Benchmarking Generative Models on Computational Thinking Tests in Elementary Visual Programming

Victor-Alexandru Pădurean
MPI-SWS
vpadurea@mpi-sws.org

Adish Singla
MPI-SWS
adishs@mpi-sws.org

Abstract

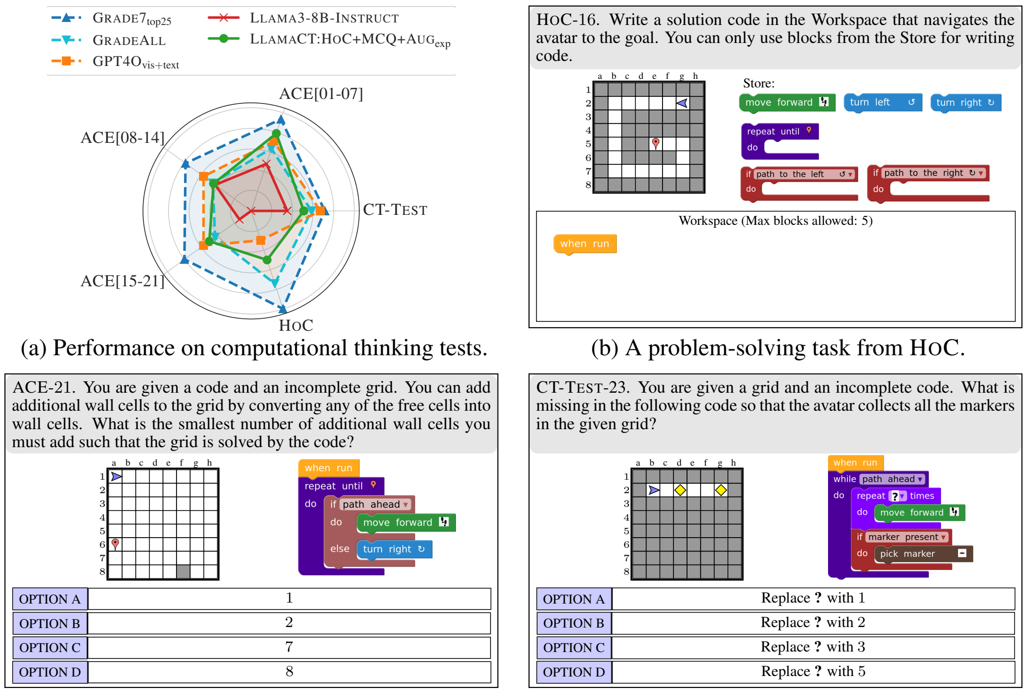
Generative models have demonstrated human-level proficiency in various benchmarks across domains like programming, natural sciences, and general knowledge. Despite these promising results on competitive benchmarks, they still struggle with seemingly simple problem-solving tasks typically carried out by elementary-level students. How do state-of-the-art models perform on standardized programming-related tests designed to assess computational thinking and problem-solving skills at schools? In this paper, we curate a novel benchmark involving computational thinking tests grounded in elementary visual programming domains. Our initial results show that state-of-the-art models like GPT-4o and Llama3 barely match the performance of an average school student. To further boost the performance of these models, we fine-tune them using a novel synthetic data generation methodology. The key idea is to develop a comprehensive dataset using symbolic methods that capture different skill levels, ranging from recognition of visual elements to multi-choice quizzes to synthesis-style tasks. We showcase how various aspects of symbolic information in synthetic data help improve fine-tuned models' performance. We will release the full implementation and datasets to facilitate further research on enhancing computational thinking in generative models.

1 Introduction

The recent advances in generative models and large language models (LLMs) have the potential to positively impact a wide variety of domains, such as medicine [1, 2, 3], arts [4, 5], and education [6, 7, 8, 9]. This potential is reflected by their success on a wide range of popular competitive benchmarks assessing their knowledge of natural sciences and day-to-day facts [10, 11, 12, 13, 14] and their skills in programming. For example, GPT-4o [10] is capable of obtaining a high accuracy on two popular programming benchmarks: 90.2% on HumanEval [15] and 87.5% on MBPP [16]. Previous studies also showed that GPT-4 [17] is capable of passing assessments in higher education programming courses, achieving course totals greater than 79% [18].

Despite these promising results, state-of-the-art models struggle with seemingly simple tasks. These models often underperform in tasks requiring mathematical reasoning, planning, and problem-solving [19, 20, 21, 22]. For example, they fail to solve planning tasks involving stacking of colored blocks [23]. Moreover, generative models often face problems with basic algebra and counting [19], or coming up with correct codes in visual programming domains [24], tasks which can successfully be carried out by elementary-level school students. These weaknesses seem to contradict the generative models' impressive performance in complex programming tasks. Based on these observations, we aim to study how generative models tackle programming tasks specifically designed to foster computational thinking and problem-solving skills in elementary-level students. This leads to our main research question: *How do state-of-the-art models perform on standardized programming-related tests designed to assess computational thinking and problem-solving skills at schools?*

38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks.



(a) Performance on computational thinking tests.

(b) A problem-solving task from HOC.

(c) A multi-choice question task from ACE.

(d) A multi-choice question task from CT-TEST.

Figure 1: **(a)** shows the performance of school students compared to various models on a scale of 0 to 100. We break down the ACE [25] test into its constituent parts: *Analyzing* (ACE[01-07]), *Evaluating* (ACE[08-14]), and *Creating* (ACE[15-21]). **(b)** shows MAZE16 from *Hour of Code:Maze Challenge* (HOC) [26, 27], an example of a solution synthesis task. **(c)** shows a *Creating* multi-choice question task from the ACE test. **(d)** shows a multi-choice question task from the CT-TEST [28, 29].

In this paper, we introduce a novel benchmark for assessing generative models' computational thinking and problem-solving capabilities. We conduct extensive experiments with various models and our results show that state-of-the-art models struggle with the computational thinking tests in our benchmark. Figure 1a illustrates how GPT-4o barely matches the performance of an average school student, with Llama3 [11] performing even worse.

We make the following contributions towards improving the models' performance on computational thinking tests: (1) We introduce a novel data generation methodology based on symbolic methods. An important aspect of the generated dataset is that it captures different skill levels, ranging from recognition of visual elements to multi-choice questions to synthesis-style tasks. (2) We fine-tune Llama3-8B and obtain the LLAMACT family of models, the best of which achieves an accuracy on par with GPT-4o (see Figure 1a). We further analyze how various aspects of symbolic information in our synthetic dataset help improve the fine-tuned models' performance. (3) We will release the data and implementation to promote further research on enhancing computational thinking in generative models.[1]

2 Related Work

We identify two key research themes in the literature: one focuses on the programming capabilities of generative models, and the other focuses on their general reasoning capabilities. To our knowledge, this paper is the first to evaluate numerous generative models on a comprehensive set of computational thinking tests grounded in elementary visual programming. Figure 2 presents a comparison between our work and the most relevant benchmarks in the literature.

Benchmarks assessing programming capabilities. Several works benchmark the programming capabilities of models, with popular examples of benchmarks including HumanEval [15], MBPP [16], and APPS [33]. For example, HumanEval [15] focuses on Python code generation but lacks mul-

[1]GitHub repo: `https://github.com/machine-teaching-group/neurips2024-benchmark-ct`.

Work	Domain	Evaluation tasks	Multimodal	Benchmark evaluation size	Trained model	Human comparison
Our Work	visual programming	code synthesis, multiple choice questions	✓	65	✓	✓
HumanEval [15]	programming	Python code writing	✗	164	✓	✗
MMCode [30]	programming	Python code writing	✓	$3,548$	✗	✗
MathVista [31]	mathematics	multiple choice questions, integer free form questions	✓	$6,141$	✗	✓
MMMU [32]	diverse	multiple choice questions, open answer questions	✓	$11,500$	✗	✓
HoC+Karel [24]	visual programming	code synthesis, tracing, grid synthesis	✗	30	✗	✗

Figure 2: Comparison of our work with related benchmarks. The first column shows the name of the benchmark and the work it was introduced in. "Domain" specifies the domain for which the benchmark was designed, and "Evaluation tasks" outlines the tasks involved. "Multimodal" indicates if the benchmark includes both visual and textual data. "Benchmark evaluation size" shows the number of samples in each benchmark. "Trained model" notes whether any model is trained in the work, and "Human comparison" indicates if model performance was compared to that of humans.

timodal elements and human comparisons, as shown in Figure 2. A more recent benchmark, MMCode [30], includes visual information in traditional coding tasks to assess multimodal model capabilities. However, this benchmark does not include comparisons between model and human performance, nor does it explore potential improvements in model performance through fine-tuning or other methods. Besides program generation, other benchmarks handle code completion, translation, summarization, debugging or explanation generation [34, 35, 36, 37], thus analyzing numerous programming-related tasks. However, these works typically focus on generating code or explanations and do not evaluate the core computational thinking and problem-solving skills of models. In contrast, our paper seeks to provide a deeper understanding of these capabilities. Additionally, we train models on our dataset and compare their performance to human counterparts, addressing gaps in previous studies.

Benchmarks assessing reasoning capabilities. For general reasoning, benchmarks like MathVista [31] and MMMU [32] assess models on tasks involving multiple-choice and free-form questions, with a focus on multimodal data. These, along with other benchmarks, evaluate reasoning in fields such as mathematics and the natural sciences [14, 20, 32, 31], planning [22, 23, 38], and causal reasoning [39, 40]. Our benchmark goes beyond these by including a variety of tasks that assess computational thinking through visual programming. This relatively underexplored area can offer intriguing insights into the reasoning capabilities of generative models. Previous efforts [24] address visual tasks without the multimodal component and do not include human comparisons. In contrast, our benchmark integrates multimodal tasks combining both programming and visual reasoning, while also comparing models directly against human performance, offering a more comprehensive evaluation of reasoning abilities in generative models.

3 Computational Thinking Tests in Elementary Visual Programming

This section first provides a background on visual programming, and then introduces the sources we use for curating our benchmark and as the basis for our synthetic dataset generation methodology.

3.1 Preliminaries and Definitions for Elementary Visual Programming

The space of grids. A visual grid, denoted as $\mathtt{G}$, includes an avatar with an initial position (row, column) and orientation (north, east, south, west), alongside free cells, wall cells, and a goal or multiple markers. The avatar is required to reach the goal or interact with the markers. The resulting grid space includes visual grids based on HoC by Code.org [26, 27], such as the grids in Figures 1b and 1c, and Karel [41], such as the grid in Figure 1d.

The space of codes. The set of valid codes $\mathbb{C}$ is defined via a domain-specific language (DSL). We adopt DSLs previously used in literature for visual programming [42, 43, 44]. A code $\mathtt{C} \in \mathbb{C}$ is characterized by its size $\mathtt{C}_{\text{size}}$, utilized constructs $\mathtt{C}_{\text{blocks}}$, and programming concepts exercised in terms of its nesting structure $\mathtt{C}_{\text{sketch}}$. For example, the code in Figure 1c uses $\mathtt{C}_{\text{size}} = 5$

Concepts	HoC	ACE			CT-Test
		Analyzing ACE[01-07]	*Evaluating* ACE[08-14]	*Creating* ACE[15-21]	
Basic actions	H01–H05	Q01	Q08	Q15	P01–P04
`Repeat{}`	H06–H09	Q02, Q05	Q12	Q16	P05, P06
`RepeatUntil{}`	H10–H13	Q06	Q09	Q17, Q18	P09, P10
`RepeatUntil{If}`	H14–H17	Q07	Q10	Q19	P13, P14
`RepeatUntil{IfElse}`	H18, H19	Q04	Q11, Q14	Q20, Q21	P17, P18
`RepeatUntil{IfElse{IfElse}}`	H20				P19, P20
`Repeat{Repeat}`			Q13		P08, P27, P28
`Repeat{If}`		Q03			
`RepeatUntil{If; If}`					P16
`RepeatUntil{Repeat}`					P11
`RepeatUntil{If{Repeat}}`					P15
`While{}; Repeat{}`					P21
`While{Repeat; Repeat}`					P22
`While{Repeat; If}`					P23
`While{If{While}}`					P24

Figure 3: Programming concepts $\texttt{C}_{\text{sketch}}$ required for solving tasks in HoC [26], ACE [25], and CT-Test [28, 29]. HoC comprises code-writing tasks. ACE and CT-Test comprise multi-choice question tasks. ACE is further split according to the higher cognitive levels of Bloom's taxonomy [45, 46].

blocks, with constructs $\texttt{C}_{\text{blocks}} = \{\texttt{move}, \texttt{turnRight}, \textsc{RepeatUntil}, \textsc{IfElse}\}$, and is structured as $\texttt{C}_{\text{sketch}} = \textsc{RepeatUntil}\{\textsc{IfElse}\}$. Executing a code on a grid generates a sequence of avatar locations, referred to as trace, along with a sequence of basic actions executed i.e., constructs from $\{\texttt{move}, \texttt{turnLeft}, \texttt{turnRight}\}$. A code is considered to solve a grid if it successfully navigates the avatar to the goal or interacts correctly with the markers (e.g., collects them when intended).

Solution synthesis tasks. A solution synthesis task is defined by the following elements: a grid $\texttt{G}$, an allowed set of constructs called $\texttt{Store}$, and a maximum code size $\texttt{maxSize}$. The objective is to write a solution code $\texttt{C}$ that successfully solves $\texttt{G}$ while respecting $\texttt{C}_{\text{blocks}} \subseteq \texttt{Store}$ and $\texttt{C}_{\text{size}} \leq \texttt{maxSize}$. Figure 1b exemplifies a solution synthesis task, where a solution code $\texttt{C}$ should solve $\texttt{G}$, have $\texttt{C}_{\text{blocks}} \subseteq \{\texttt{move}, \texttt{turnRight}, \texttt{turnLeft}, \textsc{RepeatUntil}, \textsc{IfElse}\}$ and $\texttt{C}_{\text{size}} \leq 5$.

Multi-choice question tasks. A multi-choice question (MCQ) task is defined by the following elements: a text description, a set of grids or codes, one correct option, and three distractor options. The objective is to choose the correct option out of four options. For example, Figures 1c and 1d have a text description inside the gray area, a given grid and a given code, and four options. The correct option for Figure 1c is Option A, and the correct option for Figures 1d is Option A as well.

3.2 Three Different Computational Thinking Tests

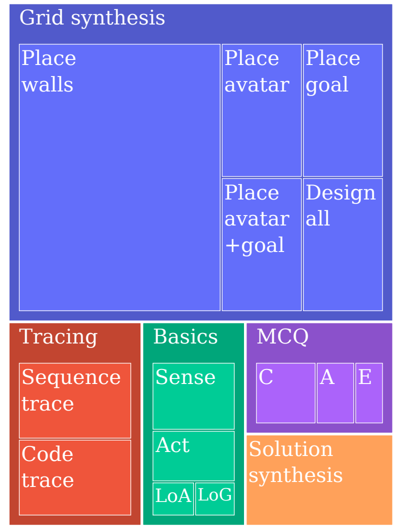
Our benchmark is based on two pedagogically validated computational thinking tests comprising multiple-choice question tasks [25, 28, 29] and a popular curriculum comprising code-writing tasks [26]. Henceforth, we refer to these three as tests, and we will use them throughout the paper to measure the performance of generative models. These tests have been carefully designed by educational experts to assess or teach a diverse set of skills in elementary visual programming within the duration of a typical one-hour school lesson. They are representative of computational thinking in this domain, providing valuable data on student performance, which we can use as a basis to benchmark the performance of generative models. Figure 3 gives an overview of the programming concepts $\texttt{C}_{\text{sketch}}$ utilized by tasks in each test. Next, we provide details for each test.

HoC. This test includes 20 code-writing tasks from Code.org's popular block-based visual programming lesson *Hour of Code:Maze Challenge* [26]. The tasks mainly cover concepts such as basic actions, `Repeat` and `RepeatUntil` loops, as well as `If` and `IfElse` branching (see Figure 3). This curriculum has been used by millions of learners to get acquainted with programming and to assess students' programming background [25, 26, 27].

ACE. This test includes 21 multi-choice question tasks from the ACE test, which was designed to evaluate higher cognitive levels of Bloom's taxonomy: *Analyzing*, *Evaluating*, and *Creating* [25, 45, 46]. These tasks were selected from a larger pool to ensure balanced coverage of cognitive levels and programming concepts, being validated using standardized pedagogical tools. Figure 3 categorizes each task by cognitive level and programming concepts covered.

Synthetic data	Original Size	Selected Size	Percentage
Solution synthesis	7,576	7,576	6.77%
Multi-choice questions (MCQ)	9,223	9,223	8.25%
Analyzing MCQ (A)	2,779	2,779	2.49%
Evaluating MCQ (E)	2,072	2,072	1.85%
Creating MCQ (C)	4,372	4,372	3.91%
Fine-grained: Basics	586,341	11,726	10.48%
Locate avatar (LoA)	65,149	1,336	1.19%
Locate goal (LoG)	65,149	1,273	1.14%
Apply action (Act)	195,447	3,930	3.51%
Sense condition (Sense)	260,596	5,187	4.64%
Fine-grained: Tracing	15,152	15,152	13.54%
Sequence trace	7,576	7,576	6.77%
Code trace	7,576	7,576	6.77%
Fine-grained: Grid synthesis	68,184	68,184	60.95%
Place avatar	7,576	7,576	6.77%
Place goal	7,576	7,576	6.77%
Place avatar+goal	7,576	7,576	6.77%
Place walls	37,880	37,880	33.87%
Design all	7,576	7,576	6.77%
Total	**586,341**	**111,861**	**100%**

(a) Distributions for synthetically generated data.



(b) Treemap of selected data distribution.

Figure 4: Our synthetically generated training dataset. Subsampling is done only in the case of basics.

CT-Test. This test is based on CT-Test, one of the earliest and most popular computational thinking tests in block-based visual programming [28, 29]. Out of 28 tasks in the original set, we curate 24 tasks compatible with our definitions and representation. Figure 3 shows the programming concepts covered, with the original task numbering: if its number is not in the table, we have not included the task.

4 Synthetic Data Generation to Fine-tune Models for Computational Thinking

In this section, we introduce our novel data generation methodology for computational thinking and problem-solving skills. With the resulting data (see Figure 4), we aim to fine-tune models to increase performance on all three tests. Next, we present our three main methods for generating data.

4.1 Synthetic Data for Solution Synthesis

We first generate data for solution synthesis tasks. Our process will start with generating a dataset of pairs (`C`,{`G`}), where `C` is a code and {`G`} is a set of grids solved by `C`. To obtain (`C`,{`G`}), we employ existing techniques for synthesizing code `C` and grid `G` [42, 43, 44]. We then split the sets into pairs of one solution code and one grid (`C`,`G`). We extract `Store` and `maxSize` from code `C`. Then, we treat (`G`, `Store`, `maxSize`) as input for the task, and keep `C` as target output.

To enhance the fine-tuning process, we aim to train the model to first produce a trace and sequence of basic actions that the avatar should execute to reach the goal, and then to produce the solution code. We refer to the trace and sequence of basic actions as an explanation for the produced answer. This method is grounded in previous research, which has shown that smaller models benefit from richer signals while being fine-tuned, leading to more careful reasoning at inference [47, 48]. However, unlike literature, we cannot rely on more powerful models like GPT-4 to produce these explanations, as state-of-the-art models struggle with computational thinking (see Figure 1a). So, we rely on symbolic methods such as executing codes on grids via an emulator to produce correct traces and basic action sequences as explanations.

4.2 Synthetic Data for Multi-choice Programming Questions

We now focus on generating MCQ tasks similar to those in ACE and CT-Test [25, 28, 29]. We generate MCQs starting from the same (`C`,{`G`}) used for generating solution synthesis tasks, using a template-based approach, with manually written text descriptions for each task type. Next, we present our task types covering all the higher cognitive levels in Bloom's taxonomy – *Analyzing*, *Evaluating*, *Creating* [25, 45, 46]. We also augment MCQs with explanations similar to the ones we use for solution synthesis tasks.

***Analyzing*.** First, we describe the process of generating *Analyzing* cognitive level tasks. For this level, we generate three task types: tasks that require selecting a solution code for a given grid, tasks that require indicating which given grids are solved by a given code, and tasks that require reasoning about the trace of a given code on a given grid and selecting the cells visited by the avatar. To offer an overview of our method, we explain the generation process of one task type, namely reasoning about the trace. We start from a pair (`C`,`G`) and the text description specific to this type of task. Then, we generate the correct option by executing `C` on `G` and selecting random cells visited by the avatar. We generate distractor options by randomly picking free cells that were not visited by the avatar. Note that this task is correct by construction, unlike some more complex task types below that need validation. Finally, we have the task containing text description, `C`, `G`, the correct option, and three distractor options.

***Evaluating*.** Second, we describe the process of generating *Evaluating* cognitive level tasks. For this level, we generate four task types: tasks that require identifying bugs, tasks that require repairing bugs, tasks that require evaluating code equivalence with no given grid, and tasks that require evaluating code equivalence given a grid. We explain generation process for the task type that requires repairing bugs. We start from pair (`C`,`G`) and corresponding text description. We generate a mutation and apply it to code `C` to obtain $\texttt{C}_{\text{mut}}$ [44, 49]. The correct option is obtained as the reverse mutation that would transform $\texttt{C}_{\text{mut}}$ back to `C`. Distractor options are obtained by generating three other mutations. We validate the task by applying reverse mutation on $\texttt{C}_{\text{mut}}$ and checking whether resulting code solves `G`. We also apply distractor mutations on $\texttt{C}_{\text{mut}}$ and make sure that resulting codes do not solve `G`. Finally, we have the task containing text description, $\texttt{C}_{\text{mut}}$, `G`, the correct option, and three distractor options.

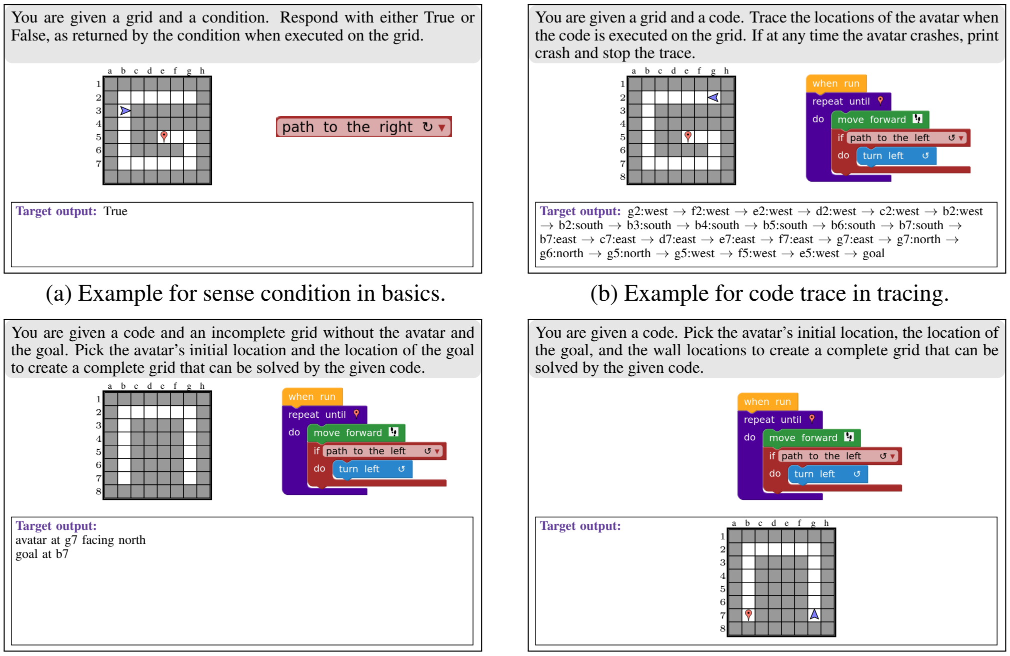
***Creating*.** Third, we describe the process of generating *Creating* cognitive level tasks. For this level, we generate six types of tasks that require reasoning about modifying an incomplete grid such that the given code solves the modified grid. We generate: tasks that require placing the avatar, tasks that require reasoning about the number of possible initial avatar locations, tasks that require placing the goal, tasks that require counting possible goal positions, tasks that require placing walls, and tasks that require counting the minimum number of walls needed. For example, a task similar to Figure 1c can be synthesized by starting from a pair (`C`,`G`) and the text description. We set the correct option by randomly picking the number of walls to remove from $\{1, 2, 3\}$, in this case 1. We remove one wall from `G`, obtaining $\texttt{G}_{\text{mut}}$. We generate three distractor options by applying arithmetic operations to the correct option. To validate the task correctness, we check whether `C` solves any grid obtained via adding all possible combinations of walls less than the correct option. For this specific example, as the correct option for the example is 1, we just need to check if the grid will be solved with no added walls. Finally, we have the task containing text description, `C`, $\texttt{G}_{\text{mut}}$, the correct option, and three distractor options.

4.3 Synthetic Data at Fine-grained Skills

We now introduce new kinds of tasks, aimed at improving fine-grained skills fundamental to making the models better understand the domains of our computational thinking tests. The main intuition for using various fine-grained skills is due to inter-task transfer observed during instruction-tuning [50, 51, 52], which can enhance performance on solution synthesis tasks and MCQ tasks. We now give details about generating three kinds of tasks for improving fine-grained skills.

Basics. We describe the process of generating tasks aimed at familiarizing models with the fundamental aspects of visual programming. We generate four types of basic tasks: tasks that require locating the avatar in a given grid, tasks that require locating the goal in a given grid, tasks that require specifying the new location of the avatar after executing a given basic action on a given grid, and tasks which require specifying the outcome of applying a given condition to a given grid. For example, we generate the input for the task in Figure 5a starting from a grid `G`, a randomly selected condition present in the DSL, and a fixed text description. The target output is obtained as the outcome of applying the condition on `G`, in this case True as the avatar has a free cell to its right. As the number of obtainable basic tasks is very large, we subsample to 2% of the original size (see Figure 4a). We have empirically chosen this percentage, analyzing the performance on a validation segment corresponding to basic tasks.

Tracing. Next, we describe the process of generating tasks aimed at enhancing the model's understanding of the interaction between the basic actions, conditions, and grids, crucial for answering the MCQs corresponding to the *Analyzing* cognitive level in tests. We generate two types of tasks: tasks requiring to produce the trace obtained by applying a sequence of basic actions to a given grid, and



(a) Example for sense condition in basics.

(b) Example for code trace in tracing.

(c) Example for place avatar+goal in grid synthesis.

(d) Example for design all in grid synthesis.

Figure 5: Illustrative examples for synthetically generated tasks with target outputs for the fine-grained skills. The tasks have been adapted for readability. For example, in **(d)**, we illustrate the target output visually, but the actual target output is in textual form.

tasks requiring to produce the trace of a given code on a given grid. For example, in Figure 5b, we use a pair (C,G) and a fixed text description as input and treat the trace of C on G as the target output.

Grid synthesis. Finally, we describe the generation of tasks aimed at boosting the model's understanding of the role of each grid element and how it can influence the execution of a code, crucial for answering the MCQs corresponding to the *Creating* cognitive level in tests. We generate five types of tasks: tasks that require placing the avatar, tasks that require placing the goal, tasks that require placing both the avatar and the goal, tasks that require placing walls, and tasks that require designing a full grid. For example, we generate the task in Figure 5c starting from a pair (C,G) and a fixed text description, removing the avatar and the goal from G, and giving the incomplete grid and C as input. The target output is the avatar and goal positions from G. Similarly, in Figure 5d, we keep only C as input, and require as output a natural language description of G. We also include tracing information and sequences of basic actions as explanations during fine-tuning in a style similar to solution synthesis and MCQ tasks.

5 Experiments

In this section, we compare performance of open-access models, OpenAI's GPT family of models, and our fine-tuned LLAMACT. We also include school students' performance based on studies in existing literature [25, 29]. We evaluate LLAMACT variants to assess the impact of training on different data segments and the use of explanations. We also provide insights into models' reasoning process.

5.1 Techniques Evaluated

We start with NAIVE technique, a baseline that generates random tokens for HoC tasks and selects most frequent answer from four options for ACE and CT-TEST tasks. Note that because of non-uniform distribution of options, NAIVE yields better results for ACE and CT-TEST than randomly choosing an option. Next, we present techniques based on generative models and performance of school students. Figure 6 shows a summary of our model-based techniques.

Open-access models. We select smaller, instruction-tuned models from the Llama family, such as the 7B parameter version of CodeLlama [53] and the 8B parameter version of Llama3 [11], alongside the 7B parameter version of Llava [54]. These are referred to as CODELLAMA-7B-INSTRUCT,

Technique	Base model	Modality		Fine-tuning data	Explanation
		Visual	Text		
CODELLAMA-7B-INSTRUCT	CodeLlama-7B [53]	✗	✓	n/a	n/a
LLAVA1.5-7B	LLaVA-v1.5-7B [54]	✓	✓	n/a	n/a
LLAMA3-8B-INSTRUCT	Llama3-8B [11]	✗	✓	n/a	n/a
GPT3.5, $\text{GPT4}_{\text{text}}$, $\text{GPT4O}_{\text{text}}$	GPT-3.5 [56], 4 [17], 4o [10]	✗	✓	n/a	n/a
GPT4_{vis}, $\text{GPT4O}_{\text{vis}}$	GPT-4V [57], 4o [10]	✓	✗	n/a	n/a
$\text{GPT4}_{\text{vis+text}}$, $\text{GPT4O}_{\text{vis+text}}$	GPT-4V [57], 4o [10]	✓	✓	n/a	n/a
LLAMACT:HOC	Llama3-8B [11]	✗	✓	Solution syn	None
LLAMACT:HOC+MCQ	Llama3-8B [11]	✗	✓	Solution syn+MCQ	None
$\text{LLAMACT:HOC}_{\text{exp}}$	Llama3-8B [11]	✗	✓	Solution syn	Train
$\text{LLAMACT:HOC+MCQ}_{\text{exp}}$	Llama3-8B [11]	✗	✓	Solution syn+MCQ	Train
$\text{LLAMACT:HOC+MCQ+AUG}_{\text{exp}}$	Llama3-8B [11]	✗	✓	Full data	Train
$\text{LLAMACT:HOC+MCQ+AUG}_{\text{exp*}}$	Llama3-8B [11]	✗	✓	Full data	Train+infer

Figure 6: Table summarizing techniques based on generative models, showing the base model and whether the input grid is represented visually or in text (modality). For fine-tuned models (e.g., LLAMACT), the table specifies the data segment used for training and whether models were trained with no explanations, to generate explanations, or to receive explanations during inference.

LLAMA3-8B-INSTRUCT, and LLAVA1.5-7B, respectively. For LLAVA1.5-7B, we incorporate both natural language and visual representations of grids to utilize its vision capabilities. All techniques are prompted to use chain-of-thought (CoT) [55].

GPT family. This group includes techniques based on GPT-3.5 [56] and GPT-4 [10, 17]. We start with GPT3.5 technique which processes tasks, including grids, only in natural language, as it has no vision capabilities. Similarly, $\text{GPT4}_{\text{text}}$ is solely based on natural language. Next, for the GPT4_{vis}, we input the grids solely as visual representation, while the rest of the task is represented through natural language. $\text{GPT4}_{\text{vis+text}}$ technique combines textual and visual representations for grids, with the rest of the task in natural language. We also include similar techniques based on the newer GPT-4o [10], namely $\text{GPT4O}_{\text{text}}$, $\text{GPT4O}_{\text{vis}}$, and $\text{GPT4O}_{\text{vis+text}}$. All techniques are prompted to use CoT.[2]

Fine-tuned models. We fine-tune the instruction-tuned 8B parameter version of the Llama3 model using LoRA [58] and obtain the LLAMACT family. LLAMACT:HOC and $\text{LLAMACT:HOC}_{\text{exp}}$ are fine-tuned using only the generated solution synthesis tasks, LLAMACT:HOC+MCQ and $\text{LLAMACT:HOC+MCQ}_{\text{exp}}$ are trained using both generated solution synthesis tasks and generated MCQ tasks, and $\text{LLAMACT:HOC+MCQ+AUG}_{\text{exp}}$ is trained on the full synthetic dataset. $\text{LLAMACT:HOC}_{\text{exp}}$, $\text{LLAMACT:HOC+MCQ}_{\text{exp}}$, and $\text{LLAMACT:HOC+MCQ+AUG}_{\text{exp}}$ are trained on target outputs enriched with explanations. Additionally, $\text{LLAMACT:HOC+MCQ+AUG}_{\text{exp*}}$ simulates an ideal scenario where the correct reasoning process is known at inference time.

Human students. We benchmark these models against the performance of students observed in literature, reporting results for one group of students for HOC and ACE [25], and for a different group of students for CT-TEST [29]. GRADEALL comprises the average performance of students across grades 3-7 for HOC and ACE, and the average performance of students across grades 5-10 for CT-TEST. $\text{GRADE7}_{\text{top25}}$ represents the top 25% of grade 7 students for HOC and ACE, and the top 25% of grade 7-8 students for CT-TEST, showing the performance of the best students.

5.2 Performance on Computational Thinking Tests

We evaluate techniques on HOC, ACE, and CT-TEST, introduced in Section 3.2. Figure 7 shows results, with accuracy computed as percentage of correctly answered tasks in one trial out of total tasks per test. We set temperature to 0 and assess over three seeds, reporting average results as mean (stderr).

Combining language and vision enhances performance. Providing input in both text and visual modality leads to better results for $\text{GPT4O}_{\text{vis+text}}$ when compared with $\text{GPT4O}_{\text{vis}}$ and $\text{GPT4O}_{\text{text}}$. Similar results hold for $\text{GPT4}_{\text{vis+text}}$ when compared with GPT4_{vis} and $\text{GPT4}_{\text{text}}$.

Symbolic information-based explanations improve outcomes. Template-based explanations derived from execution information used while training enhance reasoning at inference and boost model

[2]Few-shot prompting did not improve results. All results are based on zero-shot CoT prompting.

Technique	HoC	ACE	CT-Test	Overall
Naive	0.0	33.0	33.0	22.0
CodeLlama-7B-Instruct	0.0 (0.0)	14.3 (0.0)	29.2 (0.0)	14.3 (0.0)
Llava1.5-7B	0.0 (0.0)	28.6 (0.0)	20.8 (0.0)	16.7 (0.0)
Llama3-8B-Instruct	0.0 (0.0)	34.9 (2.0)	34.7 (5.0)	22.9 (1.3)
GPT3.5	25.0 (0.0)	31.7 (4.0)	36.1 (5.0)	31.1 (0.5)
$GPT4_{vis}$	18.3 (2.0)	31.7 (8.0)	44.4 (3.0)	31.6 (1.3)
$GPT4_{text}$	21.7 (2.0)	52.4 (6.0)	56.9 (5.0)	43.7 (2.9)
$GPT4_{vis+text}$	28.3 (2.0)	57.1 (3.0)	58.3 (5.0)	48.0 (0.4)
$GPT4O_{vis}$	20.0 (0.0)	38.1 (3.0)	52.8 (3.0)	36.9 (1.6)
$GPT4O_{text}$	30.0 (0.0)	61.9 (3.0)	59.7 (3.0)	50.7 (1.7)
$GPT4O_{vis+text}$	30.0 (0.0)	61.9 (0.0)	66.7 (0.0)	53.0 (0.0)
LlamaCT:HoC	10.0 (0.0)	30.5 (2.0)	25.0 (0.0)	21.9 (0.7)
LlamaCT:HoC+MCQ	11.7 (4.0)	44.4 (5.0)	33.3 (5.0)	29.8 (1.2)
LlamaCT:HoC_{exp}	55.0 (4.0)	27.6 (3.0)	23.1 (2.0)	35.3 (0.9)
LlamaCT:HoC+MCQ_{exp}	40.0 (9.0)	43.5 (3.0)	36.1 (2.0)	40.0 (3.6)
LlamaCT:HoC+MCQ+Aug_{exp}	50.0 (4.0)	57.8 (1.0)	51.4 (3.0)	53.0 (1.7)
LlamaCT:HoC+MCQ+Aug_{exp*}	76.7 (7.0)	74.6 (4.0)	65.3 (0.0)	72.2 (1.3)
GradeAll	74.1	50.9	58.5	61.2
$Grade7_{top25}$	99.8	84.0	71.4	85.1

Figure 7: Results on HoC, ACE, CT-Test, and overall performance.

Technique	HoC	HoC reasoning	ACE	ACE reasoning	CT-Test	CT-Test reasoning
Llama3-8B-Instruct	0.0 (0.0)	4.2 (1.0)	34.9 (2.0)	4.8 (0.0)	34.7 (5.0)	0.0 (0.0)
$GPT4O_{vis+text}$	30.0 (0.0)	35.0 (0.0)	61.9 (0.0)	28.6 (0.0)	66.7 (0.0)	39.6 (0.0)
LlamaCT:HoC+MCQ+Aug_{exp}	50.0 (4.0)	73.3 (8.0)	57.8 (1.0)	32.5 (3.0)	51.4 (3.0)	23.6 (3.0)

Figure 8: Comparison of accuracy in correctly answered tasks and reasoning correctness across domains for representative models in HoC, ACE, and CT-Test tasks, reported as mean (stderr). Reasoning correctness results are based on manual annotations done by two independent annotators.

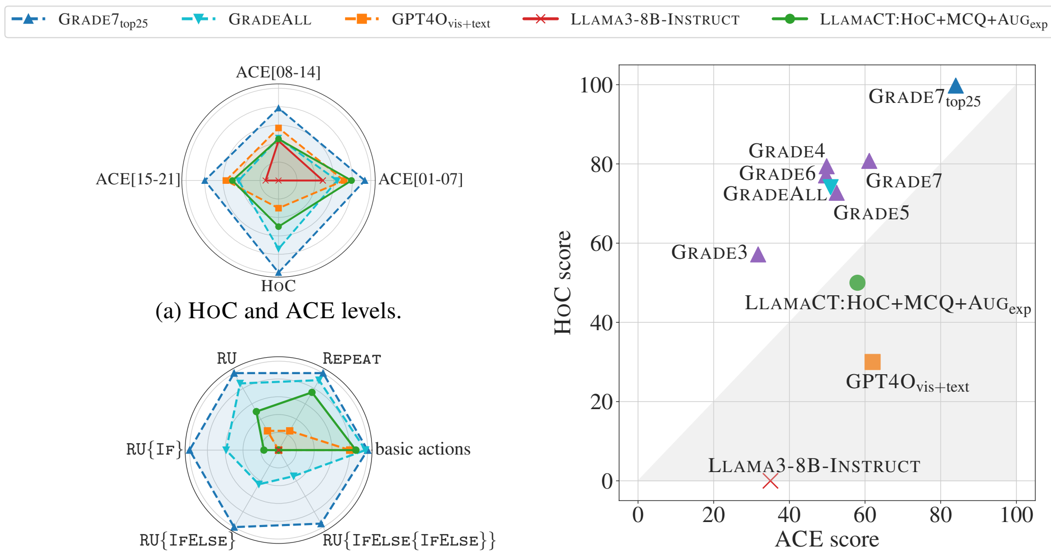
performance. Specifically, LlamaCT:HoC_{exp} and LlamaCT:HoC+MCQ_{exp}, which are trained with explanations, outperform their counterparts LlamaCT:HoC and LlamaCT:HoC+MCQ trained only for generating an answer with no explanation.

Fine-grained skills make LlamaCT comparable to GPT-4. We notice an increase of at least 10% in performance on HoC, ACE, and CT-Test for the model fine-tuned with fine-grained skills data. Fine-tuning with explanations and across the full dataset allows LlamaCT:HoC+MCQ+Aug_{exp} to achieve overall results comparable to those of $GPT4O_{vis+text}$. This shows that a better understanding of the visual domain is key to better performance on all three tests.

Reasoning for MCQ tasks is harder than for solution synthesis tasks. We analyze the reasoning capabilities of three selected models, Llama3-8B-Instruct, $GPT4O_{vis+text}$, and LlamaCT:HoC+MCQ+Aug_{exp}, through manual annotations of their reasoning process. A model's reasoning is considered correct if it respects grid constraints, correctly maps codes to traces and sequences, and avoids introducing unnecessary details. We follow a strict binary metric, where the reasoning process is marked as correct (i.e., 1) if the entire reasoning process is correct and incorrect (i.e., 0) otherwise. The reasoning processes were reviewed by two independent annotators[3]. Figure 8 compares accuracy in correctly answered tasks with reasoning correctness averaged across annotators and aggregated over three seeds, for HoC, ACE, and CT-Test tasks. Llama3-8B-Instruct often tries guessing answers without providing any reasoning, while $GPT4O_{vis+text}$ struggles with grid layouts, sometimes missing walls. LlamaCT:HoC+MCQ+Aug_{exp} traces tasks well in HoC but faces challenges with converting sequences to minimal codes and tracing in ACE and CT-Test.

Symbolic information at inference leads to human-level performance. Including explanations with a correct reasoning process in the input prompts increases performance, bringing it closer to that of school students. However, LlamaCT:HoC+MCQ+Aug_{exp*} simulates an ideal scenario, as correct explanations are usually not available at inference as input.

[3]The annotators obtained a Cohen's kappa score of 0.84, indicating high agreement [59].



(a) HOC and ACE levels.

(b) HOC concepts.

(c) HOC and ACE.

Figure 9: Comparison between the performance of the best techniques and school students (grade 3-7) on a scale of 0 to 100. For better visualization and comparison, we present results only for HOC and ACE. **(a)** shows that state-of-the-art and fine-tuned models have a similar performance to an average grade 3-7 student on the ACE, but lag behind for HOC. **(b)** shows that fine-tuning can help models' problem-solving skills get closer to an average grade 3-7 student for simpler concepts. (RU stands for REPEATUNTIL). **(c)** shows how every grade is dominating models for HOC. It also shows that state-of-the-art and fine-tuned models are close to the average grade 7 students' performance on ACE. However, the performance of the best 25% grade 7 students is still far from reach for generative models.

Human students are better at solution synthesis. Figure 9a showcases that state-of-the-art and fine-tuned models have slightly better performance than the average grade 3-7 student across three analyzed levels of Bloom's taxonomy, and that state-of-the-art models struggle with solution synthesis. Figure 9b shows a deeper analysis of performance on HOC, breaking down the performance per concept. It shows that by fine-tuning, a model's understanding of programming concepts grows similarly to that of an average student. Finally, Figure 9c compares models' performance on HOC and ACE tests with that of students from various grades. It reveals that models have not yet reached the problem-solving capabilities of grade 3 students on HOC tasks. Besides spatial reasoning, adhering to constraints such as the required size and constructs is another reason for this weak performance. Interestingly, models can match the performance of grade 7 students on ACE tests, where answer options are available.

6 Concluding Discussion

In this paper, we introduced a new benchmark for assessing generative models on computational thinking tests grounded in elementary visual programming. We made a detailed analysis of the performance of open-access models such as Llama3 and the GPT family of models, comparing it to that of school students. To boost performance of Llama3-8B, we fine-tuned it using our novel synthetic generation methodology based on symbolic information. The best fine-tuned model has a performance similar to state-of-the-art models, even though it is much smaller and does not use vision capabilities.

While our analysis gives a deep insight into the computational thinking and problem-solving capabilities of generative models, there are some limitations of our current work and directions to tackle them in future work. First, we assess multi-modal models on our benchmark but do not fine-tune them to improve performance. An interesting direction for future work is fine-tuning multi-modal models for solving computational thinking and problem-solving tasks. Second, one of our techniques naively uses correct explanations provided at inference time to help it reach an answer. An interesting direction for future work is developing techniques where generative models interact with symbolic tools to obtain this kind of information at inference time, possibly via multiple rounds of interaction.

Acknowledgements

Funded/Co-funded by the European Union (ERC, TOPS, 101039090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Karan Singhal et al. Large Language Models Encode Clinical Knowledge. *CoRR*, abs/2212.13138, 2022.

[2] Chunyuan Li, Cliff Wong, Sheng Zhang, Naoto Usuyama, Haotian Liu, Jianwei Yang, Tristan Naumann, Hoifung Poon, and Jianfeng Gao. LLaVA-Med: Training a Large Language-and-Vision Assistant for Biomedicine in One Day. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2023.

[3] Renqian Luo, Liai Sun, Yingce Xia, Tao Qin, Sheng Zhang, Hoifung Poon, and Tie-Yan Liu. BioGPT: Generative Pre-trained Transformer for Biomedical Text Generation and Mining. *Briefings in Bioinformatics*, 2022.

[4] Anqi Wang, Zhizhuo Yin, Yulu Hu, Yuanyuan Mao, and Pan Hui. Exploring the Potential of Large Language Models in Artistic Creation: Collaboration and Reflection on Creative Programming. *CoRR*, abs/2402.09750, 2024.

[5] Tyler Angert, Miroslav Ivan Suzara, Jenny Han, Christopher Lawrence Pondoc, and Hariharan Subramonyam. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the Annual Symposium on User Interface Software and Technology (UIST)*, 2023.

[6] Khan Academy. Khanmigo. `https://www.khanmigo.ai/`, 2023.

[7] Manh Hung Nguyen, Sebastian Tschiatschek, and Adish Singla. Large Language Models for In-Context Student Modeling: Synthesizing Student's Behavior in Visual Programming from One-Shot Observation. In *Proceedings of the International Conference on Educational Data Mining (EDM)*, 2023.

[8] Paul Denny, Sumit Gulwani, Neil T. Heffernan, Tanja Käser, Steven Moore, Anna N. Rafferty, and Adish Singla. Generative AI for Education (GAIED): Advances, Opportunities, and Challenges. *CoRR*, abs/2402.01580, 2024.

[9] Tung Phung, Victor-Alexandru Padurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation. In *Proceedings of the Learning Analytics and Knowledge Conference (LAK)*, 2024.

[10] OpenAI. Hello GPT-4o. `https://openai.com/index/hello-gpt-4o/`, 2024.

[11] Meta. Llama 3. `https://llama.meta.com/llama3/`, 2024.

[12] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring Massive Multitask Language Understanding. In *International Conference on Learning Representations (ICLR)*, 2021.

[13] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a Machine Really Finish Your Sentence? In *Proceedings of the Conference of the Association for Computational Linguistics (ACL) - Volume 1*, 2019.

[14] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *CoRR*, abs/1803.05457, 2018.

[15] Mark Chen et al. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107-03374, 2021.

[16] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program Synthesis with Large Language Models. *CoRR*, abs/2108.07732, 2021.

[17] OpenAI. GPT-4 Technical Report. 2023.

[18] Jaromír Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. In *Proceedings of the Conference on International Computing Education Research (ICER) - Volume 1*, 2023.

[19] Sébastien Bubeck et al. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. *CoRR*, abs/2303.12712, 2023.

[20] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving With the MATH Dataset. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2021.

[21] Boshi Wang, Xiang Yue, and Huan Sun. Can ChatGPT Defend the Truth? Automatic Dialectical Evaluation Elicits LLMs' Deficiencies in Reasoning. *CoRR*, abs/2305.13160, 2023.

[22] Karthik Valmeekam, Matthew Marquez, Alberto Olmo Hernandez, Sarath Sreedharan, and Subbarao Kambhampati. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2023.

[23] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the Planning Abilities of Large Language Models - A Critical Investigation. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[24] Adish Singla. Evaluating ChatGPT and GPT-4 for Visual Programming. In *Proceedings of the Conference on International Computing Education Research (ICER) - Volume 2*, 2023.

[25] Ahana Ghosh, Liina Malva, and Adish Singla. Analyzing-Evaluating-Creating: Assessing Computational Thinking and Problem Solving in Visual Programming Domains. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, 2024.

[26] Code.org. Hour of Code: Classic Maze Challenge. `https://studio.code.org/s/hourofcode`, 2013.

[27] Code.org. Code.org: Learn Computer Science. `https://code.org/`, 2013.

[28] Marcos Román González. Computational Thinking Test: Design Guidelines and Content Validation. In *Proceedings of the International Conference on Education and New Learning Technologies (EDULEARN)*, 2015.

[29] Marcos Román-González, Juan-Carlos Pérez-González, and Carmen Jiménez-Fernández. Which Cognitive Abilities Underlie Computational Thinking? Criterion Validity of the Computational Thinking Test. *Computers in Human Behavior*, 72, 2017.

[30] Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, and Jing Ma. MMCode: Evaluating Multi-Modal Code Large Language Models with Visually Rich Programming Problems. *CoRR*, abs/2404.09486, 2024.

[31] Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. MathVista: Evaluating Math Reasoning in Visual Contexts with GPT-4V, Bard, and Other Large Multimodal Models. *CoRR*, abs/2310.02255, 2023.

[32] Xiang Yue et al. MMMU: A Massive Multi-discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI. *CoRR*, abs/2311.16502, 2023.

[33] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence with APPS. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2021.

[34] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. OctoPack: Instruction Tuning Code Large Language Models. *CoRR*, abs/2308.07124, 2023.

[35] Tung Phung, Victor-Alexandru Padurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors. In *Proceedings of the Conference on International Computing Education Research (ICER) - Volume 2*, 2023.

[36] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Zhiyuan Liu, and Maosong Sun. DebugBench: Evaluating Debugging Capability of Large Language Models. *CoRR*, abs/2401.04621, 2024.

[37] Tianyang Liu, Canwen Xu, and Julian J. McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *CoRR*, abs/2306.03091, 2023.

[38] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2022.

[39] Zhijing Jin, Yuen Chen, Felix Leeb, Luigi Gresele, Ojasv Kamal, Zhiheng Lyu, Kevin Blin, Fernando Gonzalez Adauto, Max Kleiman-Weiner, Mrinmaya Sachan, and Bernhard Schölkopf. CLadder: A Benchmark to Assess Causal Reasoning Capabilities of Language Models. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[40] Zhijing Jin, Jiarui Liu, Zhiheng LYU, Spencer Poff, Mrinmaya Sachan, Rada Mihalcea, Mona T. Diab, and Bernhard Schölkopf. Can Large Language Models Infer Causation from Correlation? In *International Conference on Learning Representations (ICLR)*, 2024.

[41] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., 1981.

[42] Victor-Alexandru Pădurean, Georgios Tzannetos, and Adish Singla. Neural Task Synthesis for Visual Programming. *Transactions on Machine Learning Research (TMLR)*, 2024.

[43] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *International Conference on Learning Representations (ICLR)*, 2018.

[44] Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. Synthesizing Tasks for Block-based Programming. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[45] Benjamin S Bloom, Max D Engelhart, Edward J Furst, Walker H Hill, and David R Krathwohl. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook 1: Cognitive Domain*. Longman New York, 1956.

[46] David R Krathwohl. A Revision of Bloom's Taxonomy: An Overview. *Theory into Practice*, 2002.

[47] Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. Orca: Progressive Learning from Complex Explanation Traces of GPT-4. *CoRR*, abs/2306.02707, 2023.

[48] Arindam Mitra et al. Orca 2: Teaching Small Language Models How to Reason. *CoRR*, abs/2311.11045, 2023.

[49] Ahana Ghosh, Sebastian Tschiatschek, Sam Devlin, and Adish Singla. Adaptive Scaffolding in Block-Based Programming via Synthesizing New Tasks as Pop Quizzes. In *Proceeding of the International Conference on Artificial Intelligence in Education AIED*, 2022.

[50] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations (ICLR)*, 2022.

[51] Wenpeng Yin, Jia Li, and Caiming Xiong. ConTinTin: Continual Learning from Task Instructions. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) - Volume 1*, 2022.

[52] Jisoo Mok, Jaeyoung Do, Sungjin Lee, Tara Taghavi, Seunghak Yu, and Sungroh Yoon. Large-scale Lifelong Learning of In-context Instructions and How to Tackle It. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) - Volume 1*, 2023.

[53] Baptiste Rozière et al. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950, 2023.

[54] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual Instruction Tuning. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[56] OpenAI. ChatGPT. `https://openai.com/blog/chatgpt`, 2023.

[57] OpenAI. GPT-4V(ision) System Card. `https://openai.com/blog/chatgpt`, 2023.

[58] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations (ICLR)*, 2022.

[59] Jacob Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20:37 – 46, 1960.

[60] Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic Datasets for Neural Program Synthesis. In *International Conference on Learning Representations (ICLR)*, 2019.

[61] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Commun. ACM*, 64, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] See Sections 4 and 5.
   (b) Did you describe the limitations of your work? [Yes] See Section 6.
   (c) Did you discuss any potential negative societal impacts of your work? [No] We do not foresee any potential negative societal impacts of this work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See the GitHub repository.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See the supplemental material.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Section 5.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See the supplemental material.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See Section 3 and the supplemental material.
   (b) Did you mention the license of the assets? [Yes] See the supplemental material.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] See the GitHub repository.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Table of Contents

In this section, we provide a brief description of the content provided in the appendices of the paper.

- Appendix B provides a discussion of the broader impact of our work, a responsibility statement, compute resources used, and training details.
- Appendix C has a more detailed description of the generation methodology in Section 4.
- Appendix D gives more details about the origin of the curated data and more examples for each test.
- Appendix E details the synthetic dataset evaluation segment and shows results of selected techniques on this segment.
- Appendix F presents the datasheet of both the curated benchmark and the synthetically generated dataset.
- Appendix G offers an insight into the reasoning process done by models.
- Appendix H shows the templates of the prompts used for interacting with and training the models.

B Discussion

Broader impact. This paper introduces a new benchmark for assessing the generative models' performance on computational thinking and problem-solving tasks. It also includes a dataset for potentially improving newer models (e.g., via fine-tuning). We believe our proposed benchmark has the potential to bring more attention to the problems state-of-the-art generative models encounter when it comes to computational thinking and problem-solving, thus leading to an improvement in their reasoning capabilities.

Responsibility statement. The authors declare that they bear full responsibility for any violations of rights, including but not limited to copyright infringement, plagiarism, or any other legal or ethical breaches, that may arise from the content and data provided in this work.

Compute resources. All the experiments were conducted on a cluster of machines equipped with Intel Xeon Platinum 8360Y CPUs running at a frequency of 2.40GHz and 8x NVIDIA A100 80GB PCIe GPUs.

Training details. We fine-tuned Llama3 using the llama-recipes repository[4], using LoRA and FSDP with the default settings and hyperparameters, unless stated otherwise. We set LoRA r $=16$ and alpha $=32$. We pass as target modules for LoRA the following: "q_proj", 'v_proj", "k_proj", "o_proj", "gate_proj", "up_proj", "down_proj", and "lm_head". We chose these LoRA parameters to strike a balance between accuracy, inference speed, and the size of the resulting LoRA adapter. The data is naturally split into evaluation (i.e., the real-world benchmark) and training (i.e., the synthetic dataset). We further randomly split the synthetic dataset into train (90%) and validation segments (10%). We use the validation segment to check whether the loss is decreasing. As we have noticed that the decrease in loss is minimal after the second epoch, we always choose to train for 2 epochs. Training the 8B parameter version of Llama3 on the full synthetic dataset, with the resources described above takes approximately 10 hours. Training it on solution synthesis only takes approximately 1 hour, while training it on solution synthesis and multiple choice questions takes roughly 2 hours. Doing it for all the versions of LLAMACT for multiple seeds leads to approximately 78-80 hours of compute.

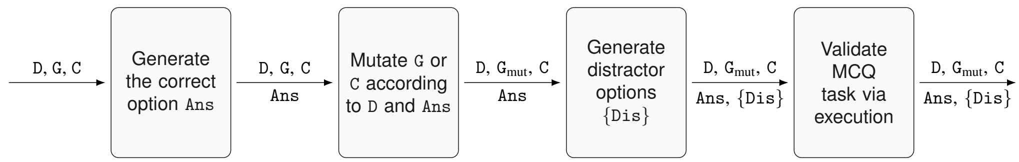
[4]Found at `https://github.com/meta-llama/llama-recipes`

C Further Details about Data Generation

In this section we give more insight into the data generation methodology described in Section 4. We first give details about the MCQ task generation process, then we continue with the generation process for the fine-grained skills tasks.

C.1 MCQ Tasks Generation

We will now describe our generation methodology exemplified with two types of tasks.



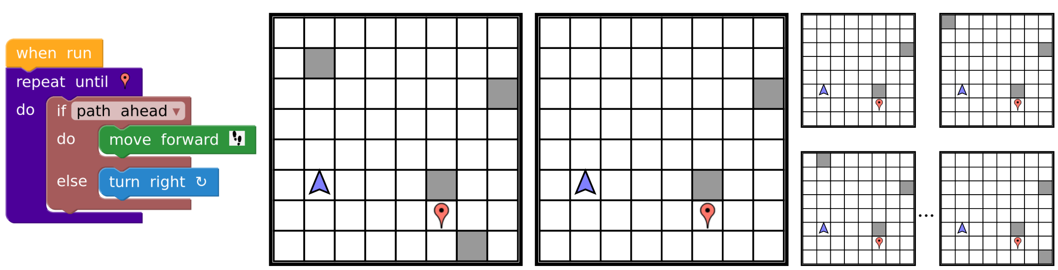
(a) Illustration of the MCQ task generation methodology for minimum wall counting.

> You are given a code and an incomplete grid. You can add additional wall cells to the grid by converting any of the free cells into wall cells. What is the smallest number of additional wall cells you must add such that the grid is solved by the code?

(b) Manually written input text description D

Correct option	2
Distractor option 1	1
Distractor option 2	3
Distractor option 3	4

(c) Answer options Ans + {Dis}

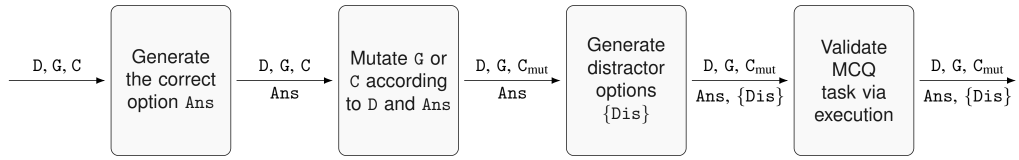


(d) Input code C (e) Input grid G (f) Mutated grid $\texttt{G}_{\text{mut}}$ (g) Combinations of walls

Figure 10: Illustration of our MCQ task generation methodology instantiated on the minimum wall counting task in *Creating* cognitive level. **(a)** gives an overview of the pipeline. **(b,d,e)** show the input components for the pipeline: text description D, grid G, and its solution code C. **(c)** shows the generated correct option Ans along with the generated distractors {Dis}, put together to obtain all four options. **(f)** shows the mutated grid $\texttt{G}_{\text{mut}}$ obtained according to the selected correct option Ans. **(g)** shows the grids obtained by combinatorially adding numbers of walls less than the correct option Ans. The MCQ task is valid if no combinatorially obtained grid can be solved by C.

Figure 10 shows an overview of our generation process for MCQ tasks. To give a better understanding, we will first exemplify the generation process for a task involving counting the minimum number of walls to add to the given grid so that the given code solves the modified grid. We start from a manually written text description for a task D, a grid G, and a code C. D decides the type of task that will be generated and what operations need to be done at each step. The first step is to generate the correct answer option. This is selected from a pool of options which are either fixed or obtained via executing C on G. In our example, we pick a number from $\{1, 2, 3\}$. Let us say that we have picked Ans $= 2$. We then proceed to the next step, namely mutating C or G according to the task type and the picked correct option Ans. In our example, we will only mutate G by removing two randomly picked walls and obtain $\texttt{G}_{\text{mut}}$. Going on to the next step, we generate distractor options based on the information we have until now. For this example, we apply arithmetic operations to Ans, thus obtaining the set of distractors {Dis}. We now have all the components of an MCQ task: the text description D, the mutated grid $\texttt{G}_{\text{mut}}$, code C, the correct option Ans, and the three distractors {Dis}.

In the final step, we validate the task via execution. In our example, we check whether C solves any grid obtained via adding all possible combinations of walls less than the correct option. We start by adding no walls and check whether C solves the grid. Then we add one wall at a time to each free cell and check if C solves the obtained grid. We discard the task in case any of the grids obtained via combinatorially adding walls is solved by C. Otherwise, the task passes validation and we keep it.



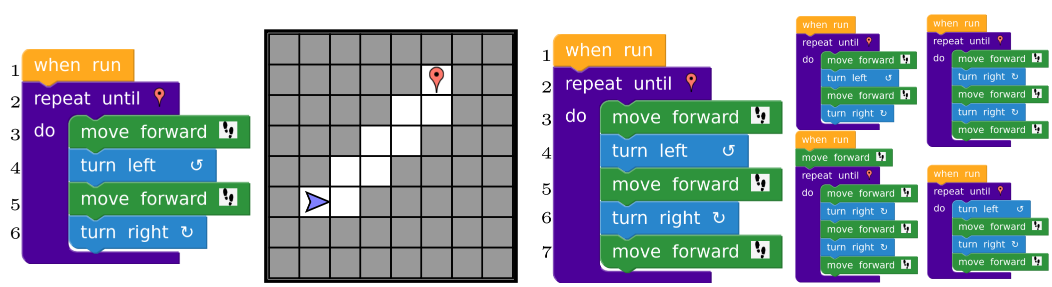
(a) Illustration of the MCQ task generation methodology for bug repair.

You are given a code and a grid. You may have to fix some errors in the code such that it solves the grid. How can you fix the code?

(b) Input text description D

Correct option	Remove line 7
Distractor option 1	Change Line 4 to turn right
Distractor option 2	Add move forward after Line 1
Distractor option 3	Remove line 3

(c) Answer options Ans + {Dis}



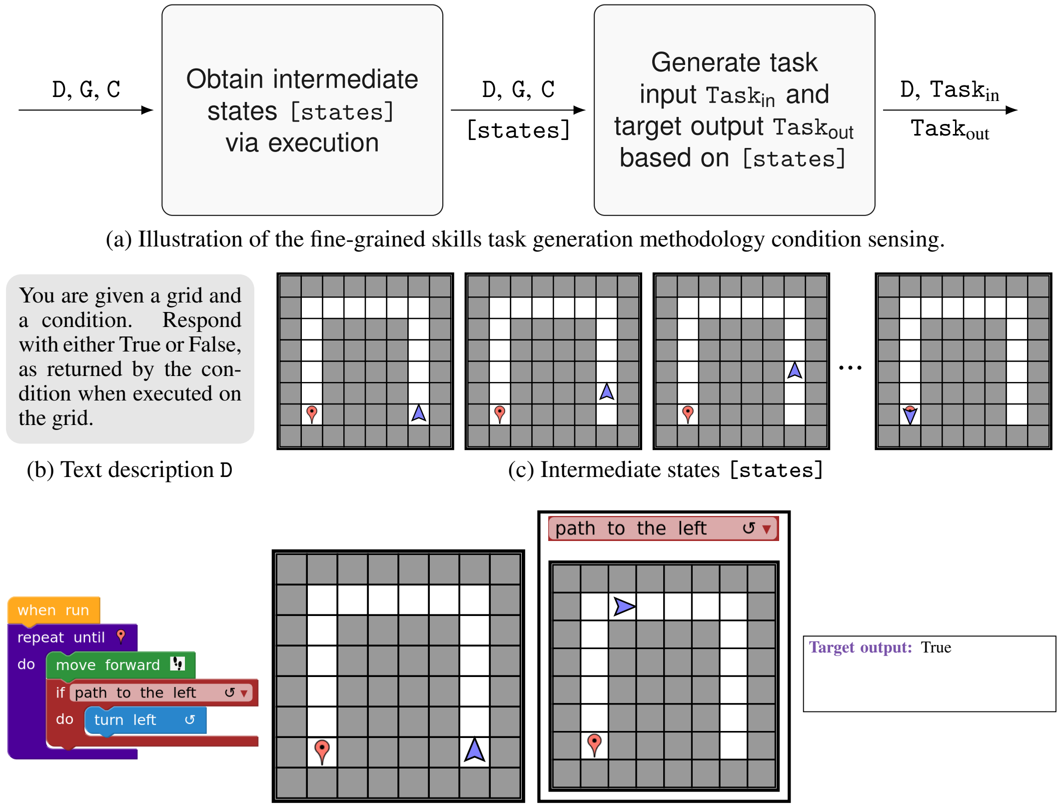
(d) Input code C (e) Input grid G (f) Mutated code C_{mut} (g) Each option applied to C

Figure 11: Illustration of our MCQ task generation methodology instantiated on the bug repair task in *Evaluating* cognitive level. **(a)** gives an overview of the pipeline. **(b,d,e)** show the input components for the pipeline: text description D, grid G, and its solution code C. **(c)** shows the generated correct option Ans along with the generated distractors {Dis}, put together to obtain all four options. **(f)** shows the mutated code C_{mut} obtained according to the selected correct option Ans. **(g)** shows the codes obtained by applying each of the options on C. The MCQ task is valid if the code obtained by applying Ans to C_{mut} solves G, and none of the codes obtained by applying {Dis} to C_{mut} solve G.

Next, we exemplify the generation process for a task involving repairing bugs in a given code in Figure 11. Similarly to the previous example, we start from a manually written text description for a task D, a grid G, and a code C. We first select a code mutation from the pool of feasible mutations. Let us say that the selected mutation is to add a `move forward` after Line 6. We generate the correct answer as the reverse mutation of the previously selected mutation i.e., Ans will be to remove Line 7. The next step is to apply the mutation adding `move forward` after Line 6 on C, thus obtaining C_{mut}. We then generate distractor options by picking three other feasible mutations that can be applied on C_{mut}, thus obtaining {Dis}. Again, after this step, we have all the components of an MCQ task, but the task has to be validated. The validation process for this example involves applying each mutation from the four options on C. A task is valid if G is solved by the code obtained by applying Ans on C, and is not solved by all the other codes obtained by applying {Dis} on C.

C.2 Fine-grained Skills Tasks Generation

We now demonstrate the generation methodology for the fine-grained skills tasks.



(a) Illustration of the fine-grained skills task generation methodology condition sensing.

(b) Text description D

(c) Intermediate states [states]

(d) Input code C

(e) Input grid G

(f) Generated $\texttt{Task}_{\text{in}}$

(g) Generated $\texttt{Task}_{\text{out}}$

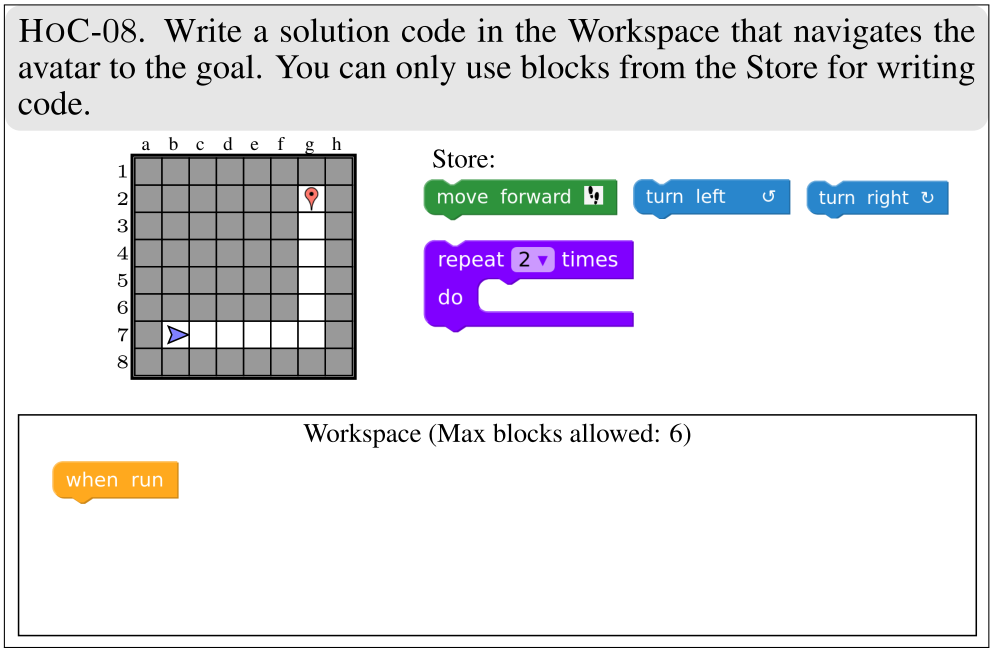
Figure 12: Illustration of our fine-grained skills task generation methodology instantiated on the condition sensing basic type of task. **(a)** gives an overview of the pipeline. **(b,d,e)** show the input components for the pipeline: text description D, grid G, and its solution code C. **(c)** shows the states generated by executing C on G. **(f)** shows the generated content $\texttt{Task}_{\text{in}}$ which will be part of the task input. **(g)** shows the generated target output $\texttt{Task}_{\text{out}}$.

Figure 12 shows an overview of our fine-grained skills generation methodology applied for a condition sensing task. We start with a code C, a grid G, and a manually written text description D which indicates what kind of task should be generated. The first step is to obtain the list of intermediate states [states] by executing C on G. The intermediate states are grids themselves, with the avatar having various locations and orientations, obtained after each execution of a basic action on the starting grid G. The next step is to use the information in [states] for generating the input for the task $\texttt{Task}_{\text{in}}$ and the target output $\texttt{Task}_{\text{out}}$. In our case, we randomly pick a state from [states], modify the orientation of the avatar for a richer set of generated situations, and select a random condition from the DSL (i.e., `path to the left`), thus obtaining $\texttt{Task}_{\text{in}}$. We execute the picked condition on the selected state to obtain the target output $\texttt{Task}_{\text{out}}$ (i.e., True). The generation process of other types of tasks is similar.

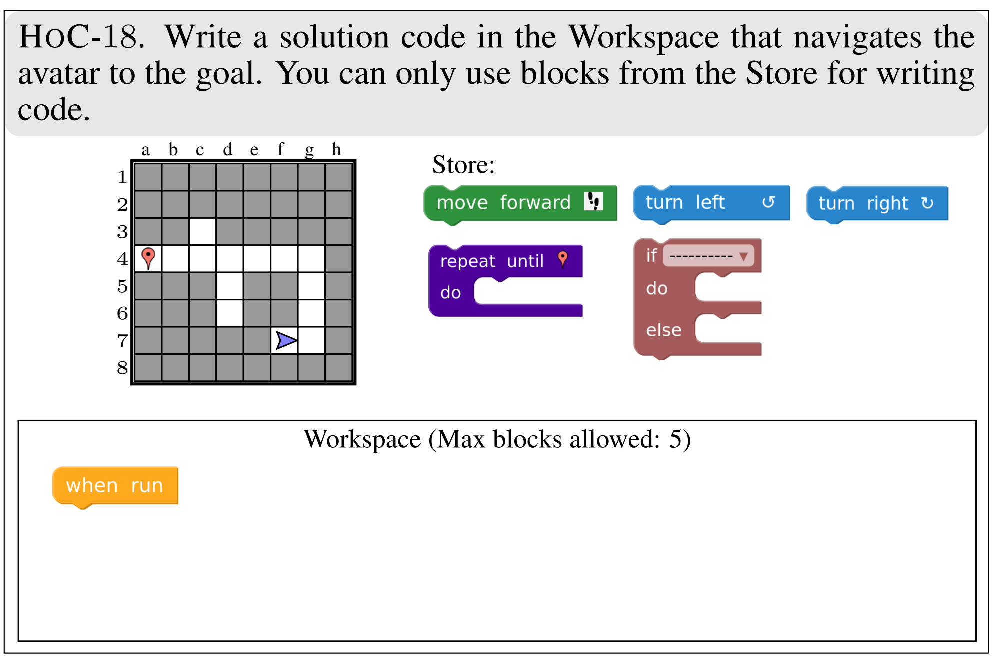
D Additional Data Details

D.1 Hour of Code:Maze Challenge

The *Hour of Code:Maze Challenge* online programming lesson is publicly accessible at `https://studio.code.org/s/hourofcode`. Use of the curriculum is permitted under a Creative Commons BY-NC-SA 4.0 license. We give two more illustrative examples of solution synthesis extracted from HoC in Figure 13.



(a) A HoC task with $\mathrm{C}_{\text{sketch}} = \textsc{Repeat}\{\}$.



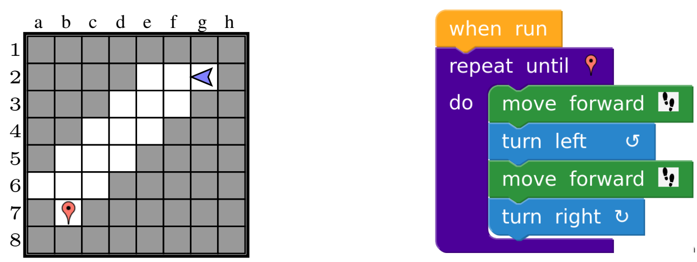
(b) A HoC task with $\mathrm{C}_{\text{sketch}} = \textsc{Repeat}\{\textsc{IfElse}\}$.

Figure 13: Solution synthesis tasks from HoC. Note that the number of times the repeat loop should be executed and the condition of the if statement can be changed.

D.2 The ACE Test

This test has been proposed in [25] and all the MCQ tasks can be found in the Appendix of the respective paper. Next, we give four more illustrative examples for ACE. Figure 14a shows a task corresponding to the *Analyzing* cognitive level. It involves reasoning about the trace, given a grid and a code. Figures 14b and 14c show tasks corresponding to the *Evaluating* cognitive level. The first one involves repairing a buggy code, while the second one involves reasoning about code equivalence without a given grid. Finally, Figure 14d shows a task corresponding to the *Creating* cognitive level. It involves placing the avatar on the incomplete grid so that the given code solves the modified grid.

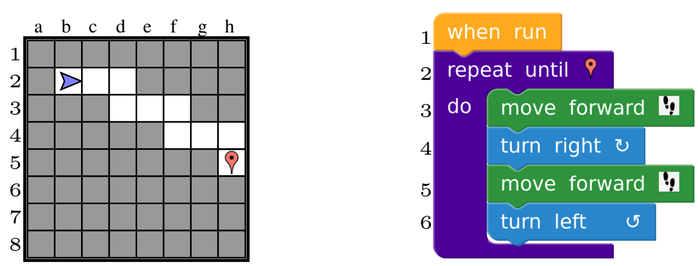
ACE-06. You are given a grid and its solution code. What happens to the avatar when the code is run on this grid?



OPTION A	The avatar will pass through the grid cells f2 and e2
OPTION B	The avatar will pass through the grid cells e3 and d3
OPTION C	The avatar will pass through the grid cells e4 and d4
OPTION D	The avatar will pass through the grid cells d4 and c4

(a) An ACE task involving tracing.

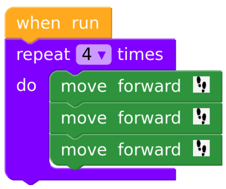
ACE-09. You are given a code and a grid. You may have to fix some errors in the code such that it solves the grid. How can you fix the code?



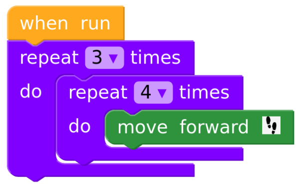
OPTION A	The code does not have any errors and it already solves the grid
OPTION B	Add move forward after Line 3
OPTION C	Add move forward after Line 5
OPTION D	Change Line 4 to turn left and Line 6 to turn right

(b) An ACE task involving bug repair.

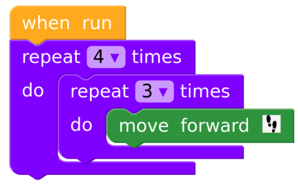
ACE-13. You are given a code CODE-1 and two smaller codes CODE-2 and CODE-3. You have to think about the AVATAR's behavior when a code is run on a grid. Which of these two smaller codes produce the same behavior as CODE-1 on any grid?



CODE-1



CODE-2

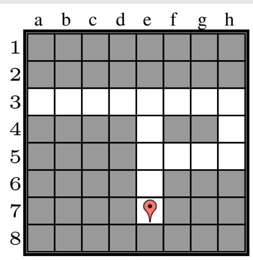


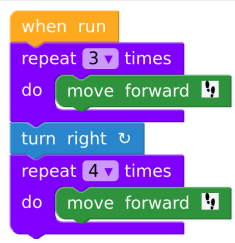
CODE-3

OPTION A	None of these two smaller codes
OPTION B	Only CODE-2
OPTION C	Only CODE-3
OPTION D	Both CODE-2 and CODE-3

(c) An ACE task involving code equivalence.

ACE-16. You are given a code and an incomplete grid without the avatar. What could be the initial position of the avatar such that the grid is solved by the code?





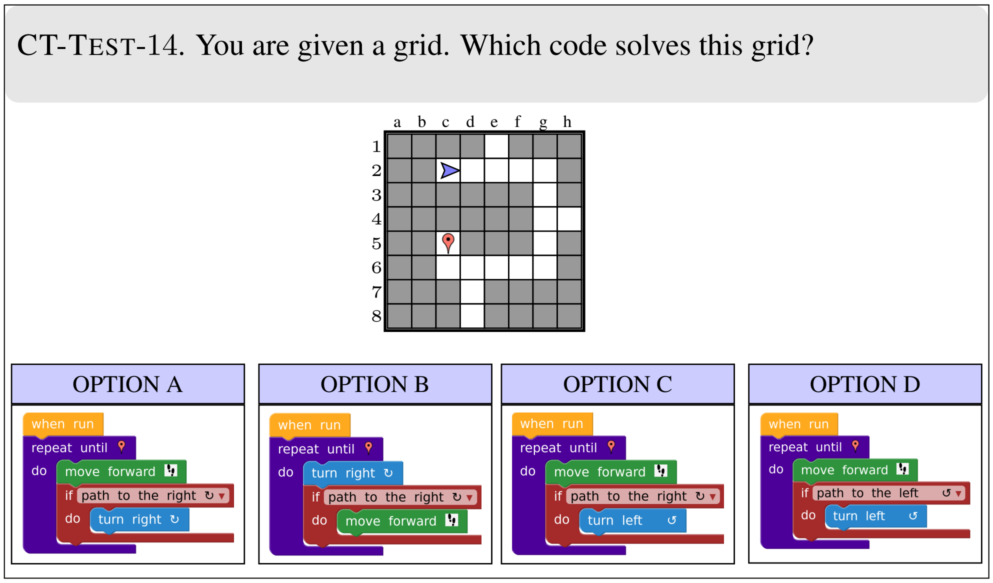
OPTION A	Grid cell h5 facing west
OPTION B	Grid cell a3 facing east
OPTION C	Grid cell b3 facing east
OPTION D	Grid cell h3 facing west

(d) An ACE task involving avatar design.

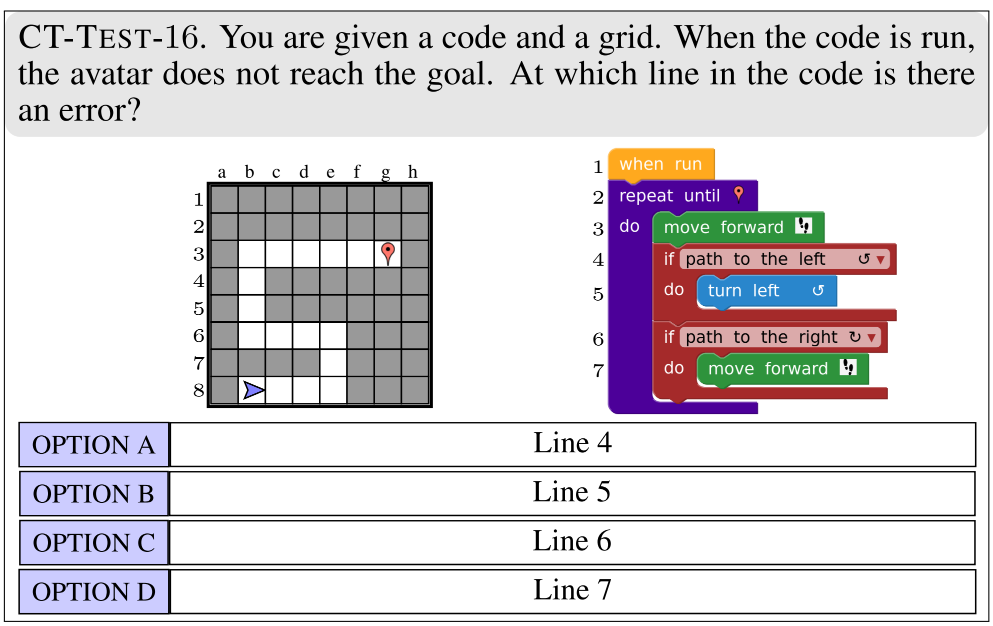
Figure 14: MCQ tasks from ACE.

D.3 The CT-TEST

This test has been proposed and refined in [28, 29], with all the MCQ tasks of the refined version being publicly accessible (in Spanish) at `http://goo.gl/IYEKMB`, free of charge, for research purposes. Next, we give two more illustrative examples for CT-TEST. Figure 15a shows a task requiring to pick the solution code for the given grid. Figure 15b shows a task requiring to point out the line that contains an error (i.e., which line should be changed so that the given code becomes a solution for the given grid).



(a) A CT-TEST task involving picking the solution code.



(b) A CT-TEST task involving pointing out the error.

Figure 15: MCQ tasks from CT-TEST. **(a)** shows a task where the correct solution for a given grid needs to be picked. Similar tasks appear in ACE. **(b)** shows a task involving pointing the line with the error in a given code.

E Synthetic Evaluation Data

Besides our real-world evaluation benchmark, we offer a larger-scale synthetic evaluation segment for HoC and ACE, as has also been considered in the literature [43, 60]. First, we offer details about the composition of our synthetic evaluation segment, then we present results for selected techniques.

E.1 Generation of the Synthetic Evaluation Segment

We generate the data using the same generation pipeline. However, we have the flexibility to vary the nature of synthetic evaluation data and how it differs from its training counterpart. We do this by conditioning the types of generated codes `C` (i.e., concepts covered, nesting structure). Thus, we obtain four parts of the synthetic evaluation dataset:

- HoC-SYNTH: this part corresponds to 758 solution synthesis tasks generated using the same codes as used for training but with newly generated grids. In particular, the samples encountered during evaluation are similar to samples encountered during training in terms of codes (output) but not in terms of grids (input).
- HoC-SYNTH$_{\text{hard}}$: this part corresponds to 378 hard solution synthesis tasks that are a subset of 758 HoC-SYNTH tasks. We selected those tasks for which the solution code requires a condition and are generally more difficult for generative models.
- HoC-FILTERED: this part corresponds to 758 solution synthesis tasks. These are also generated using the same generation pipeline as used for generating the training datasets; however, we have ensured that the samples are distinct from training both in terms of codes (output) and grids (inputs).
- HoC-OoD: this part corresponds to 100 solution synthesis tasks, which are "out-of-distribution" (OoD), meaning that the solution structures for these tasks are different and more complex than those used in training datasets. For instance, this set contains tasks that require over three for-loops or a combination of multiple for-loops and while-loops.
- ACE-SYNTH: this part corresponds to 922 MCQ tasks generated using the same codes as used for training but with newly generated grids. Similarly to HoC-SYNTH, the samples encountered during evaluation are similar to samples encountered during training in terms of codes (output) but not in terms of grids (input).

E.2 Results of Selected Models

Technique	HoC-SYNTH	HoC-SYNTH$_{\text{hard}}$	HoC-FILTERED	HoC-OoD	ACE-SYNTH
LLAMA3-8B-INSTRUCT	0.1 (0.0)	0.0 (0.0)	0.1 (0.0)	0.0 (0.0)	29.5 (0.0)
GPT4O$_{\text{vis+text}}$	12.8 (0.0)	10.3 (0.0)	20.8 (0.0)	17.0 (0.0)	47.6 (1.0)
LLAMACT:HoC+MCQ+AUG$_{\text{exp}}$	37.5 (1.0)	8.6 (0.0)	34.4 (2.0)	18.0 (3.0)	77.5 (0.0)

Figure 16: Results of selected models on different parts of the synthetic evaluation segment.

We evaluate each of the following selected techniques over three seeds: LLAMA3-8B-INSTRUCT, GPT4O$_{\text{vis+text}}$, and LLAMACT:HoC+MCQ+AUG$_{\text{exp}}$. Figure E shows the results of the selected techniuqes, averaged as mean (stderr).LLAMA3-8B-INSTRUCT struggles consistently across all parts of the synthetic evaluation dataset, showing poor performance in both simple and challenging tasks. GPT4O$_{\text{vis+text}}$ struggles more with synthetic data compared to real-world tasks (results shown in Figure 7). LLAMACT:HoC+MCQ+AUG$_{\text{exp}}$ generally outperforms both models on synthetic data, showing strong results across most tasks. It also demonstrates good generalization capabilities, outperforming GPT4O$_{\text{vis+text}}$ on the HoC-FILTERED set and achieving comparable performance on the out-of-distribution tasks (HoC-OoD).

F Datasheet

We include the datasheet [61] for both our curated benchmark and the synthetically generated dataset.

F.1 Motivation

For what purpose was the dataset created?

The benchmark was curated for assessing generative models' computational thinking and problem-solving skills on tasks designed for school students. It is comprised of three parts, namely HoC, ACE and CT-TEST. HoC is used to analyze the models' solution synthesis capabilities under strict constraints, while ACE and CT-TEST are made for assessing the models' problem-solving skills over three cognitive levels: *Analyzing*, *Evaluating*, and *Creating*.

The synthetic dataset was generated to improve the models' performance on computational thinking and problem-solving tasks (e.g., via fine-tuning). We aim to accomplish two objectives with the dataset: improve performance on tasks assessed by the benchmark (i.e., solution synthesis and multi-choice questions) and offer the models a better understanding of visual programming via training on fine-grained skills.

Who created the dataset (e.g., which team, research group) and on behalf of which entity (e.g., company, institution, organization)?

The benchmark was curated by the authors of the original paper from sources such as [25, 29, 26], while the synthetic dataset was created by the authors, using techniques from literature [42, 43, 44]. Full details regarding the authors of the original paper will be given at the time of publication.

Who funded the creation of the dataset?

Full details regarding funding will be given at the time of publication.

Any other comments?

None.

F.2 Composition

What do the instances that comprise the dataset represent (e.g., documents, photos, people, countries)?

The dataset comprises grid representations, codes (under AST structure or in Python), traces (i.e., sequences of grids), sequences of basic actions, instructions, and answer options for multi-choice questions and explanations involving reasoning about the answers. The tasks in the benchmark also contain images which are visual representations of the grids in the dataset. We also include prompt templates for easily generating prompts.

For ease of usage, we offer a method for accessing data in a uniform way. In this data representation, all records have a task type (i.e., what kind of task they represent – see Section 4), an instruction (i.e., for multi-choice questions), a set of grids, a set of codes, answer options (i.e., for multi-choice questions), and an answer. Additionally, they contain a miscellaneous field that includes any information necessary for the respective tasks (e.g., traces, explanations).

How many instances are there in total (of each type, if appropriate)?

There are $111,861$ synthetic data points for training, $2,538$ synthetic data points for evaluating, and 65 tasks in the real-world benchmark.

Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set?

Only in the case of CT-TEST, the data is a sample of 24 tasks out of a total of 28 tasks, due to incompatibility reasons.

What data does each instance consist of?

Each instance consists of the grid representation, its solution code, usually a type of task and an answer and additional data for formulating the question, answer, or explanations (e.g., trace of grid state or sequence of basic actions).

Is there a label or target associated with each instance?

Yes, possible target answers are included.

Is any information missing from individual instances?

Not to our knowledge.

For the uniform access method, some fields are empty due to the nature of the task.

Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)?

No.

Are there recommended data splits (e.g., training, development/validation, testing)?

Yes, the data is naturally split into real-world evaluation (i.e., benchmark), synthetic evaluation (i.e., synthetic data), and training (i.e., synthetic data) segments. The training data is further randomly split into train (i.e., 90% of synthetic data) and validation (i.e., 10% of synthetic data).

Are there any errors, sources of noise, or redundancies in the dataset?

The benchmark is created by experts and does not have errors to the best of our knowledge. The synthetic dataset was verified automatically while being created. However, it has not been manually checked for errors or redundancies that may get past automatic checks.

Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)?

The dataset is self-contained.

Does the dataset contain data that might be considered confidential (e.g., data that is protected by legal privilege or by doctor–patient confidentiality, data that includes the content of individuals' non-public communications)?

No.

Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety?

No.

Does the dataset identify any subpopulations (e.g., by age, gender)?

No.

Is it possible to identify individuals (i.e., one or more natural persons), either directly or indirectly (i.e., in combination with other data) from the dataset?

No.

Does the dataset contain data that might be considered sensitive in any way (e.g., data that reveals race or ethnic origins, sexual orientations, religious beliefs, political opinions or union memberships, or locations; financial or health data; biometric or genetic data; forms of government identification, such as social security numbers; criminal history)?

No.

Any other comments?

None.

F.3 Collection Process

How was the data associated with each instance acquired?

The data used for benchmarking was collected from each respective source [25, 29, 26] and translated to our representation. The synthetic dataset was created by the authors, using techniques from literature [42, 43, 44].

What mechanisms or procedures were used to collect the data (e.g., hardware apparatuses or sensors, manual human curation, software programs, software APIs?

The data used for benchmarking was collected via manual curation, while the synthetic dataset was produced by authors' own software using techniques from literature [42, 43, 44].

If the dataset is a sample from a larger set, what was the sampling strategy (e.g., deterministic, probabilistic with specific sampling probabilities)?

While curating data from CT-TEST, we kept the data compatible with our representation of a grid.

Who was involved in the data collection process (e.g., students, crowdworkers, contractors) and how were they compensated (e.g., how much were crowdworkers paid)?

The authors were the only involved in the data collection process.

Over what timeframe was the data collected?

The data was collected between February and May 2024. It is not known to the authors when the HOC lesson [26] was created. Data for ACE was published in 2024, while data for CT-TEST was published in 2017.

Were any ethical review processes conducted (e.g., by an institutional review board)?

No.

Did you collect the data from the individuals in question directly, or obtain it via third parties or other sources (e.g., websites)?

N/A.

Were the individuals in question notified about the data collection?

N/A.

Did the individuals in question consent to the collection and use of their data?

N/A.

If consent was obtained, were the consenting individuals provided with a mechanism to revoke their consent in the future or for certain uses?

N/A.

Has an analysis of the potential impact of the dataset and its use on data subjects (e.g., a data protection impact analysis) been con- ducted?

N/A.

Any other comments?

None.

F.4 Preprocessing/cleaning/labeling

Was any preprocessing/cleaning/labeling of the data done (e.g., discretization or bucketing, tokenization, part-of-speech tagging, SIFT feature extraction, removal of instances, processing of missing values)?

No.

Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data (e.g., to support unanticipated future uses)?

N/A.

Is the software that was used to preprocess/clean/label the data available?

N/A.

Any other comments?

None.

F.5 Uses

Has the dataset been used for any tasks already?

The curated data was previously used in their intended ways by the authors of each of [25, 29, 26]. The synthetic dataset was only used in the original paper.

Is there a repository that links to any or all papers or systems that use the dataset?

There will be a publicly released repository at the time of publication.

What (other) tasks could the dataset be used for?

The dataset can be used for any purpose requiring various computational thinking tasks, or the grids and codes involved. The original paper uses them for fine-tuning large language models, but researchers are free to use them in other ways to improve generative models' performance and not only.

Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses?

No.

Are there tasks for which the dataset should not be used?

Training on this data does not guarantee generalization to other domains.

Any other comments?

None.

F.6 Distribution

Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created?

Yes, it will be free and available online at the time of publication.

How will the dataset will be distributed (e.g., tarball on website, API, GitHub)?

The dataset will be available on GitHub at the time of the oringial paper's publication, under CC BY-NC-SA 4.0 license. The source code for processing data and evaluating models will be available under MIT license.

When will the dataset be distributed?

The dataset will be distributed at the time of the original paper's publication.

Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?

The data will be distributed under CC BY-NC-SA 4.0 license.

Have any third parties imposed IP-based or other restrictions on the data associated with the instances?

No.

Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?

No.

Any other comments?

None.

F.7 Maintenance

Who will be supporting/hosting/maintaining the dataset?

The authors of the paper will provide needed maintenance to the dataset. Further work may add additional domains to the data and contributors are free to submit pull requests.

How can the owner/curator/manager of the dataset be contacted (e.g., email address)?

The authors will make this data public at the time of the original paper's publication.

Is there an erratum?

No.

Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)?

Further work may choose to add more data (e.g., more tests, or synthetic data for Karel).

If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances (e.g., were the individuals in question told that their data would be retained for a fixed period of time and then deleted)?

N/A.

Will older versions of the dataset continue to be supported/hosted/maintained?

In case new data will be released, a versioning system will be used to keep track of versions.

If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?

We plan to use GitHub for maintaining both the code and the data. Contributors can either contact authors or submit pull requests.

Any other comments?

None.

G Additional Reasoning Examples

We provide reasoning examples done by LLAMACT:HOC+MCQ+AUG$_{exp}$, LLAMACT:HOC, and GPT4O$_{vis+text}$. First, we show how explanations help LLAMACT reason better spatially. For the solution synthesis task in Figure 1b, LLAMACT:HOC+MCQ+AUG$_{exp}$ provides an almost correct explanation (shown in Figure 17a), helping the model solve the task by correctly understanding the avatar's orientation. In contrast, LLAMACT:HOC (see Figure 17b) struggles with distinguishing left from right, failing to provide a correct solution despite understanding the task's general structure. Next, we provide reasoning examples for GPT4O$_{vis+text}$ for the tasks in Figures 1b and 1c. The parts where the reasoning is wrong are highlighted in red. We can notice that for HOC, the model comes up with a general code that may be able to solve a large variety of grids, but it doesn't respect `maxSize` and `Store` from Figure 1b. Additionally, the reasoning involving tracing and the sequence of basic actions is flawed, as the model doesn't take all walls into account. The model makes a very convincing argument for the ACE example, yet it reaches a wrong solution by a partially flawed reasoning process. It places walls in a manner that would block the avatar from reaching the goal in Figure 1c.

avatar locations: g2:west f2:west e2:west d2:west c2:west b2:west b2:south b3:south b4:south b5:south b6:south b7:south b7:east c7:east d7:east e7:east f7:east g7:east g7:north g6:north g5:north g5:west f5:west e5:west e5:south goal
avatar actions: move move move move move turnLeft move move move move move turnLeft move move move move move turnLeft move move turnLeft move move turnLeft
code:

```
when run:
    repeatUntil goal:
        move
        if leftIsClear:
            turnLeft
```

(a) Answer with an explanation.

code:

```
when run:
    repeatUntil goal:
        move
        if rightIsClear:
            turnRight
```

(b) Answer with no explanation.

Figure 17: Illustrative examples showing the reasoning processes of LLAMACT:HOC+MCQ+AUG$_{exp}$ and LLAMACT:HOC for HOC task in Figure 1b, with errors highlighted in red.

Sequence of Locations:
(1, 6, west) (1, 5, west) (1, 4, west) (1, 3, west) (1, 2, west) (1, 1, west) (1, 1, south) (2, 1, south) (3, 1, south) (4, 1, south) (4, 1, east) (4, 2, east) (4, 3, east) (4, 4, east)
Actions:
move move move move move turnLeft move move move turnRight move move move
Minimal Python Code:

```
when run:
    repeatUntil goal:
        if frontIsClear:
            move
        elif leftIsClear:
            turnLeft
        else:
            turnRight
```

(a) Reasoning for HOC.

To determine the smallest number of additional WALL cells needed to ensure the grid is solved by the given code, we need to trace the code and understand the avatar's movement. The avatar starts at a1 facing east. The goal is at a6. The avatar will move forward if there is a path ahead, otherwise, it will turn right.
Let's trace the avatar's movement step-by-step: a1 facing east – b1 facing east – c1 facing east – d1 facing east – e1 facing east – f1 facing east – g1 facing east – h1 facing east – h1 facing south – h2 facing south – h3 facing south – h4 facing south – h5 facing south – h6 facing south – h7 facing south – h8 facing south – h8 facing west – g8 facing west – f8 facing west (blocked cell).
At this point, the avatar will crash into the blocked cell at f8. To prevent this, we need to add WALL cells to guide the avatar to the goal at a6. To ensure the avatar reaches the goal, we can add WALL cells to block the avatar's path and force it to turn at the correct points. The critical points to block are: b1 to prevent the avatar from moving east initially; a2 to prevent the avatar from moving south after turning right. By adding these two WALL cells, the avatar will be forced to turn right at a1 and move south directly to the goal at a6.
Thus, the correct answer is Option b: 2.

(b) Reasoning for ACE.

Figure 18: Examples of reasoning done by GPT4O$_{vis+text}$.

H Prompt Templates

In this section, we give details about the prompts we used for fine-tuning LLAMACT and for inference with all models.

H.1 Domain background for HOC and ACE

We start with the prompts that offer background knowledge and familiarize the models with our representations of the grids and codes for the HOC and ACE tests. Figure 19a presents the prompt used to provide background information during fine-tuning. The same prompt was used during inference for models receiving grids under text representation. Figure 19b shows the prompt used during inference with models for which we input grids under visual representation. Finally, Figure 19c shows the prompt used during inference for models taking as input grids under both text and visual representation.

Background information and representation for text

You are an expert in the visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, a task is a 2D grid of cells, where each cell is either empty or blocked. To solve a task is to move an avatar from a given starting position to a given goal position. The avatar can move forward, turn left, or turn right, sense the path ahead, to the left, or to the right, and check if the destination is reached. If the avatar moves into a blocked cell, the avatar will crash and the task will fail. Turning does not make the avatar change coordinates, it just changes its orientation. For a given task, you will be given the size of the grid, the starting position, the destination position, and a list of blocked cells, e.g., size of the grid in terms of rows and columns is (x1,y1), avatar's location as (row, column, direction) is (x2,y2,d), goal's location as (row, column) is (x3,y3), and the list of blocked cells rows and columns is [(x4,y4), (x5,y5), ...].

You are given a list of pre-defined primitive functions that you can use to solve the task. Next, we explain how each primitive works.

The first primitive is move_forward(). This function moves the avatar one cell forward in the direction it is facing. If the cell in front of the avatar is blocked or outside of the boundaries, the avatar will crash and the task will fail. In that case, the avatar will just have 'crashed' instead of the location. If the avatar's location is (x,y,north), then move_forward() will move the avatar to (x-1,y,north). If the avatar's location is (x,y,east), then move_forward() will move the avatar to (x,y+1,east). If the avatar's location is (x,y,south), then move_forward() will move the avatar to (x+1,y,south). If the avatar's location is (x,y,west), then move_forward() will move the avatar to (x,y-1,west).

The second primitive is turn_left(). This function changes the avatar's direction by 90 degrees to the left. If the avatar's location is (x,y,north), then turn_left() will change the avatar's location to (x,y,west). If the avatar's location is (x,y,east), then turn_left() will change the avatar's location to (x,y,north). If the avatar's location is (x,y,south), then turn_left() will change the avatar's location to (x,y,east). If the avatar's location is (x,y,west), then turn_left() will change the avatar's location to (x,y,south).

The third primitive is turn_right(). This function changes the avatar's direction by 90 degrees to the right. If the avatar's location is (x,y,north), then turn_right() will change the avatar's location to (x,y,east). If the avatar's location is (x,y,east), then turn_right() will change the avatar's location to (x,y,south). If the avatar's location is (x,y,south), then turn_right() will change the avatar's location to (x,y,west). If the avatar's location is (x,y,west), then turn_right() will change the avatar's location to (x,y,north).

The fourth primitive is path_ahead(). This function checks if the cell in front of the avatar is free. If the avatar's location is (x,y,north), then path_ahead() will return "true" if the cell (x-1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,east), then path_ahead() will return "true" if the cell (x,y+1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,south), then path_ahead() will return "true" if the cell (x+1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,west), then path_ahead() will return "true" if the cell (x,y-1) is not in the list of blocked cells, and "false" otherwise.

The fifth primitive is path_to_the_left(). This function checks if the cell to the left of the avatar is free. If the avatar's location is (x,y,north), then path_to_the_left() will return "true" if the cell (x,y-1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,east), then path_to_the_left() will return "true" if the cell (x-1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,south), then path_to_the_left() will return "true" if the cell (x,y+1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,west), then path_to_the_left() will return "true" if the cell (x+1,y) is not in the list of blocked cells, and "false" otherwise.

The sixth primitive is path_to_the_right(). This function checks if the cell to the right of the avatar is free. If the avatar's location is (x,y,north), then path_to_the_right() will return "true" if the cell (x,y+1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,east), then path_to_the_right() will return "true" if the cell (x+1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,south), then path_to_the_right() will return "true" if the cell (x,y-1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,west), then path_to_the_right() will return "true" if the cell (x-1,y) is not in the list of blocked cells, and "false" otherwise.

The seventh primitive is goal(). If the avatar's location is the same as the goal's location, then goal() will return "true", and "false" otherwise.

Below is an instruction that describes a requirement, paired with an input task that provides further context. You will need to use the domain knowledge and the primitives to solve the requirement. Write a response that appropriately completes the request, carefully following the required format.

(a) Prompt used as background information for models employing text modality only.

Background information and representation for vision

You are an expert in the visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, a task is a 2D grid of cells, where each cell is either empty or blocked. To solve a task is to move an avatar from a given starting position to a given goal position. The avatar can move forward, turn left, or turn right, sense the path ahead, to the left, or to the right, and check if the destination is reached. If the avatar moves into a blocked cell, the avatar will crash and the task will fail. Turning does not make the avatar change coordinates, it just changes its orientation.

A task is represented as an image of a 8x8 visual grid. The rows are labeled from 0 to 7 (from top to bottom), and the columns are labeled from 0 to 7 (from left to right). This grid contains blocked cells, free cells, a goal cell, and the avatar (with a specific location and direction). We represent a task's 8x8 visual grid with the following symbols. A gray cell represents a wall cell. A white cell represents a free cell. An orange asterisk represents the goal cell. A blue triangle pointing to the right represents the avatar's location at the corresponding coordinates, facing east direction. A blue triangle pointing to the left represents the avatar's location at the corresponding coordinates, facing west direction. A blue triangle pointing upwards represents the avatar's location at the corresponding coordinates, facing north direction. A blue triangle pointing downwards represents the avatar's location at the corresponding coordinates, facing south direction.

You are given a list of pre-defined primitive functions that you can use to solve the task. Next, we explain how each primitive works.

[Content same as above – omitted for brevity]

(b) Prompt used as background information for models employing vision modality only.

Background information and representation for vision+text

You are an expert in the visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, a task is a 2D grid of cells, where each cell is either empty or blocked. To solve a task is to move an avatar from a given starting position to a given goal position. The avatar can move forward, turn left, or turn right, sense the path ahead, to the left, or to the right, and check if the destination is reached. If the avatar moves into a blocked cell, the avatar will crash and the task will fail. Turning does not make the avatar change coordinates, it just changes its orientation. For a given task, you will be given the size of the grid, the starting position, the destination position, and a list of blocked cells, e.g., size of the grid in terms of rows and columns is (x1,y1), avatar's location as (row, column, direction) is (x2,y2,d), goal's location as (row, column) is (x3,y3), and the list of blocked cells rows and columns is [(x4,y4), (x5,y5), ...].

Besides the textual description, I will provide with one or more images of 8x8 visual grids, corresponding to the previously described task. The rows are labeled from 0 to 7 (from top to bottom), and the columns are labeled from 0 to 7 (from left to right). This grid contains blocked cells, free cells, a goal cell, and the avatar (with a specific location and direction). We represent a task's 8x8 visual grid with the following symbols. A gray cell represents a wall cell. A white cell represents a free cell. An orange asterisk represents the goal cell. A blue triangle pointing to the right represents the avatar's location at the corresponding coordinates, facing east direction. A blue triangle pointing to the left represents the avatar's location at the corresponding coordinates, facing west direction. A blue triangle pointing upwards represents the avatar's location at the corresponding coordinates, facing north direction. A blue triangle pointing downwards represents the avatar's location at the corresponding coordinates, facing south direction.

You are given a list of pre-defined primitive functions that you can use to solve the task. Next, we explain how each primitive works.

[Content same as above – omitted for brevity]

(c) Prompt used as background information for models employing both vision and text modalities.

Figure 19: Prompts used for offering background and representation information for HoC and ACE.

H.2 Domain background for CT-TEST

Similarly, we present the prompts offering background and representation information during inference on CT-TEST. These include some additional information involving Karel tasks and another type of tasks containing colored cells. Figure 20a shows the prompt used for models receiving grids under text representation. Figure 20b shows the prompt used for models receiving grids under visual representation only. Finally, Figure 20c shows the prompt used during inference for models taking as input grids under both text and visual representation.

Background information and representation for text

You are an expert in the visual programming domain of Hour of Code: Maze Challenge from code.org and Karel. In this domain, a task is a 2D grid of cells, where each cell is either empty or blocked. To solve a task is to move an avatar from a given starting position to a given goal position. If any markers are present, the avatar should collect all of them. The avatar can move forward, turn left, or turn right, sense the path ahead, to the left, or to the right, check if the destination is reached, pick markers, and check whether any markers are present in the current cell. If the avatar moves into a blocked cell, the avatar will crash and the task will fail. Turning does not make the avatar change coordinates, it just changes its orientation. For a given task, you will be given the size of the grid, the starting position, the destination position, marker positions if any are present, and a list of blocked cells, e.g., size of the grid in terms of rows and columns is (x1,y1), avatar's location as (row, column, direction) is (x2,y2,d), goal's location as (row, column) is (x3,y3), list of markers rows, columns and how many as [(x4,y4,z4), (x5,y5,z5), ...], and the list of blocked cells rows and columns is [(x6,y6), (x7,y7), ...]. Some cells may be colored differently (i.e., blue, green), in this case, they will be indicated separately in the input.

You are given a list of pre-defined primitive functions that you can use to solve the task. Next, we explain how each primitive works.

The first primitive is move_forward(). This function moves the avatar one cell forward in the direction it is facing. If the cell in front of the avatar is blocked or outside of the boundaries, the avatar will crash and the task will fail. In that case, the avatar will just have 'crashed' instead of the location. If the avatar's location is (x,y,north), then move_forward() will move the avatar to (x-1,y,north). If the avatar's location is (x,y,east), then move_forward() will move the avatar to (x,y+1,east). If the avatar's location is (x,y,south), then move_forward() will move the avatar to (x+1,y,south). If the avatar's location is (x,y,west), then move_forward() will move the avatar to (x,y-1,west).

The second primitive is turn_left(). This function changes the avatar's direction by 90 degrees to the left. If the avatar's location is (x,y,north), then turn_left() will change the avatar's location to (x,y,west). If the avatar's location is (x,y,east), then turn_left() will change the avatar's location to (x,y,north). If the avatar's location is (x,y,south), then turn_left() will change the avatar's location to (x,y,east). If the avatar's location is (x,y,west), then turn_left() will change the avatar's location to (x,y,south).

The third primitive is turn_right(). This function changes the avatar's direction by 90 degrees to the right. If the avatar's location is (x,y,north), then turn_right() will change the avatar's location to (x,y,east). If the avatar's location is (x,y,east), then turn_right() will change the avatar's location to (x,y,south). If the avatar's location is (x,y,south), then turn_right() will change the avatar's location to (x,y,west). If the avatar's location is (x,y,west), then turn_right() will change the avatar's location to (x,y,north).

The fourth primitive is path_ahead(). This function checks if the cell in front of the avatar is free. If the avatar's location is (x,y,north), then path_ahead() will return "true" if the cell (x-1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,east), then path_ahead() will return "true" if the cell (x,y+1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,south), then path_ahead() will return "true" if the cell (x+1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,west), then path_ahead() will return "true" if the cell (x,y-1) is not in the list of blocked cells, and "false" otherwise.

The fifth primitive is path_to_the_left(). This function checks if the cell to the left of the avatar is free. If the avatar's location is (x,y,north), then path_to_the_left() will return "true" if the cell (x,y-1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,east), then path_to_the_left() will return "true" if the cell (x-1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,south), then path_to_the_left() will return "true" if the cell (x,y+1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,west), then path_to_the_left() will return "true" if the cell (x+1,y) is not in the list of blocked cells, and "false" otherwise.

The sixth primitive is path_to_the_right(). This function checks if the cell to the right of the avatar is free. If the avatar's location is (x,y,north), then path_to_the_right() will return "true" if the cell (x,y+1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,east), then path_to_the_right() will return "true" if the cell (x+1,y) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,south), then path_to_the_right() will return "true" if the cell (x,y-1) is not in the list of blocked cells, and "false" otherwise. If the avatar's location is (x,y,west), then path_to_the_right() will return "true" if the cell (x-1,y) is not in the list of blocked cells, and "false" otherwise.

The seventh primitive is goal(). If the avatar's location is the same as the goal's location, then goal() will return "true", and "false" otherwise.

The eighth primitive is pick_marker(). This function picks exactly one marker from the current cell. If there are no markers, avatar will crash. If a cell is represented as (x,y,z) and the avatar is located at (x,y), then pick_marker() will pick one marker from the cell and the cell will be represented as (x,y,z-1). If z is 0, then the cell will be empty and the avatar will crash.

The ninth primitive is marker_present(). This function checks if there are any markers in the current cell. If the cell is represented as (x,y,z) and the avatar is located at (x,y), then marker_present() will return "true" if z is greater than 0, and "false" otherwise.

The tenth and eleventh primitives are cell_red() and cell_green(). These functions check if the current cell is colored red or green, respectively. If the cell the avatar is standing on is colored red, cell_red() will return "true", and "false" otherwise. If the cell is colored green, cell_green() will return "true", and "false" otherwise.

Below is an instruction that describes a requirement, paired with an input task that provides further context. You will need to use the domain knowledge and the primitives to solve the requirement. Write a response that appropriately completes the request, carefully following the required format.

(a) Prompt used as background information for models employing text modality only.

Background information and representation for vision

You are an expert in the visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, a task is a 2D grid of cells, where each cell is either empty or blocked. To solve a task is to move an avatar from a given starting position to a given goal position. The avatar can move forward, turn left, or turn right, sense the path ahead, to the left, or to the right, and check if the destination is reached. If the avatar moves into a blocked cell, the avatar will crash and the task will fail. Turning does not make the avatar change coordinates, it just changes its orientation.

A task is represented as an image of a 8x8 visual grid. The rows are labeled from 0 to 7 (from top to bottom), and the columns are labeled from 0 to 7 (from left to right). This grid contains blocked cells, free cells, a goal cell, and the avatar (with a specific location and direction). We represent a task's 8x8 visual grid with the following symbols. A gray cell represents a wall cell. A white cell represents a free cell. An orange asterisk represents the goal cell. A blue triangle pointing to the right represents the avatar's location at the corresponding coordinates, facing east direction. A blue triangle pointing to the left represents the avatar's location at the corresponding coordinates, facing west direction. A blue triangle pointing upwards represents the avatar's location at the corresponding coordinates, facing north direction. A blue triangle pointing downwards represents the avatar's location at the corresponding coordinates, facing south direction. A number in a cell represents the number of markers in that cell. A red cell represents a cell colored red. A green cell represents a cell colored green.

You are given a list of pre-defined primitive functions that you can use to solve the task. Next, we explain how each primitive works.

[Content same as above – omitted for brevity]

(b) Prompt used as background information for models employing vision modality only.

Background information and representation for vision+text

You are an expert in the visual programming domain of Hour of Code: Maze Challenge from code.org and Karel. In this domain, a task is a 2D grid of cells, where each cell is either empty or blocked. To solve a task is to move an avatar from a given starting position to a given goal position. If any markers are present, the avatar should collect all of them. The avatar can move forward, turn left, or turn right, sense the path ahead, to the left, or to the right, check if the destination is reached, pick markers, and check whether any markers are present in the current cell. If the avatar moves into a blocked cell, the avatar will crash and the task will fail. Turning does not make the avatar change coordinates, it just changes its orientation. For a given task, you will be given the size of the grid, the starting position, the destination position, marker positions if any are present, and a list of blocked cells, e.g., size of the grid in terms of rows and columns is (x1,y1), avatar's location as (row, column, direction) is (x2,y2,d), goal's location as (row, column) is (x3,y3), list of markers rows, columns and how many as [(x4,y4,z4), (x5,y5,z5), ...], and the list of blocked cells rows and columns is [(x6,y6), (x7,y7), ...]. Some cells may be colored differently (i.e., blue, green), in this case, they will be indicated separately in the input.

Besides the textual description, I will provide with one or more images of 8x8 visual grids, corresponding to the previously described task. The rows are labeled from 0 to 7 (from top to bottom), and the columns are labeled from 0 to 7 (from left to right). This grid contains blocked cells, free cells, a goal cell, and the avatar (with a specific location and direction). We represent a task's 8x8 visual grid with the following symbols. A gray cell represents a wall cell. A white cell represents a free cell. An orange asterisk represents the goal cell. A blue triangle pointing to the right represents the avatar's location at the corresponding coordinates, facing east direction. A blue triangle pointing to the left represents the avatar's location at the corresponding coordinates, facing west direction. A blue triangle pointing upwards represents the avatar's location at the corresponding coordinates, facing north direction. A blue triangle pointing downwards represents the avatar's location at the corresponding coordinates, facing south direction. A number in a cell represents the number of markers in that cell. A red cell represents a cell colored red. A green cell represents a cell colored green.

You are given a list of pre-defined primitive functions that you can use to solve the task. Next, we explain how each primitive works.

[Content same as above – omitted for brevity]

(c) Prompt used as background information for models employing both vision and text modalities.

Figure 20: Prompts used for offering background and representation information for CT-Test.

H.3 Instructions

We continue with the prompts used to instruct the models about their task. We use the prompt in Figure 21a for inference on HoC, and the prompt in Figure 21b for inference on ACE and CT-Test. All the prompts in Figure 21 were used for fine-tuning.

Solution synthesis prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to write a code that will solve the task. First, write the sequence of locations the avatar should navigate through. Then write the primitives that the avatar should execute to solve the task. Finally, write the minimal Python code, including loops and conditionals, that will take the avatar to the goal, according to the sequence of primitives you have written. Do not exceed the maximum number of lines of code. Do not use constructs that are not allowed. If any primitives, if statements, if-else statements, while loops or for loops are not specified in the allowed constructs, they are not allowed.

###Input:

Task: [task]

Maximum number of lines of code: [size]

Allowed constructs: [constructs]

###Response:

(a) Prompt used for solution synthesis.

Multi-choice question prompt

###Instruction:

[instruction] Note that wall is a synonym for blocked cells. Use your domain knowledge to carefully reason about each answer option, tracing codes when necessary. Select the correct answer in the end, formatting it as "Thus, the correct answer is Option [your answer]".

###Input:

[input]

###Response:

(b) Prompt used for asking multi-choice questions.

Locate avatar prompt

###Instruction:

Do not solve the task. Tell me the position of the avatar as provided by the task formatted as "avatar: x:y:direction".

###Input:

[task]

###Response:

(c) Prompt used for locating the avatar in the basics fine-grained skills.

Locate goal prompt

###Instruction:

Do not solve the task. Tell me the position of the goal as provided by the task formatted as "goal: x:y".

###Input:

[task]

###Response:

(d) Prompt used for locating the goal in the basics fine-grained skills.

Apply action prompt

###Instruction:

Do not solve the task. Use your knowledge about the primitives to give me the new location of the avatar or whether it has crashed after executing the given primitive. Use the format "avatar: x:y:direction" or "avatar: crashed" if the avatar crashed.

###Input:

Task: [task]

Primitive: [action]

###Response:

(e) Prompt used for applying an action in the basics fine-grained skills.

Sense condition prompt

###Instruction:

Do not solve the task. Use your knowledge about the primitives to respond with either "true" or "false", as returned by the primitive executed on the given task.

###Input:

Task: [task]

Primitive: [condition]

###Response:

(f) Prompt used for sensing a condition in the basics fine-grained skills.

Trace sequence prompt

###Instruction:

Do not solve the task. Use your knowledge about the primitives to give me a trace of the new locations of the avatar after executing each primitive of the given code. The initial configuration of the task is given and you will start from that. Use the format "avatar: x1:y1:direction x2:y2:direction ..." to represent the trace. If at any time the avatar crashes, print "crash" and stop the trace. If the avatar reaches the goal, print "goal" and stop the trace.

###Input:

Initial configuration:

[task]

Code to execute:

[code]

###Response:

(g) Prompt used for tracing a sequence of basic actions in the tracing fine-grained skills.

Trace code prompt

###Instruction:

Do not solve the task. Use your knowledge about the primitives, conditionals and Python programming to give me a trace of the new locations of the avatar after executing each primitive of the given code. The initial configuration of the task is given and you will start from that. Use the format "avatar: x1:y1:direction x2:y2:direction ..." to represent the trace. If at any time the avatar crashes, print "crash" and stop the trace. If the avatar reaches the goal, print "goal" and stop the trace.

###Input:

Initial configuration:

[task]

Code to execute:

[code]

###Response:

(h) Prompt used for tracing a code in the tracing fine-grained skills.

Place avatar prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to pick the avatar's initial location in order to create a task that can be solved by the given code. First, write the locations the avatar should navigate through to reach the goal. Then write the sequence of primitives that the avatar will execute when executing the code on the task that you will obtain. Finally, write down the avatar's initial location under the format "avatar: x:y:direction".

###Input:

Code:

[code]

Task:[task]

###Response:

(i) Prompt used for placing the avatar in the grid synthesis fine-grained skills.

Place goal prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to pick the location of the goal in order to create a task that can be solved by the given code. First, write the locations the avatar should navigate through to reach the goal. Then write the sequence of primitives that the avatar will execute when executing the code on the task that you will obtain. Finally, write down the location of the goal under the format "goal: x:y".

###Input:

Code:

[code]

Task:[task]

###Response:

(j) Prompt used for placing the goal in the grid synthesis fine-grained skills.

Place avatar+goal prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to pick the avatar's initial location and the location of the goal in order to create a task that can be solved by the given code. First, write the locations the avatar should navigate through to reach the goal. Then write the sequence of primitives that the avatar will execute when executing the code on the task that you will obtain. Finally, write down the avatars's initial location and the location of the goal under the format "avatar: x1:y1:direction, goal: x2:y2".

###Input:

Code:

[code]

Task:[task]

###Response:

(k) Prompt used for placing both the avatar and the goal in the grid synthesis fine-grained skills.

Place walls prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to place additional blocked cells in order to create a task that can be solved by the given code. First, write the locations the avatar should navigate through to reach the goal. Then write the sequence of primitives that the avatar will execute when executing the code on the task that you will obtain. Finally, write down all the locations of the blocked cells under the format "blocked: x4:y4, x5:y5, ...".

###Input:

Code:

[code]

Task:[task]

###Response:

(l) Prompt used for placing walls in the grid synthesis fine-grained skills.

Design all prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to pick the avatar's initial location in order to create a task that can be solved by the given code. First, write the locations the avatar should navigate through to reach the goal. Then write the sequence of primitives that the avatar will execute when executing the code on the task that you will obtain. Finally, write down the original configuration of the grid under the format "size: x1:y1, avatar: x2:y2:direction, goal: x3:y3, blocked: x4:y4, x5:y5, ...".

###Input:

Code:

[code]

###Response:

(m) Prompt used for designing the full grid in the grid synthesis fine-grained skills.

Figure 21: Prompts with placeholders for input data used as instructions for each type of task, without correct reasoning provided at inference. Some of the prompts mention that the model should not solve the task. This is required for making the model reason about the given instruction, and not simply write a code that would solve the grid.

H.4 Instructions with correct reasoning at inference

Figure 22 shows the prompts replacing the ones in Figures 21a and 21b for fine-tuning and inference with LLAMACT:HOC+MCQ+AUG$_{exp*}$, that receive the correct reasoning during inference.

Solution synthesis with correct reasoning prompt

###Instruction:

Use your knowledge about the primitives, conditionals and Python programming to write a code that will solve the task. To help you, the avatar will provide a sequence of locations the avatar should navigate through and the primitives that the avatar should execute to solve the task. They are guaranteed to be correct. Using this information, write the minimal Python code, including loops and conditionals, that will take the avatar to the goal, according to the sequence of primitives given by the emulator. Do not exceed the maximum number of lines of code. Do not use constructs that are not allowed. If any primitives, if statements, if-else statements, while loops or for loops are not specified in the allowed constructs, they are not allowed.

###Input:

Task: [task]

Maximum number of lines of code: [size]

Allowed constructs: [constructs]

Emulator sequence: [sequence]

Avatar actions: [primitives]

###Response:

(a) Prompt used for solution synthesis.

Multi-choice question with correct reasoning prompt

###Instruction:

[instruction] Note that wall is a synonym for blocked cells. Emulator output will be provided in order to help you. The emulator output is guaranteed to be correct. Use your domain knowledge to carefully reason about each answer option, based on the emulator output. Select the correct answer in the end, formatting it as "Thus, the correct answer is Option [your answer]".

###Input:

[input]

###Response:

(b) Prompt used for multi-choice questions.

Figure 22: Prompts with placeholders for input data used as instructions for the two target tasks, with correct reasoning provided at inference.