Program Synthesis Benchmark for Visual Programming in XLogoOnline Environment

Chao Wen
MPI-SWS
chaowen@mpi-sws.org

Jacqueline Staub
University of Trier
staub@uni-trier.de

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

Abstract

Large language and multimodal models have shown remarkable success on various benchmarks focused on specific skills such as general-purpose programming, math word problem-solving, and visual question answering. However, it is unclear how well these models perform on tasks that require a combination of these skills. In this paper, we curate a novel program synthesis benchmark based on the real-world tasks in the XLogoOnline visual programming environment. Each task requires a combination of different skills such as spatial planning, basic programming, and logical reasoning. Our evaluation shows that current state-of-the-art models like GPT-4V and Llama3-70B struggle to solve these tasks, achieving only 20% and 2.35% success rates, respectively. Next, we develop a fine-tuning pipeline to boost the performance of models by leveraging a large-scale synthetic training dataset with over 80, 000 tasks. Moreover, we showcase how emulator-driven feedback can be used to design a curriculum over training data distribution, through which a fine-tuned Llama3-8B drastically outperforms GPT-4V and Llama3-70B models. Finally, we provide an in-depth failure analysis to understand the limitations of different models. We will publicly release the benchmark for future research on program synthesis in visual programming.

1 Introduction

In recent years, large models have shown remarkable performance in various domains, such as general-purpose programming and visual question answering (Bubeck et al., 2023). For instance, in programming, numerous tools and models use large language models (LLMs) for code generation (Chen et al., 2021; GitHub, 2021) and programming feedback generation (Phung et al., 2024, 2023a,b), revolutionizing how programmers write code and how teachers instruct

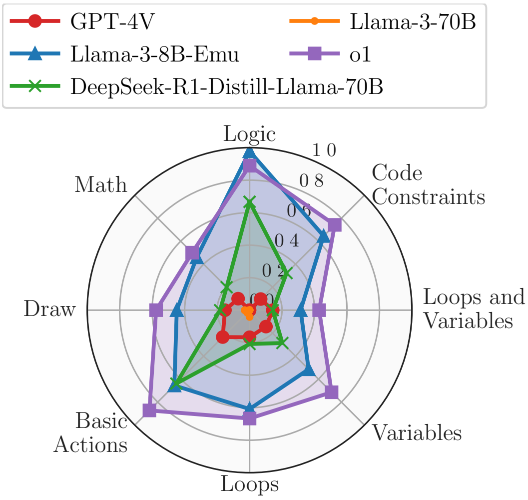


Figure 1: Success rates of different models across different skills in real-world tasks in the XLogoMiniProg benchmark.

programming (Peng et al., 2023; Denny et al., 2024). Beyond text-based tasks, the focus has expanded to multimodal models that process and generate not only text but also images, achieving significant success in domains such as visual question answering (Radford et al., 2021) and text-to-image generation (Ramesh et al., 2021).

Despite these successes, the performance of large models on tasks that require a combination of skills remains unclear. Real-world tasks often demand a blend of skills. For example, a typical task like "navigating to the kitchen to fetch ten apples" involves spatial reasoning to understand the environment and plan a path around obstacles, together with basic arithmetic to ensure that exactly ten apples are retrieved. This example illustrates the multifaceted nature of real-world tasks. While various benchmarks focus on specific skills (Chen et al., 2021; Hendrycks et al., 2021c,b; Lin et al., 2022), there is a lack of benchmarks evaluating how large models perform on tasks that require a combination of different skills.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15812–15838
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

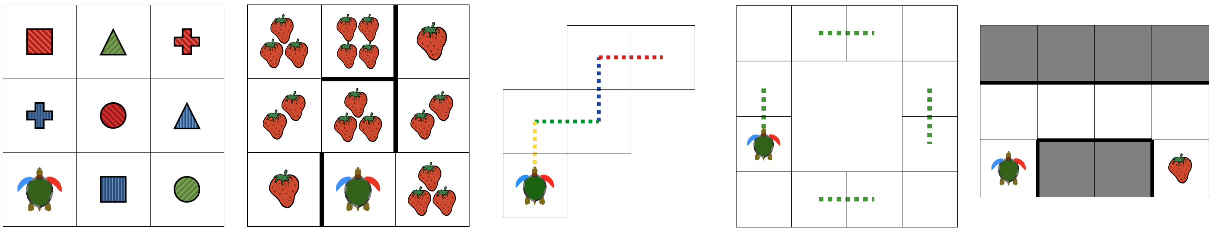
Task 28: Collect all red shapes without standing on the color green.

Task 38: Collect exactly 10 strawberries.

Task 65: Draw the picture using the colors yellow, green, blue and red.

Task 73: Draw the picture in green.

Task 87: Find the strawberry with just 6 commands.



Required Skills: Logic, Basic Actions

Required Skills: Math, Basic Actions

Required Skills: Draw, Variables, Basic Actions

Required Skills: Draw, Variables, Loops

Required Skills: Code Constraints, Loops

Solution Code:

```
def Run():
  move_forward()
  move_forward()
  move_back()
  turn_right()
  move_forward()
  move_forward()
  turn_left()
  move_forward()
```

Solution Code:

```
def Run():
  move_forward()
  turn_left()
  move_forward()
  move_back()
  turn_left()
  move_forward()
  turn_left()
  move_forward()
  turn_left()
  move_forward()
```

Solution Code:

```
def Run():
  setpc("yellow")
  move_forward()
  turn_right()
  setpc("green")
  move_forward()
  turn_left()
  setpc("blue")
  move_forward()
  turn_right()
  setpc("red")
  move_forward()
```

Solution Code:

```
def Run():
  for i in range(4):
    setpc("green")
    move_forward()
    setpc("white")
    move_forward()
    turn_right()
    move_forward()
```

Solution Code:

```
def Run():
  move_forward()
  turn_right()
  for i in range(3):
    move_forward()
  turn_right()
  move_forward()
```

Figure 2: Examples of real-world tasks, required skills, and solution codes in XLogoOnline-Mini.

To bridge this gap, we introduce XLOGOMINIPROG, a benchmark for program synthesis in the visual programming domain. Our benchmark is constructed using the Mini-level of the XLogoOnline platform (XLogoOnline, 2024), featuring 85 real-world and 1,000 synthetic visual programming tasks, each demanding a blend of diverse skills. Figure 2 illustrates examples of these tasks. Each task includes a visual grid with a turtle that needs to be directed to complete a specific goal. For example, in Task 28, the goal is to direct the turtle to collect all red shapes without stepping on the color green, requiring logical reasoning, spatial reasoning, planning, and basic programming skills. Task 38 requires additional math word problem-solving to collect 10 strawberries. These tasks provide a testbed for evaluating how large models perform on tasks that require a combination of skills, presenting a unique challenge to current large models.

We evaluate large models on these tasks and find that GPT-4V and Llama3-70B struggle, with success rates of only 20% and 2.35%, respectively. Reasoning model DeepSeek-R1-Distill-Llama3-70B performs better, achieving 44.71% success rate. However, a significant gap remains compared to human performance, as these tasks are designed for students up to 2nd grade, where humans can successfully solve almost all tasks (Staub, 2021). This indicates that current large models are not yet capable of effectively solving visual programming tasks that require a combination of skills. Figure 1 compares the performance of large models across different skill dimensions on these tasks. To improve performance, we develop a fine-tuning pipeline by leveraging a large-scale synthetic dataset containing over $80,000$ visual programming tasks. Our fine-tuned Llama3-8B model outperforms GPT-4V, Llama3-70B, and DeepSeek-R1-Distill-Llama3-70B, achieving a 54.12% success rate. Moreover, we leverage emulator feedback to design a curriculum over the training data distribution, improving performance by 6.1% over standard supervised fine-tuning.

Our contributions are as follows: First, we introduce XLOGOMINIPROG, a program synthesis benchmark based on the XLogoOnline platform to evaluate large models in visual programming, which requires a blend of skills. Second, we develop a fine-tuning pipeline that includes synthetic dataset generation and supervised fine-tuning, along with an emulator-driven fine-tuning technique that improves standard supervised fine-tuning performance by 6.1%. Third, we conduct extensive experiments to benchmark the performance of different models, providing an in-depth failure analysis and a detailed analysis of their expertise across multiple skill dimensions. We will publicly release the benchmark for future research on program synthesis in visual programming.[1]

[1] https://github.com/machine-teaching-group/acl2025-xlogominiprog

2 Related Work

Multimodal benchmarks for large models. Existing works have developed many multimodal benchmarks to evaluate the visual understanding and reasoning capabilities of multimodal models (Hendrycks et al., 2021b; Yu et al., 2024; Padlewski et al., 2024; Devlin et al., 2017). Our work falls into this broader category but focuses on program synthesis in visual programming domains that demand both visual reasoning and programming skills. Furthermore, our benchmark is built on a widely used educational programming platform, providing practical value for AI-assisted programming education.

Program synthesis benchmarks for large models. Program synthesis aims to automatically generate programs from specifications. Recently, numerous recent works have focused on training large models specifically for program synthesis (Chen et al., 2021; Rozière et al., 2023; Fried et al., 2023; Nijkamp et al., 2023). To evaluate these large models, many program synthesis benchmarks have been developed, such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021a). However, these benchmarks focus on generating code from natural language or docstrings for general programming languages such as Python (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021a). Our benchmark focuses on program synthesis in the visual programming domain. While our benchmark covers basic programming like loops and variables, it requires models to combine spatial, logical, and programming skills, posing unique challenges not addressed by these program synthesis benchmarks.

Large models for visual programming. Visual programming has been studied in various scenarios, such as task synthesis (Ahmed et al., 2020; Ghosh et al., 2022; Wen et al., 2024; Pădurean et al., 2023), program synthesis (Bunel et al., 2018; Chen et al., 2019), and student modeling (Nguyen et al., 2024). With the rise of large models, some initial works evaluate ChatGPT (OpenAI, 2023a) and GPT-4 (OpenAI, 2023b) in these scenarios, showing that large models struggle with visual programming tasks (Pădurean et al., 2023; Nguyen et al., 2024; Singla, 2023). In contrast, we provide a comprehensive benchmark that evaluates a broader range of models and skills for program synthesis in visual programming.

3 Background and Synthesis Objective

In this section, we provide background on the XLogoOnline visual programming platform and then introduce the program synthesis objective.

3.1 Background on XLogoOnline-Mini

XLogoOnline (XLogoOnline, 2024) is a visual programming platform based on the Logo programming language (Pea, 1987) and is widely used by tens of thousands of students every year (Hromkovic et al., 2017; Staub, 2021). In this work, we focus on the Mini-level (XLogoOnline-Mini). In XLogoOnline-Mini, each task includes a text description of the goal and code constraints, along with a two-dimensional visual grid. The visual grid features a turtle and various elements such as fruits, shapes, colors, lines, walls, and forbidden areas. To solve the task, one needs to write a program to direct the turtle's movement in the visual grid to achieve the specified goal. Figure 2 shows illustrative examples of tasks, the required skills, and solution codes.

Required skills for XLogoOnline-Mini. We examine the skills required for solving visual programming tasks in XLogoOnline-Mini. Specifically, the visual programming tasks in our domain cover the following skills: (i) *Logic*: Understanding underlying logical relationships specified in the goal; (ii) *Math*: Applying basic arithmetic to solve the task; (iii) *Draw*: Identifying patterns and generating the corresponding code; (iv) *Basic actions*: Moving and changing directions using only basic commands; (v) *Loops*: Utilizing loops to repeat commands multiple times; (vi) *Variables*: Using variables to set and update colors to draw lines with a specific color; (vii) *Loops and Variables*: Integrating loops with variables to solve a task; (viii) *Code Constraints*: Adhering to specific code constraints such as maximum code length.

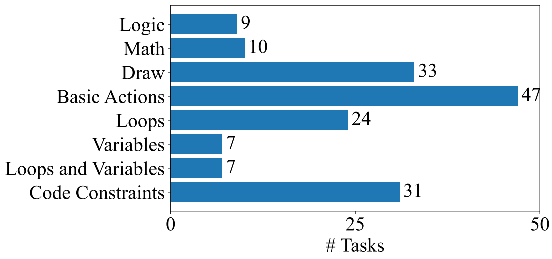
3.2 Program Synthesis Objective

Next, we formally define task and code specifications and introduce our synthesis objective.

Task specifications. In XLogoOnline-Mini, a task $\mathtt{T} := (\mathtt{G}, \mathtt{L}, \mathtt{W})$ consists of a goal $\mathtt{G}$, code constraints $\mathtt{L}$, and a visual grid world $\mathtt{W}$. The goal $\mathtt{G}$ defines the turtle's objective. The code constraints $\mathtt{L}$ specify the requirements for a solution code. There are five types of code constraints: `None` (no restrictions), `AtMost` (maximum number of commands),

Task Type	#	Code Constraints	#	Code Concepts	#	Code Length	#	Grid Size	#
`Find`	33	`None`	54	Basic Actions	47	Short (1-5)	41	$Size \leq 3$	59
`Draw`	33	`AtMost`	21	Loops	24	Medium (6-10)	29	$Size = 4$	15
`Math`	10	`Exactly`	6	Variables	7	Long (11-17)	15	$Size = 5$	4
`Logic`	9	`StartBy`	4	Loops and Variables	7			$Size = 6$	4
		`Hybrid`	0					$Size \geq 7$	3
Total	85	Total	85	Total	85	Total	85	Total	85

(a) Task distribution of BASIC dataset.



(b) Skill distribution of BASIC dataset.

```
code C := def Run() Do b

rule b := a | b; b | repeat(x) do b

action a := forward | backward | left | right | setpc(r)

color r := red | blue | green | white | black | yellow

iter x := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

(c) Code DSL.

Figure 3: Statistics of the BASIC dataset and the code DSL. (a) shows the task distribution across five dimensions within BASIC. (b) illustrates the skill distribution. To describe these skills, we extract various aspects from task type, code constraints, and code concepts as detailed in (a), and consolidated these aspects into broader categories, which we refer to as *skills*. A task may require multiple skills (see Figure 2). (c) shows the code DSL used in the XLogoOnline-Mini domain.

`Exactly` (exact number of commands), `StartBy` (initial command sequence), and `Hybrid` (combination of constraints). The visual grid world `W` is a two-dimensional visual grid featuring a turtle and various elements. We define the grid size as the maximum dimension of the grid. For example, in Figure 2 (Task 87), the goal is "Find the strawberry", the code constraint is "use just 6 commands" (`AtMost`), and the visual grid world depicts a 3×4 grid ($size = 4$) with a turtle, a strawberry, and forbidden areas marked by gray cells.

Code specifications. The code space of XLogoOnline-Mini is defined by the domain-specific language (DSL) depicted in Figure 3c. Note that while the DSL defines the formal structure and syntax, we implement it using a Python-style code representation to leverage large models' pre-trained knowledge of Python programming. A *solution code* for a given task is the code that meets the task's code constraints and achieves the specified goal when executed in the visual grid world. In Figure 2, a solution code is provided below each task.

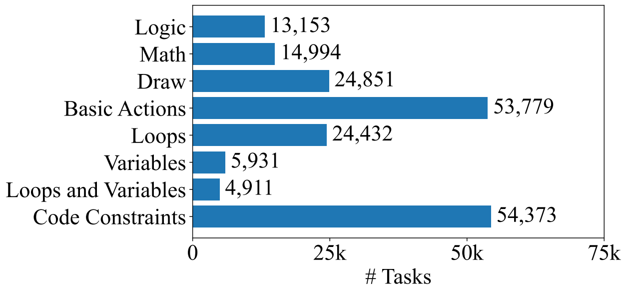
Program synthesis objective. Our objective is to develop a synthesizer function, $f : \mathtt{T} \rightarrow \mathtt{C}$, which generates a solution code `C` for a given visual programming task `T`. To evaluate f on a task `T`, we first use f to synthesize a code $\hat{\mathtt{C}}$ and then execute the synthesized code $\hat{\mathtt{C}}$ using an emulator. The emulator outputs *success* if the synthesized code $\hat{\mathtt{C}}$ successfully solves the task `T` and adheres to code constraints; otherwise, the emulator outputs *fail*. We use *success* as the main evaluation metric. Given a dataset $\mathcal{D}_{\text{eval}} = \{\mathtt{T}_i\}_{i=1}^{N}$, we calculate the success rate of f on this dataset as the overall performance. We curate a dataset BASIC of $N = 85$ real-world visual programming tasks from XLogoOnline-Mini, and we use this as one of the main datasets for evaluation. In Figures 3a and 3b, we show the overall distribution of this dataset and the number of tasks requiring specific skills, respectively.

4 Methodology for Synthetic Dataset Generation and Fine-tuning

As discussed in Section 1, existing large models such as GPT-4V and Llama3-70B struggle with visual programming tasks in XLogoOnline-Mini. To address this, we develop a two-stage fine-tuning pipeline consisting of synthetic dataset generation and supervised fine-tuning. This section details the dataset generation process and the methodology for fine-tuning large models on the synthetic dataset.

Task Type	#	Code Constraints	#	Code Concepts	#	Code Length	#	Grid Size	#
`Find`	36, 055	`None`	34, 680	Basic Actions	53, 779	Short (1-5)	20, 985	$Size \leq 3$	35, 908
`Draw`	24, 851	`AtMost`	29, 354	Loops	24, 432	Medium (6-10)	45, 682	$Size = 4$	25, 933
`Math`	14, 994	`Exactly`	16, 169	Variables	5, 931	Long (11-17)	22, 386	$Size = 5$	14, 852
`Logic`	13, 153	`StartBy`	1, 430	Loops and Variables	4, 911			$Size = 6$	8, 061
		`Hybrid`	7, 420					$Size \geq 7$	4, 299
Total	89, 053	Total	89, 053	Total	89, 053	Total	89, 053	Total	89, 053

(a) Task distribution of SIM dataset.



(b) Skill distribution of SIM dataset.

Dataset	Purpose	#
SIMTRAIN	Train	87, 053
SIMVAL	Validation	1, 000
SIMEVAL	Evaluation	1, 000
SIM		89, 053

(c) Dataset split of SIM dataset.

Figure 4: Statistics of the synthetic SIM dataset. (a) and (b) show the task distribution and the skill distribution, respectively. (c) shows the dataset split.

4.1 Synthetic Dataset Generation

Our goal is to develop a large synthetic dataset to train models (Bunel et al., 2018). To achieve this, we adopt the task synthesis techniques from (Ahmed et al., 2020; Wen et al., 2024), which were developed to automatically generate high-quality tasks in visual programming education domains. Instead of random task generation, these techniques employ symbolic execution and constraint satisfaction, enabling more controlled and systematic task synthesis, such as specifying task types, code concepts, and code lengths. However, we further adapt these techniques to generate a larger, more diverse dataset for model training (see Appendix A.2 for more details).

Dataset generation process and statistics. We use the adapted task synthesis technique to generate a synthetic dataset as follows: (i) We manually craft a solution code for each task in the BASIC dataset, resulting in a set $\{(\mathsf{T}_i, \mathsf{C}_i)\}_{i=1}^{85}$; (ii) For each $(\mathsf{T}_i, \mathsf{C}_i)$, we generate up to $1,500$ synthesized tasks and their solution codes. To ensure the quality of the dataset, we take the following processing steps: we remove any duplicate task-code pairs to maintain diversity, conduct a correctness check on the generated solution codes using the emulator, and exclude any task-code pairs present in the real-world BASIC dataset from our synthetic dataset. This last processing step guarantees that the model has not seen any tasks from the evaluation dataset during training. We ultimately produce the synthetic dataset SIM with $89,053$ task-code pairs. The statistics of this dataset are detailed in Figure 4. From this synthetic dataset, we randomly select $1,000$ samples for validation, $1,000$ samples for evaluation, and use the remaining samples for training. We use this synthetic evaluation dataset ($1,000$ samples), referred to as SIMEVAL, to complement the real-world dataset BASIC in evaluating the model's performance. We provide full details of the dataset generation process and dataset quality assessment in Appendix A.

4.2 Methodology for Fine-tuning

Supervised fine-tuning. Supervised fine-tuning is commonly used to improve large models' performance on domain tasks. In our domain, one can use the synthetic dataset SIM to fine-tune large models, where the model receives a natural language task description as input and outputs Python-style code. The model is optimized to minimize the cross-entropy loss between the predicted code and the ground truth solution code.

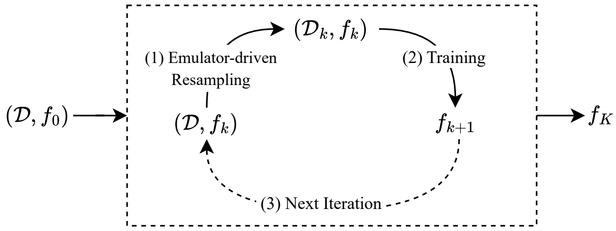
Emulator-driven feedback for fine-tuning. Standard supervised fine-tuning assigns equal weights to all training samples. However, our domain tasks require varying skills and have different difficulty levels (see Figure 4). Moreover, some skills are prerequisites for mastering advanced ones. For instance, a model typically needs to understand

You are presented with a visual programming task involving a goal, a grid, a turtle, various items (or lines). You need to write Python code that enables the turtle to accomplish the goal within the grid.

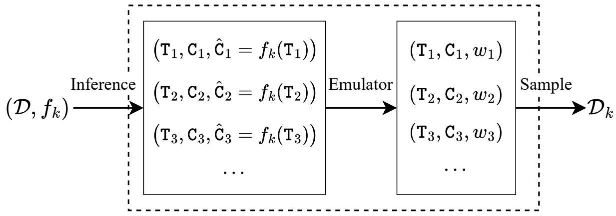
{description_of_grid_properties}
{description_of_python_functions}
Now, write a correct Python code that successfully solves the following task.

Task:
{description_of_task}
Goal:
{description_of_goal}
Correct code:

(a) Prompt template.



(b) Overview of emulator-driven fine-tuning.



(c) Emulator-driven resampling.

Figure 5: (a) shows the prompt template for fine-tuning. This prompt has several placeholders to include details for the descriptions of different aspects of the task. More details can be found in the supplementary material. (b) provides an overview of emulator-driven fine-tuning, starting with the dataset $\mathcal{D}$ and initial model f_0, and iteratively resampling and training to produce the final model f_K. (c) illustrates the resampling process in emulator-driven fine-tuning to create the dataset $\mathcal{D}_k$.

basic actions before mastering loops and variables, and it solves tasks with shorter code lengths before tackling longer ones. Thus, treating all tasks with equal importance can be suboptimal in our setting (Bengio et al., 2009). To address this, we introduce emulator-driven fine-tuning, which designs a curriculum over the training data distribution by leveraging emulator feedback. The key idea is to dynamically adjust the data distribution based on emulator evaluation, assigning higher weights to challenging tasks and progressively guiding the model from simpler to more complex problems. The overall process is shown in Figures 5b and 5c. Given an initial model f_0 and the training dataset $\mathcal{D}$, our goal is to learn a final model f_K. To achieve this, at each training epoch k, we first perform the *emulator-driven resampling* step (see Figure 5c), where model f_k infers on the training dataset $\mathcal{D}$ to obtain the predicted code $\hat{\mathtt{C}}_i$ for each task $\mathtt{T}_i$. Then, we evaluate each predicted code using an emulator and update the weight w_i for $(\mathtt{T}_i, \mathtt{C}_i)$ as follows:

$$w_i = \frac{1}{|\mathcal{D}|}\Big[1+\beta\cdot\mathbb{I}\big(\text{Emulator}(\mathtt{T}_i, \hat{\mathtt{C}}_i) = \textit{fail}\big)\Big], \quad (1)$$

where $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the predicted code fails to solve $\mathtt{T}_i$, and 0 otherwise. The hyperparameter β is adjustable, with a larger β encouraging the model to focus more on its mistakes, and $\beta = 0$ is equivalent to fine-tuning without resampling. Then, we sample from the training dataset $\mathcal{D}$ according to the categorical distribution $w'_i = w_i / \sum_{j=1}^{|\mathcal{D}|} w_j$ to obtain a resampled dataset $\mathcal{D}_k$. After resampling, we perform the *training* step, where we train model f_k on $\mathcal{D}_k$ to obtain f_{k+1}. We repeat the resampling and training steps until the model converges or reaches a predefined number of training epochs, yielding the final model f_K. To reduce computational costs, resampling can be performed at fixed intervals (set to 3 epochs in our experiments).

5 Experimental Evaluations

In this section, we present a comprehensive evaluation of large models on XLogoMiniProg. We begin by describing the experimental setup in Section 5.1, followed by the main results in Section 5.2. Next, we provide a comparative analysis in Section 5.3 and a failure analysis in Section 5.4.

5.1 Experimental Setup

Models evaluated. We evaluate four types of models: (i) *Base LLMs*, which include GPT-3.5 (OpenAI, 2023a), GPT-4 (OpenAI, 2023b), Llama2 (Touvron et al., 2023), and Llama3 (Meta, 2024) family models; (ii) *Reasoning LLMs*, which include o1 (OpenAI, 2024b) and DeepSeek-R1 (DeepSeek-AI et al., 2025) family models; (iii) *Base VLMs*, which include GPT-4V (OpenAI, 2023b), GPT-4o (OpenAI, 2024a), Llava1.5 (Liu et al., 2023a), InternVL2 (Chen et al., 2023), Qwen2-VL (Wang et al., 2024), NVLM-D (Dai et al., 2024), and Molmo (Deitke et al., 2024);

	Basic (85 tasks)			SimEval (1,000 tasks)		
	Format (%)	No-Crash (%)	Success (%)	Format (%)	No-Crash (%)	Success (%)
Base LLMs (text-only):						
GPT-3.5 (`gpt-3.5-turbo-0125`)	92.94	11.76	1.18	87.60	9.50	1.60
GPT-4 (`gpt-4-turbo-2024-04-09`)	**95.29**	**38.83**	**12.94**	**97.40**	**16.80**	**5.30**
Llama3-8B	48.24	5.88	0.00	40.90	2.80	0.60
Llama3-70B	67.06	8.24	2.35	15.50	1.20	0.30
Llama2-7B	27.06	5.88	0.00	21.90	2.90	0.40
Llama2-13B	60.00	7.06	0.00	54.40	3.50	0.40
Llama2-70B	28.24	7.06	0.00	38.30	1.10	0.10
Base VLMs (text + vision):						
GPT-4o (`gpt-4o-2024-11-20`)	**100.00**	36.47	**22.35**	**99.10**	**18.30**	**5.90**
GPT-4V (`gpt-4-turbo-2024-04-09`)	96.47	**47.06**	20.00	95.50	18.10	5.50
Llava1.5-7B	10.59	1.18	0.00	3.20	0.00	0.00
Llava1.5-13B	10.59	2.35	0.00	9.00	2.10	0.00
InternVL2-8B	0.00	0.00	0.00	56.90	3.80	0.00
InternVL2-Llama3-76B	77.65	31.76	9.41	40.50	6.10	1.50
Qwen2VL-7B	43.53	9.41	0.00	14.30	2.10	0.20
Qwen2VL-72B	28.24	11.76	0.00	36.50	4.40	0.40
NVLM-D-72B	61.18	8.24	1.18	67.40	8.30	2.00
Molmo-7B-D	75.29	8.24	0.00	66.00	7.70	0.60
Molmo-72B	4.71	1.18	1.18	6.40	0.70	0.40
Reasoning LLMs (text-only):						
o1 (`o1-2024-11-12`)	**100.00**	**97.65**	**76.47**	**99.08**	**47.38**	23.18
DeepSeek-R1-Distill-Llama-8B	38.82	21.18	12.94	28.70	9.00	5.10
DeepSeek-R1-Distill-Llama-70B	76.47	48.24	44.71	67.00	41.00	**32.90**
Fine-tuned LLMs and VLMs:						
Llava1.5-13B-Uni	68.24 ± 18.48	19.53 ± 14.98	11.99 ± 10.55	56.18 ± 15.68	13.64 ± 11.36	10.68 ± 10.23
Llama2-7B-Uni	99.76 ± 0.24	65.88 ± 1.05	45.65 ± 0.86	99.98 ± 0.02	62.64 ± 0.33	53.04 ± 0.20
Llama2-7B-Emu	$\mathbf{100} \pm 0.00$	69.41 ± 1.97	51.53 ± 0.44	99.96 ± 0.02	68.70 ± 0.49	60.10 ± 0.69
Llama3-8B-Uni	99.53 ± 0.29	$\mathbf{73.65} \pm 0.80$	54.12 ± 1.78	99.96 ± 0.04	71.26 ± 1.01	62.72 ± 1.17
Llama3-8B-Emu	99.76 ± 0.24	71.53 ± 0.78	$\mathbf{60.23} \pm 1.01$	$\mathbf{100} \pm 0.00$	$\mathbf{74.92} \pm 0.60$	$\mathbf{66.92} \pm 0.65$

Figure 6: Performance comparison of models on two evaluation datasets. **Bold** values indicate the highest performance in each column within the group. Fine-tuned models are trained using 5 different random seeds and we report the mean and standard error of the performance.

(iv) *Fine-tuned LLMs and VLMs*, which include fine-tuned Llava1.5, Llama2, and Llama3 models. We use "Uni" and "Emu" as suffixes for models fine-tuned via standard supervised learning and emulator-driven methods, respectively. Detailed versions of the evaluated models are provided in Appendix Figure 13. Additional evaluation and fine-tuning details are in Appendix B.

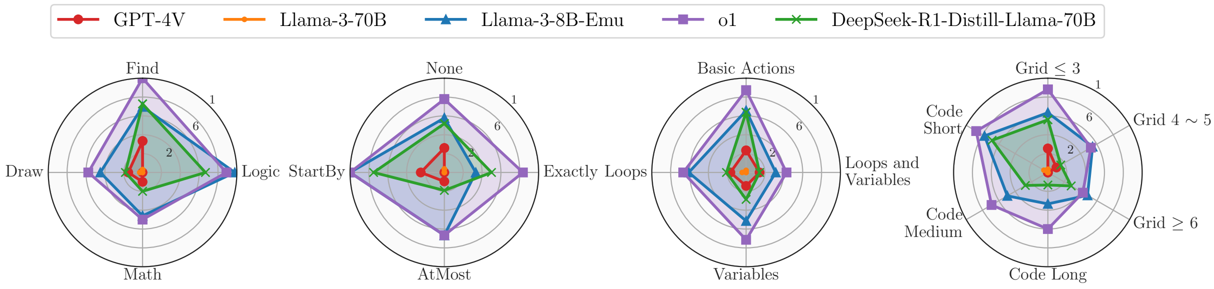
Evaluation procedure and metrics. We evaluate models using two datasets: Basic and the synthetic dataset SimEval (see Figure 4c). For each task, we convert its JSON format into a natural language description using a fixed prompt template (see Figure 5a). [2] For multimodal models (e.g., GPT-4V, Llava1.5), we additionally provide the task image as input. Then, we prompt the model to generate Python code and extract only the code portion from the model output. Finally, we evaluate the extracted code using an emulator. We use *success* as the main metric (see Section 3.2), and also consider two additional metrics: (i) *Format*, which evaluates whether the model's output adheres to the desired code format, and (ii) *No-Crash*, which evaluates whether the code runs without crashing, such as hitting walls, entering forbidden areas, or exceeding grid boundaries.

[2]The prompt template does not include few-shot examples or advanced prompting strategies. The evaluation of different prompting strategies is provided in Appendix C.1.

5.2 Main Results

As shown in Figure 6, most of the evaluated models struggle significantly on our benchmark. The best performance of base models on Basic and SimEval is 76.47% and 32.90%, respectively.

Vision capabilities provide limited benefits. Vision-enabled models, such as GPT-4V, show modest improvements over their text-only counterparts (GPT-4V's 20% vs. GPT-4's 12.94% success rate on Basic), yet other VLMs continue to struggle significantly. This suggests that while vision capabilities offer some benefits, they are not the determining factor in solving our benchmark tasks. This is likely because, in our setting, the textual



(a) Task Type (b) Code Constraints (c) Code Concepts (d) Code Length & Grid Size

Figure 7: Comparative analysis of models' capabilities across different dimensions on BASIC. Each chart highlights the models' capabilities in different aspects within a dimension. Note that code length and grid size are combined in the same chart, as both indicate the difficulty levels of the tasks.

input can already sufficiently capture all necessary visual information required for solving the tasks. Therefore, although vision capabilities offer some incremental advantages, their overall impact on performance in our benchmark remains limited.

Reasoning capabilities are crucial. Reasoning models outperform all non-reasoning base models, with o1 achieving a 76.47% success rate on BASIC, compared to 22.35% for the best non-reasoning model.[3] This large gap suggests that reasoning is essential for solving visual programming tasks.

Fine-tuning improves performance. Fine-tuning significantly improves model performance, especially for Llama models. Llama3-8B improves from 0% to 54.12% with standard fine-tuning (Uni), and further to 60.23% with emulator-driven resampling (Emu).

5.3 Comparative Analysis Across Dimensions

We evaluate model performance across various dimensions to identify strengths and weaknesses. We automatically categorize each task-code pair along different dimensions (e.g., task type) and assess model capability in specific aspects (e.g., `Math` in the task type dimension) by calculating success rates for all relevant tasks. Figure 7 presents a comparative analysis of five representative models across distinct dimensions on BASIC. Overall, o1 and Llama3-8B-Emu consistently outperform other models across almost all dimensions. However, o1 and Llama3-8B-Emu particularly struggle with draw tasks and complex code concepts (e.g., Loops and Variables), with performance degrading significantly as task complexity increases through larger grid sizes and longer code lengths.

5.4 Failure Analysis

In this section, we perform failure analysis to understand the limitations of different models. We analyze model failures through two types of failure analysis: (i) *explanation-based failure analysis*, which examines model-generated explanations to identify failure reasons, and (ii) *perturbation-based failure analysis*, which evaluates performance on simplified, perturbed tasks.

Explanation-based failure analysis. We present a failure analysis by examining output codes and explanations from GPT-4V, Llama3-70B, and DeepSeek-R1-Distill-Llama-70B. Fine-tuned models are excluded from this analysis as they generate code without explanations. To conduct the failure analysis, we first identify common failure types. Then, we systematically analyze the explanations and output codes and manually annotate the most significant reason for each failure. In cases where multiple failure reasons are identified, we attribute the failure to the most significant reason. The analysis results are shown in Figure 8. Our findings show that GPT-4V and Llama3-70B struggle most with spatial reasoning, which is caused by misunderstandings of coordinates or directions after movements or turns. DeepSeek-R1-Distill-Llama-70B often fails due to recursive reasoning, where excessive reasoning produces lengthy outputs without arriving at a final answer.

Perturbation-based failure analysis. We provide failure analysis by perturbing tasks. We consider GPT-4V, Llama3-70B, and Llama3-8B-Uni. We first select 10 tasks from the BASIC dataset that the three models consistently fail to solve. For

[3]In our evaluation, o1 failed to generate responses for 25 tasks in SIMEVAL, so we report its performance on the remaining 975 tasks.

	Repetition	Format	Goal	Code Constraints	Grid Constraints	Spatial Reasoning	Recursive Reasoning	Hallucination	Success
GPT-4V	0.00	3.53	11.76	7.06	11.76	**42.35**	0.00	3.53	20.00
Llama3-70B	34.12	1.18	5.88	3.53	8.24	**44.71**	0.00	0.00	2.35
DeepSeek-R1-Distill-Llama-70B	0.00	0.00	1.18	1.18	3.53	23.53	**25.89**	0.00	44.71

Figure 8: Percentage (%) of different failure types by analyzing model outputs on the BASIC dataset. **Bold** values highlight the most common failure type for each model. See Appendix B.1 for detailed definitions of failure types.

	T	T_{A}	T_{B}	T_{C}	$\mathsf{T}_{\mathsf{A,B}}$	$\mathsf{T}_{\mathsf{B,C}}$	$\mathsf{T}_{\mathsf{A,C}}$	$\mathsf{T}_{\mathsf{A,B,C}}$
GPT-4V	0	30.0	30.0	**50.0**	50.0	50.0	**60.0**	**60.0**
Llama3-70B	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Llama3-8B-Uni	0	0.0	**10.0**	0.0	**20.0**	**20.0**	0.0	**30.0**

Figure 9: Success rates (%) across 80 perturbed tasks (10 tasks $\times$ 8 perturbations). Perturbations are grouped by the number of components removed. The perturbations include removing code constraints (T_{A}), removing grid constraints (T_{B}), simplifying spatial relationships (T_{C}), and combinations of these perturbations ($\mathsf{T}_{\mathsf{A,B}}$, $\mathsf{T}_{\mathsf{B,C}}$, $\mathsf{T}_{\mathsf{A,C}}$, and $\mathsf{T}_{\mathsf{A,B,C}}$). **Bold** values indicate the highest success rate for each model within each perturbation group.

each task T, we consider perturbations including removing code constraints (T_{A}), removing grid constraints (T_{B}), simplifying spatial relationships (T_{C}), and combinations of these perturbations ($\mathsf{T}_{\mathsf{A,B}}$, $\mathsf{T}_{\mathsf{B,C}}$, $\mathsf{T}_{\mathsf{A,C}}$, and $\mathsf{T}_{\mathsf{A,B,C}}$). Tasks lacking certain components remain unchanged. As shown in Figure 9, GPT-4V struggles with spatial relationships. Simplifying these increases its success rate from 0% to 50% (see columns T and T_{C}). Conversely, Llama3-8B-Uni struggles with grid constraints. Removing these boosts its success rate to 10% (see column T_{B}), while removing code constraints and spatial relationships has no effect. [4]

6 Concluding Discussions

In this paper, we introduced XLOGOMINIPROG, a visual programming benchmark designed to evaluate the program synthesis capabilities of large models on visual programming tasks using the XLogoOnline environment. We found that large models struggle with visual programming tasks that require a combination of skills, even though our benchmark tasks require only basic programming skills. To improve performance, we developed a fine-tuning pipeline that involves synthetic dataset generation followed by supervised fine-tuning. This pipeline enabled the Llama3-8B model to achieve a success rate of 54.12% on the benchmark tasks. Additionally, we demonstrated that leveraging emulator-driven feedback can further enhance standard fine-tuning performance by approximately 6% in both the Llama3-8B and Llama2-7B models. Our failure analysis revealed that different models struggle with distinct issues: DeepSeek-R1-Distill-Llama-70B struggles with recursive reasoning, GPT-4V and Llama3-70B with spatial reasoning, and the fine-tuned Llama3-8B-Uni with grid constraints.

Limitations

We discuss some limitations of our work and propose ideas for addressing them in the future. First, our benchmark focuses on basic programming skills, and future work could extend it to include more complex programming tasks. This could involve tasks that require more advanced programming concepts, such as conditionals and functions. Second, our emulator-driven fine-tuning provides the model with only binary feedback on the correctness of the predicted code. In the future, it would be interesting to provide more detailed feedback, such as identifying specific errors in the generated code and then using this more informative feedback to guide the fine-tuning process.

Acknowledgments

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.

[4] Interestingly, Llama3-8B-Uni performs worse than GPT-4V in our failure analysis. This is likely because we selected tasks that all models initially failed on, which already indicates Llama3-8B-Uni's limitations with these examples. When perturbed, these tasks further diverge from the training distribution. GPT-4V, with its stronger generalization capabilities, remains robust to these distribution shifts and performs better.

References

Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. 2020. Synthesizing Tasks for Block-based Programming. In *NeurIPS*.

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. 2021. Program Synthesis with Large Language Models. *CoRR*, abs/2108.07732.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *ICML*.

Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712.

Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *ICLR*.

Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374.

Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *ICLR*.

Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, Bin Li, Ping Luo, Tong Lu, Yu Qiao, and Jifeng Dai. 2023. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. *CoRR*, abs/2312.14238.

Wenliang Dai, Nayeon Lee, Boxin Wang, Zhuoling Yang, Zihan Liu, Jon Barker, Tuomas Rintamaki, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2024. NVLM: Open Frontier-Class Multimodal LLMs. *CoRR*, abs/2409.11402.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

DeepSeek-AI et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *CoRR*, abs/2501.12948.

Matt Deitke et al. 2024. Molmo and PixMo: Open Weights and Open Data for State-of-the-Art Multimodal Models. *CoRR*, abs/2409.17146.

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

Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew J. Hausknecht, and Pushmeet Kohli. 2017. Neural Program Meta-Induction. In *NIPS*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *ICLR*.

Ahana Ghosh, Sebastian Tschiatschek, Sam Devlin, and Adish Singla. 2022. Adaptive Scaffolding in Block-Based Programming via Synthesizing New Tasks as Pop Quizzes. In *AIED*.

GitHub. 2021. Github Copilot. https://github.com/features/copilot.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021a. Measuring Coding Challenge Competence With APPS. In *NeurIPS Track on Datasets and Benchmarks*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021b. Measuring Massive Multitask Language Understanding. In *ICLR*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021c. Measuring Mathematical Problem Solving With the MATH Dataset. In *NeurIPS Track on Datasets and Benchmarks*.

Juraj Hromkovic, Giovanni Serafini, and Jacqueline Staub. 2017. XLogoOnline: A Single-Page, Browser-Based Programming Environment for Schools Aiming at Reducing Cognitive Load on Pupils. In *ISSEP*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SIGOPS*.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring How Models Mimic Human Falsehoods. In *ACL*.

Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. 2023a. Improved baselines with visual instruction tuning. *CoRR*, abs/2310.03744.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *NeurIPS*.

Meta. 2024. Llama 3. https://llama.meta.com/llama3/.

Manh Hung Nguyen, Sebastian Tschiatschek, and Adish Singla. 2024. Large Language Models for In-Context Student Modeling: Synthesizing Student's Behavior in Visual Programming from One-Shot Observation. In *EDM*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *ICLR*.

OpenAI. 2023a. ChatGPT. https://openai.com/blog/chatgpt.

OpenAI. 2023b. GPT-4. https://openai.com/index/gpt-4/.

OpenAI. 2024a. GPT-4o. https://openai.com/index/hello-gpt-4o/.

OpenAI. 2024b. o1. https://openai.com/o1/.

Piotr Padlewski et al. 2024. Vibe-Eval: A hard evaluation suite for measuring progress of multimodal language models. *CoRR*, abs/2405.02287.

Victor-Alexandru Pădurean, Georgios Tzannetos, and Adish Singla. 2023. Neural Task Synthesis for Visual Programming. *Transactions on Machine Learning Research*.

Roy D Pea. 1987. Logo Programming and Problem Solving.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from Github Copilot. *CoRR*, abs/2302.06590.

Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023a. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. In *EDM*.

Tung Phung, Victor-Alexandru Padurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023b. Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors. In *ICER V.2*.

Tung Phung, Victor-Alexandru Padurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. 2024. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation. In *LAK*.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *ICML*.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. In *ICML*.

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

Adish Singla. 2023. Evaluating ChatGPT and GPT-4 for Visual Programming. In *ICER V.2*.

Jacqueline Staub. 2021. Logo Environments in the Focus of Time. *Bulletin of EATCS*.

Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR*, abs/2307.09288.

Peng Wang et al. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. *CoRR*, abs/2409.12191.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.

Chao Wen, Ahana Ghosh, Jacqueline Staub, and Adish Singla. 2024. Task Synthesis for Elementary Visual Programming in XLogoOnline Environment. In *AIED Track on Late Breaking Results*.

XLogoOnline. 2024. XLogoOnline Platform. https://xlogo.inf.ethz.ch/.

Weihao Yu, Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Zicheng Liu, Xinchao Wang, and Lijuan Wang. 2024. MM-Vet: Evaluating Large Multimodal Models for Integrated Capabilities. In *ICML*.

A More Details About the Datasets

We provide more details about the BASIC dataset and the synthetic SIM dataset.

A.1 Details of the BASIC Dataset

The real-world visual programming tasks in the BASIC dataset are curated from the Mini level of the XLogoOnline platform. These real-world programming tasks can be accessed and viewed at https://xlogo.inf.ethz.ch/. Figure 10 shows screenshots of the platform.

A.2 Details of the Synthetic Dataset Generation

In this section, we provide more details about the generation process of the synthetic dataset SIM.

We use the adapted task synthesis technique (Ahmed et al., 2020; Wen et al., 2024) to generate a synthetic dataset. The key idea is to take a reference task and its solution code as input, and then apply symbolic execution and constraint satisfaction techniques to systematically enumerate all possible task-code outputs. The details are described as follows.

First, we manually craft a solution code for each of the $N = 85$ tasks in the BASIC dataset, resulting in a set $\{(\mathtt{T}_i, \mathtt{C}_i)\}_{i=1}^{N}$. However, our objective is to generate a large and diverse set of tasks to train large models. To achieve this, we specify an additional parameter, the difficulty level $\mathtt{D}$. This parameter enables us to generate tasks with varying levels of difficulty by specifying the desired code length, number of code constraints, and goals relative to the reference input, thereby enhancing the diversity of the dataset. The parameters are detailed as follows:

- `Easy`: The code length and number of code constraints remain the same as in the reference code and code constraints, and the goal remains unchanged.
- `Medium`: The code length is increased by 1 or 2 additional commands compared to the reference code, while the number of code constraints and the goal remain the same as in the reference task $\mathtt{T}$.
- `Hard`: The code length is increased by up to 2 additional commands, one more code constraint is added compared to the reference code constraints, and the goal may be modified.

Note that the difficulty levels mentioned above indicate the relative difficulty of the generated tasks compared to the reference task, not the absolute difficulty of the tasks.

Given the reference input $(\mathtt{T}, \mathtt{C}, \mathtt{D})$, we begin by enumerating all possible codes, code constraints, and goals that meet the specified difficulty levels. To achieve this, we first create templates for the code, constraints, and goals, each containing placeholders. These placeholders are then populated with specific values using an SMT-based constraint solver (de Moura and Bjørner, 2008). This process allows us to generate all possible combinations of code, constraints, and goals that align with the desired difficulty levels.

Next, we generate task-code pairs by combining the previously generated code, code constraints, and goals with corresponding grid worlds. To generate these grid worlds, we symbolically execute the previously generated code within an empty grid, constructing elements like walls and target items to ensure the grid can be successfully solved by the code. After the grid world is constructed, it is combined with the corresponding code, code constraints, and goal to form a task-code pair.

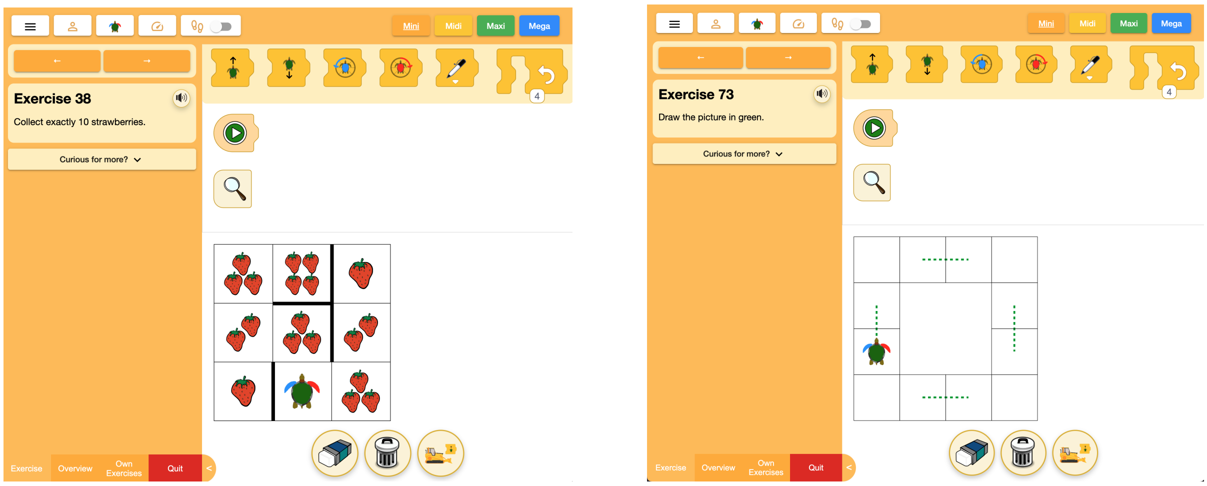
In implementation, we generate up to 3,000 tasks for each combination of code, code constraints, and goals. Subsequently, we sample 500 tasks from the pool of all generated tasks for each $(\mathtt{T}, \mathtt{C}, \mathtt{D})$, resulting in up to 500 tasks $\times$ 3 difficulty levels $=$ $1,500$ tasks for each reference input $(\mathtt{T}, \mathtt{C})$. This process is repeated for all reference inputs in the dataset, resulting in a total of up to $85 \times 1,500 = 127,500$ tasks. Finally, we apply the processing steps described in the main paper to generate the synthetic dataset, resulting in the final dataset, SIM, containing $89,053$ tasks and solution codes.

To run the adapted task synthesis technique, we use a 12-core, 3 GHz Intel Xeon E7-8857 CPU, with parallelization across 8 cores under a 64-bit Debian operating system.

A.3 Quality of the Datasets

The quality of the datasets is crucial for the success of the models trained on them. Therefore, we provide more details about the quality of the datasets. We mainly use the following two datasets for evaluation:

1. BASIC dataset (85 samples): This dataset was derived from the visual programming platform XLogoOnline. The tasks included in this plat-

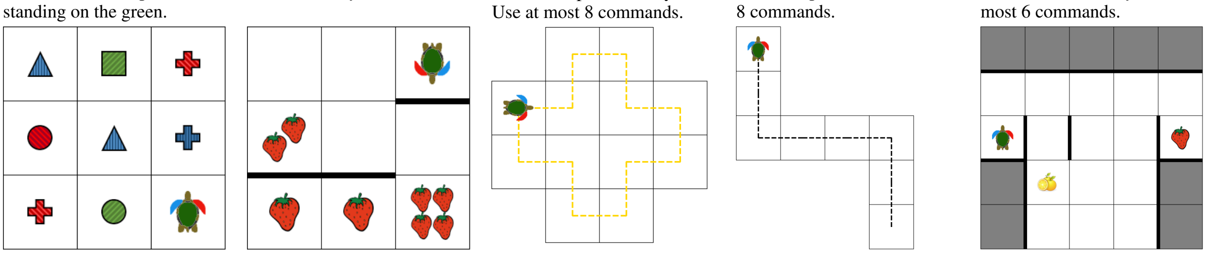


(a) Task 38

(b) Task 73

Figure 10: Example tasks from the XLogoOnline platform. Students need to drag and drop different blocks to solve the tasks.

Collect all red shapes without standing on the green.	Collect exactly 5 strawberries.	Draw the picture in yellow. Use at most 8 commands.	Draw the picture. Use at most 8 commands.	Find the strawberry. Use at most 6 commands.



Solution Code:

```
def Run():
  move_forward()
  move_forward()
  move_back()
  turn_left()
  move_forward()
  move_forward()
  turn_left()
  move_forward()
```

Solution Code:

```
def Run():
  turn_right()
  move_forward()
  turn_left()
  move_forward()
  turn_left()
  move_forward()
  turn_right()
  move_forward()
  turn_right()
  move_forward()
```

Solution Code:

```
def Run():
  setpc("yellow")
  for i in range(4):
    move_forward()
    turn_left()
    move_forward()
    turn_right()
    move_back()
    turn_left()
```

Solution Code:

```
def Run():
  for i in range(2):
    move_back()
  turn_right()
  for i in range(3):
    move_forward()
  turn_right()
  for i in range(2):
    move_forward()
```

Solution Code:

```
def Run():
  move_forward()
  turn_right()
  for i in range(4):
    move_forward()
  turn_right()
  move_forward()
```

Figure 11: Examples of synthetic tasks and their corresponding solution codes in SIMEVAL. Note that while the synthesized solution codes are correct, they may not use the minimum number of commands.

form were meticulously crafted by experts and have been used by tens of thousands of students every year (Hromkovic et al., 2017; Staub, 2021). Given this extensive use and expert involvement, the quality of the tasks in this dataset is guaranteed.

2. SIMEVAL dataset (1000 samples): This dataset was synthetically generated. However, we ensure data quality by implementing the following checks: (i) we have removed any duplicate task-code pairs; (ii) we have conducted a correctness check on the generated solution codes using the emulator; and (iii) we have excluded any task-code pairs present in the BASIC dataset from this synthetic dataset. In Figure 11, we show examples of the tasks in this dataset.

To further demonstrate the quality of our datasets, we conduct a quality annotation for both datasets. Specifically, we annotate the quality of all 85 samples in the BASIC dataset and randomly sample 5% of tasks from the SIMEVAL dataset for annotation. The following rubrics are used to evaluate each (task, code) pair:

1. *Visual appeal*
    - 0: Poor - The visual grid is highly unappealing.

	Visual Appeal	Grid Elements Utility	Code Quality	Overall Quality
BASIC	1.00	1.00	1.00	1.00
SIMEVAL	0.97	0.94	0.89	0.84

Figure 12: Quality annotation results for BASIC and SIMEVAL datasets. For BASIC, we annotate all 85 samples, while for SIMEVAL, we randomly sample 5% of the dataset for annotation.

- 0.5: Acceptable - The visual grid is moderately appealing.
- 1: Excellent - The visual grid is highly appealing.

2. *Grid elements utility*
   - 0: Poor - The distractors are neither useful nor reasonably positioned.
   - 0.5: Acceptable - Some distractors are useful, while others lack utility.
   - 1: Excellent - Most, if not all, distractors are strategically useful and sensibly placed.
3. *Code quality*
   - 0: Poor - The code is of poor quality, unable to solve the task, or violates code constraints.
   - 0.5: Acceptable - The code can solve the task but contains some unnecessary commands.
   - 1: Excellent - The code solves the task, meets code constraints, and has no redundant commands.
4. *Overall quality*: Calculated as the minimum score across visual appeal, grid elements utility, and code quality.

The results in Figure 12 demonstrate that the overall quality of the BASIC dataset is excellent. The SIMEVAL dataset, with an overall quality score of 0.84, exceeds the acceptable threshold (score = 0.5) and approaches the level of excellence (score = 1.0). Additionally, during the quality annotation, we do not find any (task, code) pair where the task is unsolvable or the code fails to successfully solve the task.

B More Details of the Failure Analysis, Fine-tuning, and Evaluation

In this section, we provide more details about the failure analysis, fine-tuning, and evaluation.

B.1 Details of Failure Analysis

We provide details of the explanation-based failure analysis. To conduct the explanation-based failure analysis, we first identify common failure types. In cases where multiple failure reasons are identified, we attribute the failure to the most significant cause. These failure types are defined as follows:

- *Repetition*: generating the same code sequences repeatedly;
- *Format*: producing code with incorrect formatting, including the use of disallowed commands;
- *Goal*: misinterpreting the goal or attempting to devise a tricky approach to achieve the goal;
- *Code constraints*: failing to adhere to specified code constraints while solving the task;
- *Grid constraints*: attempting to solve the task while ignoring walls, forbidden cells, or grid boundaries;
- *Spatial reasoning*: misunderstanding coordinates or directions after movements or turns;
- *Recursive reasoning*: failing to arrive at a final answer due to excessive or circular reasoning, leading to lengthy responses that exceed maximum token generation limits;
- *Hallucination*: generating non-existent items or code commands.

B.2 Details of the Evaluation

Versions of evaluated models. The versions of the evaluated models are provided in Figure 13.

Details of evaluation. All models are queried with a temperature of 0, except for the DeepSeek-R1 family models, which are queried with a temperature of 0.6. To evaluate GPT family models, we use the OpenAI API with a temperature of 0. For base LLMs, base VLMs, and fine-tuned models, we use the vLLM (Kwon et al., 2023) inference engine with 2 A100 GPUs, using a temperature of 0 and `max_num_seqs` of 2. We find that a smaller `max_num_seqs` value slows down inference speed but improves performance. Therefore, we choose a `max_num_seqs` value of 2 to balance performance and speed for inference. For the DeepSeek-R1 family models, we use the vLLM inference engine with 4 A100 GPUs for DeepSeek-R1-Distill-Llama-8B and 8 A100 GPUs for DeepSeek-R1-Llama-70B. We set a temperature of 0.6 and `max_num_seqs` to 2, with a maximum of 8192 tokens to enable extra reasoning tokens. For the o1 model, due to high cost and budget constraints, we set the maximum token generation limit (`max_completion_tokens`) to 4096 and the `reasoning_effort` to `Medium` when querying. After inference, we use the emulator to evaluate the models' success rates over the evaluation datasets.

B.3 Details of Fine-tuning

Details of fine-tuning Llama family models. For Llama family models, we choose non-instruction-tuned versions for fine-tuning because the base models will be fine-tuned to generate code, without requiring instruction-following capabilities. We use LoRA for parameter-efficient fine-tuning (Hu et al., 2022). To find the best LoRA rank and scaling factor, we experimented with ranks of 8, 16, 32, and 64, using a scaling factor α four times the rank in each case. We found that ranks of 32 and 64 provide the best performance. Consequently, we use a rank of 32 and a scaling factor of 128 for all fine-tuning experiments. Fine-tuning is performed with a batch size of 4 and a learning rate of 1×10^{-4}. All fine-tuning experiments are conducted on an internal cluster using 4 A100 GPUs. Each epoch of fine-tuning for the Llama3-8B and Llama2-7B models takes approximately 3.75 hours. In our experiments, all fine-tuned Llama models are trained for 8 epochs, as we observed that the validation dataset loss stabilizes around epoch 8. We train all fine-tuned Llama models using 5 different random seeds.

Details of fine-tuning Llava family model. We perform standard supervised fine-tuning on Llava1.5-13B (Liu et al., 2023a). To do this, we follow the default fine-tuning setup and code provided by the authors. Specifically, we use LoRA with a rank of 128 and a scaling factor of 256 for fine-tuning Llava1.5-13B. During fine-tuning, we use a batch size of 16, a learning rate of 2×10^{-4}, and a maximum sequence length of 2048. We fine-tune the Llava model for 3 epochs on the 87k training dataset using 5 different random seeds, utilizing 4 A100 GPUs.

Model	Version
Base LLMs (text-only):	
GPT-3.5	gpt-3.5-turbo-0125 (OpenAI, 2023a)
GPT-4	gpt-4-turbo-2024-04-09 (OpenAI, 2023b)
Llama2-7B	meta-llama/Llama-2-7b-chat (Touvron et al., 2023)
Llama2-13B	meta-llama/Llama-2-13b-chat (Touvron et al., 2023)
Llama2-70B	meta-llama/Llama-2-70b-chat (Touvron et al., 2023)
Llama3-8B	meta-llama/Meta-Llama-3-8B-Instruct (Meta, 2024)
Llama3-70B	meta-llama/Meta-Llama-3-70B-Instruct (Meta, 2024)
Base VLMs (text + vision):	
GPT-4o	gpt-4o-2024-11-20 (OpenAI, 2024a)
GPT-4V	gpt-4-turbo-2024-04-09 (OpenAI, 2023b)
Llava1.5-7B	liuhaotian/llava-v1.5-7b (Liu et al., 2023a)
Llava1.5-13B	liuhaotian/llava-v1.5-13b (Liu et al., 2023a)
InternVL2-8B	OpenGVLab/InternVL2-8B (Chen et al., 2023)
InternVL2-76B-Llama3-76B	OpenGVLab/InternVL2-Llama3-76B (Chen et al., 2023)
Qwen2-VL-7B	Qwen/Qwen2-VL-7B-Instruct (Wang et al., 2024)
Qwen2-VL-72B	Qwen/Qwen2-VL-72B-Instruct (Wang et al., 2024)
NVLM-D	nvidia/NVLM-D-72B (Dai et al., 2024)
Molmo-72B	allenai/Molmo-72B-0924 (Deitke et al., 2024)
Molmo-7B-D	allenai/Molmo-7B-D-0924 (Deitke et al., 2024)
Reasoning LLMs (text-only):	
o1	o1-2024-12-17 (OpenAI, 2024b)
DeepSeek-R1-Distill-Llama-8B	DeepSeek/DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI et al., 2025)
DeepSeek-R1-Distill-Llama-70B	DeepSeek/DeepSeek-R1-Distill-Llama-70B (DeepSeek-AI et al., 2025)
Fine-tuned LLMs and VLMs:	
Llava1.5-13B-Uni	liuhaotian/llava-v1.5-13b (supervised fine-tuning)
Llama2-7B-Uni	meta-llama/Llama-2-7b (supervised fine-tuning)
Llama2-7B-Emu	meta-llama/Llama-2-7b (emulator-driven fine-tuning)
Llama3-8B-Uni	meta-llama/Meta-Llama-3-8b (supervised fine-tuning)
Llama3-8B-Emu	meta-llama/Meta-Llama-3-8b (emulator-driven fine-tuning)

Figure 13: Evaluated models and their versions.

Details of emulator-driven fine-tuning. For emulator-driven fine-tuning, we use the same hyperparameters and setup as standard fine-tuning, with the exception of resampling every 3 epochs. Specifically, we resample the training dataset based on the emulator's evaluation results every 3 epochs. To save time and resources, we start from the checkpoint of the fine-tuned models without resampling at epoch 3. We then reuse this checkpoint to continue fine-tuning for 5 additional epochs using emulator-driven resampling, resulting in a total of 8 epochs. Emulator-driven resampling requires calculating a weight for each training sample, which involves inference over the entire training dataset. For inference, we use the vLLM inference engine (Kwon et al., 2023) with `max_num_seqs` of 8, batch size of 2, and temperature of 0. In this setting, a single iteration of inference and resampling on the 87k training dataset takes approximately 8 hours. After inference, we use the emulator to evaluate the correctness of the model's predicted code. Based on this evaluation, we calculate the weight for each training sample using a value of $\beta = 1$.

C Additional Experiments and Results

In this section, we present additional experiments and results.

C.1 Influence of Prompting Strategies

	Vanilla	3-shot	3-shot + CoT
GPT-4	12.94	10.59	18.82
GPT-4V	20	14.12	15.29

Figure 14: Success rates (%) of GPT-4 and GPT-4V with different prompting strategies on the BASIC dataset.

We conduct experiments on different prompting strategies to investigate their effectiveness in our benchmark. We consider the following prompting strategies: (i) *Vanilla* is the prompt without any additional examples or chain-of-thoughts; (ii) *3-shot* is the prompt with 3-shot examples (Brown et al., 2020); (iii) *3-shot + CoT* is the prompt with the 3-shot examples and a step-by-step chain-of-thought (CoT) for each example (Wei et al., 2022). Note that the 3-shot examples are manually designed to ensure they cover most skills, including math, logic, drawing, basic actions, variables, loops, and code constraints. These same 3-shot examples are used to prompt all tasks for both *3-shot* and *3-shot + CoT* prompting.

The results are shown in Figure 14. We observe that *3-shot* prompting by itself is not notably effective. However, when combined with CoT, it leads to performance improvements, though these gains are limited. We speculate that this is due to the nature of our visual programming tasks, which require long-term path planning, an understanding of spatial relationships, and accurate prediction of the consequences of actions. These elements are typically absent from the training data, making it difficult for the model to leverage in-context learning to solve unfamiliar visual programming tasks.

C.2 Influence of Task Representations

In this section, we investigate the influence of natural language and ASCII representations on model performance. For visual programming tasks, the 2-dimensional grid can be represented in various ways, including natural language descriptions, ASCII-based representations, and images. For the ASCII representation, we developed a template to represent the task's visual grid using ASCII characters. These ASCII characters are then provided to the model as a replacement for the natural language descriptions of the visual grid, both for fine-tuning and evaluation. An example of an ASCII-based prompt is shown in Figure 21.

The evaluation results are shown in Figure 15. Our results indicate that GPT-4 and Llama3-70B perform better with natural language (NL) representations. This might be due to their predominant training on natural language data. However, the fine-tuned Llama3-8B-Uni model performs similarly with both NL and ASCII prompts, with final success rates of 54.12% and 53.18%, respectively.

In Figure 15b, we show Llama3-8B-Uni's performance across different epochs with NL and ASCII prompts. We find that the performance of Llama3-8B-Uni with NL and ASCII prompts converges at a similar rate, suggesting that fine-tuning helps the model adapt to ASCII-based task representations, making task representations less critical for fine-tuned models in our visual programming domain.

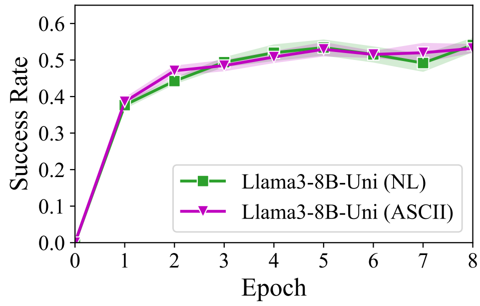
C.3 Fine-tuning Performance Across Different Epochs

Figure 16a illustrates the performance of fine-tuned models across different epochs. For the emulator-driven fine-tuning (Emu), we adjust the resampling interval to every three epochs, specifically at epochs 3 and 6. At epoch 3, we reuse the checkpoint from the standard fine-tuning (Uni) to save time and resources. As a result, the performance of the emulator-driven fine-tuning (Emu) matches that of the corresponding standard fine-tuning (Uni) up until epoch 3. Then, an emulator-driven resampling is performed at epoch 3, leading to further performance improvements compared to models without resampling. Notably, at the end of training, Llama2-7B-Emu achieves performance close to that of Llama3-8B-Uni, despite the latter being fine-tuned on a more advanced base model. This demonstrates the effectiveness of the curriculum designed by emulator-driven resampling in enhancing the performance of standard fine-tuning.

In Figure 16b, we show the fine-tuning performance across different epochs on the synthetic evaluation dataset SIMEVAL. This synthetic evaluation dataset exhibits the same distribution as the training dataset due to our splitting method. Emulator-driven resampling is performed at epochs 3 and 6 for both Llama3-8B-Emu and Llama2-7B-Emu. We find that standard fine-tuning without resampling leads to a smooth increase in performance across epochs, as seen in the Llama3-8B-Uni and

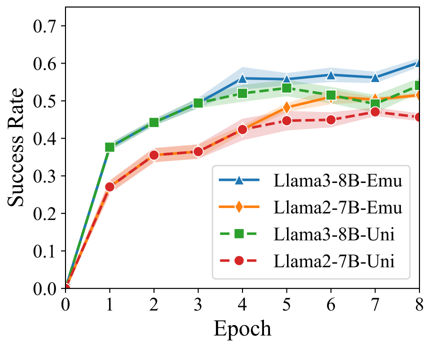
	Success Rates (%)	
	NL	ASCII
Base models		
GPT-4	**12.94**	5.88
Llama3-70B	**2.35**	1.18
Fine-tuned models		
Llama3-8B-Uni	**54.12** $\pm$ 1.78	53.18 $\pm$ 1.01

(a) Performance of base and fine-tuned models with NL and ASCII prompts.

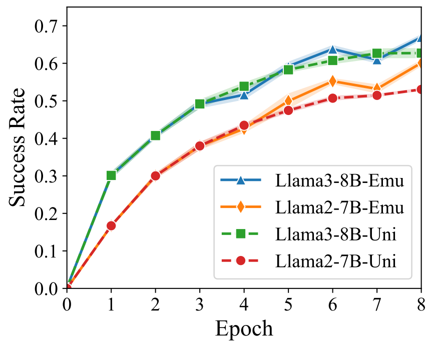


(b) Performance of Llama3-8B-Uni across epochs with NL and ASCII prompts.

Figure 15: Influence of task representations on model performance. We compare the performance of base models and fine-tuned models using natural language (NL) and ASCII prompts, respectively. (a) shows the success rates of base and fine-tuned models. (b) shows the performance of fine-tuned models across different epochs. Natural language prompts lead to better performance in base models. However, the fine-tuned Llama3-8B-Uni performs similarly with both NL and ASCII prompts.



(a) BASIC dataset.



(b) SIMEVAL dataset.

Figure 16: Fine-tuning performance across different epochs on two evaluation datasets. (a) shows the performance of fine-tuned models across different epochs on the evaluation dataset BASIC. (b) shows the fine-tuning performance across different epochs on the synthetic evaluation dataset SIMEVAL.

Llama2-7B-Uni curves. In contrast, emulator-driven fine-tuning with resampling shows slight performance fluctuations, particularly in the epochs immediately following resampling (i.e., epochs 4 and 7). The fluctuations in emulator-driven fine-tuning might be due to the resampling process altering the distribution of the training data, leading to a temporary drop in performance. However, in later epochs after resampling (e.g., epoch 8), the performance of resampling-based models outperforms that of the standard fine-tuning models, indicating the effectiveness of emulator-driven fine-tuning in improving fine-tuning performance.

C.4 Can Fine-tuned Models Learn Transferable Skills?

We explore whether fine-tuned models can develop transferable skills to solve tasks that are not seen during training. To investigate this, we first exclude all tasks involving math skills (e.g., Task 38 in Figure 2) from the training dataset, resulting in a reduced training dataset with 72k samples. Then, we fine-tune Llama3-8B on this reduced dataset using standard supervised learning, referring to the resulting model as *Llama3-8B-Uni (no-math)*. Next, we evaluate this model exclusively on math tasks from the evaluation datasets. The results are shown in Figure 17. Our results reveal that Llama3-8B-Uni (no-math) outperforms Llama3-70B, despite neither model being trained on math tasks. This suggests that the fine-tuned Llama3-8B-Uni (no-math) acquires certain transferable skills. However, compared to Llama3-8B-Uni, which was trained on the full dataset including math tasks, the no-math version performs much worse. This indicates that while Llama3-8B-Uni (no-math) learns some generalizable skills, it is less effective than the model trained directly on data that includes those skills.

	BASIC (10 tasks)	SIMEVAL (176 tasks)
Llama3-70B	0.00	0.00
Llama3-8B-Uni (no-math)	10.00 ± 10.00	6.25 ± 1.18
Llama3-8B-Uni	**40.00** ± 5.48	**38.98** ± 1.82

Figure 17: Success rates (%) of models on math tasks. Success rates of fine-tuned models are reported as mean and standard error across five seeds.

	HumanEval	HumanEval+	MBPP	MBPP+
Llama3-8B (Base)	36.6%	31.1%	62.4%	52.6%
Llama3-8B-Uni (Fine-tuned)	33.5%	26.8%	57.9%	46.8%
Δ (Fine-tuned - Base)	-3.1%	-4.3%	-4.5%	-5.8%

Figure 18: Pass@1 performance of Llama3-8B (Base) and the Llama3-8B-Uni (fine-tuned) on other program synthesis benchmarks, including HumanEval, HumanEval+, MBPP, and MBPP+. Fine-tuning on the SIM dataset leads to a performance drop of $3 \sim 6\%$ on these program synthesis benchmarks.

C.5 Impact of Domain-Specific Fine-Tuning on Other Benchmarks

We have shown that fine-tuning on the domain dataset SIM leads to performance improvements on out-of-distribution tasks within the same domain, compared to the base model without fine-tuning. However, it remains uncertain whether fine-tuning on our domain dataset would also enhance performance on tasks from different domains, such as Python program synthesis tasks.

To investigate this, we evaluate our fine-tuned Llama3-8B-Uni model on other Python program synthesis benchmarks, including HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023b), MBPP (Austin et al., 2021), and MBPP+ (Liu et al., 2023b). Unlike our benchmarks, these benchmarks focus on general Python program synthesis tasks from natural language or docstrings, without visual elements present in the benchmark tasks.

The results are presented in Figure 18. Our findings indicate that fine-tuning on our domain dataset SIM results in a slight performance drop ($3 \sim 6\%$) on these program synthesis benchmark tasks. We hypothesize that this is due to the SIM dataset's focus on visual programming tasks, which emphasize visual understanding, spatial reasoning, and planning—skills that are not directly applicable to other Python program synthesis tasks. Consequently, fine-tuning on our domain dataset does not provide additional knowledge for solving other benchmark tasks. Instead, the fine-tuning process may cause the model to forget some knowledge already acquired during the pre-training stage, leading to a performance drop in other benchmark tasks.

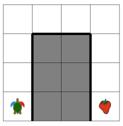
C.6 Case Study: Output Code Analysis on Perturbed Tasks

In the main paper, we presented a failure analysis by perturbing tasks and calculating the success rate. To illustrate the failure cases, we provide examples of output code from the evaluated models on these perturbed tasks, including GPT-4V, Llama3-70B, and Llama3-8B-Uni.

The output code is displayed in Figure 19. In the provided examples, we observe that GPT-4V has difficulty handling grid constraints and spatial reasoning. For example, in T and T_{A}, GPT-4V attempts to reach the strawberry by ignoring the walls. However, once the walls are removed (T_{B}), GPT-4V is able to successfully solve the task. Interestingly, GPT-4V fails to solve $\mathsf{T}_{\mathsf{A,B}}$, even though this task is conceptually simpler than T_{B} due to the absence of code constraints. Upon examining the code and the accompanying comments from GPT-4V, we found that it miscalculates the strawberry's coordinates, indicating a struggle with spatial reasoning. Additionally, we observed that moving the turtle closer to the strawberry consistently improves GPT-4V's performance, suggesting that long-path planning and spatial reasoning are challenging for GPT-4V. However, for Llama3-70B and Llama3-8B-Uni, we observe that neither model successfully solves any of the perturbed tasks.

Task	GPT-4V	Llama3-70B	Llama3-8B-Uni

Find the strawberry. Use at most 8 commands.



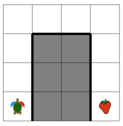
T

```
def run():
  # Turtle faces east
  turn_right()

  # Moves to (1,3)
  move_forward()

  # Moves to (2,3)
  move_forward()

  # Moves to (3,3) where the strawberry is located
  move_forward()
```

```
def run():
  move_forward()
  turn_right()
  move_forward()
  move_forward()
  turn_left()
  move_forward()
  move_forward()
  move_forward()
```

```
def run():
  move_forward()
  move_forward()
  turn_right()
  for i in range(3):
    move_forward()
  turn_right()
  for i in range(3):
    move_forward()
```

Find the strawberry.



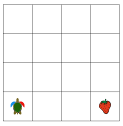
T_A

```
def run():
  # Start at (0,3) facing north
  # First, turn right to face east
  turn_right()

  # Move forward three times to reach (3,3)
  for i in range(3):
    move_forward()
```

```
def run():
  for i in range(3):
    move_forward()
    turn_right()
  move_forward()
  turn_left()
  move_forward()
  turn_right()
  move_forward()
  turn_left()
  for i in range(2):
    move_forward()
```

```
def run():
  move_forward()
  turn_right()
  move_forward()
  turn_left()
  move_forward()
  move_forward()
```

Find the strawberry. Use at most 8 commands.



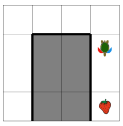
T_B

```
def run():
  turn_right() # Turtle faces east
  for i in range(3):
    move_forward() # Moves from (0,3) to (3,3)
```

```
def run():
  for i in range(3):
    move_backward()
  turn_right()
  move_forward()
  turn_left()
  move_forward()
  move_forward()
  turn_right()
  move_forward()
```

```
def run():
  move_forward()
  turn_right()
  for i in range(3):
    move_forward()
  turn_right()
  move_backward()
  for i in range(3):
    move_forward()
```

Find the strawberry. Use at most 8 commands.



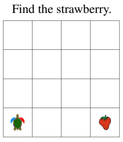
T_C

```
def run():
  move_forward() # Moves from (3,1) to (3,2)
  move_forward() # Moves from (3,2) to (3,3)
  ↪ where the strawberry is located
  turn_right() # Turns to face west
  turn_right() # Turns to face north
  turn_right() # Turns to face east
  turn_right() # Turns to face south again
  turn_right() # Turns to face west
  turn_right() # Turns to face north
```

```
def run():
  turn_left()
  move_forward()
  turn_right()
  move_forward()
  move_forward()
  turn_left()
  move_forward()
  move_forward()
```

```
def run():
  move_forward()
  turn_right()
  for i in range(3):
    move_forward()
  turn_right()
  move_forward()
  move_forward()
```

Find the strawberry.



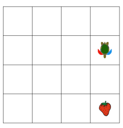
$T_{A,B}$

```
def run():
  # Turn right to face east
  turn_right()

  # Move forward three times to reach column 3
  for i in range(3):
    move_forward()

  # Turn right to face south
  turn_right()

  # Move forward to reach the strawberry at (3,3)
  move_forward()
```

```
def run():
  for i in range(3):
    move_forward()
  turn_right()
  move_forward()
  turn_left()
  for i in range(3):
    move_forward()
```

```
def run():
  move_forward()
  turn_right()
  move_forward()
  turn_right()
  move_forward()
  move_forward()
```

Find the strawberry. Use at most 8 commands.



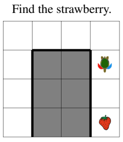
$T_{B,C}$

```
def run():
  # Moves from (3,1) to (3,2)
  move_forward()

  # Moves from (3,2) to (3,3)
  ↪ where the strawberry is located
  move_forward()
```

```
def run():
  turn_left()
  turn_left()
  for i in range(2):
    move_forward()
  turn_left()
  move_forward()
  turn_right()
  move_forward()
```

```
def run():
  move_forward()
  turn_right()
  move_forward()
  for i in range(2):
    move_forward()
    turn_right()
  turn_left()
  move_forward()
```

Find the strawberry.



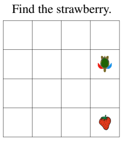
$T_{A,C}$

```
def run():
  # The turtle starts at (3,1) facing south.
  # First, move forward to (3,2).
  move_forward()

  # Then, move forward to (3,3)
  # where the strawberry is located.
  move_forward()
```

```
def run():
  turn_left()
  move_forward()
  turn_right()
  move_forward()
  turn_left()
  move_forward()
  turn_right()
  move_forward()
  turn_left()
  move_forward()
  turn_right()
  ... (repeat outputs)
```

```
def run():
  move_forward()
  turn_right()
  move_forward()
  turn_right()
  move_forward()
  move_forward()
  turn_left()
  move_forward()
  turn_right()
  move_forward()
```

Find the strawberry.



$T_{A,B,C}$

```
def run():
  # The turtle starts at (3,1) facing south.
  # Move forward to reach (3,2).
  move_forward()

  # Move forward again to reach (3,3)
  ↪ where the strawberry is located.
  move_forward()
```

```
def run():
  turn_left()
  turn_left()
  for i in range(2):
    move_forward()
  turn_left()
  for i in range(3):
    move_forward()
  turn_right()
  move_forward()
```

```
def run():
  turn_right()
  move_forward()
  turn_right()
  move_forward()
  turn_right()
  move_forward()
  move_forward()
  turn_left()
  move_forward()
  move_forward()
```

Figure 19: Output codes generated by GPT-4V, Llama3-70B, and Llama3-8B-Uni for various perturbations applied to a task T. The perturbations include removing code constraints (T_A), removing grid constraints (T_B), simplifying spatial relationships (T_C), and combinations of these perturbations ($T_{A,B}$, $T_{B,C}$, $T_{A,C}$, and $T_{A,B,C}$). Note that only the code is shown due to space limitations. The red line in the output code marks the point where the code first triggers an execution error or fails to successfully solve the task. GPT-4V successfully solves 5 out of 8 perturbed tasks, but Llama3-70B and fine-tuned Llama3-8B-Uni fail to solve any of the perturbed tasks.

D Prompts Used in the Benchmark

In this section, we present three types of prompts for program synthesis in the XLogoOnline-Mini domain. Figures 20 and 21 show examples of the prompts using natural language and ASCII representation, respectively. Figure 22 shows the prompt for the few-shot + CoT prompting.

Note that after the title "#### Available Python Functions" in prompts, we provide an explanation and two examples of the code format. This is intended for *base models*, such as GPT-family and Llama-family base models, to ensure they follow the desired code format. However, *fine-tuning models* does not need this code format in the prompt, as models are trained with formatted code directly. Therefore, we omit the code format and examples from the prompts when fine-tuning models.

Natural Language Prompt for Code Generation in XLogoOnline-Mini

You are presented with a visual programming task involving a goal, a grid, a turtle, various items (or lines). You need to write Python code that enables the turtle to accomplish the goal within the grid.

Grid and Turtle
- The task has a `m x n` grid. The coordinates of the grid cells are `(x, y)`, where `x` is the column number and `y` is the row number. The top-left cell has coordinates `(0, 0)`. - The turtle starts at a specific grid cell and faces one of four directions: North, East, South, or West.

Items
Each item in the grid is defined by three attributes:
- `count`: The number of identical items in that grid cell.
- `color`: The item's color. Options include red, green, blue, yellow, black, white, orange, purple, and pink.
- `name`: The type of the item, such as circle, rectangle, triangle, cross, strawberry, or lemon.

Lines
Sometimes, the grid doesn't contain any items but has lines with colors. You need to draw lines of the specified color to solve the task.

Grid Cell Properties
- A grid cell may be `accessible` or `forbidden`. The turtle can move to an accessible cell but not into a forbidden cell. If the turtle tries to move into a forbidden cell, it will crash and fail to solve the task.
- Grid cells can have walls on their edges (top, bottom, left, and right). The turtle cannot move through walls, otherwise it will crash and fail to solve the task.

Available Python Functions
To solve the task, you can use the following Python functions:
- `move_forward()`: This function moves the turtle forward in the direction it is facing by one grid cell. For example, if the turtle is at the position (x, y) and facing north, after executing move_forward(), the turtle will be at the position (x, y-1).
- `move_backward()`: This function moves the turtle backward in the direction it is facing by one grid cell. For example, if the turtle is at the position (x, y) and facing west, after executing `move_backward()`, the turtle will be at the position (x+1, y).
- `turn_left()`: This function makes the turtle turn left in the direction it is facing - by 90 degrees. For example, if the turtle is facing north, after executing `turn_left()`, the turtle will be facing west.
- `turn_right()`: This function makes the turtle turn right in the direction it is facing - by 90 degrees. For example, if the turtle is facing south, after executing `turn_right()`, the turtle will be facing west.
- `setpc(color)`: This function sets the pen color to the specified color. The available colors are: red, green, blue, yellow, black, white. The default pen color is black. The trajectory of the turtle is drawn with the pen color.
- `for` loop: This loop is used to repeat a set of commands a specified number of times. For example, `for i in range(4):` will repeat the commands inside the loop 4 times.
Your code should follow the format:
```python
def run():
    # Your solution code goes here
    pass
```
Here are some examples of the code:
Example 1:
```python
def run():
    move_forward()
    for i in range(4):
        move_forward()
        turn_left()
```
Example 2:
```python
def run():
    move_forward()
    setpc('red')
    for i in range(3):
        move_forward()
    turn_right()
    move_backward()
```

Now, write a CORRECT Python code that successfully solves the following task.
Task:
A 3x3 grid. The turtle starts at (1,1) facing north.
Accessible cells: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2).
Items in the grid:
- 1 red strawberry at (1,0).

Goal:
Find the strawberry.

CORRECT code:

Figure 20: An example of natural language prompt in the XLogoOnline-Mini domain.

ASCII-based Prompt for Program Synthesis in XLogoOnline-Mini

You are presented with a visual programming task involving a goal, a grid, a turtle, various items (or lines). You need to write Python code that enables the turtle to accomplish the goal within the grid.

Grid and Turtle
A task's grid contain a turtle and some items. The turtle can face one of four directions: North (`^`), South (`v`), East (`>`), or West (`<`). An item has three attributes: `count`, `color`, and `name`. The `count` indicates the number of identical items in that grid cell. The `color` specifies the item's color, and the `name` describes the item's type. Here are the possible options:
- Colors: Red (`R`), Green (`G`), Blue (`B`), Yellow (`Y`), Black (`K`), White (`W`), Orange (`O`), Purple (`U`), Pink (`P`)
- Names: Circle (`o`), Rectangle (`□`), Triangle (`△`) ,Cross (`X`), Strawberry (`S`), Lemon (`L`)
- Counts: `1`, `2`, `3`, `4`
- For example, `2RS` means two red strawberries.

We use the following symbols to describe a grid:
- `—` represents the top or bottom edge of a grid cell.
- `|` represents the left or right edge of a grid cell.
- `===` represents an upper or lower wall of a cell.
- `‖` represents a left or right wall of a cell.
- `+` represents the corner of a grid cell.
- `X` represents a forbidden cell that cannot be accessed.

Grid Cell Properties
- A grid cell may be `accessible` or `forbidden`. The turtle can move to an accessible cell but not into a forbidden cell. If the turtle tries to move into a forbidden cell, it will crash and fail to solve the task.
- Grid cells can have walls on their edges (top, bottom, left, and right). The turtle cannot move through walls, otherwise it will crash and fail to solve the task.

Available Python Functions
To solve the task, you can use the following Python functions:
- `move_forward()`: This function moves the turtle forward in the direction it is facing by one grid cell. For example, if the turtle is at the position (x, y) and facing north, after executing move_forward(), the turtle will be at the position (x, y-1).
- `move_backward()`: This function moves the turtle backward in the direction it is facing by one grid cell. For example, if the turtle is at the position (x, y) and facing west, after executing `move_backward()`, the turtle will be at the position (x+1, y).
- `turn_left()`: This function makes the turtle turn left in the direction it is facing - by 90 degrees. For example, if the turtle is facing north, after executing `turn_left()`, the turtle will be facing west.
- `turn_right()`: This function makes the turtle turn right in the direction it is facing - by 90 degrees. For example, if the turtle is facing south, after executing `turn_right()`, the turtle will be facing west.
- `setpc(color)`: This function sets the pen color to the specified color. The available colors are: red, green, blue, yellow, black, white. The default pen color is black. The trajectory of the turtle is drawn with the pen color.
- `for` loop: This loop is used to repeat a set of commands a specified number of times. For example, `for i in range(4):` will repeat the commands inside the loop 4 times.
Your code should follow the format:
```python
def run():
    # Your solution code goes here
    pass
```
Here are some examples of the code:
Example 1:
```python
def run():
    move_forward()
    for i in range(4):
        move_forward()
        turn_left()
```
Example 2:
```python
def run():
    move_forward()
    setpc('red')
    for i in range(3):
        move_forward()
    turn_right()
    move_backward()
```

Now, write a CORRECT Python code that successfully solves the following task:
Task:
```
+---+---+---+
|   |1RS|   |
+---+---+---+
|   | ^ |   |
+---+---+---+
|   |   |   |
+---+---+---+
```
Goal:
Find the strawberry.
CORRECT Code:

Figure 21: An example of ASCII-based prompt in the XLogoOnline-Mini domain.

Few-shot + CoT Prompt for Code Generation in XLogoOnline-Mini

You are presented with a visual programming task involving a goal, a grid, a turtle, various items (or lines). You need to write Python code that enables the turtle to accomplish the goal within the grid.

Grid and Turtle
- The task has a `m x n` grid. The coordinates of the grid cells are `(x, y)`, where `x` is the column number and `y` is the row number. The top-left cell has coordinates `(0, 0)`. - The turtle starts at a specific grid cell and faces one of four directions: North, East, South, or West.

Items
Each item in the grid is defined by three attributes:
- `count`: The number of identical items in that grid cell.
- `color`: The item's color. Options include red, green, blue, yellow, black, white, orange, purple, and pink.
- `name`: The type of the item, such as circle, rectangle, triangle, cross, strawberry, or lemon.

Lines
Sometimes, the grid doesn't contain any items but has lines with colors. You need to draw lines of the specified color to solve the task.

Grid Cell Properties
- A grid cell may be `accessible` or `forbidden`. The turtle can move to an accessible cell but not into a forbidden cell. If the turtle tries to move into a forbidden cell, it will crash and fail to solve the task.
- Grid cells can have walls on their edges (top, bottom, left, and right). The turtle cannot move through walls, otherwise it will crash and fail to solve the task.

Available Python Functions
To solve the task, you can use the following Python functions:
- `move_forward()`: This function moves the turtle forward in the direction it is facing by one grid cell. For example, if the turtle is at the position (x, y) and facing north, after executing move_forward(), the turtle will be at the position (x, y-1).
- `move_backward()`: This function moves the turtle backward in the direction it is facing by one grid cell. For example, if the turtle is at the position (x, y) and facing west, after executing `move_backward()`, the turtle will be at the position (x+1, y).
- `turn_left()`: This function makes the turtle turn left in the direction it is facing - by 90 degrees. For example, if the turtle is facing north, after executing `turn_left()`, the turtle will be facing west.
- `turn_right()`: This function makes the turtle turn right in the direction it is facing - by 90 degrees. For example, if the turtle is facing south, after executing `turn_right()`, the turtle will be facing west.
- `setpc(color)`: This function sets the pen color to the specified color. The available colors are: red, green, blue, yellow, black, white. The default pen color is black. The trajectory of the turtle is drawn with the pen color.
- `for` loop: This loop is used to repeat a set of commands a specified number of times. For example, `for i in range(4):` will repeat the commands inside the loop 4 times.

Your code should follow the format:
```python
def run():
    # Your solution code goes here
    pass
```

Here are some examples of the the tasks and their corresponding solution codes:
{few_shot_example_1}
{few_shot_example_2}
{few_shot_example_3}

Now, write a CORRECT Python code that successfully solves the following task.
Task:
A 3x3 grid. The turtle starts at (1,1) facing north.
Accessible cells: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2).
Items in the grid:
- 1 red strawberry at (1,0).

Goal:
Find the strawberry.

CORRECT code:

Figure 22: An example of the few-shot + CoT prompt in the XLogoOnline-Mini domain. This prompt has several placeholders to include details for few-shot examples.

{few_shot_example_1}

Task 1: A 4x4 grid. The turtle starts at (3,3) facing south.
Accessible cells: (3,0), (3,1), (3,2), (0,3), (1,3), (2,3), (3,3).
Forbidden cells: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2).
Lines in the grid:
- A red line from (3,0) to (3,1).
- A red line from (3,1) to (3,2).
- A red line from (3,2) to (3,3).
- A red line from (0,3) to (1,3).
- A red line from (1,3) to (2,3).
- A red line from (2,3) to (3,3).

Goal 1:
Draw the picture in red. Use at most 8 commands.

Code 1:
Explanation:
1. Starting Position and Initial Setup:
- The turtle starts at `(3, 3)`, facing south. To achieve the goal of drawing the picture in red, we first need to set the pen color to red using the `setpc('red')` command.

2. Plan the path:
- To draw the pattern, the path is `(3,3) -> (3,0) -> (3,3) -> (0,3)`.

3. Compiling the Path into Commands:
- The turtle starts at `(3, 3)`, facing south. To draw the first red line from `(3, 3)` to `(3, 0)`, we need to first set the pen color to red and then move back 3 times to reach from `(3, 3)` to `(3, 0)`.
- Now, the turtle is at `(3, 0)` and still facing south. To move back to `(3, 3)`, we move forward 3 times in the same direction (south).
- At `(3, 3)` and facing south, the turtle needs to turn right to face west.
- Now, the turtle is at `(3, 3)` and facing west. Move forward 3 times to reach from `(3, 3)` to `(0, 3)` facing west.
- Now the turtle has drawn the picture in red.

Putting it all together and notice that the solution code can use at most 8 commands. Here is the solution code:

```python
def run():
    setpc('red')
    for i in range(3):
        move_backward()
    for i in range(3):
        move_forward()
    turn_right()
    for i in range(3):
        move_forward()
```

Figure 23: The detail of the first few-shot + CoT example {few_shot_example_1}.

{few_shot_example_2}

Task 2:
A 3x3 grid. The turtle starts at (1,2) facing south.

Accessible cells: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2).
(1,0) has a wall at the bottom edge.
(2,0) has a wall at the bottom edge.
(1,1) has a wall at the top edge.
(2,1) has a wall at the top edge.

Items in the grid:
- 4 red strawberries at (1,0).
- 1 red strawberry at (2,0).
- 2 red strawberries at (2,1).
- 1 red strawberry at (2,2).

Goal 2:
Collect exactly 5 strawberries.

Code 2:
Explanation:
1. Starting Position and Initial Setup:
- The turtle starts at `(1, 2)`, facing south.
- The goal is to collect exactly 5 strawberries.
- There are 4 strawberries at `(1, 0)` and 1 strawberry at `(2, 0)`. By visiting these two cells, we can collect exactly 5 strawberries. We need a path to visit both `(1, 0)` and `(2, 0)` while considering walls that block movement between cells `(1, 0)` & `(1, 1)` and `(2, 0)` & `(2, 1)`.

2. Planning the Path:
- An ideal path to avoid the walls and collect strawberries is: `(1, 2) -> (1, 1) -> (0, 1) -> (0, 0) -> (2, 0)`.
- This path efficiently collects exactly 5 strawberries.

3. Compiling the Path into Commands:
- The turtle starts at `(1, 2)` facing south.
- To move from `(1, 2)` to `(1, 1)`, it needs to move backward. Now the turtle is at `(1, 1)` facing south.
- To move from `(1, 1)` to `(0, 1)`, it needs to turn right and move forward. Now it is at `(0, 1)` facing west.
- To move from `(0, 1)` to `(0, 0)`, it needs to turn right to face north, then move forward. Now it is at `(0, 0)` facing north.
- To move from `(0, 0)` to `(2, 0)`, it needs to turn right to face east, and finally, move forward twice to go from `(0, 0)` to `(2, 0)`.
- Now the turtle has collected exactly 5 strawberries.

Putting it all together. The solution code is:

```python
def run():
    move_backward()
    turn_right()
    move_forward()
    turn_right()
    move_forward()
    turn_right()
    move_forward()
    move_forward()
```

Figure 24: The detail of the second few-shot + CoT example {few_shot_example_2}.

{few_shot_example_3}

Task 3:
A 3x3 grid. The turtle starts at (2,2) facing south.
Accessible cells: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2).
Items in the grid:
- 1 blue triangle at (0,0).
- 1 red circle at (1,0).
- 1 green circle at (2,0).
- 1 blue circle at (0,1).
- 1 blue rectangle at (1,1).
- 1 red triangle at (2,1).
- 1 red cross at (0,2).
- 1 green rectangle at (1,2).

Goal 3:
Find the blue triangle without standing on a red. Use at most 3 'backward'.

Code 3:
Explanation:
1. Starting Position and Initial Setup:
- The turtle starts at `(2, 2)`, facing south.
- The goal is to find the blue triangle located at `(0, 0)` without standing on any red items.
- There are several items in the grid: red circle at `(1, 0)`, red triangle at `(2, 1)`, and red cross at `(0, 2)`.
- We need to avoid these red items and reach `(0, 0)` using at most 3 backward movements.

2. Planning the Path:
- Since we have to avoid red items and should use at most 3 backward movements, we need to plan an efficient path.
- A suitable path that avoids red items and stays within the permitted movements is: `(2, 2) -> (1, 2) -> (1, 1) -> (0, 1) -> (0, 0)`.

3. Compiling the Path into Commands:
- The turtle starts at `(2, 2)` facing south. First, turn left to face east.
- Move backward to reach `(1, 2)` facing east.
- Turn left again to face north.
- Move forward to reach `(1, 1)` facing north.
- Turn right to face east.
- Move backward to reach `(0, 1)` facing east.
- Turn right to face south.
- Move backward to reach `(0, 0)` facing south.

Putting it all together. The solution code is:

```python
def run():
turn_left()
move_backward()
turn_left()
move_forward()
turn_right()
move_backward()
turn_right()
move_backward()
```

Figure 25: The detail of the third few-shot + CoT example {few_shot_example_3}.