[7] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93.

[8] Xuefeng Bai, Jialong Wu, Yulong Chen, Zhongqing Wang, and Yue Zhang. 2023. Constituency parsing using llms. *arXiv preprint arXiv:2310.19462*.

[9] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

[10] Andrzej Banburski, Anshula Gandhi, Simon Alford, Sylee Dandekar, Sang Chin, and tomaso a poggio. 2020. Dreaming with ARC. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*.

[11] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29.

[12] Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. 2023. The reversal curse: Llms trained on" a is b" fail to learn" b is a". *arXiv preprint arXiv:2309.12288*.

[13] Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W Zhang, Michaël Defferrard, and Taco Cohen. 2023. Codeit: Abstract reasoning with iterative policy-guided program synthesis.

[14] François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

[15] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR.

[16] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*.

[17] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 420–435, New York, NY, USA. Association for Computing Machinery.

[18] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex apis. In *POPL*.

[19] Raphael Fischer, Matthias Jakobs, Sascha Mücke, and Katharina Morik. 2020. Solving abstract reasoning tasks with grammatical evolution. In *LWDA*, pages 6–10.

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

[21] Sumit Gulwani. 2016. Programming by examples (and its applications in data wrangling). In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Verification and Synthesis of Correct and Secure Systems*. IOS Press.

[22] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

[23] Martin Josifoski, Marija Sakota, Maxime Peyrard, and Robert West. 2023. Exploiting asymmetry for synthetic training data generation: Synthie and the case of information extraction. *arXiv preprint arXiv:2303.04132*.

[24] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*.

[25] Seungpil Lee, Woochang Sim, Donghyeon Shin, Sanha Hwang, Wongyu Seo, Jiwon Park, Seokki Lee, Sejin Kim, and Sundong Kim. 2024. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus.

[26] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449.

[27] Chao Lei, Nir Lipovetzky, and Krista A. Ehinger. 2024. Generalized planning for the abstraction and reasoning corpus.

[28] Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. 2024. Efficient bottom-up synthesis for programs with local variables. *Proc. ACM Program. Lang.*, 8(POPL).

[29] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding enumerative program synthesis with large language models.

[30] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195.

[31] R Thomas McCoy, Shunyu Yao, Dan Friedman, Matthew Hardy, and Thomas L Griffiths. 2023. Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*.

[32] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. 2020. Bustle: bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*.

[33] OpenAI. 2024. Hello gpt-4.0. Accessed: 2024-05-19.

[34] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630.

[35] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*, pages 74–83. Springer.

[36] Kensen Shi, David Bieber, and Rishabh Singh. 2022. Tf-coder: Program synthesis for tensor manipulations. *ACM Trans. Program. Lang. Syst.*, 44(2).

[37] Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. 2022. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*.

[38] John Chong Min Tan and Mehul Motani. 2023. Large language model (llm) as a system of multiple expert agents: An approach to solve the abstraction and reasoning corpus (arc) challenge. *arXiv preprint arXiv:2310.05146*.

[39] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296.

[40] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving llm code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632*.

[41] Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D Goodman. 2023. Hypothesis search: Inductive reasoning with language models. *arXiv preprint arXiv:2309.05660*.

[42] Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. 2024. Grounding data science code generation with input-output specifications.

[43] Johan Sokrates Wind. 2020. Arc kaggle competition, 1st place. Accessed: 2024-05-19.

[44] Yudong Xu, Elias B Khalil, and Scott Sanner. 2023. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4115–4122.

[45] Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B Khalil. 2023. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *arXiv preprint arXiv:2305.18354*.

[46] Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. 2023. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pages 40932–40945. PMLR.

```
// Solution 1, occurs 6 times
if color_of(self) = GREY ∧ is_neighbor(self, other)
        then update_color(color_of(other))

// Solution 2, occurs 1 time
if is_neighbor(self, other) ∧ color_of(other) = GREY
        then update_color(color_of(other))

// Solution 3, occurs 1 time
if color_of(self) = GREY
        then update_color(color_of(other))

// Solution 4, occurs 1 time
if not (color_of(self) = GREY) ∧ is_neighbor(self, other) ∧ color_of(other) = GREY
        then update_color(FUCHSIA)

// Solution 5, occurs 1 time
if size_of(self) = 4 then update_color(RED) ;
if size_of(self) = 4 ∧ color_of(self) = GREY then update_color(FUCHSIA) ;
if size_of(self) = 4 ∧ color_of(self) = BLUE then update_color(ORANGE) ;
if size_of(self) = 4 ∧ color_of(self) = YELLOW then update_color(CYAN)
```

Figure 6: Ten samples from GPT4o for the motivating example in Fig. 1a

A GPT4o Solutions for the Motivating Example

Recall the motivating example in Fig. 1a where the task is to update the color of the grey objects to the color of their single-pixel neighbor. As a reminder, the smallest correct solution to this task consists of the following rule:

```
if color_of(self) = GREY ∧ is_neighbor(self, x) ∧ size_of(x) = MIN
        then update_color(color_of(x))
```

Fig. 6 shows the programs we obtained by deduplicating 10 samples from GPT4o for this task. The syntax of the solutions is slightly modified for readability; our implementation uses a LISP-style s-expression syntax [30] to simplify parsing.

As you can see, the most frequent solution is almost correct, except that it does not constrain the neighbor `other` to be of size 1; this leads to the constraint being ambiguous (since every grey object has multiple neighbors of different colors), in which case the program semantics is considered undefined. That said, you can observe that the model consistently uses relevant components, such as `color_of`, `is_neighbor`, and `update_color`, which enables us to extract a useful PCFG from these solutions.

When we increased the sample size to 125, GPT4o was able to produce one correct solution (which is slightly larger than the minimal solution above):

```
if color_of(self) = GREY ∧ is_neighbor(self, other) ∧ not (color_of(other) = GREY)
        then update_color(color_of(other))
```

```
You are an assistant chatbot with human-like perception, reasoning and learning capabilities.
You can solve tasks concisely, efficiently, and moreover, correctly.
Let’s engage in perception and logic-based tasks.
You only output source code.
No explanations or any other text.
```

Figure 7: System prompt for ARC domain.

B LLM Prompt for the ARC Grammar

B.1 System Prompt

The system prompt given to the LLM for ARC domain is shown in Fig. 7.

B.2 User Prompt

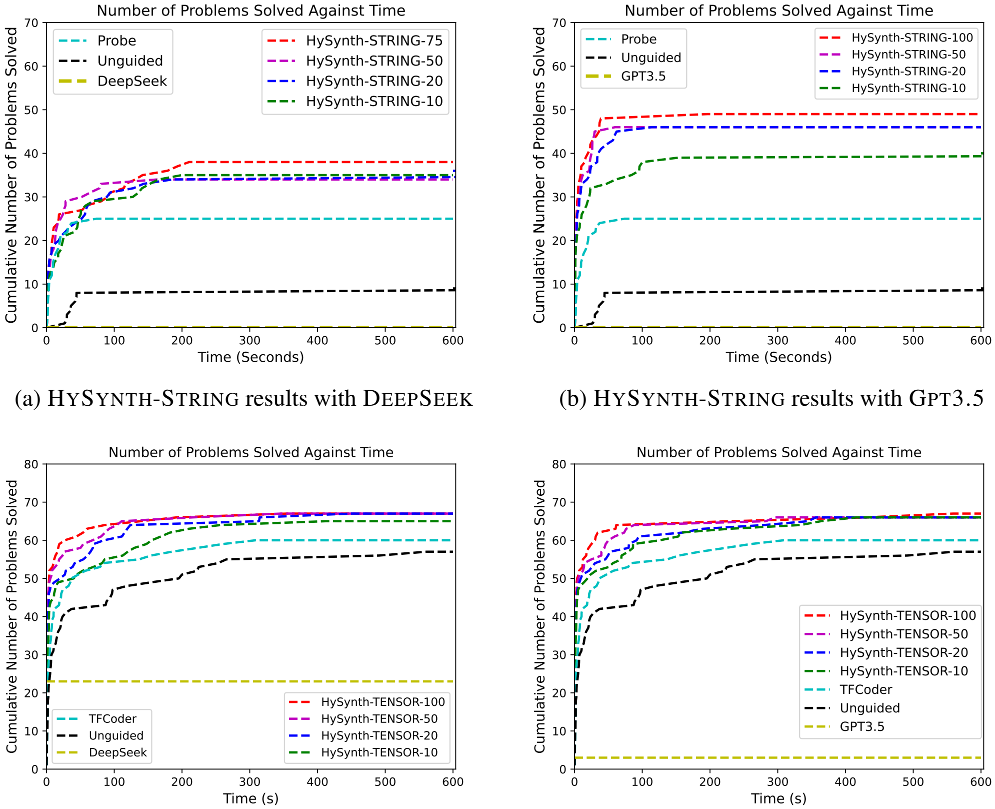
The full user prompt for the ARC domain is shown in Fig. 8. It contains the domain-specific language, four in-context examples and the query for the test task.

```
You are an efficient assistant for logical reasoning and code generation.
You will help me solve a visual perception and reasoning task.
I will first provide you with the definition of a Domain Specific Language you will use
    ↪ for writing a solution for the task.
I will then present you with the description of the task that you will be tested in.
You will then respond to the queries I make regarding the solution of the task.

This is the definition of the DSL you will use to solve the task.
It is given as a context-free grammar in the EBNF format used by the Lark parser
    ↪ generator, with some informative comments about the semantics.
You will return a string that is parseable by the ‘program‘ non-terminal of the grammar.

‘‘‘
library: "(" program∗ ")"

// Rules are executed one after another, in the order they appear.
// There could be no rules, in which case the program does nothing.
program: "(" "do" rule∗ ")"
...

⋘ DSL IMPLEMENTATION IN LARK ⋙

‘‘‘
Now we continue with the visual perception and reasoning task.
The input for the task is a small number of pairs of grids of characters.
The value of each of the cells of the grids are the colors defined in the DSL, so we can
    ↪  think of grids as images.
Each pair of images correspond to an input-output example for an unknown program P.
For each pair, the program P is evaluated on the image grid and operates on the objects
    ↪ that appear in it.
The output of the program is then the output image.
The objects in the images are easy and natural to identify for humans, so there is no
    ↪ need to define them explicitly.
However you are able to abstract them correctly, and the DSL is interpreted with the
    ↪ same correct abstraction.

Now I will show you some demonstration tasks along with the output you would be expected
    ↪  to produce for each of them.

## DEMONSTRATION TASK 1

### INPUT
PAIR 1
INPUT GRID:
0 0 0 0 0 0 0 0
0 0 0 0 0 R 0 0
0 R 0 0 0 R 0 R
0 R R 0 0 R 0 0
0 0 0 0 0 0 0 0
0 R R 0 0 0 0 0
0 R R 0 R R 0 0
0 0 0 0 0 0 0 0
```

Figure 8: User prompt for ARC domain.

```
OUTPUT GRID:
0 0 0 0 0 0 0 0
0 0 0 0 0 Y 0 0
0 Y 0 0 0 Y 0 Y
0 Y Y 0 0 Y 0 0
0 0 0 0 0 0 0 0
0 Y Y 0 0 0 0 0
0 Y Y 0 Y Y 0 0
0 0 0 0 0 0 0 0

<<< ENCODING OF EXAMPLE PAIR 2 AND 3 OF DEMO TASK 1>>>

### EXPECTED OUTPUT
{
    "nl_description": "Recolor all objects to color Y",
    "code": <<< EXPECTED CODE IN DSL >>>
}

<<< MORE DEMONSTRATION TASKS (4 IN TOTAL) >>>

Now follows task you will be evaluated on.
Output the solution as a JSON object, which should contain both a natural language
    ↪ description of the solution and the solution written in the DSL.
The code should be parseable by the DSL grammar.
The JSON must have the following structure:

{
    "nl_description": "TO_BE_FILLED",
    "code": "TO_BE_FILLED"
}

## TEST TASK

PAIR 1
INPUT GRID:
0 0 R 0 0 F 0 0 0 C
0 0 0 0 0 0 0 0 0 0
0 0 0 0 X X X X 0 0
0 0 0 0 X X X X 0 0
0 X X 0 X X X X 0 0
0 X X 0 X X X X 0 0
0 X X 0 0 0 0 0 0 0
0 X X 0 0 0 0 X X X
0 X X 0 0 0 0 X X X
0 0 0 0 0 0 0 X X X
OUTPUT GRID:
0 0 R 0 0 F 0 0 0 C
0 0 0 0 0 0 0 0 0 0
0 0 0 0 F F F F 0 0
0 0 0 0 F F F F 0 0
0 R R 0 F F F F 0 0
0 R R 0 F F F F 0 0
0 R R 0 0 0 0 0 0 0
0 R R 0 0 0 0 C C C
0 R R 0 0 0 0 C C C
0 0 0 0 0 0 0 C C C

<<< REST OF THE I/O EXAMPLES OF TEST TASK >>>
```

C Experimental results with LLMs DeepSeek and Gpt3.5



(a) HySynth-String results with DeepSeek

(b) HySynth-String results with Gpt3.5

(c) HySynth-Tensor results with DeepSeek

(d) HySynth-Tensor results with Gpt3.5

Figure 9: HySynth-String and HySynth-Tensor evaluation results with DeepSeek and Gpt3.5.

```
You are a coding assistant. Be precise and terse.
You will be provided a list of tensorflow operators, a task description, and some input/output examples.
Your task is to generate the body of a python function that will transform the input to the output.
Only use the operators provided in the list.
Your answer should be as short as possible while still being correct.
Make sure to only generate python code.
```

Figure 10: System prompt for TENSOR domain.

```
[TENSORFLOW OPERATORS]
⋘ see appendix E ⋙

[TASK DESCRIPTION]
index into the tensor

[INPUTS]
[[ 5.  2.]
 [ 1.  3.]
 [ 0. -1.]]


[OUTPUTS]
[[[ 5.  5.]
  [ 1.  1.]
  [ 0.  0.]]

 [[ 2.  2.]
  [ 3.  3.]
  [-1. -1.]]]

[PROGRAM]
def transform(in1):
```

Figure 11: User prompt for TENSOR domain

D LLM Prompt for the TENSOR Grammar

The system and user prompt for TENSOR domain are in Fig. 10 and Fig. 11.

```
You are a coding assistant. Be precise and terse.
You will be given a SyGuS grammar, a natural language specification, and a set of input-
    ↪ output examples.
Your task is to complete the provided function definition with an implementation that is
    ↪  correct according to the grammar, specification, and examples.
Your answer should be as short as possible while still being correct.
Make sure that your answer is a valid s-expression.
```

Figure 12: System prompt for STRING domain

```
[GRAMMAR]
(synth-fun f ((_arg_0 String)) String ((Start String (ntString)) (ntString String (
    ↪ _arg_0 "" " " "BRD" "DRS" "LDS" "Branding" "Direct Response" "Leads" "=" "/" "in
    ↪ " "_" "9" "." "microsoft" "windows" "apple" "mac" "-" "1" "2" "3" "4" "5" "6"
    ↪ "7" "8" "0" "," "<" ">" "/n" "%" "b" "apple" "bananas" "strawberries" "oranges"
    ↪ "LLC" "Inc" "Corporation" "Enterprises" "Company" "(" ")" "+" "name" "," (str.++
    ↪  ntString ntString) (str.replace ntString ntString ntString) (str.at ntString
    ↪ ntInt) (int.to.str ntInt) (ite ntBool ntString ntString) (str.substr ntString
    ↪ ntInt ntInt))) (ntInt Int (-1 1 2 3 4 5 6 7 8 9 0 1 0 -1 (+ ntInt ntInt) (-
    ↪ ntInt ntInt) (str.len ntString) (str.to.int ntString) (ite ntBool ntInt ntInt) (
    ↪ str.indexof ntString ntString ntInt))) (ntBool Bool (true false (= ntInt ntInt)
    ↪ (str.prefixof ntString ntString) (str.suffixof ntString ntString) (str.contains
    ↪ ntString ntString)))))

[NATURAL LANGUAGE SPECIFICATION]
; https=//exceljet.net/formula/get-top-level-domain-tld

[EXAMPLES]
www.domain.com → com
mail.net → net
www.amazon.co.uk → uk

[SOLUTION]
(define-fun f (_arg_0 String) String
```

Figure 13: User message for STRING

E LLM Prompt for STRING

The system and user prompt for STRING domain are in Fig. 12 and Fig. 13.

F The Full STRING Grammar

The full grammar for the STRING domain is detailed in Fig. 14.

$$
\begin{array}{rll}
\mathit{Start} \rightarrow & S & \\
S \rightarrow & \texttt{arg0} \mid \texttt{arg1} \mid \texttt{...} & \text{string variables} \\
\mid & \texttt{lit-1} \mid \texttt{lit-2} \mid \texttt{...} & \text{string literals} \\
\mid & (\texttt{replace}\ S\ S\ S) & \texttt{replace s x y}\text{ replaces first occurrence of }\texttt{x}\text{ in }\texttt{s}\text{ with }\texttt{y} \\
\mid & (\texttt{concat}\ S\ S) & \texttt{concat x y}\text{ concatenates }\texttt{x}\text{ and }\texttt{y} \\
\mid & (\texttt{substr}\ S\ I\ I) & \texttt{substr x y z}\text{ extracts substring of length }\texttt{z}\text{, from index }\texttt{y} \\
\mid & (\texttt{ite}\ B\ S\ S) & \texttt{ite x y z}\text{ returns }\texttt{y}\text{ if }\texttt{x}\text{ is true, otherwise }\texttt{z} \\
\mid & (\texttt{int.to.str}\ I) & \texttt{int.to.str x}\text{ converts int }\texttt{x}\text{ to a string} \\
\mid & (\texttt{at}\ S\ I) & \texttt{at x y}\text{ returns the character at index }\texttt{y}\text{ in string x} \\
B \rightarrow & \texttt{true} \mid \texttt{false} & \text{bool literals} \\
\mid & (\texttt{=}\ I\ I) & \texttt{= x y}\text{ returns true if }\texttt{x}\text{ equals }\texttt{y} \\
\mid & (\texttt{contains}\ S\ S) & \texttt{contains x y}\text{ returns true if }\texttt{x}\text{ contains }\texttt{y} \\
\mid & (\texttt{suffixof}\ S\ S) & \texttt{suffixof x y}\text{ returns true if }\texttt{x}\text{ is the suffix of }\texttt{y} \\
\mid & (\texttt{prefixof}\ S\ S) & \texttt{prefixof x y}\text{ returns true if }\texttt{x}\text{ is the prefix of }\texttt{y} \\
I \rightarrow & \texttt{arg0} \mid \texttt{arg1} \mid \texttt{...} & \text{int variables} \\
\mid & \texttt{lit-1} \mid \texttt{lit-2} \mid \texttt{...} & \text{int literals} \\
\mid & (\texttt{str.to.int}\ S) & \texttt{str.to.int x}\text{ converts string x to a int} \\
\mid & (\texttt{+}\ I\ I) & \texttt{+ x y}\text{ sums }\texttt{x}\text{ and }\texttt{y} \\
\mid & (\texttt{-}\ I\ I) & \texttt{- x y}\text{ subtracts }\texttt{y}\text{ from }\texttt{x} \\
\mid & (\texttt{length}\ S) & \texttt{length x}\text{ returns length of }\texttt{x} \\
\mid & (\texttt{ite}\ B\ I\ I) & \texttt{ite x y z}\text{ returns }\texttt{y}\text{ if }\texttt{x}\text{ is true, otherwise }\texttt{z} \\
\mid & (\texttt{indexof}\ S\ S\ I) & \texttt{indexof x y z}\text{ returns index of }\texttt{y}\text{ in }\texttt{x}\text{, starting at index }\texttt{z}
\end{array}
$$

Figure 14: The full SyGuS String grammar of the Probe benchmark suite. Integer and string variables and constants change per benchmark. Some benchmark files contain a reduced grammar.

G The Full TENSOR Grammar

```
General TensorFlow functions:
-----------------------------
tf.abs(x)
tf.add(x, y)
tf.add_n(inputs)
tf.argmax(input, axis)
tf.argmin(input, axis)
tf.argsort(values, axis, stable=True)
tf.argsort(values, axis, direction='DESCENDING', stable=True)
tf.boolean_mask(tensor, mask)
tf.broadcast_to(input, shape)
tf.cast(x, dtype)
tf.clip_by_value(t, clip_value_min, clip_value_max)
tf.concat(values, axis)
tf.constant(value)
tf.constant(value, dtype)
tf.divide(x, y)
tf.equal(x, y)
tf.exp(x)
tf.expand_dims(input, axis)
tf.eye(num_rows)
tf.eye(num_rows, num_columns)
tf.eye(num_rows, dtype)
tf.fill(dims, value)
tf.gather(params, indices)
tf.gather(params, indices, axis, batch_dims)
tf.gather_nd(params, indices)
tf.gather_nd(params, indices, batch_dims)
tf.greater(x, y)
tf.greater_equal(x, y)
tf.math.bincount(arr)
tf.math.ceil(x)
tf.math.count_nonzero(input)
tf.math.count_nonzero(input, axis)
tf.math.cumsum(x, axis)
tf.math.cumsum(x, axis, exclusive=True)
tf.math.divide_no_nan(x, y)
tf.math.floor(x)
tf.math.log(x)
tf.math.negative(x)
tf.math.reciprocal(x)
tf.math.reciprocal_no_nan(x)
tf.math.segment_max(data, segment_ids)
tf.math.segment_mean(data, segment_ids)
tf.math.segment_min(data, segment_ids)
tf.math.segment_prod(data, segment_ids)
tf.math.segment_sum(data, segment_ids)
tf.math.squared_difference(x, y)
tf.math.top_k(input, k)
tf.math.unsorted_segment_max(data, segment_ids, num_segments)
tf.math.unsorted_segment_mean(data, segment_ids, num_segments)
tf.math.unsorted_segment_min(data, segment_ids, num_segments)
tf.math.unsorted_segment_prod(data, segment_ids, num_segments)
tf.math.unsorted_segment_sum(data, segment_ids, num_segments)
```

Figure 15: List of TensorFlow operations as used in TFCODER.

```
tf.matmul(a, b)
tf.maximum(x, y)
tf.minimum(x, y)
tf.multiply(x, y)
tf.not_equal(x, y)
tf.one_hot(indices, depth)
tf.ones(shape)
tf.ones_like(input)
tf.pad(tensor, paddings, mode=’CONSTANT’)
tf.pad(tensor, paddings, mode=’CONSTANT’, constant_values)
tf.pad(tensor, paddings, mode=’REFLECT’)
tf.pad(tensor, paddings, mode=’SYMMETRIC’)
tf.range(start)
tf.range(start, limit, delta)
tf.reduce_any(input_tensor, axis)
tf.reduce_max(input_tensor)
tf.reduce_max(input_tensor, axis)
tf.reduce_mean(input_tensor)
tf.reduce_mean(input_tensor, axis)
tf.reduce_min(input_tensor)
tf.reduce_min(input_tensor, axis)
tf.reduce_prod(input_tensor, axis)
tf.reduce_sum(input_tensor)
tf.reduce_sum(input_tensor, axis)
tf.reshape(tensor, shape)
tf.reverse(tensor, axis)
tf.roll(input, shift, axis)
tf.round(x)
tf.searchsorted(sorted_sequence, values, side=’left’)
tf.searchsorted(sorted_sequence, values, side=’right’)
tf.sequence_mask(lengths)
tf.sequence_mask(lengths, maxlen)
tf.shape(input)
tf.sign(x)
tf.sort(values, axis)
tf.sort(values, axis, direction=’DESCENDING’)
tf.sqrt(x)
tf.square(x)
tf.squeeze(input)
tf.squeeze(input, axis)
tf.stack(values, axis)
tf.subtract(x, y)
tf.tensordot(a, b, axes)
tf.tile(input, multiples)
tf.transpose(a)
tf.transpose(a, perm)
tf.unique_with_counts(x)
tf.unstack(value, axis)
tf.where(condition)
tf.where(condition, x, y)
tf.zeros(shape)
tf.zeros_like(input)

SparseTensor functions:
-----------------------
tf.SparseTensor(indices, values, dense_shape)
tf.sparse.add(a, b)
tf.sparse.concat(axis, sp_inputs)
tf.sparse.expand_dims(sp_input, axis)
tf.sparse.from_dense(tensor)
tf.sparse.maximum(sp_a, sp_b)
tf.sparse.minimum(sp_a, sp_b)
tf.sparse.reduce_max(sp_input, axis, output_is_sparse)
tf.sparse.reduce_sum(sp_input, axis, output_is_sparse)
tf.sparse.reset_shape(sp_input)
tf.sparse.reshape(sp_input, shape)
tf.sparse.retain(sp_input, to_retain)
tf.sparse.slice(sp_input, start, size)
tf.sparse.split(sp_input, num_split, axis)
tf.sparse.to_dense(sp_input)
tf.sparse.to_dense(sp_input, default_value)
tf.sparse.to_indicator(sp_input, vocab_size)
tf.sparse.transpose(sp_input)
tf.sparse.transpose(sp_input, perm)

Python-syntax operations:
-------------------------
IndexingAxis1Operation: arg1[:, arg2]
IndexingOperation: arg1[arg2]
PairCreationOperation: (arg1, arg2)
SingletonTupleCreationOperation: (arg1,)
SlicingAxis0BothOperation: arg1[arg2:arg3]
SlicingAxis0LeftOperation: arg1[arg2:]
SlicingAxis0RightOperation: arg1[:arg2]
SlicingAxis1BothOperation: arg1[:, arg2:arg3]
SlicingAxis1LeftOperation: arg1[:, arg2:]
SlicingAxis1RightOperation: arg1[:, :arg2]
TripleCreationOperation: (arg1, arg2, arg3)
```

$$
\begin{array}{rl}
\mathit{Rule} \rightarrow & \texttt{\textbf{if}}\ \mathit{Filter}\ \texttt{\textbf{then}}\ \mathit{Transforms} \\
\mathit{Transforms} \rightarrow & \mathit{Transform} \mid \mathit{Transform}\ \texttt{;}\ \mathit{Transforms} \\
\mathit{Filter} \rightarrow & \mathit{Atom} \mid \texttt{not}\ \mathit{Atom} \mid \mathit{Atom} \wedge \mathit{Filter} \mid \mathit{Atom} \vee \mathit{Filter} \\
\mathit{Atom} \rightarrow & \mathit{Color} =_c \mathit{Color} \mid \mathit{Size} =_s \mathit{Size} \mid \mathit{Degree} =_d \mathit{Degree} \mid \mathit{Width} =_w \mathit{Width} \mid \mathit{Height} =_h \mathit{Height} \\
\mid & \mathit{Shape} =_S \mathit{Shape} \mid \mathit{Row} =_r \mathit{Row} \mid \mathit{Column} =_C \mathit{Column} \mid \texttt{is\_neighbor}\ \texttt{(}\mathit{Obj}, \mathit{Obj}\texttt{)} \\
\mathit{Transform} \rightarrow & \texttt{update\_color(}\mathit{Color}\texttt{)} \mid \texttt{move(}\mathit{Dir}\texttt{)} \mid \texttt{move\_max(}\mathit{Dir}\texttt{)} \mid \texttt{extend(}\mathit{Dir}, \mathit{Overlap}\texttt{)} \\
\mid & \texttt{rotate(}\mathit{Angle}\texttt{)} \mid \texttt{fill\_rectangle(}\mathit{Color}, \mathit{Overlap}\texttt{)} \mid \texttt{hollow\_rectangle(}\mathit{Color}\texttt{)} \\
\mid & \texttt{mirror(}\mathit{Axis}\texttt{)} \mid \texttt{add\_border(}\mathit{Color}\texttt{)} \mid \texttt{flip(}\mathit{Axis}\texttt{)} \mid \texttt{NoOp} \\
\mathit{Obj} \rightarrow & \texttt{\textbf{self}} \mid \texttt{x} \mid \texttt{y} \mid \ldots \\
\mathit{Color} \rightarrow & \texttt{color\_of(}\mathit{Obj}\texttt{)} \mid \texttt{GREY} \mid \texttt{RED} \mid \texttt{BLACK} \mid \texttt{BLUE} \mid \texttt{YELLOW} \mid \texttt{ORANGE} \mid \texttt{BROWN} \mid \texttt{GREEN} \mid \texttt{GREY} \mid \texttt{FUCHSIA}\ldots \\
\mathit{Dir} \rightarrow & \texttt{dir\_of(}\mathit{Obj}\texttt{)} \mid \texttt{UP} \mid \texttt{DOWN} \mid \texttt{LEFT} \mid \texttt{RIGHT} \mid \texttt{UPLEFT} \mid \texttt{DOWNLEFT} \mid \texttt{UPRIGHT} \mid \texttt{DOWNRIGHT}\ldots \\
\mathit{Axis} \rightarrow & \texttt{axis\_of(}\mathit{Obj}\texttt{)} \mid \texttt{VERTICAL} \mid \texttt{HORIZONTAL} \mid \texttt{LEFTDIAGONAL} \mid \texttt{RIGHTDIAGONAL}\ldots \\
\mathit{Overlap} \rightarrow & \texttt{TRUE} \mid \texttt{FALSE} \\
\mathit{Angle} \rightarrow & \texttt{90} \mid \texttt{180} \mid \texttt{270} \\
\mathit{Size} \rightarrow & \texttt{size\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Degree} \rightarrow & \texttt{degree\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Width} \rightarrow & \texttt{width\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Height} \rightarrow & \texttt{height\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Column} \rightarrow & \texttt{column\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Row} \rightarrow & \texttt{row\_of(}\mathit{Obj}\texttt{)} \mid \texttt{MIN} \mid \texttt{MAX} \mid \ldots \\
\mathit{Shape} \rightarrow & \texttt{shape\_of(}\mathit{Obj}\texttt{)} \mid \texttt{ENCLOSED} \mid \texttt{SQUARE} \mid \ldots
\end{array}
$$

Figure 16: The full grammar for our ARC DSL, object specific parameters like size, degree change per benchmark.

H The Full ARC DSL

The full grammar of our ARC DSL is shown in Fig. 16.

I Detailed Prompt Settings

For ARC, we sample completions with temperature 1 and 4000 max tokens. For TENSOR, we use temperature 1 and 4000 max tokens. For SYGUS, we use temperature 0.5 and 4000 max tokens. We use the same settings for all three LLMs. When prompting GPT4O, we set `response_type` to JSON.

J Broader Research Impacts

Our technique presents a powerful strategy for harnessing both syntactically valid and invalid outputs from an LLM to learn a surrogate model. Incorporating hallucinatory outputs – often erroneous generated by the model, allows us to extract insights that are discarded in standard practices. Our approach mitigates the need for large-scale sampling of completions from LLMs, promoting a more efficient and effective utilization of these models, saving resources. In addition to improving the cost effectiveness of using LLMs, it also opens up new avenues for enhancing model robustness and adaptability across different domains.

Algorithm 2 ARC Synthesis Algorithm

```
Input: A set of input-output example grids $\mathcal{E}$, transform grammar $\mathcal{G}_t$ and filter grammar $\mathcal{G}_f$
Output: A solution map $\mathcal{M}$ from each transform to the corresponding filter
procedure HYSYNTH-ARC($\mathcal{E}$, $\mathcal{G}_p$, $\mathcal{G}_t$)
    $\text{LVL}, \text{B} \leftarrow 0, \emptyset$                                  ▷ Initialize search state
    $\mathcal{S}_{\text{llm}} \leftarrow \text{LLM}(\mathcal{E})$                           ▷ Sample solutions from the LLM
    $\mathcal{G}_p, \mathcal{G}_t \leftarrow \text{INIT}(\mathcal{G}_p, \mathcal{S}_{\text{llm}}), \text{INIT}(\mathcal{G}_t, \mathcal{S}_{\text{llm}})$   ▷ Initialize both PCFGs using LLM solutions
    while not timeout do
        $\mathcal{O} \leftarrow \text{TRANSFORM-SEARCH}(\mathcal{G}_t, \mathcal{E})$          ▷ Synthesize transforms that cover all objects
        $\mathcal{M} \leftarrow \text{FILTER-SEARCH}(\mathcal{G}_f, \mathcal{E}, \mathcal{O})$      ▷ Synthesize filters for the above transforms
        if $\forall (t, f) \in \mathcal{M}, f \neq \bot$ then                ▷ Found a filter for each transform
            return $\mathcal{M}$                                    ▷ Return the complete solution
```

Algorithm 3 Transform Synthesis Algorithm

```
Input: PCFG $\mathcal{G}_t$ and input-output grids $\mathcal{E}$
Output: A map $\mathcal{O}$ from transforms to correctly changed objects
 1: procedure TRANSFORMS-SEARCH($\mathcal{G}_t, \mathcal{E}$)
 2:     $\text{LVL}, \text{B}, \text{E} \leftarrow 0, \emptyset, \emptyset$                              ▷ Initialize search state
 3:     while $\text{LVL} \leq \text{LIM}$ do
 4:         for $T \in \text{NEW-TRANSFORMS}(\mathcal{G}_t, \text{LVL}, \text{B})$ do          ▷ For all transforms with cost LVL
 5:             $\text{EVAL} \leftarrow \{[\![T]\!](\omega_i) \mid \langle i, o \rangle \in \mathcal{E}, \omega_i \in i\}$   ▷ Apply transform on objects in input grids from $\mathcal{E}$
 6:             if $\text{EVAL} \cap \bigcup_{\langle i,o \rangle \in \mathcal{E}} \{\omega_o \mid \omega_o \in o\} \neq \emptyset$ then       ▷ $T$ covers a subset of objects
 7:                 $\mathcal{O}[T] \leftarrow \text{EVAL}$                      ▷ Store the transform and objects covered by it
 8:             else if $\text{EVAL} \in \text{E}$ then
 9:                 continue                        ▷ $T$ is observationally equivalent to another transform in B
10:             if $\bigcup_{T \in \mathcal{O}} \mathcal{O}[T] = \bigcup_{\langle i,o \rangle \in \mathcal{E}} \{\omega_o \mid \omega_o \in o\}$ then       ▷ All objects are correctly transformed
11:                 return $\mathcal{O}$
12:             $\text{B}[\text{LVL}] \leftarrow \text{B}[\text{LVL}] \cup \{T\}$          ▷ Add transform to the bank, indexed by cost for later search
13:             $\text{E} \leftarrow \text{E} \cup \text{EVAL}$                                  ▷ Cache evaluation result
14:         $\text{LVL} \leftarrow \text{LVL} + 1$
15:     return $\bot$                                               ▷ Cost limit reached
```

K The ARC Synthesis Algorithm

Overall Synthesis Algorithm The overall synthesis algorithm takes as input a set of input-output grids $\mathcal{E}$, along with grammars $\mathcal{G}_t$ and $\mathcal{G}_f$. We sample candidate solutions from an LLM by constructing a prompt using $\mathcal{E}$. These solutions are used to initialize the weights of production rules in the transform and filter grammars, $\mathcal{G}_t$ and $\mathcal{G}_f$, respectively. We optimize the search by using a divide and conquer approach: first, a TRANSFORM-SEARCH procedure searches for transforms, mapping each to its correctly transformed objects in $\mathcal{O}$. Following this, a search for filters is initiated using the FILTER-SEARCH procedure. If a filter is found for each transform, the algorithm terminates and returns $\mathcal{M}$, which maps each transform to its corresponding filter. The algorithm described above terminates after the first solution is found, but we keep searching for a smaller set of transforms [6].

Transform Search Algorithm The transform synthesis algorithm in Algorithm 3 takes as input a PCFG $\mathcal{G}_t$ and $\mathcal{E}$. It enumerates transforms in the order of increasing discrete costs according to $\mathcal{G}_t$.

The algorithm starts with the following initial state: 1) a cost level (LVL) equal to 0 in order to keep track of the current cost during enumeration, 2) a program bank (B) that indexes the enumerated transforms by their cost for efficient retrieval, and 3) an evaluation cache (E) that stores the result of all evaluated transforms within B. At each iteration, the algorithm explores the space of all new transforms generated by the NEW-TRANSFORMS procedure for the current cost level.

On line 5 in Algorithm 3, the enumerated transform T is applied to each object in the input grids from $\mathcal{E}$. If T correctly transforms a subset of objects, T and the objects covered by it are stored in map $\mathcal{O}$ indexed by the transform (line 7). When the transforms in $\mathcal{O}$ cover all grid objects, $\mathcal{O}$ is returned (line 10-11). For transforms with objects bound by a filter, such as in `update_color(color_of(other))`, we consider all possible values (of color) that could be assigned and yield concrete transforms corresponding to each of those assignments.

Filter Search Algorithm The filter search algorithm takes as input a filter PCFG $\mathcal{G}_f$, $\mathcal{E}$, and the map $\mathcal{O}$ returned by the transform search in Algorithm 3. The filter search proceeds in a similar manner as the transforms search wherein it enumerates filters in the order of increasing cost as per the PCFG $\mathcal{G}_f$. It initiates a new search to find a filter for each transform in $\mathcal{O}$. Each enumerated filter expression is evaluated on all objects in the input grids. If the objects for which the filter is True are the same as the objects covered by the transform, we have found a filter for this transform. Once a filter is found for each of the transforms in $\mathcal{O}$, we return the solution map $\mathcal{M}$.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The main claims accurately reflect the paper's contributions and scope.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The authors have created a separate "Limitations" section (4.3) included in the experiments section of the paper.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: All the results presented in the paper can be reproduced using the LLM generations and synthesizers, both of which are publicly available on GitHub.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The authors provide a detailed description of the datasets. Instructions for reproducing the results, along with the actual results in the paper, are included in the supplemental material.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper and Github repository provide all the necessary details to reproduce the experimental results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The non-deterministic component of our experiments is the LLM sampling and the results are validated by performing the experiments across different models and domains, and also for different number of LLM samples.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The experimental setup details are provided in the experiments section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in the paper conforms with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The paper discusses the potential impacts of the work in the conclusion and the "Broader Impacts" section in the appendix.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.

- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The authors reference all relevant papers related to existing assets.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: The authors have included experimental results in the supplemental data and publicly released the ARC synthesizer on GitHub, complete with documentation.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.